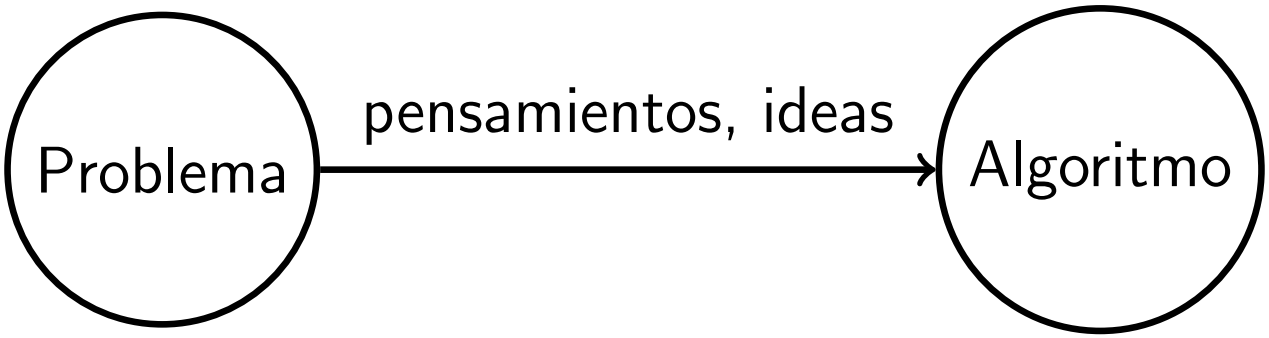
# Estrategias algorítmicas

E. Rivas

Mayo de 2014

# Introducción

Un modo de resolución de problemas con computadoras:




El objetivo de esta clase será conocer patrones de resolución de problemas, estrategias algorítmicas.

# Léxico algorítmico

En la literatura de cs. computacionales será común encontrar términos como:

- Greedy.
- Dinámica.
- Divide and conquer.
- Backtracking.

Estos términos hacen referencia a maneras de resolver problemas.

Forman un léxico, una jerga, que permite explicar la forma de resolución de un problema sin ahondar en detalles.

# Sobre el léxico

Conocer el léxico nos ayuda a entender a los demás cuando utilizan estos términos.

Pero además, nos ayuda para que nosotros pensemos también en estos términos. Al asignar una palabra a una idea o concepto, nos resulta más fácil pensar en términos de estos (ej. de Newspeak o la tesis Sapir-Whorf).

Para introducir estos términos daremos una definición intensional breve, y luego mostraremos algún ejemplo de manera de fijar esta definición.

# Problemas de optimización

Un *problema de optimización* apunta a encontrar la mejor solución de un conjunto de soluciones posibles.

En muchos casos se trata de encontrar el máximo o el mínimo de una función, sujetos a un conjunto de restricciones posibles.

$$
\begin{aligned}
\underset{x}{\text{minimize}} \quad & f(x) \\
\text{subject to} \quad & g_i(x) \leq 0, \quad i = 1, \ldots, m \\
& h_i(x) = 0, \quad i = 1, \ldots, p
\end{aligned}
$$

# Greedy: definición

Un algoritmo *greedy* busca el óptimo global mediante la búsqueda de óptimos locales en varias etapas. Suelen utilizarse para la resolución de problemas de optimización.

En cada etapa:

- Computamos el óptimo local, sin importarnos el futuro (carpe diem).
- Agregamos el óptimo local a nuestra solución final.
- Preparamos el problema para la siguiente etapa.

# Greedy: ejemplo

**interval scheduling.** dado un conjunto de tareas, descriptas por el intervalo de tiempo que ocupan, debemos encontrar la manera de maximizar la cantidad de tareas realizadas de manera que no se solapen dos tareas.

# Greedy: ejemplo

Nuestro óptimo local en este problema será la tarea que termine primero.

Agregaremos esta tarea a nuestro conjunto de tareas solución: aquellas que realizaremos.

Luego, nos prepararemos para la próxima etapa eliminando aquellas tareas incompatibles con la elegida.

Costo: $\Theta(n \log n)$

# Greedy: ejemplo

ver `greedy.c`

# Divide et impera: definición

Un algoritmo *divide et impera* busca la resolución de un problema dividiendolo en subproblemas del mismo tipo, solucionando cada uno, y luego combinandolos para obtener una solución al problema original.

El funcionamiento suele ser: si el problema es sencillo de resolver (pequeño), lo resolvemos de manera directa. En caso contrario:

- Dividimos el problema en subproblemas del mismo tipo.
- Resolvemos recursivamente cada uno.
- Combinamos las subsoluciones para obtener una solución.

# Divide et impera: ejemplo

**mergesort.** este algoritmo de ordenamiento funciona en base al método de resolución de "divide et impera":

- Si al arreglo tiene un solo elemento, ya está ordenado, y solucionamos el problema.
- En caso contrario:
  - Dividimos el arreglo en dos mitades.
  - Ordenamos recursivamente cada mitad.
  - Combinamos las mitades ordenadas.

# Divide et impera: ejemplo

ver `divandcon.c`

# Dinámica: definición

Un algoritmo *dinámico* busca la resolución la resolución mediante la combinación de subproblemas del mismo tipo. A diferencia de divide et impera, pediremos que los subproblemas se solapen.

Que los subproblemas se solapen significa que el problema lo dividimos en subproblemas que son reutilizados varias veces, o que estos subproblemas comparten subsubproblemas.

Un ejemplo típico es Fibonacci, donde al resolver `fib(50)` recursivamente, se termina recomputando muchísimas veces `fib(3)` por ej.

# Dinámica: definición (top-down)

El funcionamiento suele ser: si el problema es sencillo de resolver (pequeño), lo resolvemos de manera directa, y memorizamos la solución. En caso contrario:

- Dividimos el problema en subproblemas del mismo tipo.
- Resolvemos recursivamente cada uno.
- Combinamos las subsoluciones para obtener una solución.
- Memorizamos la solución.

De tener que resolver un caso ya resuelto, buscamos la solución memorizada.

# Dinámica: definición (bottom-up)

Existe otra manera de funcionamiento para estos algoritmos:

- Resolvemos los problemas pequeños.
- Resolvemos los problemas que se solucionan combinando solo los problemas pequeños.
- Resolvemos los problemas que se solucionan combinando los problemas ya resueltos.
- ...

# Dinámica: ejemplo

**números combinatorios.** dado $n$, $k$, deseamos calcular el número combinatorio de $n$ y $k$.

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

El algoritmo obvio (utilizando la fórmula matemática) puede resultar en problemas ya que es fácil que $n!$ haga un overflow.

# Dinámica: ejemplo

ver `dyn.c`

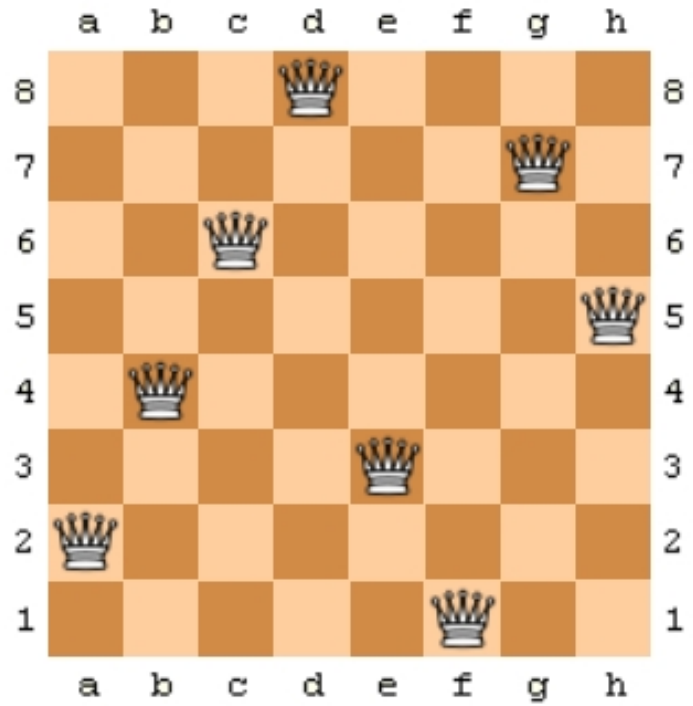
# Backtracking: definición

Un algoritmo de *backtracking* busca una solución avanzando por pasos, y retrocede parcialmente en caso de que notar que las elecciones realizadas no llegan a una solución.

- Si el nodo en el que estamos constituye una solución, entonces terminamos con solución.
- Si se puede avanzar en alguna dirección:
  - Avanzamos en alguna de estas direcciones.
  - Si recursivamente se termina con solución, entonces terminamos con solución.
  - Si no, continuamos con otra dirección posible.
- Si agotamos las direcciones, terminamos con fallo.

# Backtracking: ejemplo

**ocho reinas.** el problema consiste en posicionar ocho reinas en un tablero de ajedrez, de manera que ninguna reina ataque a la otra.



# Backtracking: ejemplo

Como argumento tomamos la fila del tablero en la que estamos posicionando a la reina.
Si estamos en la 9na. fila, es porque ya terminamos.
Si estamos en una fila $i$, empezando con la columna $j = 0$:

- Posicionamos una reina en la fila $i$, columna $j$.
- Intentamos resolver el problema para la fila $i + 1$.
- De ser exitosos, retornamos la solución.
- En caso contrario, nos novemos a la próxima columna.

Si no retornamos, retornamos fallo.

# Backtracking: ejemplo

ver `backtracking.c`

# Una demostración greedy

Retomando el problema del interval scheduling, podemos modelizarlo matemáticamente como:

1. $I$ - un conjunto de intervalos.
2. $s : I \to \mathbb{N}$ - la función que asigna a un intervalo el momento de inicio.
3. $f : I \to \mathbb{N}$ - la función que asigna a un intervalo el momento de finalización.

En general, asumimos que para todo $i \in I$ tenemos que $s(i) < f(i)$.

# Una demostración greedy

La solución de este problema consiste en encontrar un subconjunto $S$ de $I$ tal que:

$$\forall i, i' \in G.s(i) \leq s(i') \implies f(i) \leq s(i')$$

tal que $|S|$ sea máximo respecto a esta propiedad:

para todo otro conjunto $S'$ que satisfaga la propiedad, $|S'| \leq |S|$.

# Una demostración greedy

El siguiente es un esquema del funcionamiento del algoritmo greedy:

1. Sea $S = \emptyset$ la solución tentativa, $R = I$ los intervalos disponibles.
2. Elegimos $i \in R$ con el mínimo tiempo de finalización.
3. Agregamos $i$ a $S$.
4. Borramos de $R$ todos los elementos no compatibles con $i$.
5. Si $R$ es no vacío, volvemos al paso 2.

# Una demostración greedy

Para probar que el algoritmo greedy funciona efectivamente, supondremos que $G = \{i_1, i_2, \ldots, i_k\}$ es la solución que provee el algoritmo greedy. Asumiremos que las tareas están indexadas según el orden de inicio, i.e. si $k \leq k'$ entonces $s(i_k) \leq s(i_{k'})$.

Ahora supondremos que existe una solución óptima $O = \{o_1, o_2, \ldots, o_m\}$.

Probaremos por el absurdo que $|G| = |O|$.

# Una demostración greedy

Pero antes... demostraremos la siguiente propiedad:

$$r \leq k \implies f(i_r) \leq f(o_r)$$

Esta propiedad la probaremos por indución.

**Caso base:** ejercicio.

# Una demostración greedy

**Caso inductivo:**
Hipótesis: para $r$ vale $f(i_r) \leq f(o_r)$.
Tesis: $f(i_{r+1}) \leq f(o_{r+1})$.

Si $f(i_{r+1}) > f(o_{r+1})$, entonces el algoritmo greedy hubiera elegido a $o_{r+1}$ en lugar de $i_{r+1}$, ya que $o_{r+1}$ es efectivamente compatible con $o_r$ (por ser solución), y por ende es compatible con con $i_r$ (por hipótesis inductiva).

# Una demostración greedy

Ahora, en base a esta propiedad podemos demostrar que efectivamente la solución greedy es óptima.
Si la solución $G$ no es óptima, entonces $|O| > |G|$, por lo que $m > k$.

Instanciando nuestra propiedad en $r = k$, obtenemos que $f(i_k) \leq f(o_k)$. Ya que $m > k$, existe un elemento $o_{k+1} \in O$.

Puesto que $O$ es una solución válida, $o_{k+1}$ comienza después de que $o_k$ termine, y por lo tanto, después de que $i_k$ termine.

Pero entonces, el conjunto de tareas después de eliminar las tareas incompatibles con $I$ tendría como elemento a $o_{k+1}$, por lo cual el algoritmo no hubiera parado en ese punto. Absurdo.