Algoritmos y Estructuras de Datos III

Primer cuatrimestre 2022

(bienvenidos!)

Programa

1. Técnicas de diseño de algoritmos.
2. Introducción a la teoría de grafos y algoritmos sobre grafos.
3. Problema de árbol generador mínimo.
4. Problema de camino mínimo.
5. Problemas de flujo en redes.
6. Introducción a la teoría de NP-completitud.

Bibliografía

1. G. Brassard and P. Bratley, *Fundamental of Algorithmics*, Prentice-Hall, 1996.
2. F. Harary, *Graph theory*, Addison-Wesley, 1969.
3. R. Ahuja, T. Magnanti and J. Orlin, *Network Flows: Theory, Algorithms, and Applications*, Prentice-Hall, 1993.
4. M. Garey and D. Johnson, *Computers and intractability: a guide to the theory of NP- Completeness*, W. Freeman and Co., 1979.

Régimen de cursada

- Cursada:
  1. Lunes: clases teóricas.
  2. Miércoles y viernes: clases prácticas y de laboratorio.
- Evaluaciones:
  1. Dos parciales (individuales).
  2. Dos trabajos prácticos (en grupo).
  3. Un examen final (individual).
- Espacio en el campus virtual (campus.exactas.uba.ar), y comunicación por correo electrónico:
  1. Lista de docentes: `algo3-doc@dc.uba.ar`.
  2. Lista de alumnos: `algo3-alu@dc.uba.ar`.

Algoritmos

- ¿Qué es un algoritmo?
- ¿Qué es un buen algoritmo?
- Dados dos algoritmos para resolver un mismo problema, ¿cuál es mejor?
- ¿Cuándo un problema está bien resuelto?

Pseudocódigo

Describiremos los algoritmos por medio de pseudocódigo. Por ejemplo, el siguiente es un algoritmo para econtrar el máximo de un arreglo de enteros:

algoritmo *maximo*(A, n)
    entrada: un vector A con $n \geq 1$ de enteros
    salida: el elemento máximo de A

    $max \leftarrow A[0]$
    para $i = 1$ **hasta** $n - 1$ **hacer**
        si $A[i] > max$ **entonces**
            $max \leftarrow A[i]$
        fin si
    fin para
    retornar max

Análisis de algoritmos

Analizaremos los algoritmos utilizando como medida de eficiencia su tiempo de ejecución.

- Análisis empírico: implementarlos en una máquina determinada utilizando un lenguaje determinado, correrlos para un conjunto de instancias y comparar sus tiempos de ejecución (ventajas? desventajas?).
- Análisis teórico: determinar matemáticamente la cantidad de tiempo que llevará su ejecución como una función de la medida de la instancia considerada, independizándonos de la máquina sobre lá cuál es implementado el algoritmo y el lenguaje para hacerlo (ventajas? desventajas?).

Complejidad computacional

Definición informal: La *complejidad* de un algoritmo es una función que representa el tiempo de ejecución en función del tamaño de la entrada del algoritmo.

- Complejidad en el peor caso.
- Complejidad en el caso promedio.

Definición formal?

Modelo de cómputo: Máquina RAM

Definición: Máquina de registros + registro acumulador + direccionamiento indirecto.

Motivación: Modelar computadoras en las que la memoria es suficiente y donde los enteros involucrados en los cálculos entran en una palabra.

- Unidad de entrada: Sucesión de celdas numeradas, cada una con un entero de tamaño arbitrario.
- Unidad de salida: Sucesión de celdas.
- Memoria: Sucesión de celdas numeradas, cada una puede almacenar un entero de tamaño arbitrario.
- Programa no almacenado en memoria.

Modelo de cómputo: Máquina RAM

- Un programa es una secuencia de instrucciones que son ejecutadas secuencialmente.
- Hay un contador de programa, que identifica la próxima instrucción a ser ejecutada.
- Hay tantas celdas de memoria (registros) como se necesiten.
- Se pueden acceder de forma directa a cualquier celda (acceso aleatorio).
- Los enteros entran en una celda de memoria.
- Hay un registro especial, llamado acumulador, donde se realizan los cálculos.

Máquina RAM - Instrucciones

- LOAD operando - Carga un valor en el acumulador
- STORE operando - Carga el acumulador en un registro
- ADD operando - Suma el operando al acumulador
- SUB operando - Resta el operando al acumulador
- MULT operando - Multiplica el operando por el acumulador
- DIV operando - Divide el acumulador por el operando
- READ operando - Lee un nuevo dato de entrada $\rightarrow$ operando
- WRITE operando - Escribe el operando a la salida
- JUMP label - Salto incondicional
- JGTZ label - Salta si el acumulador es positivo
- JZERO label - Salta si el acumulador es cero
- HALT - Termina el programa

Complejidad en la Máquina RAM

- Asumimos que cada instrucción tiene un tiempo de ejecución asociado.
- **Tiempo de ejecución de un algoritmo A:**
  $T_A(I)$ = suma de los tiempos de ejecución de las instrucciones realizadas por el algoritmo con la *instancia* I.
- **Complejidad de un algoritmo A:**
  $f_A(n) = \text{máx}_{I:|I|=n} T(I)$ (pero debemos definir $|I|$!).

Tamaño de una instancia y tiempo de ejecución (1/2)

Modelo uniforme: Cada dato individual ocupa una celda de memoria, y cada operación básica tiene un tiempo de ejecución constante.

- Apropiado cuando el input es una estructura de datos y cada dato individual entra en una palabra de memoria.
- En general utilizaremos el modelo uniforme y como tamaño de la entrada la cantidad de elementos de la instancia de entrada.
- Este modelo es una simplificación razonable y facilita el análisis de los algoritmos.

Tamaño de una instancia y tiempo de ejecución (2/2)

Modelo logarítmico: El tamaño de la instancia es la cantidad de símbolos de un alfabeto necesaria para representarla, y el tiempo de ejecución de cada operación depende del tamaño de los operandos.

- Es apropiado para algoritmos que toman un número predeterminado de datos individuales como input (como cálculo del factorial o determinar si un entero es primo), tomando como tamaño de la entrada la cantidad de bits necesarios para representar la instancia en notación binaria.
- Por ejemplo, para almacenar $n \in \mathbb{N}$, se necesitan $L(n) = \lfloor \log_2(n) \rfloor + 1$ dígitos binarios.
- Para almacenar una lista de m enteros, se necesitan $L(m) + mL(N)$ dígitos binarios, donde N es el valor máximo de la lista.

Notación O

Dadas dos funciones $f, g : \mathbb{N} \to \mathbb{R}$, decimos que:

- $f(n) = O(g(n))$ si existen $c \in \mathbb{R}_+$ y $n_0 \in \mathbb{N}$ tales que $f(n) \leq c\, g(n)$ para todo $n \geq n_0$.
- $f(n) = \Omega(g(n))$ si existen $c \in \mathbb{R}_+$ y $n_0 \in \mathbb{N}$ tales que $f(n) \geq c\, g(n)$ para todo $n \geq n_0$.
- $f(n) = \Theta(g(n))$ si $f = O(g(n))$ y $f = \Omega(g(n))$.

Notación O usando lím

Si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = a$ con $0 \leq a < \infty$, significa que

$$\left| \frac{f(n)}{g(n)} - a \right| < \epsilon$$

para algún $\epsilon > 0$ para n suficientemente grande. Entonces, $f(n) < (\epsilon + a)\, g(n)$, y por lo tanto $f(n)$ es $O(g(n))$.

- $f(n)$ es $O(g(n))$ si y sólo si $\lim_{n\to\infty} \frac{f(n)}{g(n)} \in [0, \infty)$.
- $f(n)$ es $\Omega(g(n))$ si y sólo si $\lim_{n\to\infty} \frac{f(n)}{g(n)} \in (0, \infty]$.
- $f(n)$ es $\Theta(g(n))$ si y sólo si $\lim_{n\to\infty} \frac{f(n)}{g(n)} \in (0, \infty)$.

Notación O

- Si un algoritmo es $O(n)$, se dice lineal.
- Si un algoritmo es $O(n^2)$, se dice cuadrático.
- Si un algoritmo es $O(n^3)$, se dice cúbico.
- Si un algoritmo es $O(n^k)$, $k \in \mathbb{N}$, se dice polinomial.
- Si un algoritmo es $O(\log n)$, se dice logarítmico.
- Si un algoritmo es $O(d^n)$, $d \in \mathbb{R}_+$, se dice exponencial.

- Cualquier función exponencial es *peor* que cualquier función polinomial: Si $k, d \in \mathbb{N}$ entonces k^n no es $O(n^d)$.
- La función logarítmica es *mejor* que la función lineal (no importa la base), es decir $\log n$ es $O(n)$ pero no a la inversa.

Ejemplos

- Búsqueda secuencial: $O(n)$.
- Búsqueda binaria: $O(\log n)$.

- Ordenar un arreglo (bubblesort): $O(n^2)$.
- Ordenar un arreglo (quicksort): $O(n^2)$ en el peor caso (!).
- Ordenar un arreglo (heapsort): $O(n \log n)$.

Es interesante notar que $O(\log n)$ es la complejidad **óptima** para algoritmos de búsqueda en un arreglo, y lo mismo sucede con $O(n \log n)$ para algoritmos de ordenamiento basados en comparaciones binarias.

Ejemplo dramático: Cálculo del determinante de una matriz

Recordemos: Si $A \in \mathbb{R}^{n \times n}$, se define su *determinante* por

$$\det(A) = \sum_{j=1}^{n} (-1)^{j+1} a_{ij} \det(A_{ij}),$$

donde $i \in \{1, \ldots, n\}$ y A_{ij} es la submatriz de A obtenida al eliminar la fila i y la columna j.

Complejidad: $f(n) = \begin{cases} n\, f(n-1) + O(n) & \text{si } n > 1 \\ O(1) & \text{si } n = 1 \end{cases}$

$= O(n!)$ (oops!).

Ejemplo dramático: Cálculo del determinante de una matriz

Algoritmo alternativo: Obtener la *descomposición* LU, escribiendo $PA = LU$. Entonces,

$$\det(A) = \det(P^{-1})\ \det(L)\ \det(U),$$

y todos los determinantes del lado derecho son sencillos de calcular.

Complejidad: $f(n) = O(n^3) + 3O(n) = O(n^3)$ (!).

Problemas bien resueltos

Definición: Decimos que un problema está bien resuelto si existe un algoritmo de complejidad polinomial para el problema.

	$n = 10$	$n = 20$	$n = 30$	$n = 40$	$n = 50$
$O(n)$	0.01 ms	0.02 ms	0.03 ms	0.04 ms	0.05 ms
$O(n^2)$	0.10 ms	0.40 ms	0.90 ms	0.16 ms	0.25 ms
$O(n^3)$	1.00 ms	8.00 ms	2.70 ms	6.40 ms	0.12 sg
$O(n^5)$	0.10 sg	3.20 sg	24.30 sg	1.70 min	5.20 min
$O(2^n)$	1.00 ms	1.00 sg	17.90 min	12 días	35 años
$O(3^n)$	0.59 sg	58 min	6 años	3855 siglos	2×10^8 siglos!

Problemas bien resueltos

Conclusión: Los algoritmos polinomiales se consideran satisfactorios (cuanto menor sea el grado, mejor), y los algoritmos supra-polinomiales se consideran no satisfactorios.

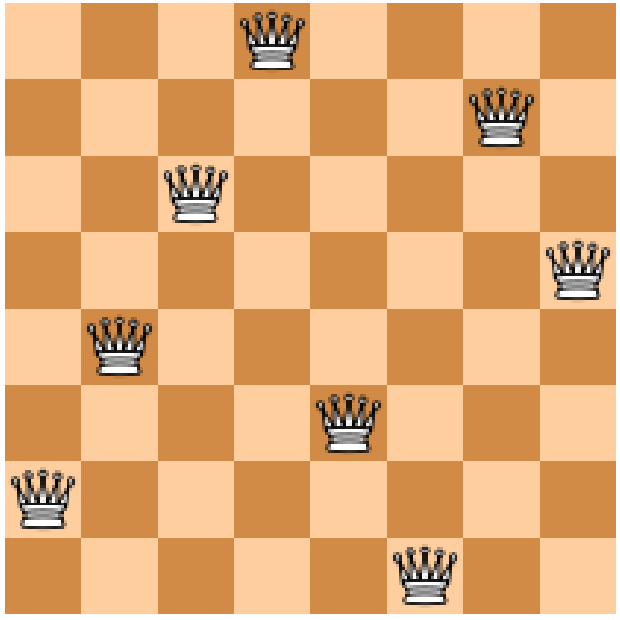
No obstante ...

- Si los tamaños de instancia son pequeños, ¿es tan malo un algoritmo exponencial?
- ¿Cómo se comparan $O(n^{85})$ con $O(1{,}001^n)$?
- ¿Puede pasar que un algoritmo de peor caso exponencial sea eficiente en la práctica? ¿Puede pasar que en la práctica sea *el mejor*?
- ¿Qué pasa si no encuentro un algoritmo polinomial?

Técnicas de diseño de algoritmos

- *Divide and conquer* (AED2)
- Fuerza bruta y *backtracking*
- Programación dinámica
- Heurísticas y algoritmos aproximados
- Algoritmos golosos
- Algoritmos probabilísticos

Fuerza bruta - Problema de las n damas



Problema: Ubicar n damas en un tablero de ajedrez de $n \times n$ casillas, de forma que ninguna dama amenace a otra.

Fuerza bruta - Problema de las n damas

- Solución por **fuerza bruta**: hallar todas las formas posibles de colocar n damas en un tablero de $n \times n$ y luego seleccionar las que satisfagan las restricciones.
- Un algoritmo de fuerza bruta (también llamado de búsqueda exhaustiva) analiza todas las posibles "configuraciones", lo cual habitualmente implica una complejidad exponencial.
- Por ejemplo, para $n = 8$ una implementación directa consiste en generar todos los subconjuntos de casillas.

$$2^{64} = 18,446,744,073,709,551,616 \text{ combinaciones!}$$

- Sabemos que dos damas no pueden estar en la misma casilla.

$$\binom{64}{8} = 4,426,165,368 \text{ combinaciones.}$$

Fuerza bruta - Problema de las n damas

- Sabemos que cada columna debe tener exactamente una dama. Cada solución parcial puede estar representada por $(a_1, \ldots, a_k)$, $k \leq 8$, con $a_i \in \{1, \ldots, 8\}$ indicando la fila de la dama que está en la columna i.

  Tenemos ahora $8^8 = 16,777,216$ combinaciones.

- Adicionalmente, cada fila debe tener exactamente una dama.

  Se reduce a $8! = 40,320$ combinaciones.

- Esto está mejor, pero se puede mejorar observando que no es necesario analizar muchas de estas combinaciones (¿por qué?).

Backtracking

Idea: Recorrer sistemáticamente todas las posibles configuraciones del espacio de soluciones de un problema computacional, eliminando las configuraciones parciales que no puedan completarse a una solución.

- Habitualmente, utiliza un vector $a = (a_1, a_2, \ldots, a_n)$ para representar una solución candidata, cada a_i pertenece un dominio/conjunto ordenado y finito A_i.
- El espacio de soluciones es el producto cartesiano $A_1 \times \ldots \times A_n$.

Backtracking

- 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 sucesores de la anterior.
- Si S_{k+1} es vacío, se *retrocede* a la solución parcial $(a_1, a_2, \ldots, a_{k-1})$.
- Se puede pensar este espacio 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.
- Permite descartar configuraciones antes de explorarlas (podar el árbol).

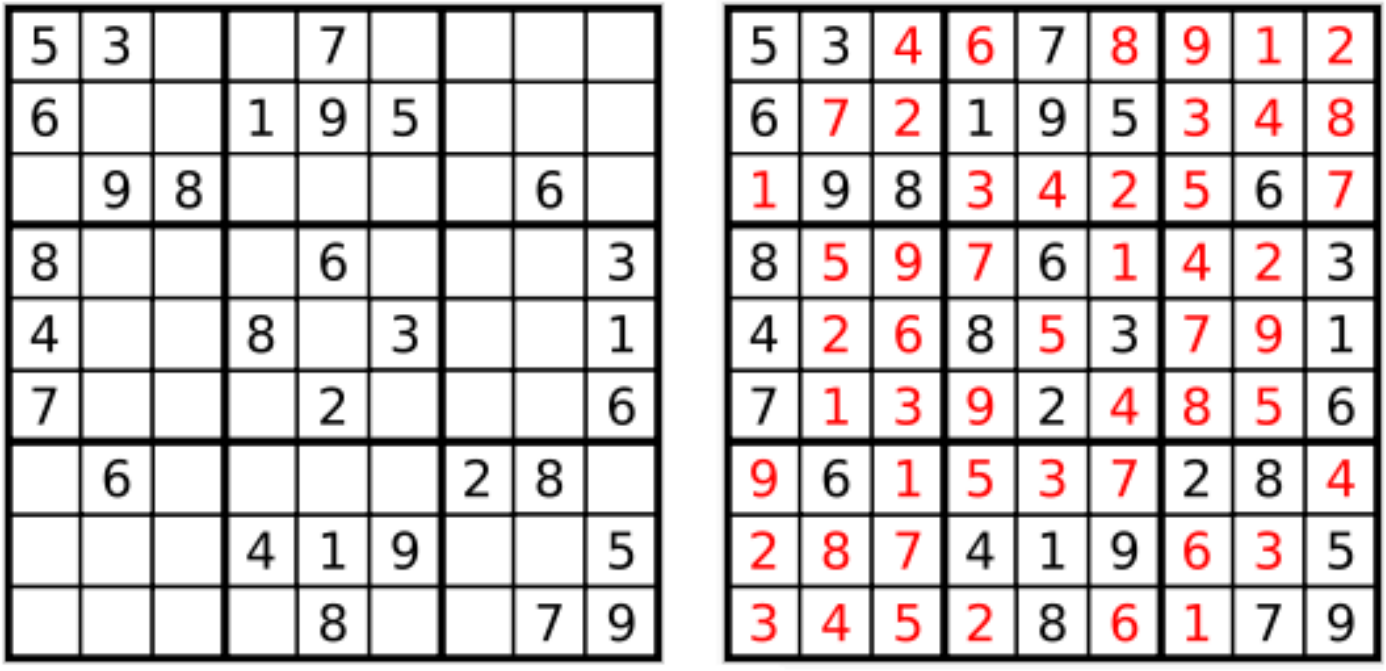
Backtracking: Todas las soluciones

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

Backtracking: Una solución

```
algoritmo BT(a, k)
    si a es solución 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
```

Backtracking - Resolver un *sudoku*



5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

El problema de resolver un *sudoku* se resuelve en forma muy eficiente con un algoritmo de *backtracking* (no obstante, el peor caso es exponencial!).

Programación dinámica

- Al igual que *divide and conquer*, se divide el problema en subproblemas de tamaños menores que se resuelven recursivamente.
- **Ejemplo.** Cálculo de coeficientes binomiales. Si $n \geq 0$ y $0 \leq k \leq n$, definimos

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

- No es buena idea computar esta definición (¿por qué?).
- **Teorema.** Si $n \geq 0$ y $0 \leq k \leq n$, entonces

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

Programación dinámica

Tampoco es buena idea implementar un algoritmo recursivo directo basado en esta fórmula (¿por qué?).

algoritmo *combinatorio*(n,k)
    entrada: dos enteros n y k
    salida: $\binom{n}{k}$

    si $k = 0$ **o** $k = n$ **hacer**
        retornar 1
    si no
        $a :=$ *combinatorio*$(n-1,\ k-1)$
        $b :=$ *combinatorio*$(n-1,\ k)$
        retornar $a + b$
    fin si

Programación dinámica

- **Superposición de estados:** El árbol de llamadas recursivas resuelve el mismo problema varias veces.
    1. Alternativamente, podemos decir que se realizan muchas veces llamadas a la función recursiva con los mismos parámetros.
- Un algoritmo de programación dinámica evita estas repeticiones con alguno de estos dos esquemas:
    1. **Enfoque top-down.** Se implementa recursivamente, pero se guarda el resultado de cada llamada recursiva en una estructura de datos (memoización). Si una llamada recursiva se repite, se toma el resultado de esta estructura.
    2. **Enfoque bottom-up.** Resolvemos primero los subproblemas más pequeños y guardamos (habitualmente en una tabla) todos los resultados.

Ejemplo: Cálculo de coeficientes binomiales

	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							

Ejemplo: Cálculo de 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 $A[n][k]$

Ejemplo: Cálculo de coeficientes binomiales

- Función recursiva:
  - Complejidad $\Omega(\binom{n}{k})$.
- Programación dinámica:
  - Complejidad $O(nk)$.
  - Espacio $\Theta(k)$: sólo necesitamos almacenar la fila anterior de la que estamos calculando.