Notas de curso
Análisis y Diseño de Algoritmos

Rodrigo Alexander Castro Campos
UAM Azcapotzalco, División de CBI
https://racc.mx
https://omegaup.com/profile/rcc

Última revisión: 24 de noviembre de 2023

1. Problemas, instancias y algoritmos

Un problema computacional es una pregunta o cálculo cuyo resultado queremos obtener con una computadora. Un problema se especifica en términos generales, parametrizando los datos de entrada del mismo y describiendo lo que esperamos obtener como resultado final. Por ejemplo:

> **Problema:** Exponenciación de naturales.
> **Entrada**: Dos números naturales a y b.
> **Salida**: Un natural r que sea el valor de a^b

Una instancia es una asignación de valores a los datos de entrada de un problema. Para el problema anterior, un ejemplo de instancia es $a = 5, b = 2$ y su resultado es $r = 25$. Un algoritmo resuelve un problema si obtiene la respuesta correcta para cualquier instancia del mismo.

En computación, difícilmente ocurre que un problema tenga un único algoritmo correcto. Dada la diversidad de opciones con las que se cuentan al momento de diseñar un algoritmo, en este curso hay tres cosas que nos preocuparán: ¿cómo saber si un algoritmo dado es correcto?, ¿cómo saber qué tan eficiente es un algoritmo? y finalmente ¿cómo diseñar un algoritmo más eficiente si el que tengo es muy lento?

Los temas de correctitud y eficiencia los veremos desde el punto de vista teórico, pero los problemas también abordaremos de forma práctica mediante implementaciones en C++. Esto implica varias cosas:

- La eficiencia teórica no necesariamente corresponde uno a uno con la eficiencia en la práctica. El tiempo de ejecución de un algoritmo puede variar (a veces drásticamente) dependiendo de las capacidades del hardware en el que se ejecuta. Sin embargo, un algoritmo que es teóricamente muchísimo más eficiente que otro, casi siempre es muchísimo más eficiente también en la práctica.
- Los lenguajes de programación cercanos al hardware tienen tipos nativos que están limitados a los conceptos que el hardware puede modelar. Por ejemplo, si $n = 10^9$ entonces $n^3 = 10^{27}$ pero este último valor no cabe en un entero de 64 bits. Un algoritmo puede ser correcto en la teoría y su implementación ser incorrecta en la práctica para algunas instancias extremas. En este curso, cada vez que se proponga un problema a implementar, también se espeficarán los límites de los valores de entrada.

El diseño de algoritmos es un tema complicado que involucra todos los aspectos anteriormente mencionados (no es útil diseñar un algoritmo incorrecto y suele no ser útil diseñar un algoritmo más ineficiente). En este curso se abordarán únicamente algunas técnicas generales de diseño de algoritmos.

2. Correctitud de algoritmos recursivos

Un algoritmo recursivo es aquél que, para calcular la respuesta de una instancia dada, primero resuelve una o más instancias más fáciles del mismo problema y luego usa sus resultados para calcular la respuesta de la instancia original. Plantearemos un algoritmo recursivo para exponenciación de naturales.

Supongamos que queremos calcular 2^{20}. Por supuesto, una forma de hacerlo es multiplicar 2 veinte veces por sí mismo, pero hay una alternativa. ¿Qué pasa si, de alguna forma, ya supieramos cuánto vale $t = 2^{10}$? En este caso, el cálculo de 2^{20} se vuelve simplemente $t \times t$ ya que $2^{20} = 2^{10} \times 2^{10}$. Nótese cómo estamos resolviendo una instancia de exponenciación de naturales, *usando otra instancia de exponenciación de naturales*, aunque la otra instancia se intuye que es más fácil porque el exponente es menor. Para calcular 2^{10} que aún no sabemos cuánto vale, podemos aplicar la misma idea: ¿qué pasa si, de alguna forma, ya supieramos cuánto vale $t' = 2^5$? entonces 2^{10} es simplemente $t' \times t'$.

Si queremos volver a aplicar la idea de calcular a^b usando $a^{\frac{b}{2}}$, debemos tener cuidado ahora al calcular 2^5. Es cierto que $2^5 = 2^{2.5} \times 2^{2.5}$, pero $2^{2.5}$ no es una instancia del problema de exponenciación de naturales y termina empeorando la situación. En general, es mejor auxiliarnos de calcular $a^{\lfloor \frac{b}{2} \rfloor}$. Cuando el exponente es impar como en 2^5, ahora tenemos $t'' = 2^{\lfloor 2.5 \rfloor} = 2^2$ y hay que observar que $2^5 = t'' \times t'' \times 2$. En un planteamiento recursivo correcto, eventualmente aparecen instancias que son tan fáciles de resolver que podemos (y debemos) resolverlas directamente. Por ejemplo, $2^0 = 1$ y no tiene caso aplicar recursión. A estas instancias se les denomina casos base. El algoritmo recursivo para exponenciación natural es:

---

subrutina ExponenciaciónNatural(natural a, natural b)
  si $b = 0$
    regresa 1
  sino
    $t \leftarrow$ ExponenciaciónNatural$(a, \lfloor \frac{b}{2} \rfloor)$
    si b es par
      regresa $t \times t$
    sino
      regresa $t \times t \times a$

En el anexo A puede consultarse una implementación en C++ de éste y todos los demás algoritmos que se especifiquen formalmente en las notas. La implementación provista del algoritmo de exponenciación natural sólo funcionará si se invoca con números naturales (en particular en el exponente) y sólo regresará el resultado correcto si éste es representable en el `int` del lenguaje. Una implementación del algoritmo anterior que use enteros de precisión arbitraria será correcta sin importar la magnitud del resultado.

Aunque la intuición nos diga que el algoritmo anterior está bien, lo demostraremos formalmente. La demostración de correctitud de algoritmos recursivos suele ser sencilla porque es una aplicación directa de inducción matemática. En inducción matemática, partimos de un primer caso que es correcto y luego demostramos simbólicamente que los siguientes casos también lo son. Dado que en la función anterior el único valor que cambia durante la recursión es el exponente b, haremos inducción sobre b. Por simplicidad notacional, en la demostración nos referiremos a la función de exponenciación natural como f.

Demostración. Primero probaremos que $f(a, 0)$ es correcto. Sabemos que a^0 es 1 y vemos que el algoritmo regresa 1 cuando $b = 0$, por lo que el caso base es correcto. Ahora supondremos que $f(a, b')$ es correcto para toda $0 \leq b' < b$ y demostraremos que *f(a, b)* es correcto. Dado que el algoritmo se comporta de forma distinta cuando b es par que cuando no lo es, necesitamos manejar dos casos:

- Si b es par, entonces $b = 2k$. Dado que $b > 0$ entonces $k < b$. El exponente que usamos en la recursión es $\lfloor \frac{b}{2} \rfloor = \lfloor \frac{2k}{2} \rfloor = k$. Por la hipótesis de inducción, $t = f(a, k) = a^k$. El algoritmo regresa $t \times t = a^k \times a^k = a^{2k} = a^b$, por lo que el algoritmo es correcto en este caso.
- Si b es impar, entonces $b = 2k + 1$. Dado que $b > 0$ entonces $k < b$. El exponente que usamos en la recursión es $\lfloor \frac{b}{2} \rfloor = \lfloor \frac{2k+1}{2} \rfloor = k$. Por la hipótesis de inducción, $t = f(a, k) = a^k$. El algoritmo regresa $t \times t \times a = a^k \times a^k \times a = a^{2k+1} = a^b$, por lo que el algoritmo es correcto en este caso.

□

Otro ejemplo que trabajaremos es el del coeficiente binomial. El coeficiente binomial $\binom{n}{k}$ es la cantidad de combinaciones que se pueden formar escogiendo k objetos de los n disponibles. El coeficiente binomial está definido para enteros $0 \leq k \leq n$ y la fórmula más conocida es $\frac{n!}{k!(n-k)!}$. Sin embargo, nosotros presentaremos un planteamiento recursivo para calcular $\binom{n}{k}$ y demostraremos que es correcto. Al igual que el ejemplo anterior, en la demostración denotaremos como f a la función recursiva propuesta.

```
subrutina CoeficienteBinomial(natural n, natural k)
    si k = 0 o k = n
        regresa 1
    sino
        regresa CoeficienteBinomial(n − 1, k) + CoeficienteBinomial(n − 1, k − 1)
```

Demostración. Primero probaremos que $f(n, 0)$ y $f(n, n)$ son correctos. Cuando $k = 0$ entonces $\frac{n!}{k!(n-k)!} = \frac{n!}{0!(n)!}$ que es igual a 1. Cuando $k = n$ entonces $\frac{n!}{k!(n-k)!} = \frac{n!}{n!(0)!}$ que es igual a 1. Como el algoritmo regresa 1 en ambos casos, los casos bases de algoritmo recursivo son correctos.

Como los casos de $k = 0$ y $k = n$ están cubiertos por los casos bases, falta demostrar el caso donde $0 < k < n$. Supondremos que $f(n', k')$ es correcto para toda $0 \leq k' \leq n' < n$ y demostraremos que *f(n, k)* es correcto. Por la hipótesis de inducción tenemos lo siguiente:

$$f(n-1,k) = \frac{(n-1)!}{k!((n-1)-k)!} = \frac{(n-1)!}{k!(n-k-1)!}$$

$$f(n-1,k-1) = \frac{(n-1)!}{(k-1)!((n-1)-(k-1))!} = \frac{(n-1)!}{(k-1)!(n-k)!}$$

La suma de las dos expresiones es

$$\begin{aligned} f(n-1,k) + f(n-1,k-1) &= \frac{(n-1)!}{k!(n-k-1)!} + \frac{(n-1)!}{(k-1)!(n-k)!} \\ &= \frac{(n-1)!}{(k-1)!(n-k-1)!}\left(\frac{1}{k} + \frac{1}{n-k}\right) \\ &= \frac{(n-1)!}{(k-1)!(n-k-1)!}\left(\frac{(n-k)+(k)}{k(n-k)}\right) \\ &= \frac{(n-1)!}{(k-1)!(n-k-1)!}\left(\frac{n}{k(n-k)}\right) \\ &= \frac{n!}{k!(n-k)!} \end{aligned}$$

por lo que $f(n-1,k) + f(n-1,k-1) = \binom{n}{k}$, lo que concluye la demostración. □

Usaremos inducción matemática repetidamente durante el curso. Su aplicación en la demostración de correctitud de algoritmos recursivos es prácticamente directa. Sin embargo, demostrar la correctitud de algoritmos iterativos es más complicado.

2.1. Ejercicios

1. Demuestra que el siguiente algoritmo regresa $\frac{n(n+1)}{2}$ para todo entero $n \geq 0$.

```
subrutina f(natural n)
    si n = 0
        regresa 0
    sino
        regresa n + f(n − 1)
```

2. Demuestra que el siguiente algoritmo regresa $2^n - 1$ para todo entero $n \geq 0$.

```
subrutina f(natural n)
    si n = 0
        regresa 0
    sino
        regresa 1 + 2f(n − 1)
```

3. Demuestra que el siguiente algoritmo regresa $(n+1)! - 1$ para todo entero $n \geq 0$.

```
subrutina f(natural n)
    si n = 0
        regresa 0
    sino
        regresa n(n!) + f(n − 1)
```

4. Demuestra que el siguiente algoritmo regresa $a + b$ para todos los enteros $a, b \geq 0$.

```
subrutina f(natural a, natural b)
    si b = 0
        regresa a
    sino
        regresa f(a + 1, b − 1)
```

3. Correctitud de algoritmos iterativos

Un algoritmo iterativo es aquél que usa ciclos. El gran inconveniente que existe al demostrar la correctitud de algoritmos iterativos es que las variables pueden cambiar su valor conforme se itera. Esto nos obliga a hacer las siguientes preguntas:

- ¿Cuánto valen las variables antes de examinar la condición del ciclo por primera vez?
- ¿Cómo cambian los valores de las variables cuando pasamos de una iteración a otra?
- ¿Cuánto valen las variables al terminar el ciclo?

Si intentamos responder estas preguntas, podremos proponer las propiedades algorítmicas que se cumplen durante la ejecución del ciclo, las cuales se denominan invariantes. Las invariantes que propongamos deben servirnos para demostrar la correctitud del algoritmo y las demostraremos a su vez mediante expresiones matemáticas que estén respaldadas por lo que el algoritmo hace.

Nos limitaremos a demostrar la correctitud de algoritmos con sólo un ciclo *mientras*, donde el ciclo debe acabar normalmente mediante la condición de control y donde además casi todo el trabajo del algoritmo se realiza dentro del ciclo. Si necesitáramos demostrar la correctitud de algoritmos con más de un ciclo o con ciclos anidados, lo más seguro es que necesitemos realizar múltiples demostraciones y luego usar sus resultados como partes de una demostración mayor.

Consideremos una variable x que está declarada antes del ciclo y que potencialmente cambia su valor durante el ciclo. Observemos que si el ciclo realiza n iteraciones, entonces la condición del ciclo se evalua $n+1$ veces y es falsa durante la última evaluación. Denotaremos como x_t al valor de x donde la t-ésima evaluación de la condición del ciclo, comenzando por $t = 0$. Esto quiere decir que x_0 corresponde con el valor de x al examinar la condición del ciclo la primera vez y x_n es el valor de x cuando termina el ciclo. La demostración de las invariantes que propongamos la haremos por inducción sobre t.

Primero probaremos que el siguiente algoritmo regresa $a - b$ para $a, b \geq 0$.

```
subrutina RestaNatural(natural a, natural b)
    mientras b > 0
        a ← a − 1
        b ← b − 1
    regresa a
```

La intuición nos dice que a se decrementa iteración tras iteración hasta alcanzar el valor de la respuesta. La cantidad de decrementos en a debe depender de la cantidad de iteraciones hechas. La intuición también nos dice que lo que evita que a se decremente de más es que b eventualmente llega a 0 y detiene el ciclo.

Demostración. Propondremos las invariantes $a_t = a - t$ y $b_t = b - t$. Sabemos que $a_0 = a$, $b_0 = b$ y verificamos las invariantes $a_0 = a - 0 = a$ y $b_0 = b - 0 = b$, por lo que éstas son correctas inicialmente.

Ahora supondremos que las invariantes son correctas para $0 \leq t-1$ y demostraremos que también lo son para t (esto implica que la condición del ciclo fue cierta en $t-1$). Por la hipótesis de inducción, $a_{t-1} = a-(t-1) = a-t+1$ y $b_{t-1} = b-(t-1) = b-t+1$. Los pasos realizados dentro del ciclo son $a_t = a_{t-1}-1$ y $b_t = b_{t-1}-1$. Desarrollando tenemos $a_t = a-t+1-1 = a-t$ y $b_t = b-t+1-1 = b-t$, por lo que las invariantes son correctas. El valor de b_t se decrementa de uno en uno ya que $b_t - b_{t-1} = (b-t)-(b-(t-1)) = -1$, por lo que el ciclo termina cuando $b_{t'} = 0$. Por la invariante tenemos que $t' = b$. El algoritmo regresa $a_b = a-b$ por lo que es correcto. □

Ahora demostraremos que el siguiente algoritmo regresa $\sum_{i=1}^{n} i$ para $n \geq 0$.

---

subrutina Sumatoria(natural n)
  $r \leftarrow 0$, $k \leftarrow 1$
  mientras $k \leq n$
    $r \leftarrow r + k$
    $k \leftarrow k + 1$
  regresa r

La intuición nos dice que k se incrementa iteración tras iteración hasta que el ciclo se detiene, por lo que ésta debe jugar el rol de i en $\sum_{i=1}^{n} i$. Con respecto a r que aumenta en k en cada iteración, debe ser cierto que el primer valor que sumamos es $k = 1$ y el último es $k = n$.

Demostración. Propondremos las invariantes $r_t = \sum_{i=1}^{t} i$ y $k_t = t+1$. Sabemos que $r_0 = 0$, $k_0 = 1$ y verificamos las invariantes $r_0 = \sum_{i=1}^{0} i = 0$ y $k_0 = 0+1 = 1$, por lo que éstas son correctas inicialmente. Ahora supondremos que las invariantes son correctas para $0 \leq t-1$ y demostraremos que también lo son para t (esto implica que la condición del ciclo fue cierta en $t-1$). Por la hipótesis de inducción, $r_{t-1} = \sum_{i=1}^{t-1} i$ y $k_{t-1} = (t-1)+1 = t$. Los pasos realizados dentro del ciclo son $r_t = r_{t-1} + k_{t-1}$ y $k_t = k_{t-1}+1$. Desarrollando tenemos $r_t = \sum_{i=1}^{t-1} i + t = \sum_{i=1}^{t} i$ y $k_t = t+1$, por lo que las invariantes son correctas. El valor de k_t se incrementa de uno en uno ya que $k_t - k_{t-1} = t+1-t = 1$, por lo que el ciclo termina cuando $k_{t'} = n+1$. Por la invariante tenemos que $t' = n$. El algoritmo regresa $r_n = \sum_{i=1}^{n} i$. □

3.1. Ejercicios

1. Demuestra que el siguiente algoritmo regresa a^b para todo entero $a, b \geq 0$.

   subrutina f(natural a, natural b)
     $r \leftarrow 1$
     mientras $b > 0$
       $r \leftarrow r \times a$
       $b \leftarrow b - 1$
     regresa r

2. Demuestra que el siguiente algoritmo regresa $\frac{n(n+1)(2n+1)}{6}$ para todo entero $n \geq 0$.

   subrutina f(natural n)
     $r \leftarrow 0$, $k \leftarrow 1$
     mientras $k \leq n$
       $r \leftarrow r + k^2$
       $k \leftarrow k + 1$
     regresa r

4. Complejidad de algoritmos

Como se explicó anteriormente, un problema computacional describe un cálculo que se desea realizar a partir de datos de entrada. En este contexto, la complejidad de un algoritmo es la cantidad de recursos, generalmente tiempo o memoria, que el algoritmo utiliza durante su ejecución. La complejidad de un

algoritmo la expresaremos como una funcion $\mathbb{T}(L)$ donde L es la cantidad de bits usados para codificar los datos de entrada del mismo. Normalmente estimaremos $\mathbb{T}(L)$ contando la cantidad de operaciones de cierto tipo (asignaciones, comparaciones, etc.) que un algoritmo ejecuta. A continuación contrastaremos la complejidad de dos algoritmos, donde ambos toman un entero codificado en w bits:

```
subrutina CuentaBits(natural n)
    r ← 0
    mientras n ≠ 0
        r ← r + n mód 2
        n ← ⌊n/2⌋
    regresa r
```

```
subrutina Sumatoria(natural n)
    r ← 0, i ← 1
    mientras i ≤ n
        r ← r + i
        i ← i + 1
    regresa r
```

La estrategia usada por el algoritmo *CuentaBits* consiste en examinar la paridad de n el cual es su bit menos significativo, sumarlo en el acumulador y recorrer todos los bits de n un lugar a la derecha rellenando con ceros los bits de la izquierda (la división entre 2 tiene este efecto en base binaria). Denotaremos como $\overleftarrow{\mathbb{T}}(L)$ a la cantidad de asignaciones que el algoritmo realiza. El algoritmo ejecuta incondicionalmente una asignación y también ejecuta dos asignaciones por iteración. Si inicialmente $n = 0$ no se hace ninguna iteración del ciclo, pero si n tiene sus w bits prendidos entonces se realizan w iteraciones. Como la entrada es de tamaño w, entonces $w = L$ y $\overleftarrow{\mathbb{T}}(L) = 1+2L$ en el peor caso. La complejidad del algoritmo en cuanto a asignaciones entonces es lineal con respecto a L.

El segundo algoritmo es una típica implementación de $\sum_{i=1}^{n} i$. Al igual que en el caso anterior, si $n = 0$ entonces no se hace ninguna iteración del ciclo, pero si n tiene sus w bits prendidos ahora la situacion es diferente, porque el máximo valor de n es $2^w - 1$. Es fácil obervar que el ciclo realiza n iteraciones (aunque i debe tener suficientes bits para superar el valor de n). El algoritmo ejecuta incondicionalmente dos asignaciones y también ejecuta dos asignaciones por iteración. Como la entrada es de tamaño w, entonces $w = L$ y $\overleftarrow{\mathbb{T}}(L) = 2 + 2(2^L - 1)$ en el peor caso, lo que es exponencial en L.

Lo anterior no implica que todo algoritmo que itere n veces es exponencial cuando n es parte de la entrada. A continuación se muestra un algoritmo que suma los elementos de un arreglo con índices que comienzan a partir de 0, tal como se definen en la mayoría de los lenguajes de programación actuales:

```
subrutina Acumulación(natural n, arreglo(entero^n) a)
    r ← 0, i ← 0
    mientras i < n
        r ← r + a[i]
        i ← i + 1
    regresa r
```

Al igual que los casos anteriores, si $n = 0$ entonces no se hace ninguna iteración del ciclo y si n tiene sus w bits prendidos entonces se hacen $2^w - 1$ iteraciones. Aquí tenemos que $\overleftarrow{\mathbb{T}}(L) = 2 + 2(2^w - 1) = 2(2^w)$. Sin embargo, también es cierto que $L = w + n(w) = w + (2^w - 1)(w) = w(1 + 2^w - 1) = w(2^w)$. Aunque el número de asignaciones $\overleftarrow{\mathbb{T}}(L) = 2(2^w)$ parece ser sublineal por un factor de $\frac{2}{w}$ con respecto a $L = w(2^w)$, en una computadora teórica cada asignacion de enteros implica copiar w bits y por consiguiente, la cantidad de bits copiados por asignaciones es $2w(2^w)$ que es lineal con respecto a L. En general, simplificaremos el análisis de un algoritmo de modo que, cuando L es una función de un entero n que también es parte de la entrada, evitaremos mencionar L y w y haremos el análisis con respecto a n. Esto lo hará más intuitivo y cercano a lo que ocurre en una computadora real, donde w es una constante.

Siempre nos interesará determinar la complejidad de un algoritmo en términos de su peor caso. En el caso de un algoritmo con múltiples subcasos, el peor caso del algoritmo coincidirá con el peor de sus subcasos; si cada una de las iteraciones de un ciclo realiza p operaciones y el ciclo realiza n iteraciones, entonces el ciclo realizará $n(p)$ operacion; finalmente, si un algoritmo evalúa una o más funciones, la cantidad de operaciones del algoritmo es la suma de las operaciones realizadas por esas funciones.

Determinar la complejidad de un algoritmo recursivo puede ser algo complicado. Por ejemplo, continuación se muestra una implementación que calcula el n-ésimo término de la secuencia Fibonacci:

```
subrutina Fibonacci(natural n)
    si n = 0 o n = 1
        regresa n
    sino
        regresa Fibonacci(n − 1) + Fibonacci(n − 2)
```

Sabemos que n tiene w bits y definiremos $\overset{\pm}{\mathbb{T}}(L)$ con $L = w$ como la cantidad de operaciones aritméticas que realiza *Fibonacci* en términos del tamaño L de la entrada. También introduciremos la función $T(n)$ que cuenta lo mismo pero en función del valor de n. Si n tiene w bits y todos ellos están prendidos, entonces $n = 2^w - 1$ en el peor caso. Tenemos que $T(0) = T(1) = 0$ y que $T(n) = T(n-1) + T(n-2) + 3$ para $n > 1$. Una aproximación burda pero relativamente buena para los primeros 40 términos de T es $T(n) \approx 1.65^n$. Esto no es coincidencia: tanto el valor de $Fibonacci(n)$ como el de $T(n)$ se parecen a 2^n, ya que aunque $T(n)$ no es exactamente el doble del caso anterior, sí es la suma de los dos anteriores. El crecimiento de $T(n)$ es exponencial en n y el valor de n es exponencial en $w = L$, por lo que, para *Fibonacci*, $\overset{\pm}{\mathbb{T}}(L)$ es doblemente exponencial con respecto a L.

4.1. Ejercicios

1. Encuentra una función $\overset{\leftarrow}{\mathbb{T}}(L)$ que cuente la cantidad de asignaciones que realiza $f(n)$ en el peor caso. Recuerda que L es el tamaño de la entrada y que un entero se codifica con w bits.

```
subrutina f(natural n)
    r ← n
    si n mód 2 = 0
        r ← 5 × r
    regresa r
```

2. Encuentra una función $\overset{\leftarrow}{\mathbb{T}}(L)$ que cuente la cantidad de asignaciones que realiza $g(n)$ en el peor caso si $n = 2^k$. Recuerda que L es el tamaño de la entrada y que un entero se codifica con w bits.

```
subrutina f(natural n)
    r ← 0, i ← 1
    mientras i < n
        r ← r + 1
        i ← 2 × i
    regresa r
```

5. Solución de recurrencias

En los algoritmos de búsqueda y ordenamiento sobre un arreglo, existen dos operaciones que generalmente dominan el tiempo de ejecución total: las comparaciones entre elementos y las asignaciones entre elementos. A su vez, muchos de estos algoritmos son recursivos pero fáciles de análizar en su peor caso, por lo que se prestan para calcular exactamente cuántas de estas operaciones realizan.

El primer ejemplo que trabajaremos es el de búsqueda binaria. Para este ejemplo, nos concentraremos en calcular la cantidad de comparaciones realizadas en el peor caso (cuando lo que buscamos nunca lo encontramos) sobre un arreglo de $n = 2^k$ elementos. Recordemos que el algoritmo de búsqueda binaria

```
subrutina BúsquedaBinaria(natural n, arreglo(entero^n) a, natural b)
    si n = 0
        regresa falso
    sino si b < a[⌊n/2⌋]
        regresa BúsquedaBinaria(⌊n/2⌋, a[0 ... ⌊n/2⌋ − 1], b)
    sino si b > a[⌊n/2⌋]
        regresa BúsquedaBinaria(n − ⌊n/2⌋ − 1, a[⌊n/2⌋ + 1 ... n − 1], b)
    sino
        regresa verdadero
```

primero compara el elemento buscado con el de enmedio del arreglo, repitiendo la búsqueda sobre la mitad izquierda o sobre la mitad derecha dependiendo del resultado de la comparación. Cuando n es par, no hay un elemento que esté exactamente a la mitad del arreglo, así que al tomar un elemento que esté aproximadamente enmedio y luego descartarlo, una de las dos mitades queda con un elemento menos que la otra. En el peor caso tendríamos hacer recursión sobre el lado más grande, el cual tiene exactamente $\frac{n}{2}$ elementos. Si $n = 2^k$ entonces el valor de n enviado en la recursión siempre es par hasta llegar a $n = 1$.

Definiremos $T(n)$ como la cantidad de comparaciones que búsqueda binaria realiza sobre un arreglo de n elementos. Obsérvese que no consideraremos $\mathbb{T}$, L ni w ya que L es lineal con respecto a n este caso (el arreglo forma parte de la entrada del algoritmo). Esto simplificará mucho el análisis.

$$T(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 + T(\lfloor \frac{n}{2} \rfloor) & \text{si } n > 0 \end{cases}$$

Una de las técnicas más empleadas para calcular una fórmula no recursiva para $T(n)$ es la técnica de sustitución repetida. En esta técnica, el valor de $T(n)$ se desarrolla recursivamente de forma repetida hasta que aparezca un patrón no recursivo. Después se deberá demostrar que el patrón en efecto es correcto, usualmente mediante inducción matemática. Desarrollando $T(n)$ para búsqueda binaria obtenemos:

$$\begin{aligned} T(n) &= 1 + T\left(\frac{n}{2}\right) \\ &= 1 + 1 + T\left(\frac{n}{4}\right) = 2 + T\left(\frac{n}{4}\right) \\ &= 1 + 1 + 1 + T\left(\frac{n}{8}\right) = 3 + T\left(\frac{n}{8}\right) \\ &= 1 + 1 + 1 + 1 + T\left(\frac{n}{16}\right) = 4 + T\left(\frac{n}{16}\right) \\ &\dots \end{aligned}$$

El patrón que aparece es $T(k) = \log_2(k) + T(\frac{n}{k})$ y la fracción se vuelve 1 para $T(n) = \log_2(n) + T(\frac{n}{n}) = \log_2(n) + T(1)$, donde al desarrollar tenemos $T(n) = \log_2(n) + 1 + T(0) = \log_2(n) + 1$. Con esto podemos proponer la siguiente definición alternativa de $T(n)$:

$$T(n) = \begin{cases} 0 & \text{si } n = 0 \\ \log_2(n) + 1 & \text{si } n > 0 \end{cases}$$

Aunque el patrón es correcto, debemos demostrarlo formalmente y lo haremos por inducción.

Demostración. Claramente, las definiciones coinciden para $T(0)$. Para el caso recursivo, demostraremos primero el caso $T(1)$. Tenemos que $T(1) = 1 + T(0) = 1$ según la definición original, mientras $T(1) = \log_2(1) + 1 = 1$ según la definición propuesta. Ahora supondremos que $T(\frac{n}{2})$ es correcta y demostraremos que $T(n)$ también lo es. Tenemos que $T(n) = 1 + T(\frac{n}{2})$ y eso es igual a $1 + 1 + \log_2(\frac{n}{2})$ por la hipótesis de inducción. Desarrollando tenemos $T(n) = 1 + \log_2(2) + \log_2(\frac{n}{2}) = 1 + \log_2(\frac{2n}{2}) = 1 + \log_2(n)$. □

El algoritmo de ordenamiento por mezcla es un algoritmo que primero ordena cada mitad de un arreglo de n elementos de forma recursiva y luego los mezcla de forma ordenada. Todas las comparaciones y asignaciones de elementos se realizan durante la mezcla, pero no siempre se hacen la misma cantidad de comparaciones. Supongamos que $n = 2^k$ y que la mezcla toma dos arreglos de $\frac{n}{2}$ elementos cada uno:

```
subrutina OrdenamientoPorMezcla(natural n, arreglo(entero^n) a)
    si n ≤ 1
        regresa a
    sino
        t_1 = OrdenamientoPorMezcla(⌊n/2⌋, a[0 ... ⌊n/2⌋ − 1])
        t_2 = OrdenamientoPorMezcla(n − ⌊n/2⌋, a[⌊n/2⌋ ... n − 1])
        regresa MezclaOrdenada(|t_1|, t_1, |t_2|, t_2)
```

```
subrutina MezclaOrdenada(natural n, arreglo(entero^n) a, natural m, arreglo(entero^m) b)
    r ← [ ], i ← 0, j ← 0
    mientras i < n y j < m
        si a[i] < b[j]
            r ← (r)(a[i])
            i ← i + 1
        sino
            r ← (r)(b[j])
            j ← j + 1
    regresa (r)(a[i ... n − 1])(b[j ... m − 1])
```

- Cuando todos los elementos del primer arreglo son menores que los del segundo (o viceversa), entonces se necesitan $\frac{n}{2}$ comparaciones para darnos cuenta que el siguiente elemento a colocar en el resultado siempre proviene del mismo arreglo. Al acabarse este arreglo, los $\frac{n}{2}$ elementos del arreglo restante pueden colocarse al final sin usar comparaciones adicionales entre elementos. La cuenta de comparaciones para el algoritmo de ordenamiento por mezcla cuando la mezcla siempre entra al mejor caso es:

$$T_1(n) = \begin{cases} 0 & \text{si } n \leq 1 \\ \frac{n}{2} + 2T_1(\frac{n}{2}) & \text{si } n > 1 \end{cases}$$

- Cuando la mezcla se obtiene intercalando los elementos de ambos arreglos, entonces cada comparacion sirve para colocar alguno de los elementos en el resultado. Cuando sólo falta un elemento por considerar, éste ya no tiene con quién compararse. La situación descrita anteriormente realiza $n-1$ comparaciones y es el peor caso del algoritmo de mezcla. La cuenta de comparaciones para el algoritmo de ordenamiento por mezcla cuando la mezcla siempre entra al peor caso es:

$$T_2(n) = \begin{cases} 0 & \text{si } n \leq 1 \\ n - 1 + 2T_2(\frac{n}{2}) & \text{si } n > 1 \end{cases}$$

- La mezcla siempre hace $\frac{n}{2} + \frac{n}{2} = n$ asignaciones porque todos los elementos de ambos arreglos deben colocarse en el resultado. La cuenta de asignaciones para el algoritmo de ordenamiento por mezcla es:

$$T_3(n) = \begin{cases} 0 & \text{si } n \leq 1 \\ n + 2T_3(\frac{n}{2}) & \text{si } n > 1 \end{cases}$$

Dado que $T_3(n)$ es la función que más crece de las tres, resolveremos ésta de forma exacta. Sea $T = T_3$, obtenemos lo siguiente mediante la técnica de sustitución repetida:

$$\begin{aligned} T(n) &= n + 2T\left(\frac{n}{2}\right) \\ &= n + \frac{2n}{2} + 4T\left(\frac{n}{4}\right) = 2n + 4T\left(\frac{n}{4}\right) \\ &= 2n + \frac{4n}{4} + 8T\left(\frac{n}{8}\right) = 3n + 8T\left(\frac{n}{8}\right) \\ &= 3n + \frac{8n}{8} + 16T\left(\frac{n}{16}\right) = 4n + 16T\left(\frac{n}{16}\right) \\ &\dots \end{aligned}$$

El patrón que aparece es $T(k) = \log_2(k)n + kT(\frac{n}{k})$ y la fracción se vuelve 1 para $T(n) = \log_2(n)n + nT(\frac{n}{n})$, donde al desarrollar tenemos $T(n) = \log_2(n)n + nT(1) = \log_2(n)n$. Con esto podemos proponer la siguiente definición alternativa de $T(n)$:

$$T(n) = \begin{cases} 0 & \text{si } n \leq 1 \\ n\log_2(n) & \text{si } n > 1 \end{cases}$$

Aunque el patrón es correcto, debemos demostrarlo formalmente y lo haremos por inducción.

Demostración. Claramente, ambas definiciones coinciden para $T(0)$ y $T(1)$. Para el caso recursivo, demostraremos primero el caso $T(2)$. Tenemos que $T(2) = 2 + 2T(\frac{2}{2}) = 2$ según la definición original, mientras $T(2) = 2\log_2(2) = 2$ según la definición propuesta. Ahora supondremos que $T(\frac{n}{2})$ es correcta y demostraremos que $T(n)$ también lo es. Tenemos que $T(n) = n + 2T(\frac{n}{2})$ y eso es igual a $n + 2(\frac{n}{2}\log_2(\frac{n}{2}))$ por la hipótesis de inducción. Desarrollando tenemos $T(n) = n + n\log_2(\frac{n}{2}) = n(1 + \log_2(\frac{n}{2})) = n(\log_2(2) + \log_2(\frac{n}{2})) = n(\log_2(\frac{2n}{2})) = n\log_2(n)$ que es lo que queríamos demostrar. □

5.1. Ejercicios

1. Resuelve exactamente la siguiente recurrencia:

$$T(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 + T(n-1) & \text{si } n > 0 \end{cases}$$

2. Resuelve exactamente la siguiente recurrencia:

$$T(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 + 2T(n-1) & \text{si } n > 0 \end{cases}$$

3. Resuelve exactamente la recurrencia T_2 de ordenamiento por mezcla para $n = 2^k$.

4. Resuelve el problema `https://omegaup.com/arena/problem/Resolviendo-una-recurrencia`.

5. Resuelve el problema `https://omegaup.com/arena/problem/Resolviendo-una-recurrencia-la-r`.

6. Notación de comportamiento asintótico

Rara vez nos interesa contar la cantidad exacta de operaciones que realiza un algoritmo y nos es suficiente conocer el comportamiento de $\mathbb{T}(L)$ a grandes rasgos. Esto es, nos interesa saber si $\mathbb{T}(L)$ es constante, logarítmica, lineal, polilogarítmica, polinomial, exponencial o factorial, por mencionar algunos grupos de funciones. Eso quiere decir que si $\mathbb{T}(L) = 2L + 8$ o si $\mathbb{T}(L) = 9L - 5$, para nosotros ambas serán simplemente funciones lineales. Para formalizar esto, introduciremos tres notaciones:

- Diremos que $f(n) \in O(g(n))$ si existen $n_0, c > 0$ tal que $f(n) \leq cg(n)$ para toda $n \geq n_0$. En este caso, $g(n)$ es una cota superior del comportamiento asintótico de $f(n)$.
- Diremos que $f(n) \in \Omega(g(n))$ si existen $n_0, c > 0$ tal que $f(n) \geq cg(n)$ para toda $n \geq n_0$. En este caso, se dice que $g(n)$ es una cota inferior del comportamiento asintótico de $f(n)$.
- Diremos que $f(n) \in \Theta(g(n))$ si $f(n) \in O(g(n))$ y $f(n) \in \Omega(g(n))$. En este caso, $g(n)$ es una cota justa del comportamiento asintótico de $f(n)$.

Como siempre, tendremos que demostrar que $f(n)$ y $g(n)$ cumplen esas relaciones. Los casos difíciles generalmente se probarán por inducción matemática. A continuación se muestran algunos ejemplos:

Ejemplo 1. Demostrar que $n^2 - 2n - 1 \in O(n^2)$.

Demostración. Si elegimos $n_0 = 1$ y $c = 1$, es inmediato ver que $n^2 - 2n - 1 \leq n^2$ para toda $n \geq 1$. □

Ejemplo 2. Demuestra que $5n + 8 \in O(n)$.

Demostración. Si elegimos $n_0 = 8$ y $c = 6$, debemos demostrar que $5n+8 \leq 6n$ para toda $n \geq 8$. Podemos reescribir $6n$ como $5n + n$ y es fácil ver que $5n + 8 \leq 5n + n$ para $n \geq 8$. □

Ejemplo 3. Demuestra que $5n + 2 \in \Omega(1)$.

Demostración. Si elegimos $n_0 = 1$ y $c = 1$, es inmediato ver que $5n + 2 \geq 1$ para toda $n \geq 1$. □

Ejemplo 4. Demuestra que $n^2 - 2n \in \Omega(n)$.

Demostración. Si elegimos $n_0 = 10$ y $c = 1$, debemos demostrar que $n^2 - 2n \geq n$ para toda $n \geq 10$. Esto podemos hacerlo demostrando que $n^2 \geq 3n$ para $n \geq 10$. Para $n = 10$ tenemos que $n^2 = 100$ y $3n = 30$, por lo que es cierto en este caso. Ahora supondremos que $n^2 \geq 3n$ es cierta y demostraremos que para $n + 1$ también lo es. Desarrollando tenemos $(n + 1)^2 = n^2 + 2n + 1 \geq 3n + 3$ y sabemos por hipótesis de inducción que $n^2 \geq 3n$, por lo que sólo falta probar que $2n + 1 \geq 3$ es directo para $n \geq 10$. □

Ejemplo 5. Demuestra que $3 \in \Theta(1)$.

Demostración. Primero demostraremos que $3 \in O(1)$. Si elegimos $n_0 = 1$ y $c = 5$, debemos demostrar que $3 \leq 5$ para toda $n \geq 1$, lo cual es directo. Ahora demostraremos que $3 \in \Omega(1)$. Si elegimos $n_0 = 1$ y $c = 1$, debemos demostrar que $3 \geq 1$ para toda $n \geq 1$, lo cual es directo. □

Es de particular interés la notación O, ya que ésta se usa frecuentemente para describir la complejidad de un algoritmo. Por ejemplo, si un algoritmo tiene $\mathbb{T}(L) = 2L + 8$ entonces $\mathbb{T}(L) \in O(L)$. Si otro algoritmo tiene $\mathbb{T}(L) = 3L^3 + 10L^2 - 550$ entonces $\mathbb{T}(L) \in O(L^3)$. Es decir, la notación O nos permite identificar cuál de los sumandos es el que más impacto tiene en el crecimiento asintótico de la función, ignorando constantes y la magnitud de los coeficientes. Por supuesto, si $\mathbb{T}(L) \in O(L^3)$ entonces también $\mathbb{T}(L) \in O(L^4)$, pero generalmente buscaremos la función $O(g(L))$ asintóticamente más justa.

Generalmente se considera que el algoritmo más rápido para un problema es el de menor crecimiento asintótico (en la práctica esto no es necesariamente cierto, ya que la notación O esconde constantes que pueden ser astronómicamente grandes). En ese sentido, si el algoritmo de menor crecimiento asintónico es $O(f(n))$, se dice que el problema también es $O(f(n))$. Por ejemplo, el problema de búsqueda lineal es $O(n)$ donde n es la cantidad de elementos sobre los que se busca, ya que el algoritmo que revisa todos los elementos realiza justamente $O(n)$ comparaciones. También se sabe que búsqueda lineal no puede resolverse más rápido que $O(n)$, por lo que el problema tambien es $\Omega(n)$ y por consiguiente es $\Theta(n)$. Desafortunadamente no se sabe la complejidad $\Theta(f(n))$ para todos los problemas de interés. Por ejemplo, la multiplicación de dos matrices cuadradas de tamaño $n \times n$ se puede hacer en tiempo $O(n^3)$ y se cree que existe un algoritmo de tiempo $O(n^2)$, pero hasta el día de hoy nadie lo ha encontrado. Claramente no se puede hacer más rápido que $O(n^2)$, ya la entrada como la salida son de esa magnitud.

6.1. Ejercicios

1. Demuestra que $2^n + 5 \in \Theta(2^n)$.
2. Demuestra que $2^n \in O(n!)$.
3. Demuestra que $n^3 - 2n^2 - 4n \in \Omega(n)$.

7. El teorema maestro para el análisis de recurrencias

El teorema maestro es un resultado que permite determinar la complejidad asintótica $O(g(n))$ de muchas recurrencias de forma casi inmediata, sin tener que demostrar nada adicional. Supongamos una recurrencia $T(n)$ de la siguiente forma:

$$T(n) = \begin{cases} k_n & \text{si } n \leq n_0 \\ aT\left(\frac{n}{b}\right) + f(n) & \text{si } n > n_0 \end{cases}$$

En esta definición, la función $T(n)$ regresa una constante k_n para todas las $n \leq n_0$, que son los casos que no requieren recursión. Para el caso recursivo, necesitaremos identificar cuál de los siguientes casos aplica según los valores de a, b y la función $f(n)$. Notemos que a es la cantidad de subproblemas en los que se divide la recursion, $\frac{1}{b}$ es la fracción del problema que enviamos a la recursión y $f(n)$ es el trabajo local que realizamos en la llamada recursiva actual. Definiremos $r = \log_b(a)$ como una medida que relaciona la cantidad de subproblemas con la fracción del problema recursivo. Entonces tenemos los siguientes casos:

1. Si $f(n) \in O(n^c)$ con $c < r$, entonces $T(n) \in \Theta(n^r)$. El trabajo local se ve opacado por el trabajo causado por la cantidad y el tamaño de los subproblemas.
2. Si $f(n) \in \Theta(n^r \log_2(n)^c)$ para $c \geq 0$ entonces $T(n) \in \Theta(n^r \log_2(n)^{c+1})$. El trabajo local es comparable con el trabajo causado por la cantidad y el tamaño de los subproblemas.
3. Si $f(n) \in \Omega(n^c)$ con $c > r$, entonces $T(n) \in \Theta(f(n))$ siempre y cuando $af(\frac{n}{b}) \leq kf(n)$ para alguna $k < 1$ y un valor suficientemente grande de n. El trabajo local domina al trabajo causado por la cantidad y el tamaño de los subproblemas.

Ignorando los casos base, un ejemplo de recurrencia del tipo 1 es $T(n) = 8T(\frac{n}{2}) + 1$. A ojo se ve que la cantidad de subproblemas es exagerada y el trabajo local es mínimo. Tenemos $r = \log_2(8) = 3$ y $f(n) \in O(n^c)$ con $c = 0$. Como $0 < 3$ entonces $T(n) \in \Theta(n^3)$.

Una recurrencia del tipo 2 es $T(n) = 2T(\frac{n}{2}) + n$. Esta es la recurrencia de ordenamiento por mezcla. Tenemos $r = \log_2(2) = 1$ y $f(n) \in O(n^r \log_2(n)^c)$ con $c = 0$. Entonces por el teorema maestro tenemos que $T(n) \in \Theta(n \log_2(n)$. Otra recurrencia del tipo 2 es $T(n) = T(\frac{n}{2}) + 1$. Esta es la recurrencia de búsqueda binaria. Tenemos $r = \log_2(1) = 0$ y $f(n) \in O(n^r \log_2(n)^c)$ con $c = 0$. Entonces por el teorema maestro tenemos que $T(n) \in \Theta(\log_2(n))$.

Una recurrencia del tipo 3 es $T(n) = T(\frac{n}{2}) + n$. A ojo se ve que por sustitución repetida el patrón que aparece es $T(n) = \frac{n}{1} + \frac{n}{2} + \frac{n}{4} + \ldots$, lo cual vale aproximadamente $2n$. Más aún, el trabajo local de $T(n)$ que es $f(n) = n$ es mayor que el trabajo total realizado en el nivel inmediato inferior. Tenemos $r = \log_2(1) = 0$ y $f(n) \in \Omega(n^1)$ con $c = 1$. Como $1 > 0$ entonces $T(n) \in \Theta(n)$.

Si una recurrencia no coincide con alguno de los casos del teorema maestro, tendrá que resolverse de la forma cotidiana. De todos modos, a partir de este momento nos interesará más conocer el crecimiento del tiempo de un algoritmo en términos asintóticos, que conocerlo de forma exacta.

7.1. Ejercicios

1. Usando el teorema maestro, determina el comportamiento asintótico de $T(n) = T(\frac{n}{4}) + n^2$.
2. Usando el teorema maestro, determina el comportamiento asintótico de $T(n) = 4T(\frac{n}{4}) + n^2$.
3. Usando el teorema maestro, determina el comportamiento asintótico de $T(n) = T(\frac{n}{2}) + \log_2(n)$.

8. Divide y vencerás

Divide y vencerás es una técnica general de diseño de algoritmos que está basada en descomponer una instancia en subintances más fáciles cuyas soluciones nos sirvan para resolver la instancia original. En pocas palabras, para resolver un problema con divide y vencerás debemos pensar en un planteamiento recursivo para el mismo. Existen problemas donde el mejor algoritmo conocido está basado en divide y vencerás, aún cuando el problema computacional aparentemente no tiene nada que ver con recursión.

Un ejemplo típico es el de multiplicación de dos enteros r y s. Supongamos que ambos enteros tienen dos cifras en base decimal. El algoritmo que todos conocemos desde la primaria es:

$$\begin{array}{rrrr} & & a & b \\ \times & & c & d \\ \hline & & ad & bd \\ + & ac & bc & \\ \hline & ac & ad & bd \\ & & + & \\ & & bc & \end{array}$$

En el ejemplo anterior, tenemos que $r = 10a+b$, $s = 10c+d$ y el resultado es $t = 100ac+10(ad+bc)+bd$. El fenómeno que es importante observar aquí es que multiplicar dos números de n dígitos cada uno requiere realizar $O(n^2)$ multiplicaciones de dígitos. Este proceso es inherentemente más lento que el de la suma de dos números de n dígitos, donde la cantidad de sumas de dígitos es $O(n)$.

Generalizando esta idea, definiremos $r = h^k a+b$ y $s = h^k c+d$ donde h es la base y k es un exponente, siendo el resultado $t = h^{2k}ac + h^k(ad + bc) + bd$. Este definición requiere realizar cuatro multiplicaciones de números producidos por la descomposición de r y s (ac, ad, bc y bd) pero es posible reducir el número de estas multiplicaciones a tres, reescribiendo

$$t = h^{2k}ac + h^k(ad + bc) + bd$$

como

$$t = h^{2k}ac + h^k((b - a)(c - d)) + ac + bd) + bd$$

donde reutilizamos los valores de ac y bd. Esta nueva definición sólo necesita tres multiplicaciones de números provenientes de la descomposición de r y s, aunque aumenta la cantidad de sumas y restas.

Ahora aplicaremos divide y vencerás con $k = \frac{n}{2}$, de modo que cada vez que tengamos realizar una multiplicación entre dos números que vengan de la descomposición de r y s, aplicaremos nuevamente la misma técnica. Es decir, si queremos multiplicar dos números de n dígitos (siendo n una potencia de 2 para simplificar el análisis), la recurrencia que nos da la complejidad del algoritmo es:

$$T(n) = \begin{cases} n & \text{si } n \leq 1 \\ 3T(\frac{n}{2}) + O(n) & \text{si } n > 1 \end{cases}$$

donde $O(n)$ es el trabajo lineal causado por sumas, restas y corrimientos. Las multiplicaciones por la base h son operaciones lineales que corresponden con corrimientos en la mayoría de las computadoras (por ejemplo, si $h = 2$ entonces las multiplicaciones por dos son corrimientos de bits). Por el teorema maestro, $T(n) = \Theta(\log_2(3)) \approx \Theta(n^{1.5849625})$ que es asintóticamente menor que $O(n^2)$. Éste es el algoritmo de Karatsuba para multiplicación de enteros.

Usar la técnica de divide y vencerás suele ser una decisión consciente, en lugar de que la idea surja naturalmente de un problema. Ésta es la técnica que está detrás de búsqueda binaria, exponenciación natural y ordenamiento por mezcla, pero también sirve para multiplicar matrices y polinomios o para determinar la pareja más cercana de puntos en un plano. Algunos autores suelen usar el nombre *reduce y vencerás* cuando la recursión utiliza un único subproblema (como en búsqueda binaria o en exponenciación), pero como en el fondo es la misma idea, no haremos distinción aquí.

El algoritmo de ordenamiento rápido (*quicksort*) también está basado en la técnica de divide y vencerás. En este algoritmo, primero se particiona la entrada en función de un valor pivote (los que son menores que el pivote se mueven a la izquierda, los que son mayores a la derecha) y luego se ordena recursivamente cada parte. Podemos modificar este algoritmo para resolver un problema distinto: dado un entero k, queremos calcular cuál sería k-ésimo elemento de un arreglo si éste estuviera ordenado, pero queremos hacerlo más rápido que $O(n \log_2 n)$. La idea es simple: ordenar únicamente la parte del arreglo de donde provenga el k-ésimo elemento. Este algoritmo recibe el nombre de selección.

```
subrutina Selección(natural n, arreglo(entero^n) a, natural k)
    si n = 1
        regresa a[0]
    sino
        (s, t, u) ← Particion(n, a, Pivote(n, a))
        si k < |s|
            regresa Selección(|s|, s, k)
        sino si k < |s| + |t|
            regresa Selección(|t|, t, k − |s|)
        sino
            regresa Selección(|u|, u, k − |s| − |t|)
```

```
subrutina Partición(natural n, arreglo(entero^n) a, natural p)
    s ← [], t ← [], u ← []
    para i ← 0 ... n − 1
        si a[i] < p
            s ← (s)(a[i])
        sino si a[i] = p
            t ← (t)(a[i])
        sino
            u ← (u)(a[i])
    regresa (s, t, u)
```

Suponiendo que siempre se elige un pivote tal que el tamaño del subproblema recursivo es la mitad del problema original, la recurrencia del algoritmo de selección es:

$$T(n) = \begin{cases} 0 & \text{si } n = 1 \\ T\left(\frac{n}{2}\right) + n & \text{si } n > 1 \end{cases}$$

que es $\Theta(n)$ por el teorema maestro.

Al diseñar un algoritmo usando la técnica de divide y vencerás, generalmente hay dos enfoques que podemos tomar para resolver una instancia I:

- Si ya tuvieramos la soluciones de todas las instancias más fáciles que I, ¿de qué nos sirven? Dos ejemplos que usan este enfoque son exponenciación entera y ordenamiento por mezcla.
- Si yo hiciera parte del trabajo, ¿qué trabajo (más fácil) le toca a alguien más hacer? Dos ejemplos que usan este enfoque son búsqueda binaria y ordenamiento rápido.

A continuación describiremos dos problemas computacionales más, uno para cada caso.

Problema: Transiciones en cadenas binarias.
Entrada: Un natural n.
Salida: Un natural t que sea el total transiciones que hay en las 2^n cadenas binarias de longitud n, donde una transición ocurre cuando dos dígitos consecutivos b_i, b_{i+1} de una cadena son distintos.

Para resolver este problema conviene pensar en una estrategia incremental, ya que las 2^n cadenas binarias de longitud n se pueden construir a partir de dos copias de las 2^{n-1} cadenas de longitud $n-1$.

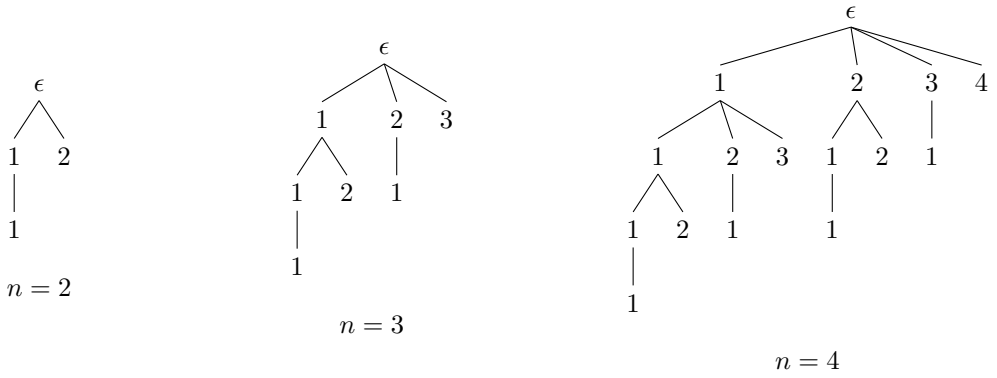
$n = 2$	$n = 3$	
00	000	100
01	001	101
10	010	110
11	011	111

Sea $f(n)$ la cantidad de transiciones para el caso n, sabemos que $f(0) = f(1) = 0$. Para valores mayores de n, podemos argumentar que $f(n) \geq 2f(n-1)$ por la observación hecha anteriormente. Lo único que falta por determinar es exactamente cuántas transiciones adicionales se producen durante la construcción de las cadenas del caso n, al agregar los 0 o los 1 a la izquierda. Para $n > 1$, cada cadena del caso $n-1$ comienza con 0 o con 1 y se copia dos veces; a la primera copia se le concatenará un 0 a la izquierda, que no provocará una transición si la cadena ya comenzaba con 0 pero sí lo hará si comenzaba con 1. De manera análoga cuando concatenamos un 1 a la segunda copia de la cadena. Como cada cadena del caso $n-1$ produce una nueva transición al ser concatenada por la izquierda en exactamente una de sus dos copias del caso n, tenemos que $f(n) = 2f(n-1) + 2^{n-1}$ para $n > 1$.

$$f(n) = \begin{cases} 0 & \text{si } n \leq 1 \\ 2f(n-1) + 2^{n-1} & \text{si } n > 1 \end{cases}$$

Problema: Suma de dígitos.
Entrada: Un natural n.
Salida: Un natural t que sea la cantidad de cadenas de dígitos del 1 al 9 que existen, tal que la suma de sus dígitos sea igual a n.

Al igual que el problema anterior, este problema se puede resolver usando un argumento de conteo, sin la necesidad de construir todas las cadenas válidas. Sin embargo, supongamos que deseamos construir tales cadenas para los casos de $n = 2, 3, 4$. Tenemos lo siguiente:



Si hacemos una pregunta ligeramente más especifica: ¿cuántas cadenas que sumen $N = 4$ comienzan con 1? entonces la respuesta corresponde con la cantidad total de cadenas para $N' = 3$, porque si queremos sumar 4 y comenzamos poniendo un 1, ¡entonces falta por sumar 3! Lo mismo ocurre si nos preguntamos ¿cuántas cadenas que sumen $N = 4$ comienzan con 2? la respuesta es la cantidad total de cadenas para $N' = 2$ porque falta por sumar 2. Siguiendo esta idea, la recurrencia que obtenemos para $f(n)$ es:

$$f(n) = \begin{cases} 1 & \text{si } n = 0 \\ \sum_{i=1}^{\min(n,9)} f(n-i) & \text{si } n > 0 \end{cases}$$

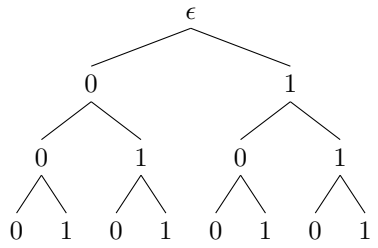
Se tiene que $f(0) = 1$ porque la cadena vacía suma 0. Las cotas de la sumatoria se obtienen de que las cadenas deben estar formadas por dígitos del 1 al 9, al mismo tiempo que no tiene sentido usar un dígito más grande que el valor de n. Ésta es la primera recurrencia que vemos donde la cantidad de llamadas recursivas no es constante. Evaluar este tipo de recurrencias tal cual en una computadora puede ser lento si usamos un lenguaje de programación imperativo.

8.1. Ejercicios

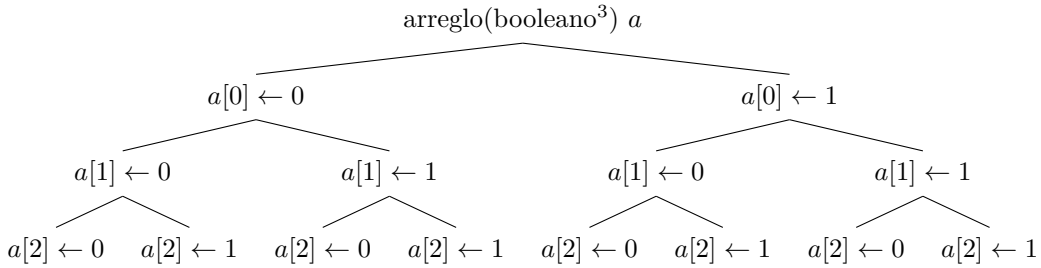
1. Resuelve el problema `https://omegaup.com/arena/problem/Transiciones-en-cadenas-binarias`.
2. Resuelve el problema `https://omegaup.com/arena/problem/Suma-de-digitos`.
3. Resuelve el problema `https://omegaup.com/arena/problem/Con-pocos-ceros-consecutivos`.
4. Resuelve el problema `https://omegaup.com/arena/problem/Cadenas-de-digitos-sin-ceros-con`.

9. El árbol de toma de decisiones

Muchos problemas que se resuelven con divide y vencerás son combinatorios, es decir, requieren encontrar alguna configuración que cumpla ciertas condiciones, contar o generar todas las configuraciones válidas del problema, o bien, encontrar la mejor configuración en función de un objetivo que se desea minimizar o maximizar. Por esta razón, primero exploraremos formas de generar configuraciones usuales tales como cadenas binarias, secuencias de enteros y permutaciones. Esto lo haremos con divide y vencerás. Comenzaremos discutiendo la generación de cadenas binarias de longitud n.



Las cadenas binarias de longitud $n = 3$



Asignaciones en un arreglo para generar las cadenas binarias de longitud $n = 3$

A continuación se presenta un algoritmo recursivo que realiza el trabajo descrito. El parámetro principal del algoritmo es un natural i que nos posiciona sobre el arreglo y el cual se va incrementando conforme cambia el nivel recursivo. Cuando el valor de i alcanza el valor de n, el cual no es una posición válida del arreglo, entonces ya se asignaron todos los valores de una rama recursiva e imprimimos el arreglo.

```
subrutina CadenasBinarias(natural n, arreglo(booleano^n) a, natural i)
    si i = n
        Imprime(n, a)
    sino
        a[i] ← 0
        CadenasBinarias(n, a, i + 1)
        a[i] ← 1
        CadenasBinarias(n, a, i + 1)
```

Sea $T(n')$ la cantidad de asignaciones que realiza el algoritmo anterior, donde n' es la cantidad de asignaciones que faltan para completar una rama recursiva (es decir $n' = n - i$ en el algoritmo), tenemos:

$$T(n') = \begin{cases} 0 & \text{si } n' = 0 \\ 2T(n'-1) + 2 & \text{si } n' > 0 \end{cases}$$

y por sustitución repetida obtenemos $T(n') = \sum_{i=1}^{n}(2^i) = 2^{n'+1} - 2 \in \Theta(2^{n'})$.

La generación de cadenas de dígitos sigue la misma idea, excepto que hay diez opciones a escribir en cada elemento del arreglo en lugar de sólo dos:

```
subrutina CadenasNuméricas(natural n, arreglo(natural^n) a, natural i)
    si i = n
        Imprime(n, a)
    sino
        para d ← 0...9
            a[i] ← d
            CadenasNuméricas(n, a, i + 1)
```

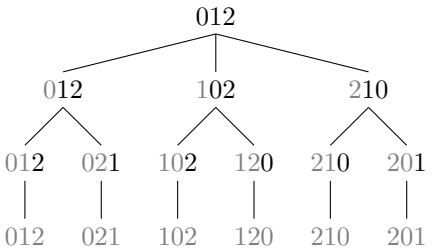
De forma similar al ejemplo anterior, la complejidad de este algoritmo está dada por:

$$T(n') = \begin{cases} 0 & \text{si } n' = 0 \\ 10T(n'-1) + 10 & \text{si } n' > 0 \end{cases}$$

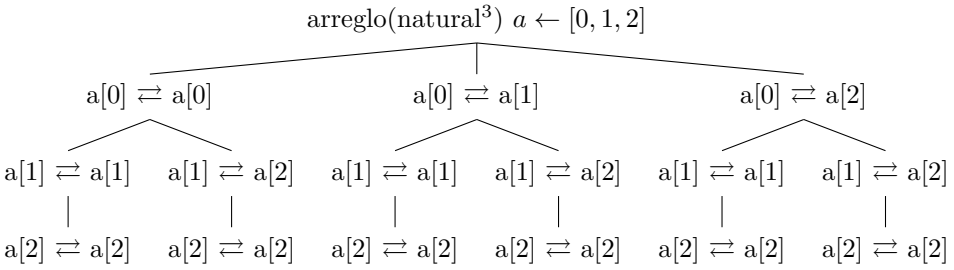
y por sustitución repetida obtenemos $T(n') = \sum_{i=1}^{n'}(10^i) \in \Theta(10^{n'})$.

La generación de permutaciones, por otra parte, es bastante más complicada que los ejemplos anteriores. Para diseñar el algoritmo, observaremos primero que todas las permutaciones de los números del 0 al $n-1$ pueden generarse mediante intercambios sobre un arreglo que ya contenga dichos valores. Por ejemplo, para $n = 3$ las $3! = 6$ permutaciones son $012, 021, 201, 210, 120, 102$ donde cada permutación se puede obtener mediante un intercambio realizado sobre la anterior. Además, cabe resaltar que cada elemento es el primero en un subconjunto de las permutaciones.

El algoritmo que diseñaremos no ejecutará la cantidad mínima de intercambios, pero de todos modos será óptimo asintóticamente. Usando la técnica de divide y vencerás, iremos fijando los elementos de la permutación de izquierda a derecha. Cada vez que fijemos un elemento haremos recursión, pero permitiremos que cada elemento por considerar tenga la oportunidad de ser fijado antes que los demás.



Las permutaciones de los elementos de 0 a 2



Intercambios en un arreglo para generar las permutaciones de 0 a 2

Las asignaciones descritas en la figura anterior suponen que los elementos del arreglo aparecen en el orden esperado, por lo que es importante que las llamadas recursivas deshagan los intercambios que ellas realizaron antes de regresar. Con esto en consideración, el algoritmo de divide y vencerás es el siguiente:

```
subrutina Permutaciones(natural n, arreglo(natural^n) a, natural i)
    si i = n
        Imprime(n, a)
    sino
        para j ← i...n − 1
            a[i] ⇄ a[j]
            Permutaciones(n, a, i + 1)
            a[i] ⇄ a[j]
```

Sea $T(n')$ la cantidad de intercambios que realiza el algoritmo anterior, donde n' es la cantidad de

intercambios que faltan para completar una rama recursiva (es decir $n' = n - i$ en el algoritmo), tenemos:

$$T(n') = \begin{cases} 0 & \text{si } n' = 0 \\ n'T(n'-1) + 2n' & \text{si } n' > 0 \end{cases}$$

El análisis de esta recurrencia es complicado, pero $n! \leq T(n) \leq 2en!$ para $n \geq 1$, por lo que $T(n) \in \Theta(n!)$.

10. Búsqueda con retroceso

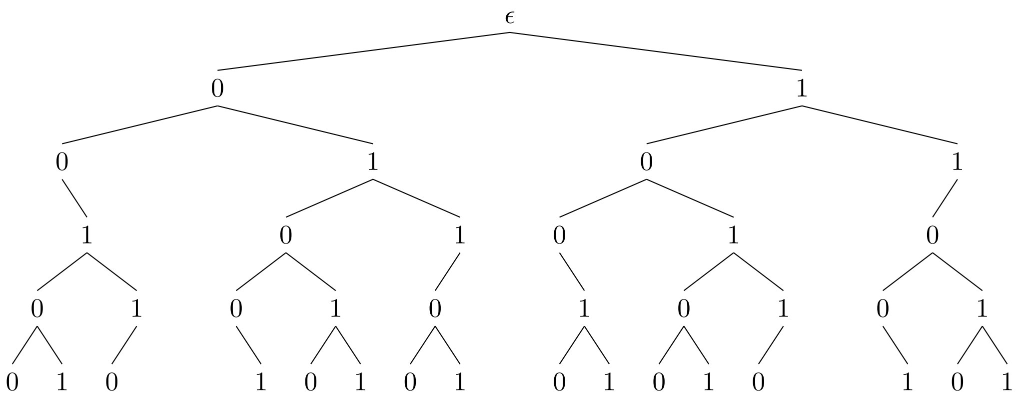
Como se mencionó en la sección anterior, un problema combinatorio consiste en encontrar alguna configuración que cumpla ciertas condiciones, en contar o generar todas las configuraciones válidas del problema, o bien, en encontrar la mejor configuración en función de un objetivo que se desea minimizar o maximizar. Una forma de resolver este tipo de problemas consiste en usar algún algoritmo que genere todas las configuraciones de una misma clase (por ejemplo, todas las cadenas binarias o todas las permutaciones de cierto tamaño) y simplemente filtrar las configuraciones que sí nos importan. Sin embargo, esta estrategia suele ser sumamente ineficiente cuando la cantidad de configuraciones generadas es mayor por órdenes de magnitud a la cantidad de configuraciones válidas. Búsqueda con retroceso es una técnica general de diseño de algoritmos que busca generar únicamente las configuraciones que sí nos importan (aunque a veces con un éxito limitado). En cierto modo, búsqueda con retroceso es todavía fuerza bruta, donde además la complejidad asintótica a veces no mejora o es muy difícil de analizar. Sin embargo, en la práctica sí se suelen conseguir aceleraciones de varias órdenes de magnitud.

Para explicar la técnica de búsqueda con retroceso, usaremos dos problemas de ejemplo. Uno de ellos tendrá que ver con la generación de cadenas binarias y el otro con la generación de permutaciones.

Problema: Cadenas binarias con pocos dígitos iguales consecutivos.
Entrada: Un natural n.
Salida: Las m cadenas binarias de longitud n que cumplan con la condición de no tener más de dos dígitos consecutivos iguales.

Un algoritmo que genere las 2^n cadenas binarias para luego validarlas tardará tiempo $O(n2^n)$. Sin embargo, es fácil ver que una cantidad considerable de estas cadenas son inválidas: para $n \geq 3$ cualquier cadena que comience 000... o 111... es inválida. Esto significa que por lo menos $2(2^{n-3})$ cadenas son inválidas y fáciles de identificar, pero en realidad hay muchísimas más. Por ejemplo, para $n = 15$ existen 32768 cadenas binarias, pero sólo 1974 son válidas. Una función que aproxima razonablemente bien el resultado para $0 \leq n \leq 40$ es 1.633^n, lo cual es mucho menor que 2^n para valores más grandes de n.

En este problema, búsqueda con retroceso modificará el algoritmo de generación de cadenas binarias para no hacer una llamada recursiva si el dígito que estamos colocando en el arreglo viola la condición del problema. Esto tiene el efecto de podar el árbol de toma de decisiones, con lo cual la magnitud del cálculo a realizar se reduce exponencialmente. Si llegamos a un caso base, entonces la cadena es válida.



El árbol de generación de cadenas binarias para $n = 4$ tras podar el árbol

En este ejemplo, podemos implementar la estrategia de búsqueda con retroceso simplemente examinando qué dígitos pusimos en las dos posiciones anteriores del arreglo, antes de hacer recursión. Si el dígito que intentamos poner es igual a los dos anteriores, entonces no debemos hacer recursión. Esta restricción no aplica cuando estamos en las dos primeras posiciones del arreglo, porque atrás no hay suficientes dígitos con quién compararnos. El algoritmo es el siguiente:

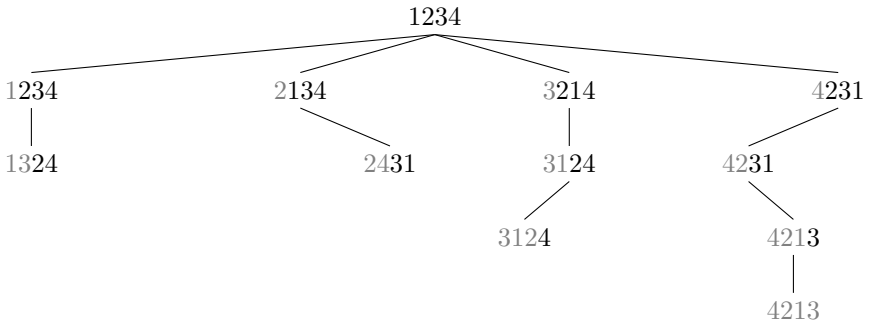
```
subrutina CadenasBinariasPDIC(natural n, arreglo(booleano^n) a, natural i)
    si i = n
        Imprime(n, a)
    sino
        si i ≤ 1 ∨ a[i − 2] ≠ 0 ∨ a[i − 1] ≠ 0
            a[i] ← 0
            CadenasBinariasPDIC(n, a, i + 1)
        si i ≤ 1 ∨ a[i − 2] ≠ 1 ∨ a[i − 1] ≠ 1
            a[i] ← 1
            CadenasBinariasPDIC(n, a, i + 1)
```

Una característica de este problema es que toda rama del árbol podado eventualmente termina en una hoja, lo cual es óptimo. Desafortunadamente, existen problemas para los que no podremos darnos cuenta que una rama recursiva no tiene futuro hasta después de haberla explorado casi por completo.

Problema: Permutaciones divisibles.
Entrada: Un natural n.
Salida: Las m permutaciones de los enteros del 1 al n que cumplan con la condición de que la suma de cada pareja de elementos contigüos sea divisible entre la posición (numerada a partir de 1) del segundo elemento de la pareja.

En este problema, una permutación $(A_1, A_2, \ldots, A_n)$ es válida para el problema si cumple que 2 divide a $A_1 + A_2$, 3 divide a $A_2 + A_3$, 4 divide a $A_3 + A_4$, etc. Un algoritmo que primero genere las $n!$ permutaciones y luego valide cada una de ellas tardará tiempo $O(n(n!))$. Sin embargo, es fácil ver que una cantidad considerable de estas permutaciones son inválidas. Para $n \geq 2$ cualquier permutación que comience con un par y un impar (o viceversa) es inválida al no cumplir la condición. Esto significa que por lo menos $\binom{n}{2}(n-2)!$ permutaciones son inválidas y fáciles de identificar, pero en realidad hay muchísimas más permutaciones inválidas. Por ejemplo, para $n = 15$ existen 1307674368000 permutaciones, pero sólo 10 son válidas. Es difícil aproximar el resultado de este problema, pero la cantidad de permutaciones válidas es menor o igual que $3n$ para los primeros 30 valores de n, lo cual es mucho menor que $n!$.

Modificaremos el algoritmo de generación de permutaciones para aplicar búsqueda con retroceso, de modo que evitemos la llamada recursiva si el entero que estamos por fijar en el arreglo provoca que se viole la condición del problema. Esto tiene el efecto de podar el árbol de toma de decisiones y reduce su tamaño se reduce enormemente. Si llegamos a un caso base, entonces la permutación es válida.



El árbol de generación de permutaciones para $n = 4$ tras podar el árbol

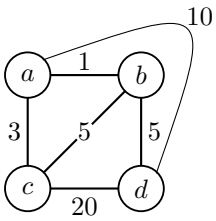
En este ejemplo, podemos implementar la estrategia de búsqueda con retroceso simplemente examinando qué entero pusimos en la posición anterior del arreglo, antes de hacer recursión. Si el entero que intentamos poner más el entero de la posición anterior no es divisible entre nuestra posición (a partir de 1), entonces no debemos hacer recursión. Esta restricción no aplica cuando estamos en la primera posición del arreglo, porque atrás no hay un elemento con quién sumarnos. El algoritmo es el siguiente:

```
subrutina PermutacionesDivisibles(natural n, arreglo(natural^n) a, natural i)
    si i = n
        Imprime(n, a)
    sino
        para j ← i ... n − 1
            a[i] ⇄ a[j]
            si i = 0 ∨ i + 1 | a[i − 1] + a[i]
                PermutacionesDivisibles(n, a, i + 1)
            a[i] ⇄ a[j]
```

Una característica de este problema es que no toda rama del árbol podado eventualmente termina en una hoja. Por ejemplo, a pesar de que ninguna permutación que comienza con 3 es válida, casi llegamos a la máxima profundidad del árbol durante la recursión. Afortunadamente, la poda del árbol es lo suficientemente efectiva como para poder resolver hasta $n = 25$ en menos de 10 segundos, aunque sólo hayan 22 permutaciones válidas para este caso.

Problema: El agente viajero (versión de camino abierto).
Entrada: Un natural n y una gráfica completa g de n vértices con pesos en sus aristas.
Salida: Un camino de costo mínimo sobre g que visite cada vértice exactamente una vez.

Éste es un problema clásico en computación y también es uno muy difícil de resolver. Tenemos la flexibilidad de comenzar en cualquier vértice de la gráfica, pero esto no facilita el problema. Tampoco funciona la estrategia de seguir la secuencia de aristas más baratas. A la fecha, nadie conoce alguna forma de resolver este problema que no degenere en una búsqueda por fuerza bruta.



Una gráfica donde el camino óptimo no usa la arista más barata

En los anteriores problemas de ejemplo usamos la técnica de búsqueda con retroceso para evitar generar soluciones inválidas, pero en este problema cualquier permutación de los vértices denota un camino válido. Nosotros lo que queremos es encontrar el camino más barato de todos, pero afortunadamente sí existe una forma de adaptar búsqueda con retroceso para acelerar la búsqueda de tal camino óptimo.

Supongamos que la gráfica no tiene aristas negativas (en caso contrario, podemos incrementar todas las aristas de la gráfica por igual hasta que desaparezcan los costos negativos; lo cual es válido porque toda solución recorre exactamente $n - 1$ aristas). También supongamos que el mejor camino que hemos encontrado hasta el momento tiene costo t y que los estamos generando con nuestro algoritmo de generación de permutaciones. El costo de un camino lo podemos calcular incrementalmente conforme decidimos en qué orden visitaremos los vértices (y en consecuencia, qué aristas recorreremos). ¿Qué pasa si el costo p de un camino parcial no es menor que t? entonces podemos ahorrarnos el completar ese camino, ya que su costo no mejorará al agregar las aristas que faltan. Si logramos completar algún camino a un costo menor que t entonces actualizaremos t, lo que volverá más efectivo nuestro criterio de poda del árbol.

```
subrutina AgenteViajeroCA(natural n, arreglo(natural^n) a, natural i, natural t, natural p)
    si i = n
        t ← p
    sino
        para j ← i ... n − 1
            a[i] ⇄ a[j]
            si i = n
                p ← p + costo(a[i − 1], a[i])
            si p < t
                AgenteViajeroCA(n, a, i + 1, t, p)
            si i = n
                p ← p − costo(a[i − 1], a[i])
            a[i] ⇄ a[j]
```

El tiempo de ejecución de este algoritmo depende en buena medida de la suerte. Si encontramos el mejor camino relativamente pronto, entonces podremos descartar el resto de los caminos de forma temprana. Por otra parte, si tenemos la mala suerte de generar primero los peores caminos, nuestro criterio de poda no será efectivo. En la práctica, este tipo de problemas se suelen atacar en dos etapas: en la primera etapa se ejecuta un algoritmo rápido que no necesariamente encuentra el mejor camino pero que encuentra uno razonablemente bueno con alta probabilidad; en la segunda etapa se usa el costo del camino encontrado anteriormente para volver más eficaz la búsqueda con retroceso.

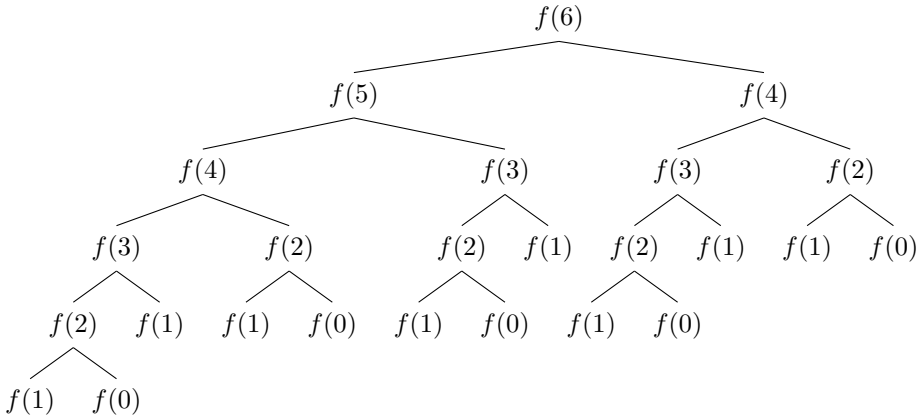
Los solucionadores de programas enteros lineales (sistemas de desigualdades lineales con variables enteras) usan las ideas discutidas en esta sección para acelerar tanto la búsqueda de una solución factible del sistema como la búsqueda de una solución óptima bajo una función objetivo. Por una parte, se aprovechan de que un sistema lineal sin variables enteras se puede resolver rápidamente, para ignorar temporalmente la integralidad de variables aún sin un valor asignado (relajar el sistema) en un intento por determinar si el sistema sigue siendo factible con las variables enteras asignadas hasta el momento. En el caso de que el sistema relajado sea infactible, el subárbol respectivo se poda. Por otra parte, si se está buscando una solución óptima donde ya se cuenta con una solución factible de valor v y la relajación de un sistema tiene un valor v' que es peor que v, entonces el subárbol respectivo también se puede podar.

10.1. Ejercicios

1. Resuelve el problema `https://omegaup.com/arena/problem/Cadenas-binarias-con-pocos-digit`.
2. Resuelve el problema `https://omegaup.com/arena/problem/Permutaciones-divisibles`.
3. Resuelve el problema `https://omegaup.com/arena/problem/El-agente-viajero`.
4. Resuelve el problema `https://omegaup.com/arena/problem/Cadenas-viborita`.
5. Resuelve el problema `https://omegaup.com/arena/problem/El-problema-de-las-N-princesas`.
6. Resuelve el problema `https://omegaup.com/arena/problem/El-recorrido-del-principe`.

11. Trabajo repetido en divide y vencerás

El fenómeno de trabajo repetido ocurre frecuentemente al usar la técnica de divide y vencerás. Las implementaciones recursivas hechas en lenguajes imperativos evalúan cada rama recursiva de forma independiente y sin usar memoria adicional a la especificada por el programador, excepto por la pila de llamadas a función que se maneja de forma transparente. La independencia entre ramas recursivas provoca que, cuando un subtrabajo aparece en más de una rama, se repita el cálculo correspondiente. La cantidad de trabajo repetido puede ser abrumador para recurrencias de aspecto inocente como Fibonacci:



El árbol de llamadas de Fibonacci para $n = 6$

El árbol anterior tiene 25 nodos, a pesar de que sólo 7 llamadas son distintas (desde $f(0)$ hasta $f(6)$). Peor aún, el árbol de llamadas para $f(n+1)$ tendrá casí el doble de nodos que el de $f(n)$ a pesar de que sólo aparece una llamada distinta más. El fenómeno de trabajo repetido es la principal razón por la que se dice (erróneamente) que un algoritmo recursivo es mucho más lento que uno iterativo: la diferencia entre recursión e iteración es por un factor constante, pero cualquier diferencia asintótica entre implementaciones recursivas e iterativas tiene su origen en el trabajo repetido.

No todo algoritmo de divide y vencerás presenta trabajo repetido. Si el algoritmo tiene una única rama recursiva entonces no habrá trabajo repetido (salvo que esté mal implementado). El algoritmo de ordenamiento por mezcla tampoco presenta trabajo repetido aún teniendo más de una rama recursiva, ya que cada rama de la recursión procesa un subarreglo distinto. Por otra parte, la recurrencia del coeficiente binomial o la del problema de suma de dígitos generan una cantidad considerable de trabajo repetido.

12. Recursión con memorización

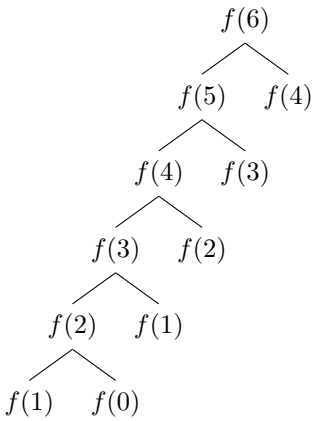
Recursión con memorización es una técnica general para eliminar el trabajo repetido que genera una recurrencia. La idea es simple: memorizar nosotros mismos los resultados de cada llamada hecha y verificar al inicio de cada llamada si ya la habíamos calculado anteriormente. Para lograrlo, necesitamos analizar la recurrencia cuidadosamente para responder lo siguiente:

- ¿Cuántos parámetros tiene? Normalmente, para memorizar los resultados de una recurrencia con n parámetros emplearemos un arreglo n-dimensional. Se debe tener cuidado, porque puede ocurrir que algunos de esos parámetros en verdad sean constantes camufladas (es decir, no cambian conforme hacemos recursión), por lo podremos la cantidad de dimensiones del arreglo empleado en la memorización.
- ¿Qué conjunto de valores puede tomar cada parámetro? El tamaño de cada dimensión del arreglo de memorización depende del conjunto de valores que puede tomar el parámetro correspondiente. Normalmente, este conjunto será un rango de valores no negativos consecutivos o casi consecutivos, lo que se presta para usar el valor del parámetro como un índice. Sin embargo, puede darse el caso que el parámetro tome valores negativos o que la mayoría de valores entre las cotas inferior y superior del dominio no formen parte del mismo (por ejemplo, un parámetro que sea una potencia de dos acotada entre 1 y 2^{30}). Este análisis se necesita para evitar apartar una cantidad exagerada de memoria. Para dominios muy irregulares, incluso podrían usarse árboles binarios de búsqueda o tablas de dispersión.
- ¿Cómo podemos distinguir entre una llamada memorizada y una llamada que es la primera vez que se calcula? Si se va a usar un arreglo n-dimensional para memorizar los resultados, lo usual es usar valores centinela para distinguir entre llamadas memorizadas y no memorizadas. Por ejemplo, si el resultado de una recurrencia no puede ser negativo, entonces el arreglo de memorización puede llenarse de -1 antes de ejecutar la recurrencia para poder distinguir entre esos dos casos. Si se usan estructuras más complicadas, éstas proveerán de métodos para identificar si una clave de búsqueda ya existe.

- ¿La implementación de la recurrencia es libre de efectos secundarios? Si vamos a memorizar un resultado para reusarlo después, debemos asegurarnos de que dicho resultado sea el correcto de forma independiente al tiempo. Si la recurrencia inspecciona y modifica variables globales de modo que $f(n) = v_1$ al tiempo t_1 y $f(n) = v_2$ al tiempo t_2 con $v_1 \neq v_2$, entonces no debemos memorizar v_1 para usarlo en el tiempo t_2. En general, para emplear la técnica de recursión con memorización debe ser cierto que el resultado de la recurrencia sólo depende de sus parámetros y de constantes globales.

A continuación se muestra un algoritmo que calcula la secuencia de Fibonacci empleando recursión con memorización y también se muestra el árbol de llamadas que se generaría. La estrategia del algoritmo es simple: primero revisamos si aún no tenemos el resultado, en cuyo caso lo calculamos y lo memorizamos. Al final del algoritmo sabemos que ya tenemos el resultado (ya sea porque lo acabamos de calcular o porque ya lo teníamos de un llamada anterior) y lo regresamos. Aunque el árbol de llamadas no tiene la cantidad óptima de nodos (porque sigue haciendo la llamada derecha aunque ésta regrese de inmediato), la cantidad de llamadas a función pasó de ser exponencial en n, a ser lineal en n.

```
subrutina FibonacciMemorizado(natural n, arreglo(entero^(n_máx+1)) r)
    si r[n] = −1
        si n = 0 o n = 1
            r[n] ← n
        sino
            r[n] ← FibonacciMemorizado(n − 1, r) + FibonacciMemorizado(n − 2, r)
     regresa r[n]
```



El árbol de llamadas de FibonacciMemorizado para $n = 6$

A continuación se presenta un algoritmo que calcula el coeficiente binomial empleando recursión con memorización. Lo normal es usar una matriz cuadrada de dimensiones $(n_{máx} + 1) \times (k_{máx} + 1)$ aunque con eso se aparte memoria para llamadas que nunca ocurrirán, como aquéllas con $n < k$. En la práctica esto no suele importar, salvo cuando se obtienen beneficios considerables en ahorro de memoria.

```
subrutina BinomialMemorizado(natural n, natural k, arreglo(entero^((n_máx+1)×(k_máx+1))) r)
    si r[n][k] = −1
        si k = 0 o k = n
            r[n][k] ← 1
        sino
            r[n][k] ← BinomialMemorizado(n − 1, k, r) + BinomialMemorizado(n − 1, k − 1, r)
     regresa r[n][k]
```

Salvo por las observaciones hechas anteriormente sobre analizar el comportamiento de los parámetros de la recurrencia, todos los algoritmos de recursión con memorización emplean la misma idea. La

sobrecarga de inicializar el arreglo de la memorización con valores centinelas ocasionalmente puede ser considerable, especialmente si realmente terminan evaluándose muy pocas llamadas recursivas distintas.

12.1. Ejercicios

1. Resuelve el problema `https://omegaup.com/arena/problem/El-rectangulo-de-domino`.
2. Resuelve el problema `https://omegaup.com/arena/problem/La-funcion-loca`.
3. Resuelve el problema `https://omegaup.com/arena/problem/Evaluando-una-funcion`.
4. Resuelve el problema `https://omegaup.com/arena/problem/Evaluando-una-funcion-algo-rara`.

13. Programación dinámica

La programación dinámica es una técnica general de diseño de algoritmos que puede aplicarse a problemas que exhiben las siguientes dos propiedades:

- Subestructura óptima: podemos descomponer una instancia del problema en instancias más fáciles del mismo problema, donde las soluciones de éstas nos permiten obtener la solución de la instancia original. Esto es, el problema puede resolverse con divide y vencerás.
- Subproblemas compartidos: en el planteamiento recursivo del problema, una instancia que aparezca en una rama recursiva puede aparecer también en otra rama, lo que resultaría en trabajo repetido si no se usa memorización.

En pocas palabras, la técnica de programación dinámica puede aplicarse en muchos de los problemas que hemos visto anteriormente, en particular en aquéllos donde aplicar recursión con memorización es una buena idea. Debemos recordar que para poder aplicar recursión con memorización, la implementación de la recurrencia debía ser libre de efectos secundarios: si vamos a memorizar un resultado para reusarlo después, debemos asegurarnos de que dicho resultado siga siendo el correcto para los mismos parámetros. En otras palabras, el estado completo de la función debe ser local a ella y todas sus dependencias deben ser parámetros explícitos de la función; si acaso, se permite depender de constantes globales.

Recursión con memorización es ideal en dos sentidos: sólo evalúa las llamadas a función que la recurrencia en verdad requiere evaluar y además evita el trabajo repetido. Sin embargo, recursión con memorización también tiene las siguientes debilidades:

- A falta de saber qué llamadas a función evaluará la recurrencia, se requiere apartar un arreglo e inicializarlo con un valor centinela, o bien, usar una estructura de datos dinámica mucho más complicada.
- Cada llamada a función ahora debe ejecutar código que verifique si el resultado ya estaba memorizado. Realizar repetidamente esta verificación tiene cierto impacto en el tiempo de ejecución total.
- Una implementación recursiva requiere ejecutar llamadas a función, lo que ees relativamente costoso: se deben colocar los parámetros en la pila, dar un salto al código de la función, regresar al término de la función y quitar los parámetros de la pila.

En contraste, programación dinámica busca evaluar y memorizar subinstancias *de forma anticipada*. Así, una instancia que necesite el resultado de otra podrá consultar directamente el resultado ya memorizado sin tener verificar si ya estaba calculado (ya lo estaba) y sin tener hacer recursión. Esto nos permite eliminar tanto la recursión como la necesidad de inicializar el arreglo de memoria con un valor centinela. Para lograr aplicar programación dinámica, necesitaremos hacer lo siguiente:

- Analizar el orden potencial en el que la recursión haría las llamadas a función. Como queremos memorizar la subinstancias de forma anticipada, necesitamos saber primero quién necesitará qué.
- Aceptar que programación dinámica con frecuencia se ”anticipará por si las dudas”, resolviendo subinstancias para las que no hay certeza si alguien realmente las necesitará. Sólo podríamos saber exactamente qué subinstancias se necesitarán si hacemos recursión, justo algo que queremos evitar.

Uno de los ejemplos más fáciles de programación dinámica es Fibonacci (una vez más). En este problema es muy fácil analizar el orden potencial en el que recursion hará las llamadas recursivas, porque $f(n)$ llama a $f(n-1)$, $f(n-1)$ llama a $f(n-2)$ y así sucesivamente. Como todas las llamadas de interés están entre $f(0)$ y $f(n)$, eso quiere decir que podemos anticiparnos a evaluar $f(0), f(1), \ldots, f(n-1)$ en ese orden para que $f(n)$ no requiera hacer recursión. Esta idea resulta en el siguiente algoritmo:

```
subrutina FibonacciPD(natural n, arreglo(entero^{n_máx+1}) r)
    para i ← 0...n
        si i = 0 o i = 1
            r[i] ← i
        sino
            r[i] ← r[i − 1] + r[i − 2]
     regresa r[n]
```

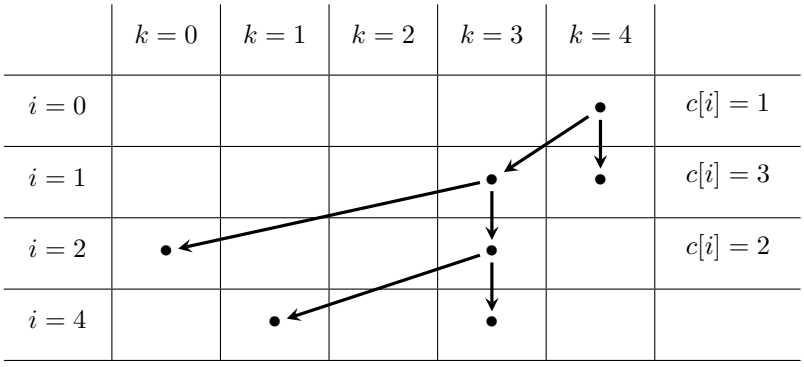
El algoritmo anterior tiene una complejidad de $O(n)$. Esto generalmente es suficiente en la práctica, aunque curiosamente existen algoritmos aún más rápidos para Fibonacci. Sin embargo, para muchos problemas es cierto que el algoritmo más rápido conocido surge de aplicar programación dinámica.

Problema: Suma de subconjuntos (versión de naturales).
Entrada: Un natural n, un natural k y un conjunto de n naturales.
Salida: Un booleano b que denote si es sumar k eligiendo un subconjunto de los n enteros.

Un algoritmo que pruebe los 2^n posibles subconjuntos sólo podrá resolver en un tiempo razonable instancias con $n \leq 40$. Sin embargo, programación dinámica puede algunas instancias con n en los cientos o miles usando la siguiente recurrencia:

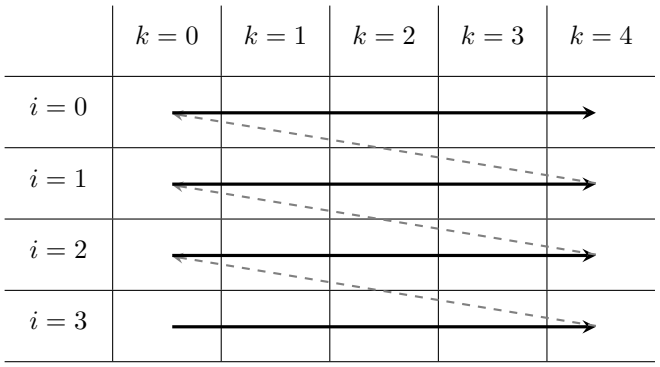
```
subrutina SumaSubconjuntos(natural n, arreglo(entero^n) c, natural i, natural k)
    si i = n
        regresa (k = 0)
    sino
        t ← SumaSubconjuntos(n, c, i + 1, k)
        si k ≥ c[i]
            t ← t ∨ SumaSubconjuntos(n, c, i + 1, k − c[i])
        regresa t
```

Para poder aplicar programación dinámica, debemos proponer un orden de evaluación que garantice que SumaSubconjuntos(n, c, i, k) siempre se evalúa después de SumaSubconjuntos$(n, c, i+1, k)$ y de SumaSubconjuntos$(n, c, i+1, k-c[i])$, que son sus dos dependencias. Este orden no necesariamente es único. Lo que generalmente se hace es dibujar la *tabla dinámica* (la estructura que almacenará los resultados memorizados) junto con un esquema que a grandes rasgos ubica a las dependencias en la tabla.



Algunas dependencias para $n = 3, k = 4$ y $c = \{1, 3, 2\}$. Se omiten el resto por simplicidad.

Un análisis rápido de la figura anterior y de la recurrencia como tal, nos dice que las dos posibles dependencias de SumaSubconjuntos(n, c, i, k) están en la fila de abajo (una en la misma columna, la otra algunas columnas a la izquierda). Un orden adecuado para evaluar las celdas de la tabla dinámica es por filas, de abajo para arriba y de izquierda a derecha en cuanto a columnas:



Un orden válido para llenar la tabla dinámica en el problema de suma de subconjuntos.

```
subrutina SumaSubconjuntosPD(natural n, arreglo(natural^n) c, natural k)
    r ← arreglo(booleano^((n+1)×(k+1)))[ ]
    para i ← n ... 0
        para k' ← 0 ... k
            si i = n
                r[i][k'] ← (k' = 0)
            sino
                t ← r[i + 1][k']
                si k' ≥ c[i]
                    t ← t ∨ r[i + 1][k' − c[i]]
                r[i][k'] ← t
     regresa r[0][k]
```

Existen otros órdenes que son válidos, por ejemplo, por columnas de izquierda a derecha y de abajo para arriba en cuanto a filas. ¡Incluso se podría evaluar la tabla dinámica en diagonal!

El algoritmo anterior tiene una complejidad de $O(nk)$, lo cual suele ser mucho menor que $O(2^n)$. Por otra parte, esto se logra a cambio de involucrar el valor de k en la complejidad del algoritmo, lo que provoca que éste siga siendo exponencial con respecto al número de bits de la entrada. Sin embargo, para este problema, el planteamiento de programación dinámica es el más usado en la práctica.

Problema: Subsecuencia común más larga.
Entrada: Dos cadenas a y b.
Salida: Dos secuencias de enteros $x_1, x_2, \ldots, x_n$ y $y_1, y_2, \ldots, y_n$ tales que $0 \leq x_i < x_{i+1} < |a|$, $0 \leq y_i < y_{i+1} < |b|$ y $a[x_i] = b[y_i]$.

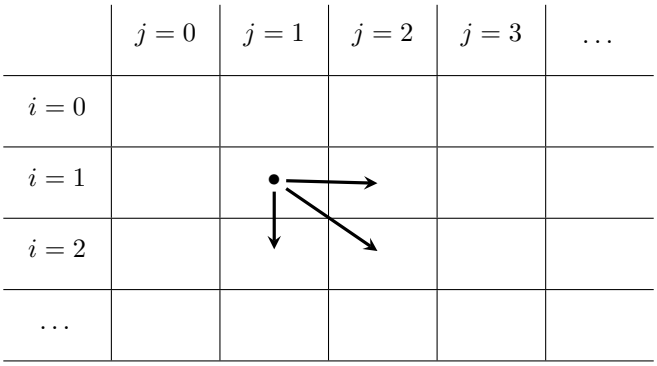
La subsecuencia común más larga de dos cadenas a y b es una buena medida de similitud entre ellas. Por esta razón, este problema tiene aplicaciones en bioinformática y en detección de plagio.

La idea para resolver este problema con divide y vencerás es la siguiente: supongamos que $f(i, j)$ es la longitud de la subsecuencia común más larga que se puede usando los caracteres de a a partir de i y los caracteres de b a partir de j. Claramente, si $a[i] = b[j]$ entonces tenemos una coincidencia en ambas cadenas, podemos contarla y avanzar en las dos. Si $a[i] \neq b[j]$ entonces no queda claro cuál de las dos posiciones podria estar en una solución óptima, si es que alguna lo está. Por ejemplo, para las cadenas "gatito" y "atitog" es mala idea forzar la coincidencia de la "g" porque eso nos impediría hacer cualquier otra coincidencia. Por lo mismo, la recurrencia explorará ambas opciones (permanecer en i y avanzar en

j, avanzar en i y permanecer en j) y se quedará con la mejor alternativa. Los casos base ocurren cuando ya no nos quedan caracteres en las cadenas por hacer coincidir. La recurrencia es:

```
subrutina SubsecuenciaComún(cadena a, cadena b, natural i, natural j)
    si i = |a| ∨ j = |b|
        regresa 0
    sino si a[i] = b[j]
        regresa 1 + SubsecuenciaComún(a, b, i + 1, j + 1)
    sino
        regresa máx(SubsecuenciaComún(a, b, i + 1, j), SubsecuenciaComún(a, b, i, j + 1))
```

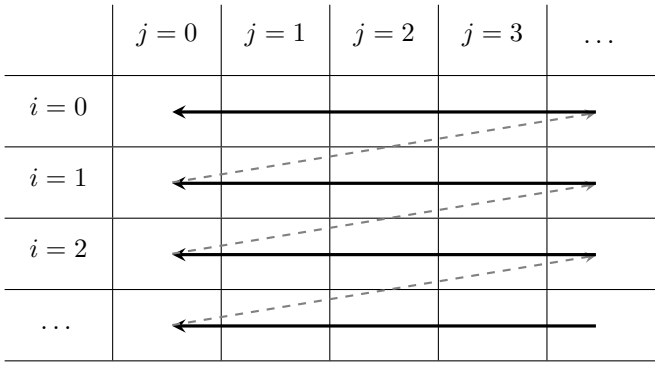
La llamada inicial de la recurrencia sería SubsecuenciaComún$(a, b, 0, 0)$ y la recurrencia exhibe trabajo repetido. Por esta razón, este problema se puede resolver con programación dinámica en tiempo $O(nm)$ donde $n = |a|$ y $m = |b|$. A grandes rasgos, las dependencias de SubsecuenciaComún(a, b, i, j) son:



Dependencias de la recurrencia. La diagonal se requiere si $a[i] = b[j]$ y las otras dos si $a[i] \neq b[j]$.

Un orden válido para llenar la tabla dinámica de este problema es por filas de arriba para abajo, y de derecha a izquierda en columnas. Debemos hacer notar que la recurrencia propuesta calcula exclusivamente la longitud de la subsecuencia común más larga, por lo que aún nos queda por responder la pregunta ¿cómo calculamos las secuencias de posiciones a coincidir? y ¿en verdad necesitamos una matriz de orden cuadrático si sólo queremos calcular la longitud?

Para recuperar la secuencia de posiciones que coinciden, podemos navegar en la tabla dinámica r partiendo de $i = 0$ y $j = 0$. Si $a[i] = b[j]$ entonces la pareja (i, j) es una coincidencia e incrementamos ambos índices. Si $a[i] \neq b[j]$ entonces no hay coincidencia y revisaremos la tabla dinámica: como sabemos que el valor de $r[i][j]$ coincide ya sea con $r[i + 1][j]$ o con $r[i][j + 1]$, incrementamos el índice respectivo para movernos a la que sea igual (y en caso de empate, incrementamos cualquiera). Cuando lleguemos a un caso base, terminamos el proceso. La longitud de la secuencia debe coincidir con el valor de $r[0][0]$.



Un orden válido para llenar la tabla dinámica en el problema de subsecuencia común más larga.

```
subrutina SubsecuenciaComúnPD(cadena a, cadena b)
    r ← arreglo(natural^((|a|+1)×(|b|+1)))[ ]
    para i ← |a| ... 0
        para j ← |b| ... 0
            si i = |a| ∨ j = |b|
                r[i][j] ← 0
            sino si a[i] = b[j]
                r[i][j] ← 1 + r[i + 1][j + 1]
            sino
                r[i][j] ← máx(r[i + 1][j], r[i][j + 1])

    i ← 0, j ← 0, s ← [ ]
    mientras i < |a| ∧ j < |b|
        si a[i] = b[j]
            s ← (s)((i, j))
            i ← i + 1, j ← j + 1
        sino si r[i][j] = r[i + 1][j]
            i ← i + 1
        sino
            j ← j + 1
     regresa (r[0][0], s)
```

Por otra parte, si sólo queremos calcular la longitud de la subsecuencia común más larga (por ejemplo, porque nos interesa usarla como medida de similitud y no porque nos interese conocer la subsecuencia en sí), podemos reducir drásticamente el consumo de memoria de la siguiente manera. Según el análisis de la tabla dinámica, las dependencias de cada celda están en su misma fila y en la fila contigüa de abajo. La siguiente implementación sólo retiene en memoria la fila $i + 1$ mientras calcula la fila i.

```
subrutina SubsecuenciaComúnPDMemoriaLineal(cadena a, cadena b)
    r ← arreglo(natural^((2)×(|a|+1)))[ ]
    para i ← |a| ... 0
        para j ← |b| ... 0
            si i = |a| ∨ j = |b|
                r[i mód 2][j] ← 0
            sino si a[i] = b[j]
                r[i mód 2][j] ← 1 + r[(i + 1) mód 2][j + 1]
            sino
                r[i mód 2][j] ← máx(r[(i + 1) mód 2][j], r[i mód 2][j + 1])
     regresa r[0][0]
```

Problema: Multiplicación encadenada de matrices.
Entrada: Un entero positivo n y n parejas de enteros positivos que denoten las dimensiones de n matrices $m_1, m_2, \ldots, m_n$.
Salida: Un natural r que sea la cantidad total mínima de multiplicaciones entre los elementos de las matrices que se requieren para evaluar el producto $m_1 \times m_2 \times \cdots \times m_n$ con parentización óptima.

Supongamos que tenemos tres matrices m_1, m_2, m_3 con dimensiones $(4, 3)$, $(3, 2)$ y $(2, 1)$ respectivamente. El producto $(m_1 \times m_2) \times m_3$ usa $(4)(3)(2)+(4)(2)(1) = 32$ multiplicaciones de elementos, mientras que el producto $m_1 \times (m_2 \times m_3)$ usa $(3)(2)(1)+(4)(3)(1) = 18$ multiplicaciones. Este problema lo podemos resolver con programación dinámica como se describe a continuación.

Diremos que un producto de matrices tiene un orden de evaluación explícito si está totalmente parenterizado (es decir, cada pareja de matrices a multiplicar están dentro de paréntesis, excepto por la

multiplicación más externa). Por ejemplo, $m_1 \times (m_2 \times m_3) \times m_4$ no es totalmente explícito, mientras que $(m_1 \times (m_2 \times m_3)) \times m_4$ sí lo es. Los paréntesis más externos parten la instancia en dos subinstancias: el subproducto izquierdo y el derecho. Sea $f(i, j)$ la cantidad óptima de multiplicaciones de elementos que se necesitan para evaluar el producto $m_i \times m_{i+1} \times \cdots \times m_{j-1}$, necesitamos encontrar el lugar óptimo en el cual partir la instancia. Por supuesto, cada subinstancia se resolverá a su vez con divide y vencerás.

```
subrutina MultiplicaciónEncadenada(natural n, arreglo((entero^2)^n) d, natural i, natural j)
    si j − i = 1
        regresa 0
    sino
        t ← ∞
        para k ∈ i + 1 . . . j − 1
            t_1 = MultiplicaciónEncadenada(n, d, i, k)
            t_2 = MultiplicaciónEncadenada(n, d, k, j)
            t ← mín(t, t_1 + t_2 + (d[i]_0)(d[k]_0)(d[j − 1]_1))
        regresa t
```

Dibujar las dependencias de la recurrencia anterior sobre la tabla dinámica es dificil, pero podemos proponer un orden de evaluación haciendo las siguientes observaciones: un subproblema está descrito por el subintervalo de las matrices a multiplicar, y la partición de un problema genera subproblemas cuyos subintervalos son menores en tamaño. Dicho esto, propondremos un orden que resuelva primero todos los subproblemas con intervalos de tamaño 1, luego los subproblemas con intervalos de tamaño 2, etc. así hasta resolver la instancia original. Esto nos da el siguiente algoritmo de programación dinámica que tiene una complejidad temporal $O(n^3)$ y usa $O(n^2)$ de memoria:

```
subrutina MultiplicaciónEncadenadaPD(natural n, arreglo((entero^2)^n) d)
    r ← arreglo(natural^{(n)×(n+1)})[ ]
    para s ← 1 . . . n
        para i ← 0 . . . n − s
            j ← i + s
            si j − i = 1
                r[i][j] ← 0
            sino
                t ← ∞
                para k ∈ i + 1 . . . j − 1
                    t_1 = r[i][k]
                    t_2 = r[k][j]
                    t ← mín(t, t_1 + t_2 + (d[i]_0)(d[k]_0)(d[j − 1]_1))
                r[i][j] ← t
    regresa r[0][n]
```

13.1. Ejercicios

1. Resuelve el problema `https://omegaup.com/arena/problem/Bajando-por-el-arbol`.
2. Resuelve el problema `https://omegaup.com/arena/problem/interminable-mermelada`.
3. Resuelve el problema `https://omegaup.com/arena/problem/Los-arboles-del-bosque-de-cedros`.
4. Resuelve el problema `https://omegaup.com/arena/problem/El-ladron-precavido`.

14. Problemas con ciclos en el espacio de búsqueda

Hasta ahora hemos estudiado problemas donde cada toma de decisión nos lleva a un subproblema más fácil que el original. Sin embargo, también existen problemas con mucho menos estructura: las tomas de decisión podrían dificultar más el problema, podrían alejarnos arbitrariamente de la solución o podrían "hacernos caminar en círculos". Una característica de estos problemas es que la toma de decisiones equivale a navegar una gráfica cíclica, en lugar de un árbol.

Problema: Exponenciación con multiplicaciones y divisiones.
Entrada: Un natural n.
Salida: Un natural r que sea la cantidad mínima de operaciones que se necesitan para calcular a^n a partir de a^0, si sólo se permiten multiplicaciones por a, divisiones entre a y elevaciones al cuadrado.

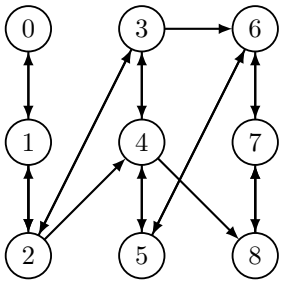
La primera observación que debemos hacer es que a es una variable simbólica que no forma parte de la entrada y que debemos concentrarnos en el cálculo del exponente. Si actualmente estamos en a^k, entonces multiplicar por a produce a a^{k+1}, dividir entre a produce a^{k-1} y elevar al cuadrado produce a^{2k}. Un intento descuidado de atacar este problema con divide y vencerás resultaría en la siguiente recurrencia, donde n es una constante implícita:

$$f(k) = \begin{cases} 0 & \text{si } k = n \\ 1 + \max(f(k+1), f(k-1), f(2k)) & \text{en otro caso} \end{cases}$$

Sin embargo, la recurrencia anterior nunca acabará: algunas ramas hacen recursión hacia $f(-\infty)$, otras hacen recursión hacia $f(+\infty)$ y otras ramas van de $f(k)$ a $f(k+1)$ al multiplicar por a y luego de $f(k+1)$ a $f(k)$ al dividir entre a. Incluso $f(0)$ hace recursión a sí misma, al elevar 0 al cuadrado. Estos problemas se pueden resolver parcialmente con búsqueda con retroceso:

- Si $k > n$ entonces la forma más rápida de acabar es dividir $k - n$ veces entre a.
- Si $k - 1 < 0$ o $2k = k$ entonces no debemos hacer recursión.
- Podemos agregar un parámetro adicional a la función para que sólo se exploren n niveles de profundidad del árbol (la mejor solución no puede ser peor que multiplicar n veces por a).

Sin embargo, persiste el problema de que el árbol contiene muchas llamadas a función con el mismo valor de k ¡incluso en la misma rama! Esto es mucho peor que el trabajo repetido observado en otras recurrencias, donde éste al menos ocurría en ramas recursivas distintas. En este problema, el espacio de búsqueda es una gráfica dirigida cíclica como la siguiente:



La subgráfica del espacio de búsqueda con los vértices de $k = 0$ a $k = 8$.

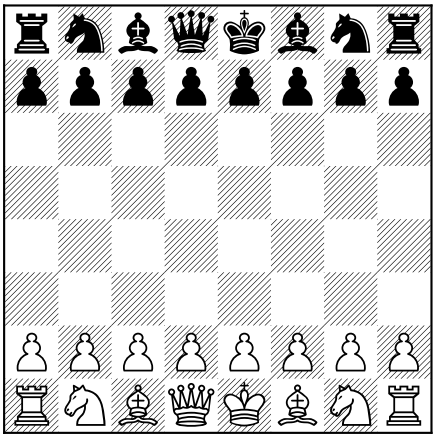
Es importante hacer notar que la gráfica anterior es implícita: ésta no forma parte de la entrada, pero podemos construirla y recorrerla mientras tomamos decisiones. Cuando estemos en el vértice k, nos dirigiremos simultáneamente a los vértices $k + 1$, $k - 1$ y $2k$. El problema de visitar repetidamente el mismo vértice se puede evitar si memorizamos la lista de vértices por los que ya pasamos (junto con su distancia más corta con respecto al vértice inicial), de modo que visitemos un vértice sólo si no lo

```
subrutina ExponenciaciónMultiplicacionesDivisiones(natural n)
    d ← arreglo(natural^{2n+1})[∞ ...]
    c ← [0], d[0] ← 0
    mientras d[n] = ∞
        s ← []
        para cada v ∈ c
            para cada v' ∈ {v + 1, v − 1, 2v}
                si 0 ≤ v' ≤ 2n ∧ d[v'] = ∞
                    d[v'] ← d[v] + 1
                    s ← (s)(v')
        c ← s
    regresa d[n]
```

habíamos hecho con anterioridad. En este problema, la solución óptima está dada por el camino más corto del vértice 0 al vértice n. Esto es un recorrido en amplitud sobre la gráfica del espacio de búsqueda.

El algoritmo anterior corre en tiempo y memoria $O(n)$, ya que la cantidad de vértices que tiene sentido visitar es reducido y sólo se visita cada vértice una vez.

Prácticamente cualquier problema de búsqueda se puede modelar haciendo uso de una gráfica que represente el espacio de búsqueda; algunos problemas en los que las tomas de decisión forman un árbol fueron casos especiales que resolvimos con divide y vencerás. Sin embargo, en algunos problemas el espacio de búsqueda es astronómicamente grande y sólo es factible visitar una pequeña porción de la gráfica. Un ejemplo es el juego del ajedrez: visto como problema, resolver el ajedrez significa determinar si existe una estrategia ganadora que incluso un oponente perfecto no pueda detener. El vértice inicial del espacio de búsqueda denota el estado inicial del tablero y el vecindario de un vértice son los tableros que se pueden obtener tras elegir uno de entre todos los movimientos posibles para el jugador en turno.

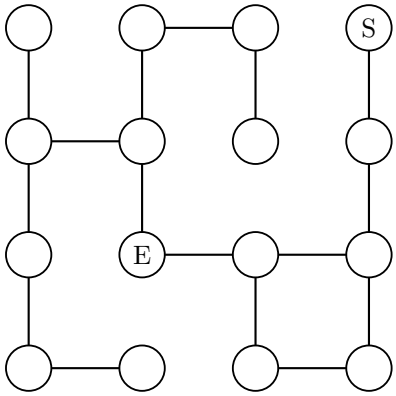


El tablero inicial de ajedrez tiene veinte vértices vecinos.

La gráfica del espacio de búsqueda del ajedrez tiene aproximadamente 10^{47} vértices. De todos modos, el recorrido en amplitud puede emplearse en problemas específicos del ajedrez, tales como encontrar la secuencia de k movimientos que llevan al mate a partir de un tablero dado, siempre y cuando el valor k sea suficientemente pequeño. La gran mayoría de las inteligencias artificiales que juegan ajedrez emplean búsqueda en amplitud en ciertos momentos de una partida.

Problema: Camino más corto en una gráfica de aristas sin pesos.
Entrada: Una gráfica g de n vértices y m aristas con dos vértices señalados v_e y v_s.
Salida: Un natural r que sea la longitud del camino más corto que existe entre v_e y v_s.

La búsqueda del camino más corto de un vértice a otro en una gráfica sin pesos se puede resolver con un recorrido en amplitud. Éste suele ser el problema subyacente de muchos otros problemas para los que la gráfica del espacio de búsqueda es implícita. Por lo mismo, éste aparece en situaciones muy diversas, tales como encontrar la secuencia de jugadas más corta que lleva a un juego a un determinado resultado o encontrar una ruta que permita escapar de un laberinto. Tomando el ejemplo del laberinto, supondremos que nos dan una gráfica incrustada en una rejilla entera con un par de vértice señalados como la entrada y la salida del laberinto, respectivamente.



Ejemplo de gráfica que denota un laberinto.
Cada vértice tiene coordenadas enteras.

La siguiente descripción del algoritmo de búsqueda en amplitud hace uso de dos estructuras de datos: un arreglo que lleva el registro de la distancia más corta encontrada para cada vértice y una cola en la que podemos agregar y sacar vértices que debemos visitar. En la cola guardaremos tanto el vértice a visitar como la distancia a la que llegaríamos a él usando la arista recorrida. En búsqueda en amplitud, la cola sigue la política de "primero en entrar, primero en salir" y la primera vez que llegamos a un vértice, también llegamos a él con la distancia más corta.

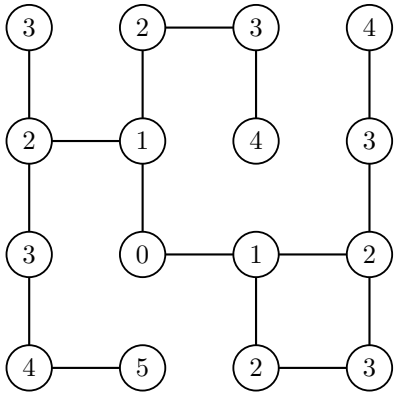
```
subrutina CaminoMásCorto(gráfica g, natural n, natural m, natural v_e, natural v_s)
    d ← arreglo(natural^{n+1})[∞ ...]
    q ← [ (v_e, 0) ]
    mientras q ≠ ∅ ∧ d[v_s] = ∞
        (v, t) ← saca(q)
        para cada v' ∈ vecindario(g, v)
            si d[v'] = ∞
                d[v'] ← t + 1
                q ← (q)((v', t + 1))
     regresa d[v_s]
```

Para calcular el camino en sí y no sólo su longitud, podemos usar arreglo de distancias para reconstruir el camino partiendo en sentido contrario, de v_s hasta v_e, siguiendo la secuencia de vértices con distancias $d[v_s], d[v_s]-1, d[v_s]-2, \ldots, d[v_e]$. En caso de empates en distancia entre vértices, se puede elegir cualquiera.

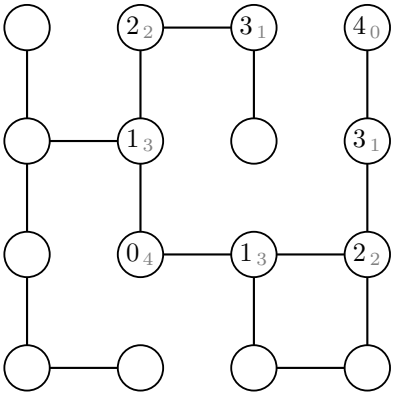
En cierto modo, una búsqueda en amplitud es ciega porque siempre procesa los vértices en todas direcciones, sin tomar en cuenta la posición del vértice destino. Sin embargo, en este ejemplo podemos implementar una política distinta. Sean i_v y j_v las coordenadas del vértice v en la rejilla entera, la distancia Manhattan $||v, u||$ de dos vértices v, u se define como $|i_v - i_u| + |j_v - j_u|$. Es fácil ver que $||v, u||$ es una cota inferior de la distancia real entre v y u. Podemos volver q una cola de prioridad si modificamos la función *saca* para que prefiera el vértice v cuya suma $d[v] + ||v.v_s||$ sea la menor de la cola. La primera vez que visitamos un vértice, se sigue garantizando que el camino usado es el más corto.

En esta variante, cuando el camino más corto de v_e a v_s es relativamente directo entonces la búsqueda evitará crear caminos que se alejen de v_s en lugar de acercarse. Por otra parte, si el camino más corto

de un vértice a otro es bastante rebuscado, la búsqueda de todos modos intentará sesgarse hacia ciertos vértices, sólo para darse cuenta que ninguno de ellos nos lleva a la salida. En este último caso, la cantidad de vértices visitados será virtualmente igual que en búsqueda en amplitud.



Distancias más cortas desde el vértice de entrada. Esto corresponde con la búsqueda en amplitud.



Búsqueda usando la distancia Manhattan (en subíndice). Algunos vértices no se visitan.

La generalización de la idea anterior se conoce como algoritmo A* y se usa con frecuencia en inteligencia artificial para reducir considerablemente el tiempo de cómputo de lo que hubiera sido una búsqueda en amplitud ordinaria. Por otra parte, cuando la gráfica es extraordinariamente grande y recorrer en amplitud incluso una pequeña fracción de ella es computacionalmente infactible, entonces conviene simplemente ignorar la mayor parte de la gráfica y concentrarnos en las partes que aparentemente son las más prometedoras. A esta idea se le conoce como búsqueda local.

14.1. Ejercicios

1. Resuelve el problema `https://omegaup.com/arena/problem/Exponenciacion-con-multiplicacio`.
2. Resuelve el problema `https://omegaup.com/arena/problem/audencia_salon_marciano`.
3. Resuelve el problema `https://omegaup.com/arena/problem/Saltando-hacia-la-pared`.
4. Resuelve el problema `https://omegaup.com/arena/problem/lsp`.

15. Algoritmos glotones

En cierto sentido, la técnica de programación dinámica es fuerza bruta bien implementada. Sin embargo, en un problema complicado es frecuente que se nos ocurran estrategias naturales o intuitivas (alguien podría llamarlas "humanas") que no hacen fuerza bruta. La pregunta que surge en esta situación es ¿la estrategia intuitiva nos dará la respuesta correcta o en verdad necesito hacer fuerza bruta?

Un algoritmo glotón es uno que, en lugar de ver cuáles son las consecuencias de cada posible decisión, elige a cada paso únicamente la alternativa que parece más atractiva o intuitiva a corto plazo. Por esta razón, los algoritmos glotones suelen ser mucho más rápidos que programación dinámica. Si nuestra intuición nos dice que un algoritmo glotón funciona, la principal dificultad suele ser demostrarlo. Por otra parte, para el mismo problema puede haber más de una estrategia glotona y posiblemente sólo una de ellas sea correcta, si es que alguna lo es. Un ejemplo clásico de cómo dos variantes de un mismo problema tienen algoritmos completamente distintos es el problema de la mochila. En este problema se quiere maximizar el valor total que se puede cargar en una mochila, si podemos guardar objetos de pesos y valores diversos y la mochila impone una restricción de capacidad total de peso.

Problema: El problema de la mochila (versión discreta).
Entrada: Dos naturales n, c y dos secuencias p, v de n enteros positivos cada una.

Salida: Un natural r que sea el valor máximo de $\sum_{i \in s}(v_i)$ sujeto a $\sum_{i \in s}(p_i) \leq c$, donde s es una subsecuencia de los índices de 0 a $n-1$.

Se cree que este problema computacionalmente difícil, ya que no se ha encontrado ningún algoritmo correcto que no sea equivalente a hacer fuerza bruta. Una estrategia que decida elegir siempre el objeto con mayor o menor v_i o con mayor o menor p_i fracasará en la instancia $n = 4$, $c = 4$, $p = (4, 2, 2, 1)$ y $v = (5, 3, 3, 1)$ donde el óptimo se consigue eligiendo los dos objetos con peso 2 y valor 3. La variante discreta del problema de la mochila se puede resolver con programación dinámica en tiempo $O(nc)$, lo cual de todos modos es exponencial con respecto al tamaño de la entrada porque involucra el valor de c, haciendo uso de la siguiente recurrencia donde n, p y v son constantes:

```
subrutina MochilaDiscreta(natural n, arreglo(entero^n) p, arreglo(entero^n) v, natural i, , natural c)
    si i = n
        regresa 0
    sino
        r ← Mochila(n, p, v, i + 1, c)
        si c ≥ p[i]
            r ← máx(r, v[i] + Mochila(n, p, v, i + 1, c − p[i]))
        regresa r
```

Problema: El problema de la mochila (versión continua).
Entrada: Dos naturales n, c y dos secuencias p, v de n enteros positivos cada una.
Salida: Un real r que sea el valor máximo de $\sum_{i \in s}(\alpha_i v_i)$ sujeto a $\sum_{i \in s}(\alpha_i p_i) \leq c$, donde s es una subsecuencia de los índices de 0 a $n-1$ y α_i es un real tal que $0 \leq \alpha_i \leq 1$.

En esta variable del problema, los reales α_i nos permiten simular el poder guardar en la mochila una parte fraccionaria de un objeto (a diferencia de la versión discreta, donde el objeto se guarda completo o no se guarda). Ante esta posibilidad, la elección de qué objeto guardar primero debe resultar obvia: conviene guardar lo más que se pueda del objeto con la mejor relación $\frac{v_i}{p_i}$. Un algoritmo que primero ordene los objetos y después intente guardarlos en la mochila tendrá una complejidad de $O(n \log_2(n))$, lo cual sí es polinomial y en general es mucho mejor que $O(nc)$.

```
subrutina MochilaContinua(natural n, arreglo(entero^n) p, arreglo(entero^n) v, natural i, natural c)
    ordena_>((p, v), λ((p[i], v[i]) ↦ v[i]/p[i]))
    r ← 0
    para i ← 0 . . . n − 1
        t ← mín(c, p[i])
        r ← r + t v[i]/p[i]
        c ← c − r
    regresa r
```

Para cada iteración del ciclo del algoritmo anterior, se tiene que $\alpha_i = \frac{t}{p_i}$, por lo que tras ordenar los objetos descendentemente por $\frac{v_i}{p_i}$, se tiene que $\frac{v_i}{p_i} \geq \frac{v_{i+1}}{p_{i+1}}$ y $\alpha_i \geq \alpha_{i+1}$. La demostración de que el algoritmo glotón es óptimo viene de observar que se desea maximizar el valor de $\sum_{i \in s}(\alpha_i v_i)$ y de que cualquier solución cuya secuencia de α_i no esté ordenada de forma ascendente se puede mejorar.

Demostración. Dadas las secuencias p y v, supondremos sin pérdida de generalidad que $\frac{v_i}{p_i} \geq \frac{v_{i+1}}{p_{i+1}}$. Supondremos que la solución óptima es r donde existe una k tal que $\alpha'_k < \alpha'_{k+1}$. Podemos construir una segunda solución factible r' con $\alpha'_k = \alpha_k + \varepsilon_k$ y $\alpha'_{k+1} = \alpha_{k+1} - \varepsilon_{k+1}$ donde $\varepsilon_k p_k = \varepsilon_{k+1} p_{k+1}$. Tenemos que $r' - r = v_k \varepsilon_k - v_{k+1} \varepsilon_{k+1} = \frac{v_k}{p_k} \varepsilon_{k+1} p_{k+1} - \frac{v_{k+1}}{p_{k+1}} \varepsilon_k p_k > 0$, por lo que $r' > r$ y se contradice la hipótesis. □

Problema: Planificación de intervalos.
Entrada: Un entero n y dos secuencias a, u de n enteros positivos cada una, tales que $a_i \leq u_i$.
Salida: Un real r que sea el valor máximo de $|s|$ tal que $u_j \leq a_i \vee u_i \leq a_j$ para todo $i, j \in s : i \neq j$, donde s es una subsecuencia de los índices de 0 a $n-1$.

Este problema se puede interpretar como sigue: si hay n eventos con horarios de inicio y fin a_i, u_i, ¿cuál es la mayor cantidad de eventos (completos) a los que podemos asistir? La restricción de asistir a un evento completo nos obliga a que los eventos que elijamos no tengan empalmes de horario. Si todos los eventos tuvieran la misma duración, entonces la estrategia es obvia: ir primero al que comienza primero. Sin embargo, si los eventos tienen duraciones distintas y aporta el mismo beneficio ir a un evento corto que a uno largo, puede ser que el evento que comienza más temporano también sea el más largo, lo que nos impediría asistir a otros eventos cortos que inicien y terminen en ese transcurso de tiempo. Ir primero al evento más corto tampoco funciona, porque puede ser que éste inicie demasiado tarde. Sin embargo, sí existe una estrategia glotona que resuelve este problema: ir primero al evento que termine más temprano.

```
subrutina PlanificaciónDeIntervalos(natural n, arreglo(entero^n) a, arreglo(entero^n) u)
    ordena_<((a, u), λ((a[i], u[i]) ↦ u_i)
    r ← 0, h ← −∞
    para i ← 0 ... n − 1
        si h ≤ a[i]
            r ← r + 1
            h ← u[i]
     regresa r
```

El algoritmo anterior tiene una complejidad de $O(n \log_2(n))$ y el siguiente argumento nos puede convencer de que la estrategia anterior es óptima: si asistimos primero al evento que termina antes, puede ser que algún otro evento comenzara más temprano pero terminará más tarde por definición; cualquiera de los dos eventos cuenta como una asistencia, pero el evento que termina más temprano nos da más posibilidades de poder asistir a otro evento posterior. Definiremos g como un arreglo de n booleanos tal que $g_i = 1$ si $i \in s$ y $g_i = 0$ en otro caso. Si desde la entrada los intervalos ya estuvieran ordenados de forma ascendente por horario de fin, entonces la subsecuencia s óptima produce un arreglo g que es el lexicográficamente mayor de entre aquéllos inducidos por subsecuencias factibles.

Demostración. Dadas las secuencias a y u, supondremos sin pérdida de generalidad que $u_i \leq u_{i+1}$. Supondremos que la solución óptima es $r_h = |s_h|$ y que la solución del algoritmo glotón es $r_g = |s_g|$ donde $r_h > r_g$. Sea h el arreglo de booleanos inducido de s_h y g el arreglo inducido de s_g. Como $r_h > r_g$ pero g es lexicográficamente mayor que h, existe una k donde $h_k < g_k$ y existe una primera $k' > k$ y una primera $k'' > k'$ donde $h_{k'} > g_{k'}$ y $h_{k''} > g_{k''}$. Como $h_{k'} = h_{k''} = 1$ y $u_{k'} \leq u_{k''}$, entonces $u_{k'} \leq a_{k''}$. Es posible modificar h sin perder factibilidad redefiniendo $h_k = 1$ y $h_{k'} = 0$, ya que $u_k \leq u'_k \leq a_{k''}$. Con respecto a la h original, la h modificada coincide en dos posiciones más con g. Repetimos el proceso anterior con la nueva h tantas veces como sea necesario, hasta que no sea posible encontrar una k que cumpla la definición dada. Esto contradice la existencia de una s_h tal que $|s_h| > |s_g|$. □

15.1. Ejercicios

1. Resuelve el problema `https://omegaup.com/arena/problem/El-baile-de-las-langostas`.
2. Resuelve el problema `https://omegaup.com/arena/problem/El-problema-de-la-mochila-c`.
3. Resuelve el problema `https://omegaup.com/arena/problem/Horarios-empalmados`.
4. Resuelve el problema `https://omegaup.com/arena/problem/Voltea-monedas-consecutivas`.

16. La clase de complejidad NP

Sabemos por intuición que existen problemas computacionales fáciles y difíciles. Esta intuición actualmente está formalizada en un concepto que se denomina clases de complejidad. La importancia de su estudio radica en que se han identificado cientos de problemas con utilidad práctica que también se sabe que son extraordinariamente difíciles, al mismo tiempo que se sabe de la existencia de problemas que son imposibles de resolver. Ante un problema nuevo, es útil saber en qué caso estamos.

Se dice que un problema es de decisión si su respuesta es *sí* o *no*. Por ejemplo, un problema de decisión es el de determinar si un natural n es par. Un problema pertenece a la clase de complejidad $\mathcal{NP}$ si:

- Es un problema de decisión.
- Existe un algoritmo no determinísta que lo resuelve en tiempo polinomial.
- Cuando la respuesta al problema de decisión es *sí*, existe un certificado de tamaño polinomial y un algoritmo determinista de tiempo polinomial que verifica la validez de la respuesta usando el certificado.

En realidad, los últimos dos puntos son equivalentes. Las siglas $\mathcal{NP}$ vienen de la frase en inglés *Nondeterministic Polynomial time*. Si además un problema en $\mathcal{NP}$ tiene un algoritmo determinista que lo resuelve en tiempo polinomial, se dice que el problema pertenece a la subclase $\mathcal{P}$. Sorpresivamente, Stephen Cook y Leonid Levin demostraron a inicios de 1970 un teorema que establece la existencia de una clase de problemas llamada $\mathcal{NP}$-Completo, la cual incluye a los problemas más difíciles en $\mathcal{NP}$ en el sentido de que cualquier algoritmo determinista de tiempo polinomial para un problema $\mathcal{NP}$-Completo implicaría que $\mathcal{P} = \mathcal{NP}$. El primer problema que se demostró ser $\mathcal{NP}$-Completo es el problema SAT.

Problema: Satisfacibilidad booleana (SAT).
Entrada: Dos enteros n, m y una fórmula booleana b de n incógnitas y m operadores lógicos.
Salida: Un booleano r que sea verdadero si y sólo existe una asignación de valores de verdad para las n incógnitas tal que b evalúe a verdadero.

Claramente, el problema SAT es un problema de decisión y un certificado de veracidad para cuando la fórmula es verdadera es justamente la asignacion de valores de verdad para las incógnitas. La idea detrás de la demostración de que SAT es $\mathcal{NP}$-Completo consiste en ver a los certificados para los problemas en $\mathcal{NP}$ como cadenas de bits sobre n incógnitas y a los algoritmos que verifican los certificados como fórmulas booleanas de tamaño polinomial. Con esto se argumenta que si existiera una forma eficiente de hacerle ingeniería inversa a tales fórmulas booleanas, se podrían resolver todos los problemas en $\mathcal{NP}$ de forma indirecta y eficiente. Desafortunadamente, al día de hoy no se conoce ningún algoritmo determinista subexponencial para resolver ningún problema $\mathcal{NP}$-Completo.

La forma usual de demostrar que un problema p es $\mathcal{NP}$-Completo es demostrar que cualquier instancia de p se puede transformar en tiempo polinomial en una instancia de un problema q que es $\mathcal{NP}$-Completo, y que la solución a la instancia de q se puede transformar en tiempo polinomial en la solución de la instancia de p. En este contexto, se dice que el problema p reduce a q. Entre los problemas que se sabe pertenecen a la clase $\mathcal{NP}$-Completo están el problema de suma de subconjuntos, la versión discreta del problema de la mochila y el problema del agente viajero (sus versiones de decisión se obtienen cambiando la optimización del resultado r por la pregunta de si r es igual a cierta k). A continuación se presenta un nuevo problema y se demuestra que es $\mathcal{NP}$-Completo mediante una reducción al problema de la mochila.

Problema: Partición balanceada.
Entrada: Un entero n y una secuencia s de n naturales.
Salida: Un booleano r que sea verdadero si y sólo existe una partición de la secuencia en dos grupos g_1 y g_2 tales que $\sum(g_1) = \sum(g_2)$.

Demostración. El problema está en $\mathcal{NP}$ porque es un problema de decisión y porque un certificado de veracidad se puede representar mediante una cadena binaria $x_1 x_2 \dots x_n$ donde $x_i = 1 \iff s_i \in g_1$.

Además, el certificado descrito es verificable en tiempo $O(n)$. Claramente, la respuesta a una instancia del problema es falsa si $\sum(s)$ es impar. En caso contrario, haremos una reducción. Una instancia del problema de partición balanceada corresponde con una instancia del problema de la mochila donde tanto los pesos como los valores de los objetos están dados por s y donde la capacidad c de la mochila está dada por $\frac{\sum(s)}{2}$. En la versión de decisión de este problema, buscamos determinar si es posible llenar la mochila. La reescritura de la instancia de un problema al otro se puede hacer en tiempo $O(n)$. Si la respuesta a la instancia del problema de la mochila es afirmativa, la solución del problema de partición balanceada tiene en g_1 a los objetos que llenaron la mochila y en g_2 a los objetos que quedaron fuera. □

Adicionalmente, los problemas de la clase de complejidad $\mathcal{NP}$-Duro son al menos tan difíciles como los de la clase $\mathcal{NP}$-Completo (y a menudo, mucho más difíciles) y no son necesariamente problemas de decisión. Formalmente, un problema p es $\mathcal{NP}$-Duro si cualquier problema $\mathcal{NP}$-Completo se puede reducir a p en tiempo polinomial. Con esta definición, todos problemas $\mathcal{NP}$-Completos también son $\mathcal{NP}$-Duros. Actualmente se cree que $\mathcal{P} \neq \mathcal{NP}$, por lo que estas clases de complejidad se verían de la siguiente forma:

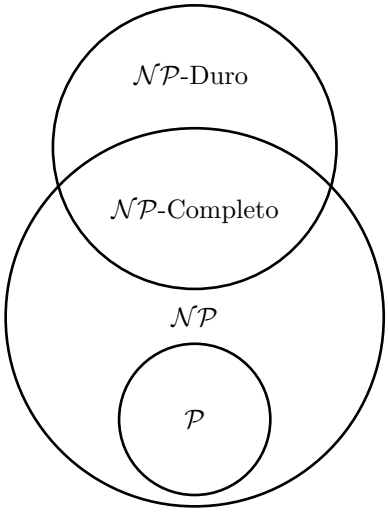


Diagrama de clases de complejidad.

Como la definición de la clase $\mathcal{NP}$-Duro es una cota inferior en la dificultad de un problema, por lo que esta clase incluye incluso problemas imposibles de resolver (formalmente llamados problemas indecidibles). El problema más famoso de este tipo es el problema de paro y a continuación se demuestra por qué no existe algún algoritmo que pueda resolverlo.

Problema: El problema de paro.
Entrada: Un natural n y una cadena de n dígitos binarios que denota un programa p.
Salida: Un booleano r que sea verdadero si y sólo si la ejecución de p termina.

Demostración. Supondremos que existe un algoritmo u oráculo $a(p)$ que puede resolver en un tiempo finito el problema de paro para el programa p. Ahora supondremos que p es de la forma:

```
subrutina p( )
    si a(p)
        para i ← 1 ... ∞
            continúa
    sino
        regresa
```

por lo que p contradice al oráculo a, lo que a su vez contradice su existencia. □

A. Implementaciones en C++

A.1. Exponenciación de naturales por elevación al cuadrado

```
int exponenciacion_natural(int a, int b) {
   if (b == 0) {
      return 1;
   } else {
      int t = exponenciacion_natural(a, b / 2);
      if (b % 2 == 0) {
         return t * t;
      } else {
         return t * t * a;
      }
   }
}
```

A.2. Coeficiente binomial recursivo

```
int coeficiente_binomial(int n, int k) {
   if (k == 0 || k == n) {
      return 1;
   } else {
      return coeficiente_binomial(n - 1, k) + coeficiente_binomial(n - 1, k - 1);
   }
}
```

A.3. Resta iterativa mediante decrementos

```
int resta_natural(int a, int b) {
   while (b != 0) {
      --a, --b;
   }
   return a;
}
```

A.4. Sumatoria iterativa

```
int sumatoria(int n) {
   int r = 0;
   for (int i = 1; i <= n; ++i) {
      r += i;
   }
   return r;
}
```

A.5. Cuenta lineal de bits prendidos

```
int cuenta_bits(int n) {
   int r = 0;
   while (n != 0) {
      r += n % 2;
      n /= 2;
   }
   return r;
}
```

A.6. Acumulación de un arreglo

```
int acumulacion(int n, int arr[]) {
   int r = 0;
   for (int i = 0; i < n; ++i) {
      r += arr[i];
   }
   return r;
}
```

A.7. Fibonacci recursivo

```
int fibonacci(int n) {
   if (n <= 1) {
      return n;
   } else {
      return fibonacci(n - 1) + fibonacci(n - 2);
   }
}
```

A.8. Búsqueda binaria

```
bool busqueda_binaria(int* ini, int* fin, int b) {
   if (ini == fin) {
      return false;
   }

   int* mitad = ini + (fin - ini) / 2;
   if (b < *mitad) {
      return busqueda_binaria(ini, mitad, b);
   } else if (b > *mitad) {
      return busqueda_binaria(mitad + 1, fin, b);
   } else {
      return true;
   }
}
```

A.9. Mezcla ordenada

```
void mezcla_ordenada(int* ini1, int* fin1, int* ini2, int* fin2, int* w) {
   while (ini1 != fin1 && ini2 != fin2) {
      *w++ = *(*ini1 <= *ini2 ? ini1 : ini2)++;
   }
   std::copy(ini1, fin1, w);
   std::copy(ini2, fin2, w);
}
```

A.10. Ordenamiento por mezcla

```
void ordenamiento_mezcla(int* ini, int* fin) {
   if (fin - ini <= 1) {
      return;
   }

   int* mitad = ini + (fin - ini) / 2;
   ordenamiento_mezcla(ini, mitad);
```

```
   ordenamiento_mezcla(mitad, fin);

   int temp[fin - ini];  // cuidado
   mezcla_ordenada(ini, mitad, mitad, fin, temp);
   std::copy(temp, temp + (fin - ini), ini);
}
```

A.11. Partición estable por predicado binario

```
template<typename F>
int* particion_estable(int* ini, int* fin, F pred) {
   int* p = ini;
   for (int* i = ini; i != fin; ++i) {
      if (pred(*i)) {
         std::swap(*p++, *i);
      }
   }
   return p;
}
```

A.12. Algoritmo de selección con último elemento como pivote

```
int* seleccion(int* ini, int* fin, int k) {
   int* p = std::partition(ini, fin - 1, [&](int v) {
      return v < *(fin - 1);
   });
   std::swap(*p, *(fin - 1));

   if (k < p - ini) {
      return seleccion(ini, p, k);
   } else if (k > p - ini) {
      return seleccion(p + 1, fin, k - (p - ini) - 1);
   } else {
      return p;
   }
}
```

A.13. Generación de cadenas binarias

```
int n;
bool arr[MAX];

// llamada inicial: cadenas_binarias(0)
void cadenas_binarias(int i) {
   if (i == n) {
      for (int i = 0; i < n; ++i) {
         std::cout << arr[i];
      }
      std::cout << "\n";
   } else {
      arr[i] = false;
      cadenas_binarias(i + 1);
      arr[i] = true;
      cadenas_binarias(i + 1);
   }
}
```

A.14. Generación de cadenas numéricas

```
int n;
int arr[MAX];

// llamada inicial: cadenas_numericas(0)
void cadenas_numericas(int i) {
   if (i == n) {
      for (int i = 0; i < n; ++i) {
         std::cout << arr[i] << " ";
      }
      std::cout << "\n";
   } else {
      for (int d = 0; d <= 9; ++d) {
         arr[i] = d;
         cadenas_numericas(i + 1);
      }
   }
}
```

A.15. Generación de permutaciones

```
int n;
int arr[MAX]; // inicializar con { 0, 1, ..., n - 1 }

// llamada inicial: permutaciones(0)
void permutaciones(int i) {
   if (i == n) {
      for (int i = 0; i < n; ++i) {
         std::cout << arr[i] << " ";
      }
      std::cout << "\n";
   } else {
      for (int j = i; j < n; ++j) {
         std::swap(arr[i], arr[j]);
         permutaciones(i + 1);
         std::swap(arr[i], arr[j]);
      }
   }
}
```

A.16. Cadenas binarias con pocos dígitos iguales consecutivos

```
int n;
int arr[MAX];

// llamada inicial: cadenas_binarias_pdic(0)
void cadenas_binarias_pdic(int i) {
   if (i == n) {
      for (int i = 0; i < n; ++i) {
         std::cout << arr[i];
      }
      std::cout << "\n";
   } else {
      if (i <= 1 || arr[i - 2] != 0 || arr[i - 1] != 0) {
         arr[i] = 0;
```

```
         cadenas_binarias_pdic(i + 1);
      }
      if (i <= 1 || arr[i - 2] != 1 || arr[i - 1] != 1) {
         arr[i] = 1;
         cadenas_binarias_pdic(i + 1);
      }
   }
}
```

A.17. Permutaciones divisibles

```
int n;
int arr[MAX]; // inicializar con { 1, 2, ..., n }

// llamada inicial: permutaciones_divisibles(0)
void permutaciones_divisibles(int i) {
   if (i == n) {
      for (int i = 0; i < n; ++i) {
         std::cout << arr[i] << " ";
      }
      std::cout << "\n";
   } else {
      for (int j = i; j < n; ++j) {
         std::swap(arr[i], arr[j]);
         if (i == 0 || (arr[i - 1] + arr[i]) % (i + 1) == 0) {
            permutaciones_divisibles(i + 1);
         }
         std::swap(arr[i], arr[j]);
      }
   }
}
```

A.18. Agente viajero (versión de camino abierto) con búsqueda con retroceso

```
int n;
int arr[MAX];  // inicializar con { 0, 1, ..., n - 1 }
int costos[MAX][MAX];
int res_global = INT_MAX;
int res_local = 0;

// llamada inicial: agente_viajero_ca(0)
void agente_viajero_ca(int i) {
   if (i == n) {
      res_global = res_local;
   } else {
      for (int j = i; j < n; ++j) {
         std::swap(arr[i], arr[j]);
         res_local += (i == 0 ? 0 : costos[arr[i - 1]][arr[i]]);
         if (res_local < res_global) {
            agente_viajero_ca(i + 1);
         }
         res_local -= (i == 0 ? 0 : costos[arr[i - 1]][arr[i]]);
         std::swap(arr[i], arr[j]);
      }
   }
}
```

A.19. Fibonacci recursivo con memorización

```
int mem[MAX + 1];  // inicializar con -1

int fibonacci_memorizado(int n) {
   if (mem[n] == -1) {
      if (n <= 1) {
         mem[n] = n;
      } else {
         mem[n] = fibonacci_memorizado(n - 1) + fibonacci_memorizado(n - 2);
      }
   }
   return mem[n];
}
```

A.20. Coeficiente binomial recursivo con memorización

```
int mem[MAX_N + 1][MAX_K + 1];  // inicializar con -1

int binomial_memorizado(int n, int k) {
   if (mem[n][k] == -1) {
      if (k == 0 || k == n) {
         mem[n][k] = 1;
      } else {
         mem[n][k] = binomial_memorizado(n - 1, k) + binomial_memorizado(n - 1, k - 1);
      }
   }
   return mem[n][k];
}
```

A.21. Fibonacci con programación dinámica

```
int fibonacci_pd(int n) {
   int mem[std::max(2, n + 1)];
   for (int i = 0; i <= n; ++i) {
      if (i <= 1) {
         mem[i] = i;
      } else {
         mem[i] = mem[i - 1] + mem[i - 2];
      }
   }
   return mem[n];
}
```

A.22. Suma de subconjuntos

```
int n;
int arr[MAX];

// llamada inicial: suma_subconjuntos(0, k_original)
bool suma_subconjuntos(int i, int k) {
   if (i == n) {
      return (k == 0);
   } else {
      bool res = suma_subconjuntos(i + 1, k);
      if (k >= arr[i]) {
```

```
         res = res || suma_subconjuntos(i + 1, k - arr[i]);
      }
      return res;
   }
}
```

A.23. Suma de subconjuntos con programación dinámica

```
bool suma_subconjuntos_pd(int n, int arr[], int k_original) {
   bool mem[n + 1][k_original + 1];
   for (int i = n; i >= 0; --i) {
      for (int k = 0; k <= k_original; ++k) {
         if (i == n) {
            mem[i][k] = (k == 0);
         } else {
            bool res = mem[i + 1][k];
            if (k >= arr[i]) {
               res = res || mem[i + 1][k - arr[i]];
            }
            mem[i][k] = res;
         }
      }
   }
   return mem[0][k_original];
}
```

A.24. Subsecuencia común más larga

```
std::string a, b;

// llamada inicial: lcs(0, 0)
int lcs(int i, int j) {
   if (i == a.size( ) || j == b.size( )) {
      return 0;
   } else if (a[i] == b[j]) {
      return 1 + lcs(i + 1, j + 1);
   } else {
      return std::max(lcs(i, j + 1), lcs(i + 1, j));
   }
}
```

A.25. Subsecuencia común más larga con programación dinámica

```
auto lcs(const std::string& a, const std::string& b) {
   int mem[a.size( ) + 1][b.size( ) + 1];
   for (int i = a.size( ); i >= 0; --i) {
      for (int j = b.size( ); j >= 0; --j) {
       if (i == a.size( ) || j == b.size( )) {
            mem[i][j] = 0;
         } else if (a[i] == b[j]) {
            mem[i][j] = 1 + mem[i + 1][j + 1];
         } else {
            mem[i][j] = std::max(mem[i][j + 1], mem[i + 1][j]);
         }
      }
   }
```

```
   std::vector<std::pair<int, int>> res;
   int i = 0, j = 0;
   while (i < a.size( ) && j < b.size( )) {
      if (a[i] == b[j]) {
         res.emplace_back(i++, j++);
      } else if (mem[i][j] == mem[i][j + 1]) {
         ++j;
      } else {
         ++i;
      }
   }
   return res;
}
```

A.26. Subsecuencia común más larga con memoria lineal

```
int lcs(const std::string& a, const std::string& b) {
   int mem[2][b.size( ) + 1];
   int *actual = mem[0], *previo = mem[1];
   for (int i = a.size( ); i >= 0; --i, std::swap(actual, previo)) {
      for (int j = b.size( ); j >= 0; --j) {
       if (i == a.size( ) || j == b.size( )) {
            actual[j] = 0;
         } else if (a[i] == b[j]) {
            actual[j] = 1 + mem[1][j + 1];
         } else {
            actual[j] = std::max(actual[j + 1], mem[1][j]);
         }
      }
   }
   return previo[0];
}
```

A.27. Multiplicación encadenada de matrices

```
struct matriz {
   int f, c;
};

matriz arr[MAX];

int multiplica_matrices(int i, int f) {
   if (f - i == 1) {
      return 0;
   } else {
      int res = INT_MAX ;
      for (int j = i + 1; j < f; ++j) {
         int t1 = multiplica_matrices(i, j);
         int t2 = multiplica_matrices(j, f);
         res = std::min(res, t1 + t2 + (arr[i].f * arr[j].f * arr[f - 1].c));
      }
      return res;
   }
}
```

A.28. Multiplicación encadenada de matrices con programación dinámica

```
struct matriz {
   int f, c;
};

int multiplica_matrices(int n, matriz arr[]) {
   int memoria[n][n + 1];
   for (int t = 1; t <= n; ++t) {
      for (int i = 0; i <= n - t; ++i) {
         int f = i + t;
         if (f - i == 1) {
            memoria[i][f] = 0;
         } else {
            int res = INT_MAX;
            for (int j = i + 1; j < f; ++j) {
               auto t1 = memoria[i][j];
               auto t2 = memoria[j][f];
               res = std::min(res, t1 + t2 + (arr[i].f * arr[j].f * arr[f - 1].c));
            }
            memoria[i][f] = res;
         }
      }
   }
   return memoria[0][n];
}
```

A.29. Exponenciación con multiplicaciones y divisiones

```
int exponenciacion_multiplicaciones_divisiones(int n) {
   int tam = 2 * n + 1, distancia[tam];
   std::fill(distancia, distancia + tam, INT_MAX);
   std::vector<int> actual = { 0 };
   distancia[0] = 0;

   while (distancia[n] == INT_MAX) {
      std::vector<int> siguiente;
      for (int vertice : actual) {
         int vecinos[] = { vertice + 1, vertice * 2, vertice - 1 };
         for (int checar : vecinos) {
            if (0 <= checar && checar < tam && distancia[checar] == INT_MAX) {
               distancia[checar] = distancia[vertice] + 1;
               siguiente.push_back(checar);
            }
         }
      }
      actual = siguiente;
   }
   return distancia[n];
}
```

A.30. Longitud del camino más corto con búsqueda en amplitud

```
int camino_mas_corto(const std::vector<int> adyacencia[], int n, int e, int s) {
   int distancia[n + 1];
   std::fill(distancia, distancia + n + 1, INT_MAX);
```

```
   std::deque<std::pair<int, int>> cola = { {e, 0} };
   distancia[e] = 0;

   while (!cola.empty( ) && distancia[s] == INT_MAX) {
      auto [v, t] = cola.front( );
      cola.pop_front( );
      for (int vecino : adyacencia[v]) {
         if (distancia[vecino] == INT_MAX) {
            distancia[vecino] = t + 1;
            cola.emplace_back(vecino, t + 1);
         }
      }
   }
   return distancia[s];
}
```

A.31. El problema de la mochila (versión discreta)

```
struct objeto {
   int peso, valor;
};

int n;
objecto arr[MAX];

// llamada inicial: mochila(0, c)
// esta recurrencia puede emplearse para implementar un algoritmo de programación dinámica
int mochila(int i, int c) {
   if (i == n) {
      return 0;
   } else {
      int res = mochila(i + 1, c);
      if (c >= arr[i].peso) {
         res = std::max(res, arr[i].valor + f(i + 1, c - arr[i].peso));
      }
      return res;
   }
}
```

A.32. El problema de la mochila (versión continua)

```
struct objeto {
   int peso, valor;
};

double mochila(int n, objecto arr[], int c) {
   std::sort(arr, arr + n, [](const objeto& a, const objeto& b) {
      return double(a.valor) / a.peso > double(b.valor) / b.peso;
   });

   double res = 0;
   for (int i = 0; i < n; ++i) {
      double quitar = std::min(c, double(arr[i].peso));
      res += quitar * double(arr[i].valor) / arr[i].peso;
      c -= quitar;
   }
```

```
   return res;
}
```

A.33. Planificación de intervalos

```
struct intervalo {
   int ini, fin;
};

int planificacion_intervalos(int n, intervalo arr[]) {
   std::sort(arr, arr + n, [](const evento& e1, const evento& e2) {
      return e1.fin < e2.fin;
   });

   int res = 0, h = 0;
   for (int i = 0; i < n; ++i) {
      if (h <= arr[i].ini) {
         res += 1;
         h = arr[i].fin;
      }
   }
   return res;
}
```

B. Soluciones de ejercicios

En general, siempre hay más de una posible respuesta correcta para cada pregunta. Razonen cuidadosamente las respuestas que propongo para que decidan si las suyas son equivalentes.

Soluciones de 2.1

1. Demuestra que el siguiente algoritmo regresa $\frac{n(n+1)}{2}$ para todo entero $n \geq 0$.

```
subrutina f(natural n)
    si n = 0
        regresa 0
    sino
        regresa n + f(n − 1)
```

> **Solución:** El caso base es $n = 0$ y observamos que $\frac{0(0+1)}{2} = 0$ que es lo que regresa el algoritmo. Supondremos que $f(n')$ es correcta para $0 \leq n' < n$ y demostraremos que $f(n)$ es correcta. Para $n > 0$ tenemos $f(n) = n + \text{f}(n-1)$ que por hipótesis de inducción es $n + \frac{(n-1)((n-1)+1)}{2}$. Desarrollando obtenemos $\frac{2n}{2} + \frac{(n-1)((n-1)+1)}{2} = \frac{2n+n^2-n}{2} = \frac{n^2+n}{2} = \frac{n(n+1)}{2}$ que es correcto.

2. Demuestra que el siguiente algoritmo regresa $2^n - 1$ para todo entero $n \geq 0$.

```
subrutina f(natural n)
    si n = 0
        regresa 0
    sino
        regresa 1 + 2f(n − 1)
```

> **Solución:** El caso base es $n = 0$ y observamos que $2^0 - 1 = 0$ que es lo que regresa el algoritmo. Supondremos que $f(n')$ es correcta para $0 \leq n' < n$ y demostraremos que $f(n)$ es correcta. Para $n > 0$ tenemos $f(n) = 1+2\text{f}(n-1)$ que por hipótesis de inducción es $1+2(2^{n-1}-1)$. Desarrollando obtenemos $1 + 2(2^{n-1}) - 2(1) = 1 + 2^n - 2 = 2^n - 1$ que es correcto.

3. Demuestra que el siguiente algoritmo regresa $(n + 1)! - 1$ para todo entero $n \geq 0$.

```
subrutina f(natural n)
    si n = 0
        regresa 0
    sino
        regresa n(n!) + f(n − 1)
```

> **Solución:** El caso base es $n = 0$ y observamos que $(0 + 1)! - 1 = 0$ que es lo que regresa el algoritmo. Supondremos que $f(n')$ es correcta para $0 \leq n' < n$ y demostraremos que $f(n)$ es correcta. Para $n > 0$ tenemos $f(n) = n(n!) + \text{f}(n-1)$ que por hipótesis de inducción es $n(n!) + ((n-1)+1)! - 1$. Desarrollando obtenemos $n(n!) + n! - 1 = n!(n+1) - 1 = (n+1)! - 1$.

4. Demuestra que el siguiente algoritmo regresa $a + b$ para todos los enteros $a, b \geq 0$.

```
subrutina f(natural a, natural b)
    si b = 0
        regresa a
    sino
        regresa f(a + 1, b − 1)
```

Solución: El caso base es $b = 0$ y observamos que $a + 0 = a$ que es lo que regresa el algoritmo. Supondremos que $f(a', b')$ es correcta para $0 \leq b' < b$ y demostraremos que $f(a, b)$ es correcta. Para $b > 0$ tenemos $f(a, b) = f(a + 1, b - 1)$ que por hipótesis de inducción es $a + 1 + (b - 1)$. Desarrollando obtenemos $a + 1 + (b - 1) = a + b$ que es correcto.

Soluciones de 3.1

1. Demuestra que el siguiente algoritmo regresa a^b para todo entero $a, b \geq 0$.

subrutina f(natural a, natural b)
  $r \leftarrow 1$
  mientras $b > 0$
    $r \leftarrow r \times a$
    $b \leftarrow b - 1$
  regresa r

Solución: Proponemos las invariantes $r_t = a^t$ y $b_t = b - t$. Primero probaremos que son correctas para $t = 0$, donde tenemos $r_0 = a^0 = 1$ y $b_0 = b - 0 = b$ que son los valores de las variables antes del ciclo. Ahora supondremos que las invariantes son ciertas para $t - 1$ y demostraremos que también lo son para t, lo que implica que la $(t - 1)$-ésima evaluación de la condición del ciclo fue verdadera. Por el ciclo sabemos que $r_t = r_{t-1} \times a$ y que $b_t = b_{t-1} - 1$. Por la hipótesis de inducción tenemos $r_t = a^{t-1} \times a = a^t$ y $b_t = (b - (t - 1)) - 1 = b - t$, por lo que son correctas. Dado que $b \geq 0$ y se decrementa en 1 por cada iteración del ciclo, el ciclo termina cuando $b_{t'} = 0$. Como $b_t = b - t$ entonces $t' = b$. Tenemos que $r_b = a^b$ que es lo que queríamos obtener.

2. Demuestra que el siguiente algoritmo regresa $\frac{n(n+1)(2n+1)}{6}$ para todo entero $n \geq 0$.

subrutina f(natural n)
  $r \leftarrow 0$, $k \leftarrow 1$
  mientras $k \leq n$
    $r \leftarrow r + k^2$
    $k \leftarrow k + 1$
  regresa r

Solución: Proponemos las invariantes $r_t = \frac{t(t+1)(2t+1)}{6}$ y $k_t = 1 + t$. Primero probaremos que son correctas para $t = 0$, donde tenemos $r_0 = \frac{0(0+1)(2(0)+1)}{6} = 0$ y $k_0 = 1 + 0 = 1$ que son los valores de las variables antes del ciclo. Ahora supondremos que las invariantes son ciertas para $t-1$ y demostraremos que también lo son para t, lo que implica que la $(t-1)$-ésima evaluación de la condición del ciclo fue verdadera. Por el ciclo sabemos que $r_t = r_{t-1} + k_{t-1}^2$ y que $k_t = k_{t-1} + 1$. Por la hipótesis de inducción tenemos

$$\begin{aligned} r_t &= \frac{(t-1)((t-1)+1)(2(t-1)+1)}{6} + (1 + (t-1))^2 \\ &= \frac{(t-1)(t)(2t-1)}{6} + t^2 \\ &= \frac{(t-1)(t)(2t-1) + 6t^2}{6} \\ &= \frac{(2t^3 - t^2 - 2t^2 + t) + 6t^2}{6} \\ &= \frac{2t^3 + 3t^2 + t}{6} = \frac{(t)(t+1)(2t+1)}{6} \end{aligned}$$

y $k_t = (1 + (t - 1)) + 1 = t + 1$, por lo que son correctas. Dado que k se incrementa en 1 por cada

iteración del ciclo, el ciclo termina cuando $k_{t'} = n+1$. Como $k_t = 1+t$ entonces $t' = n$. Tenemos que $r_n = \frac{n(n+1)(2n+1)}{6}$ que es lo que queríamos obtener.

Soluciones de 4.1

1. Encuentra una función $\overleftarrow{\mathbb{T}}(L)$ que cuente la cantidad de asignaciones que realiza $f(n)$ en el peor caso. Recuerda que L es el tamaño de la entrada y que un entero se codifica con w bits.

```
subrutina f(natural n)
    r ← n
    si n mód 2 = 0
        r ← 5 × r
    regresa r
```

Solución: La función realiza una asignación incondicional y una asignación condicional. En el peor caso, se ejecutarán las dos. Entonces $\overleftarrow{\mathbb{T}}(L) = 2$, lo cual es independiente de L. En este caso, se dice que $\overleftarrow{\mathbb{T}}(L)$ es constante con respecto a L.

2. Encuentra una función $\overleftarrow{\mathbb{T}}(L)$ que cuente la cantidad de asignaciones que realiza $g(n)$ en el peor caso si $n = 2^k$. Recuerda que L es el tamaño de la entrada y que un entero se codifica con w bits.

```
subrutina f(natural n)
    r ← 0, i ← 1
    mientras i < n
        r ← r + 1
        i ← 2 × i
    regresa r
```

Solución: Primero necesitamos saber cuántas veces se ejecuta el ciclo, por lo que propondremos la invariante $i_t = 2^t$. Primero probaremos que es correcta para $t = 0$, donde tenemos $i_0 = 2^0 = 1$ que es el valor de la variable antes del ciclo. Ahora supondremos que la invariante es cierta para $t-1$ y demostraremos que también lo es para t, lo que implica que la $(t-1)$-ésima evaluación de la condición del ciclo fue verdadera. Por el ciclo sabemos que $i_t = 2 \times i_{t-1}$. Por la hipótesis de inducción tenemos $i_t = 2 \times (2^{t-1}) = 2^t$, por lo que es correcta. Dado que $i_t = 2^t$ y $n = 2^k$, el ciclo termina cuando $i_{t'} = n$ con $t' = k$. El ciclo ejecuta k iteraciones y el algoritmo realiza dos asignaciones incondicionales y dos asignaciones por iteración, por lo que en total realiza $2 + 2k$ asignaciones. Ahora tenemos que $k = \log_2(n)$ y $n = 2^{w-1}$ en el peor caso. Entonces $k = \log_2(2^{w-1}) = w - 1$ y $L = w$, por lo que $\overleftarrow{\mathbb{T}}(L) = 2 + 2(L-1)$ que es lineal en L.

Soluciones de 5.1

1. Resuelve exactamente la siguiente recurrencia:

$$T(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 + T(n-1) & \text{si } n > 0 \end{cases}$$

Solución: Mediante sustitución repetida obtenemos:

$$\begin{aligned} T(n) &= 1 + T(n-1) \\ &= 1 + 1 + T(n-2) = 2 + T(n-2) \\ &= 2 + 1 + T(n-3) = 3 - T(n-3) \\ &\dots \end{aligned}$$

El patrón que aparece es $T(k) = k + T(n-k)$ y el argumento de la función se vuelve 0 para $T(n) = n + T(n-n)$, donde al desarrollar tenemos $T(n) = n + T(0) = n$. Con esto proponemos la definición alternativa de $T(n) = n$. Aprovechamos que $T(0) = 0$ en la definición original y usamos este caso como caso base de la inducción. Ahora supondremos que $T(n-1)$ es correcta y probaremos que $T(n)$ también lo es. Tenemos que $T(n) = 1 + T(n-1)$ y por la hipótesis de inducción tenemos $T(n) = 1 + (n-1) = n$, por lo que es correcta.

2. Resuelve exactamente la siguiente recurrencia:

$$T(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 + 2T(n-1) & \text{si } n > 1 \end{cases}$$

Solución: Mediante sustitución repetida obtenemos:

$$\begin{aligned} T(n) &= 1 + 2T(n-1) \\ &= 1 + 2(1 + 2T(n-2)) = 3 + 4T(n-2) \\ &= 3 + 4(1 + 2T(n-3)) = 7 + 8T(n-3) \\ &= 7 + 8(1 + 2T(n-4)) = 15 + 16T(n-4) \\ &\dots \end{aligned}$$

El patrón que aparece es $T(k) = (2^k - 1) + 2^k T(n-k)$ y el argumento de la función se vuelve 0 para $T(n) = (2^n - 1) + 2^n T(n-n)$, donde al desarrollar tenemos $T(n) = (2^n - 1) + 2^n T(0) = 2^n - 1$. Con esto proponemos la definición alternativa de $T(n) = 2^n - 1$. Aprovechamos que $T(0) = 0$ en la definición original y usamos este caso como caso base de la inducción. Ahora supondremos que $T(n-1)$ es correcta y probaremos que $T(n)$ también lo es. Tenemos que $T(n) = 1 + 2T(n-1)$ y por la hipótesis de inducción tenemos $T(n) = 1 + 2(2^{n-1} - 1) = 1 + 2(2^{n-1}) - 2 = 2^n - 1$.

3. Resuelve exactamente la recurrencia T_2 de ordenamiento por mezcla para $n = 2^k$.

Solución: La recurrencia T_2 corresponde con el peor caso de ordenamiento por mezcla en cuanto al número de comparaciones entre elementos:

$$T_2(n) = \begin{cases} 0 & \text{si } n \leq 1 \\ n - 1 + 2T_2(\frac{n}{2}) & \text{si } n > 1 \end{cases}$$

Sea $T = T_3$, obtenemos lo siguiente mediante sustitución repetida:

$$\begin{aligned} T(n) &= n - 1 + 2T\left(\frac{n}{2}\right) \\ &= n - 1 + \frac{2n}{2} - 2 + 4T\left(\frac{n}{4}\right) = 2n - 3 + 4T\left(\frac{n}{4}\right) \\ &= 2n - 3 + \frac{4n}{4} - 4 + 8T\left(\frac{n}{8}\right) = 3n - 7 + 8T\left(\frac{n}{8}\right) \\ &= 3n - 7 + \frac{8n}{8} - 8 + 16T\left(\frac{n}{16}\right) = 4n - 15 + 16T\left(\frac{n}{16}\right) \\ &\dots \end{aligned}$$

El patrón que aparece es $T(k) = \log_2(k)n - (k-1) + kT(\frac{n}{k})$ y la fracción se vuelve 1 para $T(n) = \log_2(n)n - (n-1) + nT(\frac{n}{n})$, donde al desarrollar tenemos $T(n) = \log_2(n)n - n + 1 + nT(1) = \log_2(n)n - n + 1$. Con esto podemos proponer la siguiente definición alternativa de $T(n)$:

$$T(n) = \begin{cases} 0 & \text{si } n \leq 1 \\ n\log_2(n) - n + 1 & \text{si } n > 1 \end{cases}$$

Claramente, ambas definiciones coinciden para $T(0)$ y $T(1)$. Para el caso recursivo, demostraremos primero el caso $T(2)$. Tenemos que $T(2) = 2-1+2T(\frac{2}{2}) = 1$ según la definición original, mientras $T(2) = 2\log_2(2) - 2 + 1 = 1$ según la definición propuesta. Ahora supondremos que $T(\frac{n}{2})$ es correcta y demostraremos que $T(n)$ también lo es. Tenemos que $T(n) = n - 1 + 2T(\frac{n}{2})$ y eso es igual a $n - 1 + 2(\frac{n}{2}\log_2(\frac{n}{2}) - \frac{n}{2} + 1)$ por la hipótesis de inducción. Desarrollando tenemos $T(n) = n-1+n\log_2(\frac{n}{2})-n+2 = n\log_2(\frac{n}{2})+1 = n(\log_2(n)+\log_2(\frac{1}{2}))+1 = n\log_2(n)-n+1$.

4. Resuelve el problema `https://omegaup.com/arena/problem/Resolviendo-una-recurrencia`.

Solución: La recurrencia del problema toma una $n = 2^k$ y es

$$T(n) = \begin{cases} c & \text{si } n = 1 \\ aT\left(\frac{n}{2}\right) + b & \text{si } n > 1 \end{cases}$$

Mediante la técnica de sustitución repetida, obtenemos lo siguiente:

$$\begin{aligned} T(n) &= b + aT\left(\frac{n}{2}\right) \\ &= b + a\left(b + aT\left(\frac{n}{4}\right)\right) = b + ab + a^2T\left(\frac{n}{4}\right) \\ &= b + ab + a^2\left(b + aT\left(\frac{n}{8}\right)\right) = b + ab + a^2b + a^3T\left(\frac{n}{8}\right) \\ &= b + ab + a^2b + a^3\left(b + aT\left(\frac{n}{16}\right)\right) = b + ab + a^2b + a^3b + a^4T\left(\frac{n}{16}\right) \\ &\dots \end{aligned}$$

El patrón que aparece es $T(k) = \sum_{i=0}^{\log_2(k)-1}(a^ib) + a^{\log_2(k)}T\left(\frac{n}{k}\right)$ y la fracción se vuelve 1 en $T(n) = \sum_{i=0}^{\log_2(n)-1}(a^ib) + a^{\log_2(n)}T\left(\frac{n}{n}\right)$, donde $T(1) = c$. Entonces, los valores a imprimir son $t = \log_2(n) - 1, u = a, w = b, x = a, y = \log_2(n), z = c$.

```
#include <cmath>
#include <iostream>

int main( ) {
   int a, b, c, n;
   std::cin >> a >> b >> c >> n;
   std::cout << std::log2(n) - 1 << " " << a << " "
             <<                    b << " " << a << " "
             <<     std::log2(n) << " " << c << "\n";
}
```

5. Resuelve el problema `https://omegaup.com/arena/problem/Resolviendo-una-recurrencia-la-r`.

Solución: La recurrencia del problema toma una $n = 2^k$ y es

$$T(n) = \begin{cases} c & \text{si } n = 1 \\ T\left(\frac{n}{2}\right) + an & \text{si } n > 1 \end{cases}$$

Mediante la técnica de sustitución repetida, obtenemos lo siguiente:

$$\begin{aligned} T(n) &= an + T\left(\frac{n}{2}\right) \\ &= an + \frac{an}{2} + T\left(\frac{n}{4}\right) \\ &= an + \frac{an}{2} + \frac{an}{4} + T\left(\frac{n}{8}\right) \\ &= an + \frac{an}{2} + \frac{an}{4} + \frac{an}{8} + T\left(\frac{n}{16}\right) \\ &\dots \end{aligned}$$

El patrón que aparece es $T(k) = an\sum_{i=0}^{\log_2(k)-1}\left(\frac{1}{2^i}\right) + T\left(\frac{n}{k}\right)$ y la fracción se vuelve 1 en $T(n) = an\sum_{i=0}^{\log_2(n)-1}\left(\frac{1}{2^i}\right) + T\left(\frac{n}{n}\right)$, donde $T(1) = c$. Entonces, los valores a imprimir son $w = a, x = n, y = \log_2(n) - 1, z = c$ aunque se permite intercambiar los valores de w, x.

```
#include <cmath>
#include <iostream>

int main( ) {
   int a, c, n;
   std::cin >> a >> c >> n;
   std::cout << a << " " << n << " " << std::log2(n) - 1 << " " << c << "\n";
}
```

Soluciones de 6.1

1. Demuestra que $2^n + 5 \in \Theta(2^n)$.

Solución: Primero demostraremos que $2^n + 5 \in O(2^n)$. Si elegimos $n_0 = 3$ y $c = 2$, debemos demostrar que $2^n + 5 \leq 2(2^n)$ para toda $n \geq 3$. Podemos reescribir lo anterior como $2^n + 5 \leq 2^n + 2^n$ donde es fácil ver que $5 \leq 2^n$ para $n \geq 3$. Ahora demostraremos que $2^n + 5 \in \Omega(2^n)$. Si elegimos $n_0 = 1$ y $c = 1$, debemos demostrar que $2^n + 5 \geq 2^n$ para toda $n \geq 1$, lo cual es directo.

2. Demuestra que $2^n \in O(n!)$.

Solución: Si elegimos $n_0 = 5$ y $c = 1$, debemos demostrar que $2^n \leq n!$ para toda $n \geq 5$. Haremos la demostración por inducción y primero probaremos el caso de $n = 5$ donde tenemos que $2^5 = 32 \leq 5! = 120$. Ahora supondremos que $2^n \leq n!$ es cierta para n y demostraremos que también lo es para $n+1$. Reescribiendo tenemos $2^{n+1} = 2(2^n)$ y $(n+1)! = (n+1)(n!)$, por lo que deseamos probar que $2(2^n) \leq (n+1)(n!)$ donde sabemos que $2^n \leq n!$ por hipótesis de inducción y que $2 \leq n + 1$ para $n \geq 5$, por lo que es cierto que $2(2^n) \leq (n+1)(n!)$.

3. Demuestra que $n^3 - 2n^2 - 4n \in \Omega(n)$.

Solución: Si elegimos $n_0 = 6$ y $c = 1$, debemos demostrar que $n^3 - 2n^2 - 4n \geq n$ para toda $n \geq 6$. Dividiendo ambos lados entre n, esto es equivalente a demostrar que $n^2 - 2n - 4 \geq 1$. Haremos la demostración por inducción y primero probaremos el caso de $n = 6$ donde tenemos que $6^2 - 2(6) - 4 = 20 \geq 1$. Ahora supondremos que $n^2 - 2n - 4 \geq 1$ es cierto para n y demostraremos que también lo es para $n + 1$. Evaluando tenemos que $(n+1)^2 - 2(n+1) - 4 = n^2 - 5$, donde es claro que $n^2 - 5 \geq 1$ para $n \geq 6$.

Soluciones de 7.1

1. Usando el teorema maestro, determina el comportamiento asintótico de $T(n) = T(\frac{n}{4}) + n^2$.

Solución: Usaremos el caso 3 del teorema maestro, donde $f(n) = \Omega(n^c)$ con $c = 2$ y $r = \log_4(1) = 0$, por lo que $c > r$. Más aún, el trabajo local de $T(n)$ que es $f(n) = n^2$ es mayor que $af(\frac{n}{b}) = (\frac{n}{4})^2 = \frac{n^2}{16}$. Por el teorema maestro, tenemos que $T(n) \in \Theta(n^2)$.

2. Usando el teorema maestro, determina el comportamiento asintótico de $T(n) = 4T(\frac{n}{4}) + n^2$.

Solución: Usaremos el caso 3 del teorema maestro, donde $f(n) = \Theta(n^c)$ con $c = 2$ y $r = \log_4(4) = 1$, por lo que $c > r$. Más aún, el trabajo local de $T(n)$ que es $f(n) = n^2$ es mayor que $af(\frac{n}{b}) = 4(\frac{n}{4})^2 = \frac{n^2}{4}$. Por el teorema maestro, tenemos que $T(n) \in \Theta(n^2)$.

3. Usando el teorema maestro, determina el comportamiento asintótico de $T(n) = T(\frac{n}{2}) + \log_2(n)$.

Solución: Usaremos el caso 2 del teorema maestro, donde $f(n) = \Theta(n^r \log_2(n)^c)$ con $c = 1$ y $r = \log_2(1) = 0$. Por el teorema maestro, tenemos que $T(n) \in \Theta(\log_2(n)^2)$.

Soluciones de 8.1

1. Resuelve el problema `https://omegaup.com/arena/problem/Transiciones-en-cadenas-binarias`.

Solución: Para resolver este problema basta implementar la estrategia descrita en las notas. Sin embargo, se deben usar enteros de al menos 64 bits para los cálculos.

```
#include <iostream>

long long f(int n) {
   if (n <= 1) {
      return 0;
   } else {
      return 2 * f(n - 1) + (1LL << (n - 1));
   }
}

int main( ) {
   int n;
   std::cin >> n;
   std::cout << f(n) << "\n";
}
```

2. Resuelve el problema `https://omegaup.com/arena/problem/Suma-de-digitos`.

Solución: Para resolver este problema basta implementar la estrategia descrita en las notas.

```
#include <algorithm>
#include <iostream>

int f(int n) {
```

```
   if (n == 0) {
      return 1;
   } else {
      int res = 0;
      for (int i = 1; i <= std::min(n, 9); ++i) {
         res += f(n - i);
      }
      return res;
   }
}

int main( ) {
   int n;
   std::cin >> n;
   std::cout << f(n) << "\n";
}
```

3. Resuelve el problema `https://omegaup.com/arena/problem/Con-pocos-ceros-consecutivos`.

Solución: Una idea para resolver este problema es imaginar que nosotros "escribimos" uno o más dígitos sobre la cadena binaria y luego delegamos el resto del trabajo. Para que la recursión no tenga que preocuparse por los dígitos que nosotros "escribimos", debemos garantizar que el último dígito que escribimos es 1. Si estamos en el caso $f(n)$ y escribimos un 1, podemos delegar el resto del trabajo a $f(n-1)$; si escribimos 01 podemos delegar el resto del trabajo a $f(n-2)$; si escribimos 001 podemos delegar el resto del trabajo a $f(n-3)$. Esta estrategia nos obliga a definir varios casos base para evitar caer en recursión con n negativa.

```
#include <iostream>

int f(int n) {
   if (n == 1) {
      return 2;
   } else if (n == 2) {
      return 4;
   } else if (n == 3) {
      return 7;
   } else {
      return f(n - 1) + f(n - 2) + f(n - 3);
   }
}

int main( ) {
   int n;
   std::cin >> n;
   std::cout << f(n);
}
```

4. Resuelve el problema `https://omegaup.com/arena/problem/Cadenas-de-digitos-sin-ceros-con`.

Solución: Para resolver este problema conviene que nuestra función tome dos parámetros: n que es la cantidad de dígitos que faltan por escribir y k que es la cantidad de ceros que aún podemos

escribir. Aplicaremos la idea del problema anterior de imaginar que "escribimos" una parte de la cadena y luego delegamos el resto del trabajo. Un caso base es cuanto ya no podemos escribir ceros, por lo que para cada caracter restante sólo tenemos 9 opciones disponibles. El caso general (cuando faltan varios caracteres por escribir y aún tenemos ceros disponibles) se plantea como sigue: si escribimos algo distinto de 0 entonces podemos delegar el resto del trabajo a $f(n-1,k)$; si escribimos 0 escribiremos también algo distinto de 0 para que, al delegar el resto del trabajo a $f(n-2,k-1)$, ésta no tenga que protegerse de formar ceros consecutivos. Los otros casos base se definen para evitar caer en recursión con n negativa. Se deben usar enteros de 64 bits.

```
#include <cmath>
#include <iostream>

long long f(int n, int k) {
   if (k == 0) {
      return std::pow(9, n);
   } else if (n == 0) {
      return 1;
   } else if (n == 1) {
      return 10;
   } else {
      return 9 * f(n - 1, k) + 9 * f(n - 2, k - 1);
   }
}

int main( ) {
   int n, k;
   std::cin >> n >> k;
   std::cout << f(n, k) << "\n";
}
```

Soluciones de 10.1

1. Resuelve el problema `https://omegaup.com/arena/problem/Cadenas-binarias-con-pocos-digit`.

Solución: Para resolver este problema basta implementar la estrategia descrita en las notas.

```
#include <iostream>

int n;
int arr[25];

void cadenas_binarias_pdic(int i) {
   if (i == n) {
      for (int i = 0; i < n; ++i) {
         std::cout << arr[i];
      }
      std::cout << "\n";
   } else {
      if (i <= 1 || arr[i - 2] != 0 || arr[i - 1] != 0) {
         arr[i] = 0;
         cadenas_binarias_pdic(i + 1);
      }
```

```
      if (i <= 1 || arr[i - 2] != 1 || arr[i - 1] != 1) {
         arr[i] = 1;
         cadenas_binarias_pdic(i + 1);
      }
   }
}

int main( ) {
   std::cin >> n;
   cadenas_binarias_pdic(0);
}
```

2. Resuelve el problema `https://omegaup.com/arena/problem/Permutaciones-divisibles`.

Solución: Para resolver este problema basta implementar la estrategia descrita en las notas.

```
#include <algorithm>
#include <iostream>

int n;
int arr[20];

void genera(int i) {
   if (i == n) {
      for (int i = 0; i < n; ++i) {
         std::cout << arr[i] << " ";
      }
      std::cout << "\n";
   } else {
      for (int j = i; j < n; ++j) {
         std::swap(arr[i], arr[j]);
         if (i == 0 || (arr[i - 1] + arr[i]) % (i + 1) == 0) {
            genera(i + 1);
         }
         std::swap(arr[i], arr[j]);
      }
   }
}

int main( ) {
   std::cin >> n;
   for (int i = 0; i < n; ++i) {
      arr[i] = i + 1;
   }
   genera(0);
}
```

3. Resuelve el problema `https://omegaup.com/arena/problem/El-agente-viajero`.

Solución: Para resolver este problema basta implementar la estrategia descrita en las notas. Sólo hay que tener cuidado en observar que, al comenzar el programa, no hemos calculado el costo de ninguna solución. Por esta razón, el valor inicial del mejor resultado encontrado debe comenzar

en un valor muy grande; así, cualquier solución válida será considerada mejor.

```
#include <algorithm>
#include <climits>
#include <iostream>

int n;
int arr[15];
int costos[15][15];
int res_global = INT_MAX;
int res_local = 0;

void genera(int i) {
   if (i == n) {
      res_global = res_local;
   } else {
      for (int j = i; j < n; ++j) {
         std::swap(arr[i], arr[j]);
         res_local += (i == 0 ? 0 : costos[arr[i - 1]][arr[i]]);
         if (res_local < res_global) {
            genera(i + 1);
         }
         res_local -= (i == 0 ? 0 : costos[arr[i - 1]][arr[i]]);
         std::swap(arr[i], arr[j]);
      }
   }
}

int main( ) {
   std::cin >> n;
   for (int i = 0; i < n; ++i) {
      arr[i] = i;
   }
   for (int i = 0; i < n; ++i) {
      for (int j = 0; j < n; ++j) {
         std::cin >> costos[i][j];
      }
   }

   genera(0);
   std::cout << res_global << "\n";
}
```

4. Resuelve el problema `https://omegaup.com/arena/problem/Cadenas-viborita`.

Solución: La diferencia en valor absoluto de dos dígitos contiguos debe ser 1, por lo que conviene verificar que el dígito que nosotros queremos anotar en la cadena sea compatible con el dígito de atrás. También debemos validar que el último dígito de la cadena sea igual que el primero, pero esto lo podemos hacer ya que terminamos de generar la cadena completa. Como tal, nuestra poda generará soluciones inválidas que debemos filtrar, pero la poda por dígitos contigüos es suficiente para resolver el problema en tiempo.

```
#include <cstdlib>
```

```
#include <iostream>

int n;
int arr[25];
int res = 0;

void genera(int i) {
   if (i == n) {
      if (arr[0] == arr[n - 1]) {
         res += 1;
      }
   } else if (i == 0) {
      for (int d = 0; d <= 9; ++d) {
         arr[i] = d;
         genera(i + 1);
      }
   } else {
      int anterior = arr[i - 1];
      if (anterior != 0) {
         arr[i] = anterior - 1;
         genera(i + 1);
      }
      if (anterior != 9) {
         arr[i] = anterior + 1;
         genera(i + 1);
      }

   }
}

int main( ) {
   std::cin >> n;
   genera(0);
   std::cout << res;
}
```

5. Resuelve el problema `https://omegaup.com/arena/problem/El-problema-de-las-N-princesas`.

Solución: Para resolver este problema, se debe observar que es obligatorio que en cada fila y en cada columna exista exactamente una princesa. El problema entonces lo podemos plantear como: "la princesa de fila actual, ¿en qué columna debe ser colocada?" Como todas columnas deben aparecer pero no pueden repetirse, la asignación de columnas a filas puede modelarse como una permutación. Los ataques en diagonal se resuelven con la restricción de que las permutaciones no puedan tener elementos contiguos (que denotan las columnas elegidas para filas contiguas) que estén a distancia 1 entre sí. El árbol podado no es óptimo pero sí es lo suficientemente chico.

```
#include <algorithm>
#include <iostream>

int n;
int arr[12];
int res = 0;
```

```
void genera(int i) {
   if (i == n) {
      res += 1;
   } else {
      for (int j = i; j < n; ++j) {
         std::swap(arr[i], arr[j]);
         if (i == 0 || std::abs(arr[i] - arr[i - 1]) != 1) {
            genera(i + 1);
         }
         std::swap(arr[i], arr[j]);
      }
   }
}

int main( ) {
   std::cin >> n;
   for (int i = 0; i < n; ++i) {
      arr[i] = i;
   }
   genera(0);
   std::cout << res << "\n";
}
```

6. Resuelve el problema `https://omegaup.com/arena/problem/El-recorrido-del-principe`.

Solución: Para resolver este problema, se debe llevar el registro de las celdas visitadas por el príncipe durante su recorrido. También se debe modelar el movimiento del príncipe en el tablero. Si una celda que deseamos visitar ya había sido visitada durante el recorrido actual, no debemos hacer recursión. Al variar la decisión de qué celda es la siguiente a visitar, es importante desmarcar la celda anteriormente elegida. El árbol podado no es óptimo pero sí es lo suficientemente chico.

```
#include <initializer_list>
#include <iostream>
#include <utility>

int n, visitados;
int res = 0;
bool arr[6][6] = { };

void cuenta(int i, int j) {
   if (visitados == n * n) {
      res += 1;
   } else {
      for (auto p : { std::pair(i - 1, j), std::pair(i + 1, j),
                      std::pair(i, j - 1), std::pair(i, j + 1) }) {
         if (0 <= p.first && p.first < n && 0 <= p.second && p.second < n &&
             !arr[p.first][p.second]) {
            visitados += 1;
            arr[p.first][p.second] = true;
            cuenta(p.first, p.second);
            arr[p.first][p.second] = false;
            visitados -= 1;
         }
```

```
      }
   }
}

int main( ) {
   std::cin >> n;
   visitados = 1;
   arr[0][0] = true;
   cuenta(0, 0);
   std::cout << res << "\n";
}
```

Soluciones de 12.1

1. Resuelve el problema `https://omegaup.com/arena/problem/El-rectangulo-de-domino`.

Solución: Para $n = 1$ sólo hay una solución (la ficha vertical) y para $n = 2$ hay dos soluciones (dos fichas verticales o dos horizontales). El caso general de $f(n)$ se deduce como sigue: podemos poner una ficha vertical y delegar el trabajo a $f(n-1)$ o poner dos horizontales y delegar el trabajo a $f(n-2)$. La recurrencia resultante es casi la misma que Fibonacci y exhibe una gran cantidad de trabajo repetido. Podemos aplicar recursión con memorización inicializando el arreglo con un centinela menor o igual a 0 y debemos tener cuidado en usar un entero de al menos 64 bits.

```
#include <iostream>

long long memoria[50 + 1];

long long f(int n) {
   if (memoria[n] == -1) {
      if (n <= 2) {
         memoria[n] = n;
      } else {
         memoria[n] = f(n - 1) + f(n - 2);
      }
   }
   return memoria[n];
}

int main( ) {
   int n;
   std::cin >> n;
   for (int i = 1; i <= n; ++i) {
      memoria[i] = -1;
   }
   std::cout << f(n) << "\n";
}
```

2. Resuelve el problema `https://omegaup.com/arena/problem/La-funcion-loca`.

Solución: Aunque el rango de los dos parámetros es de 0 a 10^5, es inviable declarar una matriz cuadrada de esa magnitud. Debemos darnos cuenta que la recurrencia siempre divide sus parámetros entre 2 (excepto en el caso base), por lo que en realidad sólo hay una cantidad logarítmica

de valores distintos que pueden tomar los parámetros durante la evaluación. Podemos añadirle a la recurrencia la cantidad de veces que hemos divido a x y a con respecto a la invocación original y usar esa información para acceder a la matriz de memoria, la cual ahora puede ser muy pequeña.

```
#include <iostream>

long long memoria[20][20];

long long f(int x, int a, int dx, int da) {
   if (memoria[dx][da] == -1) {
      if (a == 0) {
         memoria[dx][da] = (x + 1) / 2;
      } else if (x == 0) {
         memoria[dx][da] = 2 * a;
      } else {
         memoria[dx][da] = f(x / 2, a, dx + 1, da) +  f(x, a / 2, dx, da + 1);
      }
   }
   return memoria[dx][da];
}

int main( ) {
   int x, a;
   std::cin >> x >> a;
   for (int i = 0; i < 20; ++i) {
      for (int j = 0; j < 20; ++j) {
         memoria[i][j] = -1;
      }
   }
   std::cout << f(x, a, 0, 0);
}
```

3. Resuelve el problema `https://omegaup.com/arena/problem/Evaluando-una-funcion`.

Solución: La dificultad de este problema es que la n de la recurrencia puede ser negativa. Por otra parte, debemos observar que el valor absoluto de n jamás aumenta, por lo que basta un arreglo de memoria de tamaño $2n + 1$ y basta mapear los accesos al arreglo a un rango no negativo.

```
#include <iostream>

long long memoria[100 + 1];

long long f(int n) {
   if (memoria[n + 50] == 0) {
      if (n == 0) {
         memoria[n + 50] = 1;
      } else {
         memoria[n + 50] = (n < 0 ? f(-n) + f(n + 1) + 2 : f(-n + 1) + 1);
      }
   }
   return memoria[n + 50];
}
```

```
int main( ) {
   int n;
   std::cin >> n;
   std::cout << f(n) << "\n";
}
```

4. Resuelve el problema `https://omegaup.com/arena/problem/Evaluando-una-funcion-algo-rara`.

Solución: En este problema, uno de los dos parámetros de la recurrencia se mantiene constante durante su evaluación, por lo que podemos omitir una de las dimensiones de lo que hubiera sido una matriz gigantesca, para así volvera un arreglo. Sin embargo, el problema pide evaluar la recurrencia varias veces, asi que el arreglo de memoria puede no ser válido si la constante de la recurrencia cambia. Otra dificultad es que la recurrencia puede tomar valores muy grandes y se necesita evaluar con enteros de 64 bits sin signo (donde el resultado es módulo 2^{64}), por lo que no obvio cuál podría ser un buen valor centinela. Afortunadamente, es poco frecuente que la recurrencia evalúe a 0, por lo que se puede usar este valor como centinela e ignorar el trabajo repetido que aparecería al confundir el centinela con un valor memorizado que también vale 0.

```
#include <iostream>

unsigned long long memoria[10000 + 1];

unsigned long long f(int x, int y) {
   if (memoria[y] == -1) {
      if (y <= 2) {
         memoria[y] = x + y;
      } else {
         memoria[y] = x + f(x, y - 1) + 5 * f(x, y - 2);
      }
   }
   return memoria[y];
}

int main( ) {
   int n;
   std::cin >> n;

   for (int i = 0; i < n; ++i) {
      int x, y;
      std::cin >> x >> y;
      for (int i = 0; i <= y; ++i) {
         memoria[i] = -1;
      }
      std::cout << f(x, y) << "\n";
   }
}
```

Soluciones de 13.1

1. Resuelve el problema `https://omegaup.com/arena/problem/Bajando-por-el-arbol`.

Solución: En este problema, conviene observar que n sólo sirve para determinar la altura del árbol completo en cuestión. Definiremos $f(h, d)$ como la cantidad total de nodos alcanzables que existen en un árbol de altura h si sólo podemos bajar d veces por la derecha. La llamada inicial de la siguiente recurrencia es $f(\text{popcount}(n), d)$:

$$f(h,d) = \begin{cases} 0 & \text{si } h = 0 \\ 1 + f(h-1,d) & \text{si } d = 0 \\ 1 + f(h-1,d) + f(h-1,d-1) & \text{en otro caso} \end{cases}$$

Dado que h siempre se decrementa conforme avanza la recursión, podemos llenar la tabla dinámica por filas comenzando por $h = 0$.

```
#include <iostream>

int main( ) {
   int n, d;
   std::cin >> n >> d;
   int h = __builtin_popcount(n);

   int arr[h + 1][d + 1];
   for (int i = 0; i <= h; ++i) {
      for (int j = 0; j <= d; ++j) {
         if (i == 0) {
            arr[i][j] = 0;
         } else {
            arr[i][j] = 1 + arr[i - 1][j] + (j != 0 ? arr[i - 1][j - 1] : 0);
         }
      }
   }

   std::cout << arr[h][d] << "\n";
}
```

2. Resuelve el problema `https://omegaup.com/arena/problem/interminable-mermelada`.

Solución: En este problema, hay tres decisiones que podemos tomar con respecto a la mermelada en los estantes: ignorar la mermelada de ambos estantes, comer la mermelada del lado izquierdo o comer la mermelada del lado derecho. Definiremos $f(i)$ como la mayor cantidad de mermelada que podemos comer de los estantes e_0 y e_1 desde el piso i hasta el piso $n-1$. En la recurrencia, e y n son constantes implícitas. La llamada inicial de la siguiente recurrencia es $f(0)$:

$$f(i) = \begin{cases} 0 & \text{si } i \geq n \\ \max(f(i+1), \max\limits_{k \in \{0,1\}} \{e_k[i] + f(i + \max(1, e_k[i]))\}) & \text{en otro caso} \end{cases}$$

Dado que i siempre se incrementa conforme avanza la recursión, podemos llenar la tabla dinámica por filas comenzando por $i = n$. La complejidad del algoritmo propuesto es $O(n)$.

```
#include <algorithm>
#include <iostream>
```

```
int main( ) {
   int n;
   std::cin >> n;

   int botes[2][n];
   for (int i = 0; i < n; ++i) {
      std::cin >> botes[0][i] >> botes[1][i];
   }

   int res[n + 1];
   for (int i = n; i >= 0; --i) {
      if (i == n) {
         res[n] = 0;
      } else {
         res[i] = std::max({ res[i + 1],
            botes[0][i] + res[std::min(n, i + std::max(1, botes[0][i]))],
            botes[1][i] + res[std::min(n, i + std::max(1, botes[1][i]))]
         });
      }
   }

   std::cout << res[0] << "\n";
}
```

3. Resuelve el problema `https://omegaup.com/arena/problem/Los-arboles-del-bosque-de-cedros`.

Solución: Para cada árbol, podemos elegir entre ignorarlo o llevárnoslo. Definiremos $f(i, c)$ como la menor cantidad de la madera que podemos recolectar del bosque a desde el árbol i hasta el árbol $n-1$, siendo que ya recolectamos c unidades y necesitamos recolectar al menos m unidades. En la recurrencia, a, n y m son constantes implícitas. Como deseamos minimizar, una solución inválida regresará ∞ en su caso base. La llamada inicial de la siguiente recurrencia es $f(0, 0)$:

$$f(i,c) = \begin{cases} c & \text{si } c \geq m \\ \infty & \text{si } c < m \wedge i = n \\ \text{mín}(f(i+1,c), f(i+1,c+a[i])) & \text{en otro caso} \end{cases}$$

Dado que i siempre se incrementa conforme avanza la recursión, podemos llenar la tabla dinámica por filas comenzando por $i = n$. La complejidad del algoritmo propuesto es $O(nm)$.

```
#include <algorithm>
#include <climits>
#include <iostream>

int main( ) {
   int n, m;
   std::cin >> n >> m;

   int arboles[n];
   for (int i = 0; i < n; ++i) {
      std::cin >> arboles[i];
   }
```

```
   int tabla[n + 1][m + 350 + 1];
   for (int i = n; i >= 0; --i) {
      for (int c = m + 350; c >= 0; --c) {
         if (c >= m) {
            tabla[i][c] = c;
         } else if (i == n) {
            tabla[i][c] = INT_MAX;
         } else {
            tabla[i][c] = std::min(tabla[i + 1][c], tabla[i + 1][c + arboles[i]]);
         }
      }
   }

   std::cout << tabla[0][0] << "\n";
}
```

4. Resuelve el problema `https://omegaup.com/arena/problem/El-ladron-precavido`.

Solución: Para cada objeto, podemos elegir entre ignorarlo o llevárnoslo. Cuando nos lo llevamos, estamos obligados a ignorar el siguiente. Definiremos $f(i, c)$ como el mayor peso total que podemos recolectar de la bóveda a desde el objeto i hasta el objeto $n-1$, siendo que sólo tenemos capacidad suficiente para recolectar c unidades. En la recurrencia, a y n son constantes implícitas. La llamada inicial de la siguiente recurrencia es $f(0, c)$:

$$f(i,c) = \begin{cases} 0 & \text{si } i \geq n \\ f(i+1,c) & \text{si } i < n \wedge c < a[i] \\ \text{máx}(f(i+1,c), a[i] + f(i+2, c-a[i])) & \text{en otro caso} \end{cases}$$

Dado que i siempre se incrementa conforme avanza la recursión, podemos llenar la tabla dinámica por filas comenzando por $i = n$. La complejidad del algoritmo propuesto es $O(nc)$.

```
#include <algorithm>
#include <iostream>

int main( ) {
   int n, c;
   std::cin >> n >> c;

   int pesos[n];
   for (int i = 0; i < n; ++i) {
      std::cin >> pesos[i];
   }

   int memoria[n + 2][c + 1];
   for (int i = n + 1; i >= 0; --i) {
      for (int j = 0; j <= c; ++j) {
         if (i >= n) {
            memoria[i][j] = 0;
         } else {
            int res1 = memoria[i + 1][j];
            int res2 = (j >= pesos[i] ?
               memoria[i + 2][j - pesos[i]] + pesos[i] : -1
```

```
            );
            memoria[i][j] = std::max(res1, res2);
         }
      }
   }

   std::cout << memoria[0][c] << "\n";

   int i = 0;
   while (i < n) {
      if (c >= pesos[i] &&
          memoria[i][c] == memoria[i + 2][c - pesos[i]] + pesos[i]) {
         std::cout << i << " ";
         c -= pesos[i];
         i += 2;
      } else {
         i += 1;
      }
   }
}
```

Soluciones de 14.1

1. Resuelve el problema `https://omegaup.com/arena/problem/Exponenciacion-con-multiplicacio`.

Solución: Para resolver este problema, basta con aplicar la estrategia descrita en las notas.

```
#include <algorithm>
#include <climits>
#include <iostream>
#include <vector>

int main( ) {
   int n;
   std::cin >> n;

   int tam = 2 * n + 1, distancia[tam];
   std::fill(distancia, distancia + tam, INT_MAX);
   std::vector<int> actual = { 0 };
   distancia[0] = 0;

   while (distancia[n] == INT_MAX) {
      std::vector<int> siguiente;
      for (int vertice : actual) {
         int vecinos[] = { vertice + 1, vertice * 2, vertice - 1 };
         for (int checar : vecinos) {
            if (0 <= checar && checar < tam && distancia[checar] == INT_MAX) {
               distancia[checar] = distancia[vertice] + 1;
               siguiente.push_back(checar);
            }
         }
      }
```

```
      actual = siguiente;
   }
   std::cout << distancia[n];
}
```

2. Resuelve el problema `https://omegaup.com/arena/problem/audencia_salon_marciano`.

Solución: Para resolver este problema, basta con aplicar búsqueda en amplitud sobre una gráfica implícita formada por las coordenadas válidas dentro del salón marciano.

```
#include <iostream>
#include <vector>

struct coord {
   int i, j;
};

int main( ) {
   int n, m, c, s;
   std::cin >> n >> m >> c >> s;

   int matriz[n][m];
   for (int i = 0; i < n; ++i) {
      for (int j = 0; j < m; ++j) {
         matriz[i][j] = -1;
      }
   }

   std::vector<coord> actual = { {0, 0} };
   matriz[0][0] = 0;

   while (matriz[n - 1][m - 1] == -1 && !actual.empty( )) {
      std::vector<coord> siguiente;
      for (coord celda : actual) {
         coord vecinos[] = {
            {celda.i - c, celda.j},
            {celda.i - s, celda.j},
            {celda.i + c, celda.j},
            {celda.i + s, celda.j},
            {celda.i, celda.j - c},
            {celda.i, celda.j - s},
            {celda.i, celda.j + c},
            {celda.i, celda.j + s},
         };
         for (coord checar : vecinos) {
            if (0 <= checar.i && checar.i < n &&
                0 <= checar.j && checar.j < m &&
                matriz[checar.i][checar.j] == -1) {
               matriz[checar.i][checar.j] = matriz[celda.i][celda.j] + 1;
               siguiente.push_back(checar);
            }
         }
      }
```

```
      actual = siguiente;
   }

   std::cout << matriz[n - 1][m - 1] << "\n";
}
```

3. Resuelve el problema `https://omegaup.com/arena/problem/Saltando-hacia-la-pared`.

Solución: Para resolver este problema, basta con aplicar búsqueda en amplitud teniendo cuidado de no salirse del rango de posiciones factibles (no podemos ir más atrás de nuestra posición inicial ni superar la pared). A cada paso, hay cuatro movimientos a contemplar: un salto corto a ambas direcciones y un salto largo también en ambas direcciones.

```
#include <deque>
#include <iostream>
#include <map>

int main( ) {
   int c, s, d;
   std::cin >> c >> s >> d;

   std::map<int, int> tabla = { { 0, 0 } };
   std::deque<int> cola = { 0 };
   while (tabla.find(d) == tabla.end( ) && !cola.empty( )) {
      int actual = cola.front( );
      cola.pop_front( );
      for (int checar : std::initializer_list<int>{
         actual - c, actual - s, actual + c, actual + s
      }) {
         if (0 <= checar && checar <= d &&
             tabla.emplace(checar, checar - actual).second) {
            cola.push_back(checar);
         }
      }
   }

   auto iter = tabla.find(d);
   if (iter == tabla.end( )) {
      std::cout << -1 << "\n";
   } else {
      std::deque<int> saltos;
      for (int pos = d, salto; (salto = tabla[pos]) != 0; pos -= salto) {
         saltos.push_front(salto);
      }

      std::cout << saltos.size( ) << "\n";
      for (int salto : saltos) {
         std::cout << salto << " ";
      }
   }
}
```

4. Resuelve el problema `https://omegaup.com/arena/problem/lsp`.

Solución: Este problema se resuelve con búsqueda en amplitud, pero la existencia de lobos y el requerimiento de no enfrentar a más de L de ellos complica bastante el problema. Puede ocurrir que una ruta que llegue rápidamente una celda cercana a la salida pero haya enfrentando una cantidad exagerada de lobos resulte infactible si para alcanzar la salida necesitos enfrentar aún más lobos. En ese sentido, no podemos saber de antemano si una ruta rápida que enfrente muchos lobos es mejor que una ruta larga que enfrente pocos. Llevaremos el registro de lobos enfrentados en cada ruta de la búsqueda y no visitaremos ni muros ni el que hubiera sido el lobo $L + 1$ de la ruta. La primera vez que lleguemos a una celda del laberinto, registraremos la cantidad de lobos que enfrentamos usando la ruta actual y permitiremos que rutas más largas visiten de nuevo esa celda si se enfrentaron una cantidad menor de lobos (actualizando la información de la celda para endurecer el requerimiento de llegar con pocos lobos).

```
#include <algorithm>
#include <climits>
#include <iostream>
#include <vector>

struct coord {
   int i, j, p, w;
};

struct registro {
   int p, w;
};

int main( ) {
   int w, n;
   std::cin >> w >> n;

   char tablero[n][n];
   coord entrada, salida;
   for (int i = 0; i < n; ++i) {
      for (int j = 0; j < n; ++j) {
         std::cin >> tablero[i][j];
         if (tablero[i][j] == 'E') {
            entrada = {i, j};
         } else if (tablero[i][j] == 'S') {
            salida = {i,j};
         }
      }
   }

   registro matriz[n][n];
   for (int i = 0; i < n; ++i) {
      for (int j = 0; j < n; ++j) {
         matriz[i][j] = {INT_MAX,INT_MAX};
      }
   }

   matriz[entrada.i][entrada.j] = {0,0};
   std::vector<coord> actual = {entrada};
   while (matriz[salida.i][salida.j].p == INT_MAX &&
          !actual.empty( )) {
```

```
      std::vector<coord> siguiente;
      for (coord celda : actual) {
         coord vecinos[] = {
            {celda.i - 1, celda.j, celda.p + 1, celda.w},
            {celda.i + 1, celda.j, celda.p + 1, celda.w},
            {celda.i, celda.j - 1, celda.p + 1, celda.w},
            {celda.i, celda.j + 1, celda.p + 1, celda.w}
         };
         for (coord checar : vecinos) {
            if (0 <= checar.i && checar.i < n &&
                0 <= checar.j && checar.j < n &&
                tablero[checar.i][checar.j] != '#') {
               checar.w += (tablero[checar.i][checar.j] == '*');
               if (checar.w <= w &&
                   (checar.p < matriz[checar.i][checar.j].p ||
                    checar.w < matriz[checar.i][checar.j].w)) {
                  matriz[checar.i][checar.j].p = std::min(
                     matriz[checar.i][checar.j].p,
                     checar.p
                  );
                  matriz[checar.i][checar.j].w = std::min(
                     matriz[checar.i][checar.j].w,
                     checar.w
                  );
                  siguiente.push_back(checar);
               }
            }
         }
      }
      actual = siguiente;
   }

   int res = matriz[salida.i][salida.j].p;
   std::cout << (res == INT_MAX ? -1 : res);
}
```

Soluciones de 15.1

1. Resuelve el problema `https://omegaup.com/arena/problem/El-baile-de-las-langostas`.

Solución: Como no podemos repetir animales por filas, una estrategia correcta consiste en ordenar los animales (o contar cuántos hay de cada tipo) y asignarlos ordenadamente en una matriz por columnas, sin cambiar de columna hasta llenar la anterior.

```
#include <algorithm>
#include <iostream>

int main( ) {
   int f, c;
   std::cin >> f >> c;

   int n = f * c, arr[n];
   for (int i = 0; i < n; ++i) {
```

```
      std::cin >> arr[i];
   }
   std::sort(arr, arr + n);

   for (int i = 0; i < f; ++i) {
      for (int j = 0; j < c; ++j) {
         std::cout << arr[f * j + i] << ' ';
      }

      std::cout << '\n';
   }
}
```

2. Resuelve el problema `https://omegaup.com/arena/problem/El-problema-de-la-mochila-c`.

Solución: Para resolver este problema, basta aplicar la estrategia descrita en las notas.

```
#include <algorithm>
#include <cstdio>
#include <iostream>

struct objeto {
   int peso, valor;
};

int main( ) {
   int n; double c;
   std::cin >> n >> c;

   objeto arr[n];
   for (int i = 0; i < n; ++i) {
      std::cin >> arr[i].peso >> arr[i].valor;
   }
   std::sort(arr, arr + n, [](const objeto& a, const objeto& b) {
      return double(a.valor) / a.peso > double(b.valor) / b.peso;
   });

   double res = 0;
   for (int i = 0; i < n; ++i) {
      double quitar = std::min(c, double(arr[i].peso));
      res += quitar * double(arr[i].valor) / arr[i].peso;
      c -= quitar;
   }

   std::printf("%.2f\n", res);
}
```

3. Resuelve el problema `https://omegaup.com/arena/problem/Horarios-empalmados`.

Solución: Para resolver este problema, basta aplicar la estrategia descrita en las notas.

```
#include <algorithm>
```

```cpp
#include <iostream>

struct evento {
   int ini, fin;
};

int main( ) {
   int n;
   std::cin >> n;

   evento arr[n];
   for (int i = 0; i < n; ++i) {
      std::cin >> arr[i].ini >> arr[i].fin;
   }

   std::sort(arr, arr + n, [](evento e1, evento e2) {
      return e1.fin < e2.fin;
   });

   int h = 0, res = 0;
   for (int i = 0; i < n; ++i) {
      if (arr[i].ini >= h) {
         res += 1;
         h = arr[i].fin;
      }
   }

   std::cout << res;
}
```

4. Resuelve el problema `https://omegaup.com/arena/problem/Voltea-monedas-consecutivas`.

Solución: Para resolver este problema, debemos comenzar observando que el estado final de cada moneda depende únicamente de la cantidad de veces que las operaciones la afectan. Por otra parte, debemos observar que aplicar dos veces una operación en el mismo intervalo tiene el efecto de deshacer la operación anterior, ya que $1 + 1 \equiv 0 \pmod 2$. Como además la suma es conmutativa, sólo tiene caso aplicar cero veces o una vez la operación en cada intervalo posible. Las moneda de los extremos sólo son afectadas por los intervalos extremos, por lo que la decisión de si aplicar o no una operación la podemos hacer de izquierda a derecha y es de hecho, una decisión obligada. Al final sólo verificaremos si la fila completa de monedas ya está completamente en soles.

```cpp
#include <algorithm>
#include <iostream>

int main( ) {
   int n, m;
   std::cin >> n >> m;

   bool arr[n];
   for (int i = 0; i < n; ++i) {
      std::cin >> arr[i];
   }
```

```
    int res = 0;
    for (int i = 0; i <= n - m; ++i) {
        if (arr[i] == 0) {
            res += 1;
            for (int j = 0; j < m; ++j) {
                arr[i + j] = !arr[i + j];
            }
        }
    }

    if (std::count(arr, arr + n, 1) == n) {
        std::cout << res << "\n";
    } else {
        std::cout << -1 << "\n";
    }
}
```