DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Algoritmos y Estructuras de Datos III

Primer cuatrimestre 2021 *(dictado a distancia)*

Técnicas de diseño de algoritmos

Técnicas de diseño de algoritmos

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

Fuerza bruta

Idea: El método consiste en analizar *todas* las posibilidades.

- Fáciles de inventar.
- Fáciles de implementar.
- Siempre *funcionan*.
- Generalmente muy ineficientes.

Fuerza bruta - Problema de las n reinas

Problema: 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.

- Solución por FB: hallar *todas* las formas posibles de colocar n reinas en un tablero de $n \times n$ y luego seleccionar las que satisfagan las restricciones.
- 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.

Algoritmos golosos

Idea: Construir una solución seleccionando en cada paso *la mejor alternativa posible localmente* según una función de selección, sin considerar (o haciéndolo débilmente) las implicancias de esta selección.

- 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.
- Fáciles de inventar e implementar y generalmente eficientes.
- Pero no siempre *funcionan*: algunos problemas no pueden ser resueltos por este enfoque.
  - Habitualmente, proporcionan *heurísticas* sencillas para *problemas de optimización*.
  - En general permiten construir soluciones razonables, pero sub-óptimas.
- Función de selección.

Algoritmos golosos - El problema de la mochila (*continuo*)

Datos de entrada:

- Capacidad $C \in \mathbb{R}_+$ de la mochila (peso máximo).
- Cantidad $n \in \mathbb{N}$ de objetos.
- Peso $p_i \in \mathbb{R}_+$ del objeto i, para $i = 1, \ldots, n$.
- Beneficio $b_i \in \mathbb{R}_+$ del objeto i, para $i = 1, \ldots, n$.

Problema: 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*).

Algoritmos golosos - El problema de la mochila (*continuo*)

Algoritmo(s) goloso(s): Mientras no se haya excedido el peso de la mochila, agregar a la mochila el objeto i que ...

- ... tenga mayor beneficio b_i
- ... tenga menor peso p_i
- ... tenga mayor beneficio por unidad de peso, b_i/p_i

Algoritmos golosos - El problema de la mochila (*continuo*)

Datos de entrada:

$$C = 100,\ n = 5$$

	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

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

Algoritmos golosos - El problema de la mochila (*continuo*)

- ¿Qué podemos decir en cuanto a la *calidad* de las soluciones obtenidas por estos algoritmos?
- Se puede demostrar que la selección según máximo beneficio por unidad de peso, b_i/p_i, da una solución óptima.
- ¿Qué podemos decir en cuanto a su *complejidad*?
- ¿Qué sucede si los elementos se deben poner enteros en la mochila?

Algoritmos golosos - El problema del cambio

Problema: 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 cuatro de un centavo.

Algoritmo goloso: 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).

Algoritmos golosos - El problema del cambio

```
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
```

Algoritmos golosos - El problema del cambio

- Este algoritmo siempre produce la mejor solución *para estos valores de monedas*, 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 cuatro de 1 centavos, mientras que la solución óptima es retornar dos 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 esto puede llevar a una mala solución, y nunca modifica una decisión tomada.

Algoritmos golosos - Tiempo de espera total en un sistema

Problema: Un servidor tiene n clientes para atender, y los puede atender en cualquier orden. Para $i = 1, \ldots, n$, el tiempo necesario 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

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

Algoritmos golosos - Tiempo de espera total en un sistema

Algoritmo goloso: En cada paso, atender al cliente pendiente que tenga menor tiempo de atención.

- Retorna una permutación $I_{\text{GOL}} = (i_1, \ldots, i_n)$ tal que $t_{i_j} \leq t_{i_{j+1}}$ para $j = 1, \ldots, n-1$.
- ¿Cuál es la *complejidad* de este algoritmo?
- Este algoritmo proporciona la *solución óptima*!

Algoritmos recursivos

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

- 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.
- 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, dependiente del número de parámetros, del tipo de éstos, del problema concreto.
- Cada llamada *se debe acercar más* a un caso base, que al ser alcanzado finaliza con la recursión.

Algoritmos recursivos

- Supongamos que tenemos una función $f : \mathbb{N} \to \mathbb{N}$, que satisface $f(0) = 0$ y $f(n) = 2f(n-1) + n^2$. Esta declaración de f 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)
        si  n = 0  entonces
                resu ← 0
        sino
                resu ← 2*f(n − 1) + n^2
        fin si
        retornar  resu
```

Algoritmos recursivos

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

`sin_fin(`n`)`
entrada: $n \in \mathbb{N}$
salida: ???
retornar `sin_fin(`$n-1$`)`

¿Y con este?

`sin_fin2(`n`)`
entrada: $n \in \mathbb{N}$
salida: ???
si $n = 1$ **entonces**
retornar 0
sino
retornar `sin_fin2(`n`)`
fin si

No son algoritmos, porque *¡¡¡no terminan!!!*

Algoritmos recursivos: Factorial

```
factorial(n)
        entrada: n ∈ ℕ
        salida: n!
        si n = 1 entonces
                resu ← 1
        sino
                resu ← n*factorial(n − 1)
        fin si
        retornar resu
```

- El caso base corresponde a $n = 1$.
- El caso recursivo a $n > 1$.
- Como medida podemos tomar el valor de n.
- 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.

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.
- Luego, utilizando herramientas matemáticas, encontrar la fórmula cerrada (que no dependa de la complejidad de instancias más chicas) de esta ecuación.
- En el algoritmo del ejemplo del cálculo delfactorial:

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

Que es equivalente a:

$$T(n) = 1 + \sum_{k=2}^{n} \mathcal{O}(resu \leftarrow k * factorial(k-1))$$

$$\leq 1 + (n-1) * \mathcal{O}(resu \leftarrow n * factorial(n-1)).$$

Correctitud de algoritmos recursivos

Para demostrar la correctitud de un algoritmo tenemos que demostrar:

- que termina (o sea, que es un algoritmo)
  - los procesos repetitivos terminan cuando se aplican a instancias que cumplen la precondición.
- y que cumple la especificación
  - si el estado inicial satisface la precondición, entonces el estado final cumplirá la postcondición.
  - es muy frecuente recurrir al principio de inducción matemática.

Correctitud de algoritmos recursivos

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 1.

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

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

Correctitud de algoritmos recursivos

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
```

Correctitud de algoritmos recursivos

Cuadrado de un número entero no negativo

Terminación: 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 de algoritmos recursivos

Cuadrado de un número entero no negativo

Correctitud: 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: $n = 0$. En este caso el algoritmo es trivialmente correcto, ya que cuadrado$(n) = 0$ y $n^2 = 0$.

Paso inductivo: $n \geq 1$.
HI: el algoritmo es correcto cuando el parámetro es $n-1$:

$$\text{cuadrado}(n-1) = (n-1)^2.$$

Por definición del algoritmo,

$$\text{cuadrado}(n) = 2 * n + \text{cuadrado}(n-1) - 1.$$

Aplicando HI

$$\text{cuadrado}(n) = 2*n+(n-1)^2-1 = 2*n+n^2-2*n+1-1 = n^2.$$

Dividir y conquistar *(divide and conquer)*

- Si la instancia I de entrada es pequeña, entonces utilizar un algoritmo ad hoc para el problema.

- En caso contrario:
  - *Dividir* I en sub-instancias $I_1, I_2, \ldots, I_r$ más pequeñas.
  - Resolver *recursivamente* las r sub-instancias.
  - *Combinar* las soluciones para las r sub-instancias para obtener una solución para la instancia original I.

Dividir y conquistar *(divide and conquer)*

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.

Dividir y conquistar - Búsqueda binaria

Problema: Decidir si un elemento x se encuentra en un arreglo de n elementos A, ordenado de forma creciente.

Algoritmo de D&C:

- Primero decidir si x debería estar en la primera mitad del arreglo o en la segunda: para esto, 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.
- Si es mayor en la segunda.

Dividir y conquistar - Búsqueda binaria

```
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
```

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 $\mathcal{O}(1)$).
- Para los casos recursivos podemos identificar tres puntos críticos:
  - cantidad de llamadas recursivas, 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:

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

Complejidad de algoritmos dividir y conquistar

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

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

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

Dividir y conquistar - Búsqueda binaria

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

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

donde $g(n)$ es $\mathcal{O}(1) = \mathcal{O}(n^0)$

Como $r = 1$, $b = 2$, $k = 0$, tenemos que $r = b^k$, aplicando el teorema maestro, obtenemos que

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

Dividir y conquistar - Mergesort

Problema: Ordenar un arreglo A de n elementos.

Algoritmo de D&C (von Neumann, 1945):

- Si n es pequeño, ordenar por cualquier método sencillo.
- Si n es grande:
  - $A_1 :=$ primera mitad de A.
  - $A_2 :=$ segunda mitad de A.
  - Ordenar recursivamente A_1 y A_2 por separado.
  - Combinar A_1 y A_2 para obtener los elementos de A ordenados (apareo de arreglos).

Dividir y conquistar - 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)])
        A ← unir(A[0...(n div 2 − 1)], A[(n div 2)...(n − 1)])
    fin si
    retornar A
```

Dividir y conquistar - Mergesort

```
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
```

Dividir y conquistar - Mergesort

- Es fácil ver que `unir` es $\mathcal{O}(dim(U) + dim(V)) = \mathcal{O}(dim(A))$.
- 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 $\mathcal{O}(n)$.
- Como en este caso $r = 2$, $b = 2$ y $k = 1$, entonces $r = b^k$ y aplicando el Teorema maestro, obtenemos que:

$$T(n) \text{ es } \mathcal{O}(n \log n).$$

Backtracking

Idea: Técnica para recorrer sistemáticamente todas las posibles configuraciones del espacio de soluciones.

- 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.
- Para que el algoritmo sea correcto debemos estar seguros de no dejar de examinar configuraciones que estamos buscando.

Backtracking

- Habitualmente, 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 $A_1 \times \ldots \times A_n$.
- 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 el conjunto de soluciones sucesoras es vacío, esa rama no se continua explorando.

Backtracking

- Se puede pensar este espacio de búsqueda como un árbol dirigido donde cada vértice representa una solución parcial.
- 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.

Backtracking

- 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).

Backtracking: Esquema General - Todas las soluciones

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

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 sol
            fin si
        fin para
    fin si
    retornar
```

- *sol* variable global que guarda la solución.
- *encontro* variable booleana global que indica si ya se encontró

Backtracking

- 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.

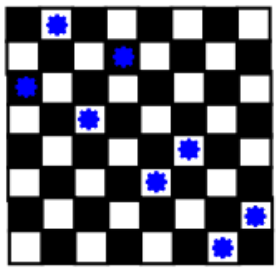
- Para el cálculo de la complejidad debemos acotar la cantidad de vértices que tendrá el árbol y considerar el costo de procesar cada uno.

Backtracking - Problema de las 8 reinas

Problema: Ubicar 8 reinas en el tablero de ajedrez (8×8) sin que ninguna *amenace* a otra.

- ¿Cuántas combinaciones del tablero hay que considerar?

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

- Sabemos que cada fila debe tener exactamente una reina.
- 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.
-  está representada por $(2, 4, 1, 3, 6, 5, 8, 7)$.
- Cada solución parcial puede estar representada por $(a_1, \ldots, a_k)$, $k \leq 8$.
- Tenemos ahora $8^8 = 16777216$ combinaciones.

Backtracking - Problema de las 8 reinas

- Una misma columna debe tener exactamente una reina.

  Se reduce 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.

- Se cumple 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.

Backtracking - Problema de las 8 reinas

- Dada una solución parcial $(a_1, \ldots, a_k)$, $k \leq 8$ (sin reinas que se amenacen), construiremos el conjunto de sus vértices hijos, $\texttt{Sucesores}(a, k)$, asegurando que siga sin haber reinas que se amenacen.

- Entonces la nueva reina, la (k)-ésima (correspondiente a la fila k), no podrá estar ubicada en ninguna de las columnas ni en ninguna de las diagonales donde están ubicadas las k anteriores.

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

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

Backtracking - Suma de subconjuntos

Problema: 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.

- Una solución puede estar representada por $a = (a_1, \ldots, a_r)$, 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 $\{c_2, c_4, c_5\}$.
- Queremos evitar esto porque estaríamos agrandando aún más el árbol de búsqueda.

Backtracking - Suma de subconjuntos

- Los vértices de nivel 1 del árbol de búsqueda se corresponderán a las n soluciones de subconjuntos de un elemento, (c_i), para $i = 1, \ldots, n$.

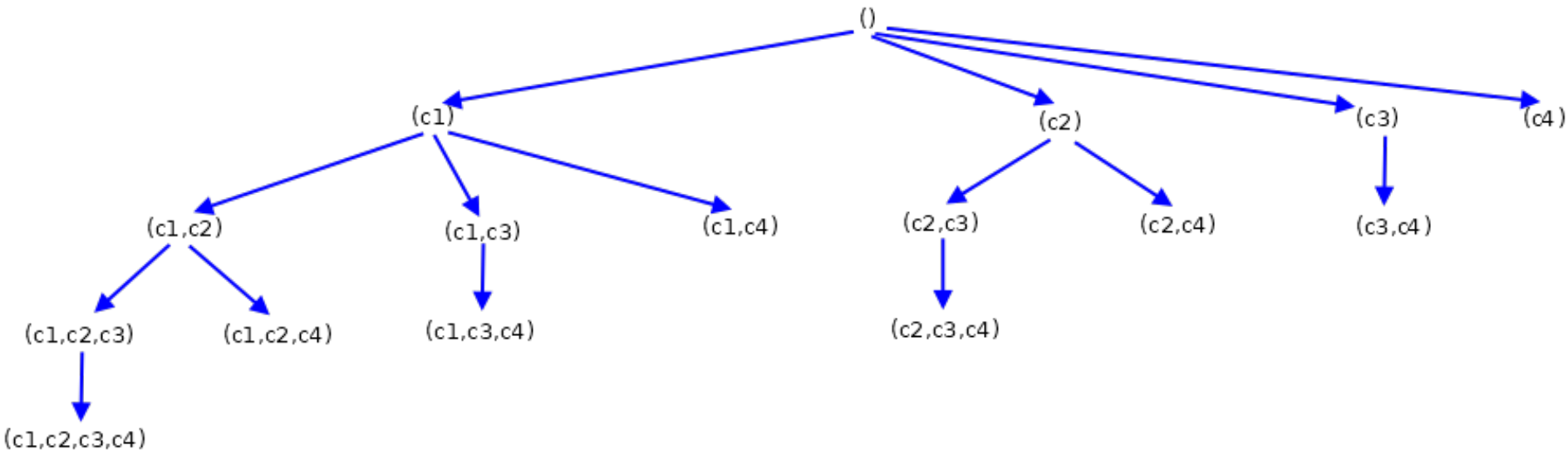
- En el segundo nivel, no quisiera generar los vectores (c_i, c_j) y (c_j, c_i), sino sólo uno de ellos porque representan la misma solución.

- 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.

Backtracking - Suma de subconjuntos

Si $n = 4$, el árbol de búsqueda es:



Backtracking - Suma de subconjuntos

- 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.
- Se cumple la propiedad dominó: Si una solución parcial $(a_1, \ldots, a_r)$ suma más que k, entonces toda extensión 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.
- Si la suma excede k, esa rama se puede podar, ya que los elementos de C son positivos.

Backtracking - Suma de subconjuntos

Podemos definir los sucesores de una solución como:

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

Programación Dinámica

Idea: Al igual que en D&C, el problema es dividido 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.

- Es aplicada típicamente a problemas de optimización combinatoria.
- También resulta adecuada para algunos problemas de naturaleza recursiva.
- Es adecuada para problemas que tienen las característas que le molestan a D&C.
- **Evita repetir llamadas recursivas almacenando los resultados calculados.**
- Esquema de *memoización*.
- Funciones recursivas top-down, pero algunas veces una implementación no recursiva bottom-up puede llegar a resultar en algoritmos con mejor tiempo de ejecución.

Programación Dinámica - Coeficientes binomiales

- **Definición:** Si $n \geq 0$ y $0 \leq k \leq n$, se define

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

- **Teorema:** Si $n \geq 0$ y $0 \leq k \leq n$, entonces

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

Programación Dinámica - Coeficientes binomiales

Algoritmo recursivo

combiRec(n, k)
entrada: $n, k \in N$
salida: $\binom{n}{k}$

si $k = 0$ **o** $k = n$ **hacer**
    `retornar 1`
else
    `retornar combiRec(`$n-1, k-1$`) + combiRec(`$n-1, k$`)`
fin si

Programación Dinámica - Coeficientes binomiales

Algoritmo recursivo con memoización (top-down)

combiRec(n, k, T)
entrada: $n, k \in N$
salida: $\binom{n}{k}$

si $T[n][k] = NULL$ **hacer**
  si $k = 0$ **o** $k = n$ **hacer**
    $T[n][k] \leftarrow 1$
  else
    $T[n][k] \leftarrow$ `combiRec(`$n-1, k-1, T$`) + combiRec(`$n-1, k, T$`)`
  fin si
fin si
`retornar` $T[n][k]$

Programación Dinámica - Coeficientes binomiales

$n \backslash k$	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							

Programación Dinámica - Coeficientes binomiales

algoritmo *combinatorio*(*n*,*k*)

entrada: dos enteros n y k

salida: $\binom{n}{k}$

para $i = 1$ **hasta** n **hacer**

$A[i][0] \leftarrow 1$

fin para

para $j = 0$ **hasta** k **hacer**

$A[j][j] \leftarrow 1$

fin para

para $i = 2$ **hasta** n **hacer**

para $j = 2$ **hasta** $\min(i-1, k)$ **hacer**

$A[i][j] \leftarrow A[i-1][j-1] + A[i-1][j]$

fin para

fin para

retornar $A[n][k]$

Programación Dinámica - Multiplicación de n matrices

Problema: Por la propiedad asociativa del producto de matrices

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

puede hacerse de muchas formas. Queremos determinar la que minimiza el número de operaciones necesarias.

Si las dimensión de A es de 13×5, la de B de 5×89, la de C de 89×3 y la de D de 3×34, tenemos

- $((AB)C)D$ 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.

Programación Dinámica - Multiplicación de n matrices

- 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.

- 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.

Programación Dinámica - Multiplicación de n matrices

Llamamos $m[i][j]$ solución del subproblema $M_i \times M_{i+1} \times \ldots M_j$. Suponemos 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 $i = 1, 2, \ldots, n$, $m[i][i] = 0$
- Para $i = 1, 2, \ldots, n-1$, $m[i][i+1] = d[i-1]d[i]d[i+1]$
- Para $s = 2, \ldots, n-1$, $i = 1, 2, \ldots, n-s$,

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

La solución del problema es $m[1][n]$.

Programación Dinámica - Subsecuencia común más larga

Dada una secuencia, una subsecuencia se obtiene eliminando 0 o más símbolos. Por ejemplo, $[4, 7, 2, 3]$ y $[7, 5]$ son subsecuencias de $[4, 7, 8, 2, 5, 3]$, $[2, 7]$ no lo es.

Problema: Encontrar la subsecuencia común mas larga (scml) de dos secuencias dadas.

- Es decir, dadas dos secuencias A y B, queremos encontrar entre todas secuencia 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.

Programación Dinámica - Subsecuencia común más larga

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.

Programación Dinámica - Subsecuencia común más larga

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 $a_i \neq b_j$: $l[i][j] = \text{máx}\{l[i-1][j], l[i][j-1]\}$

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

Programación Dinámica - Subsecuencia común más larga

scml(A, B)

entrada: A, B `secuencias`

salida: `longitud de a scml entre` A `y` B

$l[0][0] \leftarrow 0$
para $i = 1$ **hasta** r **hacer** $l[i][0] \leftarrow 0$
para $j = 1$ **hasta** s **hacer** $l[0][j] \leftarrow 0$
para $i = 1$ **hasta** r **hacer**
  para $j = 1$ **hasta** s **hacer**
    si $A[i] = B[j]$
      $l[i][j] \leftarrow l[i-1][j-1] + 1$
    sino
      $l[i][j] \leftarrow \text{máx}\{l[i-1][j], l[i][j-1]\}$
    fin si
  fin para
fin para
retornar $l[r][s]$

Heurísticas

- 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 el óptimo.

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

Algoritmos aproximados

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

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