Teórica 1 - 2da parte: Técnicas de diseño de algoritmos

1. Técnicas de diseño de algoritmos

En el transcurso de esta clase estudiaremos las siguientes técnicas:

- Fuerza bruta
- Algoritmos golosos
- Recursividad
- Dividir y conquistar (*divide and conquer*)
- Búsqueda con retroceso (*backtracking*)
- Programación dinámica
- Heurísticas y algoritmos aproximados

2. Fuerza bruta

Muchos problemas pueden resolverse buscando una solución de forma fácil, pero, a la vez, generalmente ineficiente. El método consiste en analizar ***todas*** las posibilidades.

Estos algoritmos son fáciles de inventar e implementar y siempre ***funcionan***. Pero generalmente son muy ineficientes.

Veamos un ejemplo.

Ejemplo 1. ***Problema de las n reinas****: Queremos hallar todas las formas posibles de colocar n reinas en un tablero de ajedrez de $n \times n$ casillas, de forma que ninguna reina amenace a otra. Es decir, no puede haber dos reinas en la misma fila, columna o diagonal.*

Una solución inmediata es aplicar fuerza bruta, es decir hallar ***todas*** *las formas posibles de colocar n reinas en un tablero de $n \times n$ y luego seleccionar las que satisfagan las restricciones del problema. En cada configuración tenemos que elegir las n posiciones donde ubicar a las reinas de los n^2 posibles casilleros. Entonces, el número de configuraciones que analizará el algoritmo es:*

$$\binom{n^2}{n} = \frac{n^2!}{(n^2-n)!n!}$$

Pero fácilmente podemos ver que la mayoría de las configuraciones que analizaríamos no cumplen las restricciones del problema y que trabajamos de más.

3. Algoritmos golosos

El método goloso es la técnica de diseño de algoritmos más simple. Generalmente los algoritmos golosos son utilizados para resolver problemas de optimización. Estos algoritmos son fáciles de desarrollar e implementar y, cuando ***funcionan***, son eficientes. Sin embargo, muchos problemas no pueden ser resueltos mediante esta técnica. En esos casos, proporcionan ***heurísticas*** sencillas que en general permiten construir soluciones razonables, pero sub-óptimas.

La idea es construir una solución paso por paso, de manera de hacer ***la mejor elección posible localmente*** según una función de selección, sin considerar (o haciéndolo débilmente) las implicancias de esta selección. Es decir, en cada etapa se toma la decisión que parece mejor basándose en la información disponible en ese momento, sin tener en cuenta las consecuencias futuras. Una decisión tomada nunca es revisada y no se evaluan alternativas.

Ejemplo 2. ***Problema de la mochila continuo:*** *Contamos con una mochila con una capacidad máxima C (peso máximo), donde queremos cargar las objetos (todo el objeto o una fracción de él, por eso lo de continuo en el nombre del problema) de n elementos. Cada elemento i, pesa un determinado peso p_i y llevarlo nos brinda un beneficio b_i.*

Queremos determinar qué objetos debemos incluir en la mochila, sin excedernos del peso máximo C, de modo tal de maximizar el beneficio total entre los objetos seleccionados. En la versión más simple de este problema vamos a suponer que podemos poner parte de un objeto en la mochila (continuo).

Un algoritmo goloso para este problema, agregaría objetos a la mochila mientras esta tenga capacidad libre. En líneas generales:

```
llenerMachila(C,n,P,B)
    entrada: C capacidad, n cantidad de elementos,
             P arreglo con los pesos, B arreglo con los beneficios
    salida: arreglo X, donde X[i] proporción del elemento i colocado en la mochila,
             benef beneficio obtenido

    ordenar los elementos según algún criterio
    mientras C > 0 hacer
        i ← siguiente elemento
        X[i] ← min(1, C/P[i])
        C ← C − P[i] × X[i]
        benef ← benef + B[i] × X[i]
    fin mientras
    retornar X,benef
```

Algunos criterios posibles para ordenar los objetos son:

- *ordenar los objetos en orden decreciente de su beneficio*
- *ordenar los objetos en orden creciente de su peso*
- *ordenar los objetos en orden decreciente según ganancia por unidad de peso (b_i/p_i)*

¿Sirven estas ideas? ¿Dan el resultado correcto?

Supongamos que $n = 5$, $C = 100$ y los beneficios y pesos están dados en la tabla

	1	*2*	*3*	*4*	*5*
p	*10*	*20*	*30*	*40*	*50*
b	*20*	*30*	*66*	*40*	*60*
b/p	*2.0*	*1.5*	*2.2*	*1.0*	*1.2*

Los resultados que obtendríamos ordendando los ítems según cada criterio son:

- *mayor beneficio* b_i*:* $66 + 60 + 40/2 = 146$.
- *menor peso* p_i*:* $20 + 30 + 66 + 40 = 156$.
- *maximice* b_i/p_i*:* $66 + 20 + 30 + 0{,}8 \cdot 60 = 164$.

Se puede demostrar que la selección según beneficio por unidad de peso, b_i/p_i*, da una solución óptima.*

Si los elementos deben ponerse completos en la mochila la situación es ***muy*** *diferente. Eso lo veremos más adelante.*

Ejemplo 3. ***Problema del cambio:*** *Supongamos que queremos dar el vuelto a un cliente usando el mínimo número de monedas posibles, utilizando monedas de 1, 5, 10 y 25 centavos. Por ejemplo, si el monto es* \$0,69*, deberemos entregar 8 monedas: 2 monedas de 25 centavos, una de 10 centavos, una de 5 centavos y 4 de un centavo.*

Un algoritmo goloso para resolver este problema es seleccionar la moneda de mayor valor que no exceda la cantidad restante por devolver, agregar esta moneda a la lista de la solución, y sustraer la cantidad correspondiente a la cantidad que resta por devolver (hasta que sea 0).

```
darCambio(cambio)
    entrada: cambio ∈ ℕ
    salida: M conjunto de enteros

    suma ← 0
    M ← {}
    mientras suma < cambio hacer
        proxima ← masgrande(cambio,suma)
        M ← M ∪ {proxima}
        suma ← suma + proxima
    fin mientras
    retornar M
```

Este algoritmo siempre produce la mejor solución, es decir, retorna la menor cantidad de monedas necesarias para obtener el valor cambio. Sin embargo, si también hay monedas de 12 centavos, puede ocurrir que el algoritmo no encuentre una solución óptima: si queremos devolver 21 centavos, el algoritmo retornará una solución con 6 monedas, una de 12 centavos, 1 de 5 centavos y 4 de 1 centavos, mientras que la solución óptima es retornar 2 monedas de 10 centavos y una de 1 centavo.

El algoritmo es goloso porque en cada paso selecciona la moneda de mayor valor posible, sin preocuparse que ésto puede llevar a una mala solución, y nunca modifica una decisión tomada.

Ejemplo 4. ***Minimización del tiempo de espera en un sistema:*** *Un servidor tiene* n *clientes para atender. Se sabe que el tiempo requerido para atender al cliente* i *es* $t_i \in \mathbb{R}_+$*. El objetivo es determinar en qué orden se deben atender los clientes para minimizar la suma de los tiempos de espera de los clientes.*

Si $I = (i_1, i_2, \ldots, i_n)$ es una permutación de los clientes que representa el orden de atención, entonces la suma de los tiempos de espera es

$$\begin{aligned} T &= t_{i_1} + (t_{i_1} + t_{i_2}) + (t_{i_1} + t_{i_2} + t_{i_3}) + \ldots \\ &= \sum_{k=1}^{n} (n - k + 1) t_{i_k}. \end{aligned}$$

Un algoritmo goloso para resolver este problema, atendería al cliente que requiera el menor tiempo de atención entre todos los que todaviía están en la cola. Retorna una permutación $I_{GOL} = (i_1, \ldots, i_n)$ tal que $t_{i_j} \leq t_{i_{j+1}}$ para $j = 1, \ldots, n-1$.

Este enfoque da un algoritmo correcto.

Ejemplo 5. *Sean $P_1, P_2, \ldots, P_n$ programas que se quieren almacenar en una cinta. El programa P_i requiere s_i Kb de memoria. La cinta tiene capacidad para almacenar todos los programas. Se conoce la frecuencia π_i con que se usa el programa P_i. La densidad de la cinta y la velocidad del drive son constantes. Después que un programa se carga desde la cinta, la misma se rebobina hasta el principio. Si los programas se almacenan en orden $i_1, i_2, \ldots, i_n$ el tiempo promedio de carga de un programa es*

$$T = c \sum_j \left(\pi_{i_j} \sum_{k \leq j} s_{i_k} \right)$$

donde la constante c depende de la densidad de grabado y la velocidad del dispositivo.

Un algoritmo goloso para determinar el orden en que se almacenan los programas que minimice T es el siguiente: para cada programa P_i calcular π_i / s_i y colocar los programas en orden no creciente de este valor.

Otras posibilidades son colocarlos en orden no decreciente de los s_i o en orden no creciente de los π_i, pero estos dos criterios NO resultan en algoritmos correctos.

4. Recursividad

Un algoritmo recursivo se define en términos de sí mismo, esto es, en su cuerpo aparece una aplicación suya.

Supongamos que tenemos una función (matemática) $f : \mathbb{N} \to \mathbb{N}$, que satisface $f(0) = 0$ y $f(n) = 2f(n-1) + n^2$. La declaración de f anterior no tendría sentido si no se incluye el hecho que $f(0) = 0$.

Esto no es una definición circular, porque, si bien definimos una función en términos de sí misma, no definimos un caso particular de la función en términos de sí mismo. La evaluación de $f(5)$ calculando $f(5)$ sería circular. La evaluación de $f(5)$ basándose en $f(4)$ no es circular, a menos que $f(4)$ se evalúe a partir de $f(5)$. El algoritmo inmediato para calcular la función f es el siguiente:

```
f(n)
    entrada: n ∈ ℕ
    salida: f(n)

    si n = 0 entonces
        resu ← 0
    sino
        resu ← 2*f(n − 1) + n²
    fin si
    retornar resu
```

Si f es llamada con valor 3, entonces la asignación del cuerpo del **sino** requiere del cálculo de $2 * f(2) + 3^2$. Entonces se hace una llamada a f para calcular $f(2)$. Nuevamente, se requiere del cálculo de $2 * f(1) + 2^2$. Por lo tanto, se hace otra llamada para calcular $f(1)$. Y como esto requiere evaluar $2 * f(0) + 1^2$, se hace una última llamada para calcular $f(0)$. En esta aplicación de la función f se ejecuta el cuerpo del **si**, asignando a la variable *resu* el valor 0. Luego la aplicación finaliza retornando este valor. Esto permite calcular $2 * f(0) + 1 * 1$ y, siguiendo el orden inverso de las llamadas, terminar calculando $f(3)$.

Ejemplo 6. *Cálculo recursivo del factorial de un número:*

```
factorial(n)
    entrada: n ∈ ℕ
    salida: n!

    si n = 1 entonces
        resu ← 1
    sino
        resu ← n*factorial(n − 1)
    fin si
    retornar resu
```

¿Pero qué sucede con este *supuesto algoritmo* recursivo?

```
sin_fin(n)
    entrada: n ∈ ℕ
    salida: ???

    retornar sin_fin(n − 1)
```

No es un algoritmo, porque ***¡¡¡no termina!!!***

Para que no suceda esto, debe haber por lo menos un valor de los parámetros que se considere ***caso base*** o elemental. Cuando se llega a este caso la función no recurre, sino que calcula su valor de otra forma.

Además, las llamadas recursivas deben aplicarse sobre parámetros ***más pequeños*** que los iniciales. La cualidad de ***pequeñez*** se mide en cada problema de una forma distinta, dependiendo del número de parámetros, del tipo de éstos, del problema concreto. Cada llamada ***se debe acercar má*** a un caso base, que al ser alcanzado finaliza con

la recursión.

En el caso recursivo de la función factorial, el caso base corresponde a $n = 1$ y el caso recursivo a $n > 1$. Como medida podemos tomar el valor de n. Entonces en cada llamada recursiva disminuye en 1 la medida del parámetro. Si se parte de un entero > 1, al irse decrementando en las sucesivas recursiones, acabará alcanzando el valor 1, que es el caso base, y termina el proceso recursivo.

También hay que tener cuidado que las llamadas recursivas se apliquen sobre datos que satisfagan la precondición del algoritmo. Esto, aunque parece obvio, suele ser una fuente frecuente de errores al diseñar algoritmos recursivos.

4.1. Complejidad de algoritmos recursivos

La complejidad de los algoritmos recursivos se puede describir mediante ecuaciones de recurrencia. En la definición de la complejidad para una instancia se utiliza la complejidad para instancias más chicas.

Ejemplo 7. *En el algoritmo del ejemplo 6 del cálculo recursivo del factorial:*

$$T(n) = \begin{cases} 1 & si\ n = 1 \\ T(n-1) + O(resu \leftarrow n * factorial(n-1)) & si\ n > 1 \end{cases}$$

Que es equivalente a:

$$T(n) = 1 + \sum_{k=1}^{n} O(resu \leftarrow k * factorial(k-1))$$

Si consideramos el modelo logarítmico, como vimos en el ejemplo de la clase pasada, $resu \leftarrow k * factorial(k-1))$ *es* $O(k * log_2^2(k))$*, obtenemos:*

$$T(n) \leq 1 + \sum_{k=2}^{n} O(k * log_2^2(k))) \leq 1 + n^2 log_2^2(k)$$

que es $O(t^2 4^t)$*, siendo* $t = log_2(n)$ *el tamaño de la entrada.*

De forma general, para calcular la complejidad de un algoritmo recursivo, debemos plantear primero la ecuación de recurrencia y luego, utilizando herramientas matemáticas, encontrar la fórmula cerrada (que no dependa de la complejidad de instancias más chicas) de esta ecuación.

4.2. Correctitud de algoritmos recursivos

Para demostrar la correctitud de un algoritmo tenemos que demostrar que termina (o sea, que es un algoritmo) y que cumple la especificación. Para demostrar que termina, tenemos que demostrar que los procesos repetitivos terminan cuando se aplican a instancias que cumplen la precondición. Para demostrar la correctitud, debemos asegurar que, si el estado inicial satisface la precondición, entonces el estado final cumplirá la postcondición.

El siguiente *supuesto algoritmo* para calcular el número factorial de un natural sería correcto si el proceso terminara, pero esto no sucede si n es mayor que cero.

```
factorial_malo(n)
    entrada: n ∈ ℕ
    salida: ???

    si n = 1 entonces
        resu ← 1
    sino
        resu ←factorial_malo(n + 1) div (n + 1)
    retornar resu
```

Para demostrar la correctitud de algoritmos recursivos es muy frecuente recurrir al principio de inducción matemática. Veamos en el siguiente ejemplo como aplicarla en el caso de la función cuadrado.

Ejemplo 8. *Cálculo recursivo del cuadrado de un número entero no negativo:*

```
cuadrado(n)
    entrada: n ∈ ℤ≥0
    salida: n²

    si n = 0 entonces
        resu ← 0
    sino
        resu ← 2 * n+cuadrado(n − 1) − 1
    fin si
    retornar resu
```

***Terminación**: Primero veamos que para toda instancia que cumple la precondición el procedimiento termina. La precondición en este caso es $n \geq 0$. Analicemos dos posibilidades:*

- *$n = 0$: Es el caso base de la recursión, no produce una llamada recursiva y por lo tanto el proceso termina.*
- *$n > 0$: En este caso, se produce una llamada recursiva con parámetro $n - 1$. Por lo tanto, en cada recursión el parámetro decrece en 1, y como originalmente es mayor que 0, en una cantidad finita de recursiones el parámetro de la llamada recursiva será 0, finalizando la recursión por ser un caso base de la recursión.*

***Correctitud**: Ahora vamos a demostrar la correctitud del algoritmo. Para esto tenemos que asegurar que para todo entero no negativo n, la aplicación de la función cuadrado retorna n^2. Esto lo haremos por inducción.*

***Caso base:** El caso base de nuestra demostración es cuando n es igual a 0. En este caso el algoritmo es trivialmente correcto, ya que $cuadrado(n) = 0$ y $n^2 = 0$.*

***Paso inductivo:** Para demostrar el paso inductivo, consideremos un entero $n \geq 1$. Nuestra hipótesis inductiva es: **el algoritmo es correcto cuando el parámetro es** $n-1$**: cuadrado**$(n-1) = (n-1)^2$. Por definición del algoritmo, $cuadrado(n) = 2 * n +$ $cuadrado(n - 1) - 1$. Y aplicando la hipótesis inductiva,*

$$cuadrado(n) = 2 * n + (n - 1)^2 - 1 = 2 * n + n^2 - 2 * n + 1 - 1 = n^2.$$

Con esto terminamos la demostración.

Veamos un ejemplo donde usaremos el principio de inducción matemática generalizado para demostrar su correctitud.

Ejemplo 9. *Algoritmo recursivo para multiplicar dos número enteros positivos:*

```
producto(n,m)
    entrada: n, m ∈ ℕ
    salida: n * m

    si m = 1 entonces

        resu ← n
    sino
        resu ←producto(n * 2, m div 2)
        si impar(m) entonces
            resu ← resu + n
        fin si
    fin si
    retornar resu
```

***Terminación**: La precondición es $n \geq 1$ y $m \geq 1$. Nuevamente analicemos dos casos:*

- *$m = 1$: Para cualquier valor de n, este es un caso base de la recursión, no produce llamadas recursivas y por lo tanto el proceso termina.*
- *$m > 1$: En este caso, se produce una llamada recursiva con parámetros $n * 2$ y m **div** 2. La operación m **div** 2 da como resultado un valor estrictamente menor a m y mayor o igual a 1. Entonces, luego de una cantidad finita de recursiones, el segundo parámetro de la llamada recursiva debe valer 1, llegando a un caso base de la recursión y, por lo tanto, finalizando el proceso.*

***Correctitud**: Para demostrar la correctitud del algoritmo aplicaremos inducción sobre m.*

Caso base: El caso base de la demostración es cuando $m = 1$. Se ejecuta el cuerpo de la sentencia **si**, *asignando el valor n a la variable resultado y el algoritmo termina retornando el valor de la variable resu. Por lo tanto, $producto(n, 1) = n$ que es el resultado correcto.*

*Paso inductivo: Consideremos cualquier $m \geq 2$ y cualquier entero positivo n. Nuestra hipótesis inductiva es: **$producto(s, t) = s * t$ para cualquier entero positivo s y cualquier entero positivo $t < m$.***

Vamos a considerar dos casos:

- *m par: El algoritmo retorna $producto(n * 2, m$ **div** $2)$, porque le asigna este valor a la variable resu y luego termina retornando este valor. Esto es $producto(n, m)$=$producto(n * 2, m$ **div** $2)$. Aplicando la hipótesis inductiva, sabemos que $producto(n * 2, m$ **div** $2) = (n * 2) * (m$ **div** $2)$. Como m **div** 2 es entero (porque estamos en el caso de m par), $(n * 2) * (m$ **div** $2) = n * m$, obtenemos:*

$$producto(n, m) = producto(n * 2, m \textbf{ div } 2) = \\ = (n * 2) * (m \textbf{ div } 2) = n * m$$

- *m impar: El algoritmo retorna $producto(n * 2, m$ **div** $2) + n$, porque le asigna $producto(n * 2, m$ **div** $2)$ a la variable resu y luego le suma n a ese valor. Como m es impar, $m = 2 * (m$ **div** $2) + 1$. Igual que antes, aplicando la hipótesis inductiva, sabemos que $producto(n * 2, m$ **div** $2) = (n * 2) * (m$ **div** $2)$. Juntando esto, obtenemos:*

$$producto(n, m) = producto(n * 2, m \textbf{ div } 2) + n =$$
$$= (n * 2) * (m \textbf{ div } 2) + n = n * (2 * (m \textbf{ div } 2) + 1) = n * m$$

5. Dividir y conquistar *(divide and conquer)*

Dividir y conquistar es una de las técnicas más utilizadas. Se basa en la descomposición de un problema en subproblemas. Si un problema es demasiado difícil para resolverlo directamente, una alternativa es dividirlo en partes más pequeñas que sean más fáciles de resolver y luego uniendo todas las soluciones parciales obtener la solución final. Es un caso particular de los algoritmos recursivos.

Formalmente, dado un problema a resolver para una entrada de tamaño n, se divide la entrada en r subproblemas. Estos subproblemas se resuelven de forma independiente y después se combinan sus soluciones parciales para obtener la solución del problema original. En esta técnica, los subproblemas deben ser de la misma clase que el problema original, permitiendo una resolución recursiva. Por supuesto, deben existir algunos casos sencillos, cuya solución pueda calcularse directamente. Se distinguen 3 partes en un algoritmo dividir y conquistar:

- Una forma directa de resolver casos sencillos.
- Una forma de dividir el problema en 2 o más subproblemas de menor tamaño.
- Una forma de combinar las soluciones parciales para llegar a la solución completa.

El esquema general es:

```
d&c(x)
    entrada: x
    salida: y

    si x es suficientemente fácil entonces
        y ← calcular directamente
    sino
        descomponer x en instancias más chicas x_1, ..., x_r
        para i = 1 hasta r hacer
            y_i ← d&c(x_i)
        fin para
        y ← combinar los y_i
    fin si
    retornar y
```

La cantidad de subproblemas, r, generalmente es chica e independientemente de la instancia particular que se resuelve. Para obtener un algoritmo eficiente, la ***medida de los subproblemas debe ser similar*** y no necesitar resolver más de una vez el mismo subproblema.

5.1. Complejidad de algoritmos dividir y conquistar

Estos algoritmos son algoritmos recursivos, por lo que vamos a seguir el mismo razonamiento para el cálculo de su complejidad. Generalmente las instancias que son caso base toman tiempo constante c (es O(1)) y para los casos recursivos podemos identificar tres puntos críticos:

- cantidad de llamadas, que llamaremos r
- medida de cada subproblema, n/b para alguna constante b
- tiempo requerido por D&C para ejecutar descomponer y combinar para una instancia de tamaño n, $g(n)$

Entonces, tiempo total $T(n)$ consumido por el algoritmo está definido por la siguiente ecuación de recurrencia:

$$T(n) = \begin{cases} c & \text{si } n \text{ es caso base} \\ rT(n/b) + g(n) & \text{si } n \text{ es caso recursivo} \end{cases}$$

Ahora debemos resolver esta ecuación de recurrencia. Para esta tarea, vamos a presentar una versión simplificada del teorema maestro:

Si existe un entero k tal que $g(n)$ es $O(n^k)$ se puede demostrar que:

$$t(n) \text{ es } \begin{cases} O(n^k) & \text{si } r < b^k \\ O(n^k \log n) & \text{si } r = b^k \\ O(n^{\log_b r}) & \text{si } r > b^k \end{cases}$$

Veamos ahora cómo utilizarlo.

Ejemplo 10. *Búsqueda binaria: El problema consiste en decidir si un elemento x se encuentra en un arreglo de n elementos A, ordenado de forma no decreciente ($A[i] \leq A[j]$ si $1 \leq i < j \leq n$).*

Una forma de hacer ésto es recorrer secuencialmente el arreglo A, hasta encontrar x, o llegar al final del arreglo, o encontrar un elemento mayor a x. Este algoritmo es $O(n)$ en el peor caso.

Una alternativa es primero decidir si x debería estar en la primera mitad del arreglo o en la segunda. Para ésto, basta comparar x con el elemento de la posición media del arreglo. Si x es menor a ese elemento, la búsqueda debe continuar en la primera mitad del arreglo y si es mayor en la segunda. Este algoritmo, que es el método natural cuando por ejemplo se busca una palabra en un diccionario, utiliza la técnica dividir y conquistar.

```
busquedabinaria(A,x)
    entrada: arreglo A, valor x
    salida: Verdadero si x está en A, Falso caso contrario

    si dim(A) = 1 entonces
        retornar x = A[0]
    sino
        k ← dim(A) div 2
        si x ≤ A[k − 1] entonces
            retornar busquedabinaria (A[0 . . . k − 1],x)
        sino
            retornar busquedabinaria (A[k . . . dim(A) − 1],x)
        fin si
    fin si
```

Cuando $n > 1$, el tiempo total consumido por el algoritmo es:

$$T(n) = T(n/2) + g(n)$$

donde $g(n)$ es $O(1) = O(n^0)$. Como $r = 1$, $b = 2$, $k = 0$, tenemos que $r = b^k$, aplicando el teorema maestro, obtenemos que:

$$T(n) \text{ es } O(n^0 \log n) = O(\log n)$$

Ejemplo 11. *Ordenamiento: Dado un arreglo A de n elementos, el problema consiste en ordenar los elementos de A en orden creciente. Una alternativa es utilizar el algoritmo de selección visto la clase pasada.*

La forma directa de hacerlo utilizando la técnica de dividir y conquistar es dividir el arreglo en dos partes de medida lo más parecida posible, ordenar cada uno de estos subarreglos con llamadas recursivas, y luego unir apropiadamente las soluciones de cada una conservando el orden. El caso trivial es un arreglo de longitud 1, porque ya está ordenado, pero también podríamos utilizar un algoritmo relativamente simple cuando la cantidad de elementos del arreglo es suficientemente pequeña sin necesidad de ser 1. Este algoritmo se llama mergesort.
La forma directa de hacerlo utilizando la técnica de dividir y conquistar es dividir el arreglo en dos partes de medida lo más parecida posible, ordenar cada uno de estos subarreglos con llamadas recursivas, y luego unir apropiadamente las soluciones de cada una conservando el orden. El caso trivial es un arreglo de longitud 1, porque ya está ordenado, pero también podríamos utilizar un algoritmo relativamente simple cuando la cantidad de elementos del arreglo es suficientemente pequeña sin necesidad de ser 1. Este algoritmo se llama mergesort.

```
mergesort(A)
    entrada: arreglo A de dimensión n
    salida: A ordenado de forma creciente

    si n > 1 entonces
        mergesort(A[0 ... (n div 2 − 1)])
        mergesort(A[(n div 2) ... (n − 1)])
        unir(A[0 ... (n div 2 − 1)], A[(n div 2) ... (n − 1)])
    fin si
```

En el algoritmo anterior, la descomposición de la entrada en dos subproblemas es sumamente fácil, mientras que la unión de los resultados de estos subproblemas es la parte difícil del algoritmo. Un posible algoritmo para realizar la unión es:

```
unir(U,V)
    entrada: arreglos U y V ordenados
    salida: arreglo T ordenado con todos los elementos de U y V

    i ← 0
    j ← 0
    U[dim(U)] ← ∞
    V[dim(V)] ← ∞
    para k = 0 hasta dim(U) + dim(V) − 1 hacer
        si U[i] < V[j] entonces
            T[k] ← U[i]
            i ← i + 1
        sino
            T[k] ← V[j]
            j ← j + 1
        fin si
    fin para
    retornar T
```

Es fácil ver que `unir` *es* $O(dim(U) + dim(V)) = O(dim(A))$.

Si $n = dim(A)$*, cuando* $n > 1$*, el tiempo total consumido por mergesort es:*

$$T(n) = 2T(n/2) + g(n)$$

donde $g(n)$ *es* $O(n)$*. Como en este caso* $r = 2$*,* $b = 2$ *y* $k = 1$*, entonces* $r = b^k$ *y* $T(n)$ *es* $O(n \log n)$*.*

Ejemplo 12. *Exponenciación: Queremos calcular* a^n *para un* n *grande.*

Obviamente, el algoritmo más simple es realizar $n - 1$ *multiplicaciones, calculando* $a \times a \times \ldots \times a$*. Si consideramos que* $n = \lfloor n/2 \rfloor + \lceil n/2 \rceil$*, entonces* $a^n = (a^{n/2})^2$ *si* n *es par y* $a^n = a(a^{\lfloor n/2 \rfloor})^2$ *si* n *es impar. De esta forma, el exponente se redujo a la mitad agregando a lo sumo dos multiplicaciones, resultando en el siguiente algoritmo.*

```
potencia(a,n)
    entrada: a, n ∈ Z≥0
    salida: a * n

    si n = 0 entonces
        x ← 1
    sino
        x ← potencia(a,ndiv2)
        si par(n) entonces
            x ← x × x
        sino
            x ← a × x × x
        fin si
    fin si
    retornar x
```

Cuando n no es una potencia de dos, el problema no puede ser siempre dividido en subproblemas totalmente uniformes, pero la diferencia de uno entre los dos no debería causar un gran desbalance.

6. Backtracking

Esta técnica recorre sistemáticamente todas las posibles configuraciones del espacio de soluciones de un problema buscando aquellas que cumplen las propiedades deseadas. A estas configuraciones las llamaremos ***soluciones válidas***. Puede utilizarse para resolver problemas de factibilidad (donde se quiere encontrar cualquier solución válida o todas ellas) o problemas optimización (donde de todas las configuraciones válidas se quiere la ***mejor***).

Ya vimos el método de fuerza bruta, pero, excepto para instancias muy pequeñas, su costo computacional es prohibitivo. ***Backtracking*** es una modificación de esa técnica que aprovecha propiedades del problema para evitar analizar todas las configuraciones. Obviamente, para que el algoritmo sea correcto debemos estar seguros de no dejar de examinar configuraciones que estamos buscando.

La idea básica es tratar de extender una solución parcial del problema hasta, eventualmete, llegar a obtener una solución completa, que podría ser válida o no. Habitualmente, se utiliza un ***vector*** $a = (a_1, a_2, \ldots, a_n)$ para representar una solución candidata, cada a_i pertenece a un dominio/conjunto finito A_i. El espacio de soluciones es el producto cartesiano $A_1 \times \ldots \times A_n$.

El procedimiento construye este vector elemento a elemento: primero considera un posible valor para a_1; después, uno para a_2, y así sucesivamente. Cuando una solución parcial no puede ser extendida, se retrocede (se eliminan valores de los a_i en orden inverso). Este retroceso se detiene al encontrar una configuración que permita avanzar de forma diferente. Este retroceso es lo que le da el nombre al método: backtracking.

En síntesis, en cada paso se extienden las soluciones parciales $a = (a_1, a_2, \ldots, a_k)$, $k < n$, agregando un elemento más, $a_{k+1} \in S_{k+1} \subseteq A_{k+1}$, al final del vector a. Las nuevas soluciones parciales son sucesoras de la anterior. Si S_{k+1} (conjunto de soluciones sucesoras) es vacío, esa rama no se continua explorando.

Se puede pensar este espacio de búsqueda como un árbol dirigido donde cada vértice representa una solución parcial y un vértice x es hijo de y si la solución parcial x se puede extender desde la solución parcial y. La raíz del árbol se corresponde con el vector vacío (la solución parcial vacía). Los vértices del primer nivel del árbol serán las soluciones parciales que ya tienen defenidas el primer elemento. Los de segundo nivel las que tiene los dos primeros y así siguiendo.

Si el vértice x corresponde a la solución parcial $a = (a_1, a_2, \ldots, a_k)$, por cada valor posible que puede tomar a_{k+1} se ramifica el árbol, generando tantos hijos de x como posibilidades haya para a_{k+1}. Las soluciones completas (cuandos todos los a_i tienen valor) corresponden a las hojas del árbol.

El proceso de backtracking recorre este árbol en profundidad. Cuando podemos deducir que una solución parcial no nos llevará a una solución válida, no es necesario seguir explorando esa rama del árbol de búsqueda (se ***poda*** el árbol) y se retrocede hasta encontrar un vértice con un hijo válido por donde seguir la exploración. Esta poda puede ser por:

- Factibilidad: ninguna extensión de la solución parcial derivará en una solución válida del problema.
- Optimalidad (en problemas de optimización): ninguna extensión de la solución parcial derivará en una solución del problema óptima.

De esta poda depende el éxito del método aplicado a un problema. Para poder aplicar la poda por factibilidad, la representación de las soluciones debe cumplir que, si una solución parcial $a = (a_1, a_2, \ldots, a_k)$ no cumple las propiedades deseadas, tampoco lo hará cualquier extensión posible de ella (propiedad dominó).

Podemos estar interesados en encontrar todas las soluciones válidas de nuestro problema, o nos puede alcanzar con sólo encontrar una. Los esquemas generales son los siguientes:

Backtracking: Esquema General - Todas las soluciones

```
BT(a,k)
entrada: a = (a_1,...,a_k) solución parcial
salida: se procesan todas las soluciones válidas

si k == n+1 entonces
        procesar(a)
        retornar
sino
        para cada a' ∈ Sucesores(a,k)
                BT(a',k+1)
        fin para
fin si
retornar
```

Backtracking: Esquema General - Una solución

```
BT(a,k)
        entrada: a = (a_1,...,a_k) solución parcial
        salida: sol = (a_1,...,a_k,...,a_n) solución válida

        si k == n+1 entonces
                sol ← a
                encontro ← true
        sino
                para cada a' ∈ Sucesores(a,k)
                        BT(a',k+1)
                        si encontro entonces
                                retornar
                        fin si
                fin para
        fin si
        retornar
```

Donde *sol* variable global que guarda la solución y *encontro* variable booleana global que indica si ya se encontró una solución (inicialmente está en **false**).

Para demostrar la correctitud de un algoritmo de backtracking, debemos demostrar que se enumeran todas las posibles configuraciones válidas. Es decir, que las ramificaciones y podas son correctas.

Para el cálculo de la complejidad debemos acotar la cantidad de vértices que tendrá el árbol y considerar el costo

de procesar cada uno.

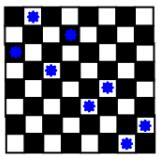
Ejemplo 13. *Problema de las 8 reinas: Volvamos a este problema, que es el ejemplo estándar para introducir esta técnica. Recordemos que el objetivo es ubicar 8 reinas en el tablero de ajedrez (8×8) sin que ninguna "amenace" a otra. Una reina amenaza a otra si ambas se encuentran en la misma fila, en la misma columna o en la misma diagonal.*

Veamos cuántas posibles configuraciones tenemos. Hay que elegir las 8 casillas donde ubicaremos nuestras reinas de entre las 64 casillas que tiene el tablero. Esto es:

$$\binom{64}{8} = 442616536$$

Un algoritmo de fuerza bruta, generaría todas estas posibilidades y luego analizaría cada una para ver si cumple las restricciones del problema.

Pero sabemos que, en las soluciones que nos interesan, cada fila debe tener exactamente una reina. Entonces, una solución puede estar representada por $(a_1, \ldots, a_8)$, con $a_i \in \{1, \ldots, 8\}$ indicando la columna de la reina que está en la fila i. Una solución parcial puede estar representada por $(a_1, \ldots, a_k)$, $k \leq 8$.

La configuración  *está representada por* $(2, 4, 1, 3, 6, 5, 8, 7)$.

Tenemos ahora $8^8 = 16777216$ *combinaciones.*

Podemos mejorar este cálculo considerando que una columna debe tener exactamente una reina. Es decir, todos los a_i, para $i = 1, \ldots, 8$, deben ser distintos. Ahora redujimos a $8! = 40320$ *combinaciones. No todas estas configuraciones serán solución de nuestro problema, ya que falta verificar que no haya reinas que se amenacen por estar en la misma diagonal.*

Cumplimos la propiedad dominó: Si una solución parcial $(a_1, \ldots, a_k)$, $k \leq 8$, tiene reinas que se amenazan, entonces toda extensón de ella seguro tendrá reinas que se amenazan. Por lo tanto es correcto podar una solución parcial que tiene reinas que se amenzan.

Dada una solución parcial $(a_1, \ldots, a_k)$, $k \leq 8$ (sin reinas que se amenacen), construiremos el conjunto de sus vértices hijos, `Sucesores`(a, k)*, asegurando que siga sin haber reinas que se amenacen (en lugar de construir todos y luego podar). Entonces la nueva reina, la $(k+1)$-ésima (correspondiente a la fila $k+1$), no podrá estar ubicada en ninguna de las columnas ni en ninguna de las diagonales donde están ubicadas las k anteriores.*

$$\texttt{Sucesores}(a, k) = \{(a, a_k) : a_k \in \{1, \ldots, 8\}, |a_k - a_j| \notin \{0, k-j\} \forall\, j \in \{1, \ldots, k-1\}\}.$$

Que no haya dos reinas en la misma fila, está implícito en la representación, en el número de coordenada. Esto es, la coordenada 1 de a, o sea a_1, va a tener la columna de la reina que está en la primera fila. La coordenada 2 de a, o sea a_2, va a tener la columna de la reina que está en la segunda fila. De forma general, la coordenada k de a, o sea a_k, va a tener la columna de la reina que está en la k-ésima fila. Esto implícitamente está prohibiendo que haya dos reinas en la misma fila (porque ya la representación hace que no se pueda).

Que no haya dos reinas en la misma columna, lo estamos exigiendo con $a_k - a_j \notin \{0\}$, es decir $a_k \neq a_j$, para $j = 1, \ldots, k-1$.

Que no haya dos reinas en la misma diagonal, lo estamos asegurando al pedir que $|a_k - a_j| \notin \{k - j\}$*, es decir* $|a_k - a_j| \neq k - j$*, para* $j = 1, \ldots, k-1$*. Esto sería que la diferencia entre las filas de dos reinas* $(k - j)$*, sea diferente que la diferencia entre sus columnas* $(|a_k - a_j|)$*. Notar que, que se cumpla* $k - j = |a_k - a_j|$ *es lo mismo que decir que las reinas de las posiciones* (j, a_j) *y* (k, a_k) *están en la misma diagonal.*

Ahora ya estamos en condición de implementar un algoritmo para resolver el problema.

Ejemplo 14. ***Suma de subconjuntos:*** *Dado un conjunto de naturales* $C = \{c_1, \ldots, c_n\}$ *(sin valores repetidos) y* $k \in \mathbb{N}$*, queremos encontrar un subconjunto de* C *(o todos los subconjuntos) cuyos elementos sumen* k*.*

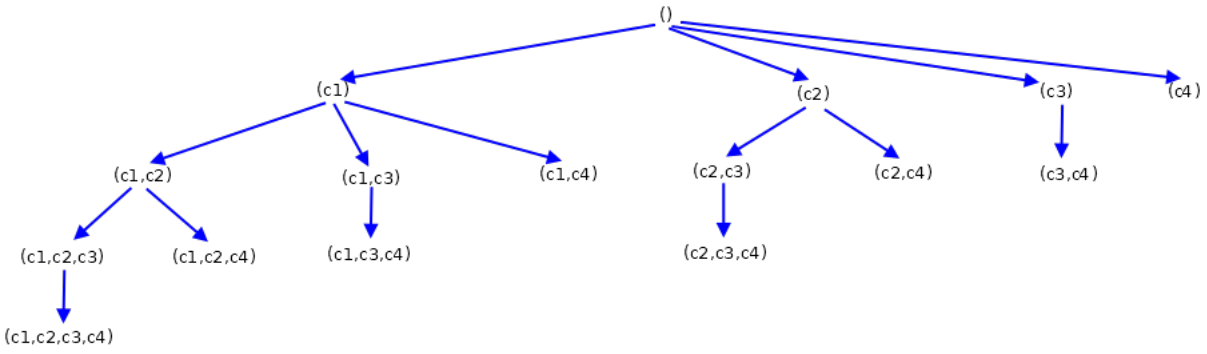
Si resolvemos este problema mediante un algoritmo de fuerza bruta, generaríamos los 2^n *posibles subconjuntos y luego verificaríamos si alguno suma* k*.*

Una posible representación de una solución es $a = (a_1, \ldots, a_n)$*, con* $a_i \in \{V, F\}$ *indicando para cada elemento del conjunto* C *si pertenece o no a la solución.*

Una representación alternativa, que parece más prometedora, es $a = (a_1, \ldots, a_r)$*, con* $r \leq n$*,* $a_i \in C$ *y* $a_i \neq a_j$ *para* $0 \leq i, j \leq r$*. Es decir, el vector a contiene los elementos del subconjunto de* C *que representa. En este caso, todos los vértices de árbol corresponden a posibles soluciones (no necesariamente válidas), no sólo las hojas. Hay que adaptar a esto el esquema general que presentamos previamente.*

Como las soluciones son conjuntos, no nos interesa el orden en que los elementos fueron agregados a la solución. Entonces (c_4, c_5, c_2) *y* (c_2, c_4, c_5) *representarían la misma solución, el subconjunto* $\{c_2, c_4, c_5\}$*. Y cualquier permutación de* (c_4, c_5, c_2) *también. Obviamente queremos evitar esto, no queremos generar más de un vector que represente la misma solución (ni parcial ni total), porque estaríamos agrandando aún más el árbol de búsqueda. Veamos cómo evitar esto.*

Los vértices de nivel 1 del árbol de búsqueda se corresponderán a las n *soluciones de subconjuntos de un elemento,* (c_i)*, para* $i = 1, \ldots, n$*. En el segundo nivel, no quisiera generar los vectores* (c_i, c_j) *y* (c_j, c_i)*, sino sólo uno de ellos porque representan la misma solución. Para esto, para la solución parcial* (c_i) *podemos sólo crear los hijos* (c_i, c_j) *con* $j > i$*. Así evitaríamos una de las dos posibilidades. De forma general, cuando extendemos una solución, podemos pedir que el nuevo elemento tenga índice mayor que el último elemento de* a*. Por ejemplo, si* $n = 4$*, el árbol de búsqueda es:*



En el subárbol encabezado por (c_1) *estarán todas las permutaciones que incluyan a* c_1*. En el encabezado por* (c_2)*, todas las que incluyan a* c_2 *pero no a* c_1*. En el encabezado por* (c_2, c_4)*, todas las permutaciones que contengan a* c_2 *y* c_4 *pero no a* c_1 *ni a* c_3*.*

Para cada vértice del árbol podemos ir guardando la suma de los elementos que están en la solución. Si encontramos un vértice que sume k, la solución correspondiente será una solución válida. Si la suma excede k, esa rama se puede podar, ya que los elementos de C son positivos.

Entonces podemos definir los sucesores de una solución como:

$$\texttt{Sucesores}(a, k) = \{(a, a_k) : a_k \in \{c_{s+1}, \ldots, c_n\},\ si\ a_{k-1} = c_s\ y\ \sum_{i=1}^{k} a_i \leq k\}.$$

*Cumplimos la propiedad dominó: Si una solución parcial $(a_1, \ldots, a_r)$ suma más que k, entonces toda extensón de ella seguro también sumará más que k. Es decir, las soluciones no se **arreglan** al extenderlas. Esto no sería cierto si hay elementos negativos en C.*

Y podemos mejorar esto. Si ordenamos los elementos de C en orden creciente y al extender una solución nos encontramos que cuando a_k toma un posible valor c_j ($c_j \in \{s+1, \ldots, n\}$) sucede que $\sum_{i=1}^{k} a_i \geq k$, podemos estar seguros que para ningún $j' > j$ obtendremos una solución válida. Por lo cual no es necesario verificar esto para esos valores.

7. Programación Dinámica

Esta técnica fue introducida en 1953 por Richard Bellman. Es aplicada típicamente a problemas de optimización combinatoria, donde puede haber muchas soluciones factibles, cada una con un valor (o costo) asociado y prentendemos obtener la solución con mejor valor (o menor costo). Pero además también resulta adecuada para algunos problemas de naturaleza recursiva, como el cálculo de los números combinatorios.

Programación dinámica, al igual que dividir y conquistar, divide al problema en subproblemas de tamaños menores, que son más fáciles de resolver. Una vez resueltos estos subproblemas, se combinan las soluciones obtenidas para generar la solución del problema original. D&C resulta eficiente cuando los subproblemas son de de igual tamaño o parecido y, además, no se es necesario resolver más de una vez el mismo subproblema.

Por el contrario, PD es adecuada para problemas que tienen estas característas que le molestan a D&C, permitiendo reducir la complejidad computacional propia de una resolución puramente recursiva. Esta técnica evita repetir llamadas recursivas almacenando los resultados que ya han sido calculados para su posterior reutilización. La gran diferencia es que programación dinámica es ***bottom up*** y no es recursivo, sino iterativo.

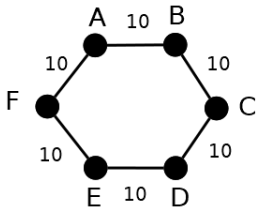
No todo problema se puede resolver mediante programación dinámica. Para que esta técnica sea aplicable, el problema debe cumplir el Principio de optimalidad de Bellman.

> ***Principio de optimalidad de Bellman:*** Un problema de optimización satisface este principio si en una solución óptima cada subsolución es a su vez óptima del subproblema correspondiente. Es decir, dada una secuencia óptima de decisiones, toda subsecuencia de ella es, a su vez, óptima.

Si miramos una subsolución de la solución óptima, debe ser solución del subproblema asociado a esa subsolución. El principio de optimalidad es condición necesaria para poder usar la técnica de programación dinámica.

Por ejemplo, el problema de camino mínimo (sin distancias negativas) cumple este principio. Si sabemos que el camino más corto desde la ciudad A a la B pasa por la C, para transitar ese camino más corto desde A a B, vamos a tener que transitar el camino más corto desde A a C y el más corto desde C a B. Entonces la solución óptima se construye *anexando* soluciones óptimas. Esto es que cumpla el principio de Bellman.

Pensemos ahora en el problema de camino máximo. En la siguiente figura tenemos las ciudades A, B, C, D, E y F. Las únicas rutas que las unen son las que se muestran en el dibujo, cada tramo con distancia 10. Entonces, el camino más largo entre la ciudad A y la B, es hacer A $\rightarrow$ F $\rightarrow$ E $\rightarrow$ D $\rightarrow$ C $\rightarrow$ B, con una distancia de 50. Este camino más largo pasa, por ejemplo, por la ciudad E. Pero, sin embargo, el camino más largo de A a B no es la concatenación del camino más largo de A a E y del de E a B. Es decir, si de la solución óptima me quedo con la subsolución (un pedacito de solución) que es el camino entre A y E, es mentira que es solución óptima del problema aplicado entre A y E (camino más largo entre A y E). Entonces este problema no cumple el principio de optimalidad de Bellman.



Que se cumpla este principio en un problema es lo que garantiza que podemos construir la solución óptima de una instancia, *juntando* de alguna manera, soluciones óptimas de instancias más chicas.

Ejemplo 15. ***Multiplicación de n matrices:*** *Dadas n matrices, $M_1, M_2, \ldots, M_n$, queremos calcular*

$$M = M_1 \times M_2 \times \ldots M_n$$

Por la propiedad asociativa del producto de matrices, ésto puede hacerse de muchas formas. De todas estas posibles formas, queremos determinar la que minimiza el número de multiplicaciones necesarias. Este es un problema de optimización combinatoria, de todas las posibilidades queremos la mejor. Por ejemplo, si la dimensión de A es 13×5, la de B es 5×89, la de C es 89×3 y la de D es 3×34. Tenemos que:

- $((AB)C)D$ *requiere* 10582 *multiplicaciones.*
- $(AB)(CD)$ *requiere* 54201 *multiplicaciones.*
- $(A(BC))D$ *requiere* 2856 *multiplicaciones.*
- $A((BC)D)$ *requiere* 4055 *multiplicaciones.*
- $A(B(CD))$ *requiere* 26418 *multiplicaciones.*

Veamos que se cumple el principio de optimalidad de Bellman en este problema. Para multiplicar todas las matrices de forma óptima, deberemos multiplicar las matrices 1 a i por un lado y las matrices $i+1$ a n por otro lado y luego multiplicar estos dos resultados, para algún $1 \leq i \leq n-1$, que es justamente lo que queremos determinar. En la solución óptima de $M = M_1 \times M_2 \times \ldots M_n$, estos dos subproblemas, $M_1 \times M_2 \times \ldots M_i$ y $M_{i+1} \times M_{i+2} \times \ldots M_n$ deben estar resueltos, a su vez, de forma óptima, es decir realizando la mínima cantidad de operaciones. Entonces se cumple el principio de optimalidad.

Llamaremos $m[i][j]$ a la cantidad mínima de multiplicaciones necesarias para calcular $M_i \times M_{i+1} \times \ldots M_j$. Por simplicidad vamos a calcular la cantidad de multiplicaciones mínima y no la forma de hacerlo en sí. Es sencillo modificar el algoritmo resultante para calcular la forma de hacer la multiplicación.

Por comodidad, supongamos que las dimensiones de las matrices están dadas por un vector $d \in N^{n+1}$, tal que la matriz M_i tiene $d[i-1]$ filas y $d[i]$ columnas para $1 \leq i \leq n$, entonces:

- *Para $i = 1, \ldots, n$, $m[i][i] = 0$*

- *Para* $i = 1, \ldots, n-1$, $m[i][i+1] = d[i-1]d[i]d[i+1]$
- *Para* $s = 2, \ldots, n-1$, $i = 1, \ldots, n-s$,

$$m[i][i+s] = \min_{i \leq k < i+s} (m[i][k] + m[k+1][i+s] + d[i-1]d[k]d[i+s])$$

Y la solución del problema será $m[1][n]$.

Analicemos si la forma de ir calculando los valores de m *es correcta. Esto es, que cuando quiera calcular un valor ya tenga calculado el valor para los subproblemas que necesito.*

Los dos primeros ítems de la fórmula, no son recursivos, así que los podemos llenar desde el comienzo.

Para el último ítem, cuando querramos calcular $m[i][i+s]$*, vamos a necesitar ya tener calculados* $m[i][k]$ *y* $m[k+1][s]$ *para todo* $k = i, \ldots, i+s-1$*. Entonces, con lo que ya calculamos desde el comienzo, podemos calcular* $m[i][i+2]$ *(es decir* $s = 2$*) para todos* $i = 1, 2, \ldots, n-2$ *(porque el único valor que puede tomar* k *es* $i+1$ *y ya tenemos a* $m[i][i+1]$ *y* $m[i+2][i+2]$*).*

Ahora, ya teniendo estos valores, podemos calcular $m[i][i+3]$ *(es decir* $s = 3$*) para todos* $i = 1, 2, \ldots, n-3$ *(porque los únicos valores que puede tomar* k *son* $i+1$ *y* $i+2$ *y ya tenemos a* $m[i][i+1]$*,* $m[i+2][i+3]$*,* $m[i][i+2]$ *y* $m[i+3][i+3]$*).*

Y así siguiendo. Entonces, para cada $s = 2, \ldots, n-1$ *recorremos todos los* $i = 1, 2, \ldots, n-s$*.*

Esto ya es el algoritmo:

Multiplicación de n matrices

```
mult(d)
  entrada: d ∈ N^n  dimensión de las matrices
  salida: cantidad mínima de multiplicaciones

  para i = 1 hasta n hacer
      m[i][i] ← 0
  fin para
  para i = 1 hasta n − 1 hacer
      m[i][i + 1] ← d[i − 1] ∗ d[i] ∗ d[i + 1]
  fin para
  para s = 2 hasta n − 1 hacer
      para i = 1 hasta n − s hacer
          min ← ∞
          para k = i hasta i + s − 1 hacer
              si m[i][k] + m[k + 1][i + s] + d[i − 1] ∗ d[k]d[i + s] < min hacer
                  min ← m[i][k] + m[k + 1][i + s] + d[i − 1] ∗ d[k] ∗ d[i + s]
              fin si
          fin para
          m[i][i + s] ← min
      fin para
  fin para
  retornar m[1][n]
```

Este algoritmo es $O(n^3)$ y requiere $O(n^2)$ espacio para almacenar m.

Veamos otro ejemplo de aplicación de PD.

Ejemplo 16. *Subsecuencia común más larga: Dada una secuencia, una subsecuencia de ella se obtiene eliminando 0 o más símbolos (sin modificar el orden de los símbolos que quedan). Por ejemplo, $[4, 7, 2, 3]$ y $[7, 5]$ son subsecuencias de $[4, 7, 8, 2, 5, 3]$, $[2, 7]$ no lo es.*

En el problema de la subsecuencia común más larga (scml) el objetivo es encontrar la subsecuencia común mas larga de dos secuencias dadas. Es decir, dadas dos secuencias A y B, queremos encontrar entre todas las secuencias que son tanto subsecuencia de A como de B la de mayor longitud.

Por ejemplo, si $A = [9, 5, 2, 8, 7, 3, 1, 6, 4]$ y $B = [2, 9, 3, 5, 8, 7, 4, 1, 6]$ las scml es $[9, 5, 8, 7, 1, 6]$.

Si resolvemos este problema por fuerza bruta, listaríamos todas las subsecuencias de S_1, todas las de S_2, nos fijaríamos cuales tienen en común, y entre esas elegiríamos la más larga.

Por simplicidad en la escritura, vamos a plantear el problema donde queremos buscar la longitud de la scml y no la subsecuencia. Fácilmente se podría adaptar el algoritmo que desarrollaremos al problema original.

Dadas las dos secuencias $A = [a_1, \ldots, a_r]$ y $B = [b_1, \ldots, b_s]$, existen dos posibilidades, $a_r = b_s$ o $a_r \neq b_s$. Analicemos cada caso:

1. *$a_r = b_s$: La scml entre A y B se obitene colocando al final de la scml entre $[a_1., \ldots, a_{r-1}]$ y $[b_1., \ldots, b_{s-1}]$ al elemento a a_r (o b_s porque son iguales).*
2. *$a_r \neq b_s$: La scml entre A y B será la más larga entre las scml entre $[a_1., \ldots, a_{r-1}]$ y la scml entre $[b_1., \ldots, b_s]$ y $[a_1., \ldots, a_r]$ y $[b_1., \ldots, b_{s-1}]$. Esto es, calculamos el problema aplicado a $[a_1., \ldots, a_{r-1}]$ y $[b_1., \ldots, b_s]$ y, por otro lado, el problema aplicado a $[a_1., \ldots, a_r]$ y $[b_1., \ldots, b_{s-1}]$, y nos quedamos con la más larga de ambas.*

Nuevamente, esta forma recursiva de resolver el problema ya nos conduce al algoritmo. Si llamamos $l[i][j]$ a la longitud de la scml entre $[a_1., \ldots, a_i]$ y $[b_1., \ldots, b_j]$, entonces:

- $l[0][0] = 0$
- *Para $j = 1, \ldots, s$, $l[0][j] = 0$*
- *Para $i = 1, 2, \ldots, r$, $l[i][0] = 0$*
- *Para $i = 1, \ldots, r$, $j = 1, \ldots, s$*

  si $a_i = b_j$**:** $l[i][j] = l[i-1][j-1] + 1$

  si $a_i \neq b_j$**:** $l[i][j] = \max\{l[i-1][j], l[i][j-1]\}$

Y la solución del problema será $l[r][s]$.

Veamos ahora como ir llenando los valores de l. Los casos no recursivos son nuestra base y los podemos calcular desde el comienzo.

Luego, con esto ya podemos calcular $l[1][1]$, porque solo necesita los casos base. Ahora ya estamos en condiciones de calcular $l[1][2]$ y $l[2][1]$. Siguiendo, podemos calcular $l[1][3]$, $l[2][2]$ y $l[3][1]$.
De esta forma, obtenemos el siguiente algoritmo:

Subsecuencia común más larga

```
scml(A,B)
  entrada: A, B secuencias
  salida: longitud de a scml entre A y B

  l[0][0] ← 0
  para i = 1 hasta r hacer
      l[i][0] ← 0
  fin para
  para j = 1 hasta s hacer
      A[0][j] ← 0
  fin para
  para i = 1 hasta r hacer
      para j = 1 hasta s hacer
          l[i][j] ← máx{l[i − 1][j], l[i][j − 1]}
      fin para
  fin para
  retornar l[r][s]
```

Este algoritmo es $O(n^2)$ y requiere $O(n^2)$ espacio para almacenar m.

En el próximo ejemplo, aplicaremos PD a un problema que no es de optimización combinatoria.

Ejemplo 17. ***Coeficientes binomiales:*** *Si $n \geq 0$ y $0 \leq k \leq n$, se define*

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Para calcular $\binom{n}{k}$, usaremos la siguiente propiedad:

$$\binom{n}{k} = \begin{cases} 1 & \text{si } k = 0 \text{ o } k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{si } 0 < k < n \\ 0 & \text{caso contrario} \end{cases}$$

y armaremos una tabla con los números combinarios para valores más pequeños de n y k. Por la propiedad anterior, podemos inicializar la tabla con 1 para $k = 0$ y $k = n$:

$n \backslash k$	0	1	2	3	4	$\cdots$	$k-1$	k
0	1							
1	1	1						
2	1		1					
3	1			1				
4	1				1			
$\vdots$	$\vdots$					$\ddots$		
$k-1$	1						1	
k	1							1
$\vdots$	$\vdots$							
$n-1$	1							
n	1							

Ahora, teniendo ya calculados $\binom{1}{0}$ *y* $\binom{1}{1}$*, aplicando la propiedad calculamos* $\binom{2}{1}$*.*

$n \backslash k$	0	1	2	3	4	$\cdots$	$k-1$	k
0	1							
1	**1**	**1**						
2	1	2	1					
3	1			1				
4	1				1			
$\vdots$	$\vdots$					$\ddots$		
$k-1$	1						1	
k	1							1
$\vdots$	$\vdots$							
$n-1$	1							
n	1							

Luego, teniendo $\binom{2}{0}$ *y* $\binom{2}{1}$*, calculamos* $\binom{3}{1}$*, y con* $\binom{2}{1}$ *y* $\binom{2}{2}$*, calculamos* $\binom{3}{2}$*.*

$n \backslash k$	0	1	2	3	4	$\cdots$	$k-1$	k
0	1							
1	1	1						
2	**1**	**2**	**1**					
3	1	3	3	1				
4	1				1			
$\vdots$	$\vdots$					$\ddots$		
$k-1$	1						1	
k	1							1
$\vdots$	$\vdots$							
$n-1$	1							
n	1							

Y después, con $\binom{3}{0}$ *y* $\binom{3}{1}$*, calculamos* $\binom{4}{1}$*, con* $\binom{3}{1}$ *y* $\binom{3}{2}$*, calculamos* $\binom{4}{2}$ *y con con* $\binom{3}{2}$ *y* $\binom{3}{3}$*, calculamos* $\binom{4}{3}$*.*

$n \backslash k$	0	1	2	3	4	$\cdots$	$k-1$	k
0	1							
1	1	1						
2	1	2	1					
3	1	3	3	1				
4	1	4	6	4	1			
$\vdots$	$\vdots$					$\ddots$		
$k-1$	1						1	
k	1							1
$\vdots$	$\vdots$							
$n-1$	1							
n	1							

Y así siguiendo...

Esta idea deriva en el siguiente pseudocódigo:

Coeficientes binomiales

```
combinatorio(n,k)
  entrada: n, k ∈ N
  salida: (n k)

  para i = 1 hasta n hacer
      A[i][0] ← 1
  fin para
  para j = 0 hasta k hacer
      A[j][j] ← 1
  fin para
  para i = 2 hasta n hacer
      para j = 2 hasta min(i − 1,k) hacer
          A[i][j] ← A[i − 1][j − 1] + A[i − 1][j]
      fin para
  fin para
  retornar A[n][k]
```

Un algoritmo recursivo para calcular el $\binom{n}{k}$ sería $\Omega(\binom{n}{k})$. En cambio, el algoritmo de programación dinámica recién presentado tiene complejidad $O(nk)$ y necesita espacio $O(k)$, ya que sólo necesitamos almacenar la fila anterior de la que estamos calculando.

En la biliografía pueden encontrar una gran cantidad de algoritmo de PD, por ejemplo el problema del cambio y el problema de la mochila versión discreta (que vimos al iniciar la clase).

8. Algoritmos heurísticos y aproximados

Por ahora sólo vamos a mencionar qué es un algoritmo heurístico. Hacia la mitad del cuatrimestre vamos a entrar en detalle en esto.

Dado un problema Π , un algoritmo heurístico es un algoritmo que intenta obtener soluciones de buena calidad para el problema que se quiere resolver pero no necesariamente lo hace en todos los casos.

Sea Π un problema de optimización, I una instancia del problema, $x^*(I)$ el valor óptimo de la función a optimizar en dicha instancia. Un algoritmo heurístico obtiene una solución con un valor que se espera sea cercano a ese óptimo pero no necesariamente va a ser el óptimo.

Si H es un algoritmo heurístico para un problema de optimización llamamos $x^H(I)$ al valor que devuelve la heurística.

Decimos que H es un algoritmo ϵ – aproximado para el problema Π si para algún $\epsilon > 0$

$$|x^H(I) - x^*(I)| \leq \epsilon |x^*(I)|$$

9. Bibliografía recomendada

- Capítulos 6, 7, 8 y 9.6 de G. Brassard and P. Bratley, *Fundamental of Algorithmics*, Prentice-Hall, 1996.

- Secciones I.4, IV.15 IV.16 de T. Cormen, C. Leiserson, R. Rivest, C. Stein, *Introduction to Algorithms*, The MIT Press, McGraw-Hill, 2001.