Algoritmos y Estructuras de Datos III

Segundo cuatrimestre 2013

Programa

1. Algoritmos:

- Definición de algoritmo. Máquina RAM. Complejidad. Algoritmos de tiempo polinomial y no polinomial. Límite inferior.
- Técnicas de diseño de algoritmos: *divide and conquer*, *backtracking*, algoritmos golosos, programación dinámica.
- Algoritmos aproximados y algoritmos heurísticos.

Programa

2. Grafos:

- Definiciones básicas. Adyacencia, grado de un nodo, isomorfismos, caminos, conexión, etc.
- Grafos eulerianos y hamiltonianos.
- Grafos bipartitos.
- Árboles: caracterización, árboles orientados, árbol generador.
- Planaridad. Coloreo. Número cromático.
- Matching, conjunto independiente, recubrimiento. Recubrimiento de aristas y vértices.

Programa

3. Algoritmos en grafos y aplicaciones:

- Representación de un grafo en la computadora: matrices de incidencia y adyacencia, listas.
- Algoritmos de búsqueda en grafos: BFS, DFS, A*.
- Mínimo árbol generador, algoritmos de Prim y Kruskal.
- Algoritmos para encontrar el camino mínimo en un grafo: Dijkstra, Ford, Floyd, Dantzig.
- Planificación de procesos: PERT/CPM.
- Algoritmos para determinar si un grafo es planar. Algoritmos para coloreo de grafos.
- Algoritmos para encontrar el flujo máximo en una red: Ford y Fulkerson.
- Matching: algoritmos para correspondencias máximas en grafos bipartitos. Otras aplicaciones.

Programa

4. Complejidad computacional:

- Problemas tratables e intratables. Problemas de decisión. P y NP. Máquinas de Turing no determinísticas. Problemas NP-completos. Relación entre P y NP.
- Problemas de grafos NP-completos: coloreo de grafos, grafos hamiltonianos, recubrimiento mínimo de las aristas, corte máximo, etc.

Bibliografía

1. G. Brassard and P. Bratley, *Fundamental of Algorithmics*, Prentice-Hall, 1996.
2. F. Harary, *Graph theory*, Addison-Wesley, 1969.
3. J. Gross and J. Yellen, *Graph theory and its applications*, CRC Press, 1999.
4. R. Ahuja, T. Magnanti and J. Orlin, *Network Flows: Theory, Algorithms, and Applications*, Prentice-Hall, 1993.
5. M. Garey and D. Johnson, *Computers and intractability: a guide to the theory of NP- Completeness*, W. Freeman and Co., 1979.

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?

Algoritmo

Secuencia de pasos que termina en un tiempo finito. Deben estar formulados en términos de pasos sencillos que sean:

- precisos: se indica el orden de ejecución de cada paso
- bien definidos: en toda ejecución del algoritmo se debe obtener el mismo resultado bajo los mismo parámetros
- finitos: el algoritmo tiene que tener un número determinado de pasos

Los describiremos mediante pseudocódigo.

Pseudocódigo

Encontrar el máimo de un vector de enteros:

algoritmo $\mathtt{maximo}(S, n)$
  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

En general, analizaremos los algoritmos utilizando como medida de eficiencia su tiempo de ejecución.

- Análisis empírico: implementarlos en una máquina determinada utilizando un lenguaje determinado, correlos para un conjunto de instancias y comparar sus tiempos de ejecución. Desventajas:
  - pérdida de tiempo y esfuerzo de programador
  - pérdida de tiempo de cómputo
  - conjunto de instancias acotado
- 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. Para esto necesitamos definir:
  - un modelo de cómputo
  - un lenguaje sobre este modelo
  - instancias relevantes
  - tamaño de la instancia

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 (aún así es una máquina programable!).

Modelo de cómputo: Máquina RAM

Suponemos que:

- Un programa es ua secuencia de intrucciones que son ejecutadas secuencialmente, comenzando por la primera isntrucción.
- 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 random).
- 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

Máquina RAM - Operandos

- LOAD $= a$: Carga en el acumulador el entero a.
- LOAD i: Carga en el acumulador el contenido del registro i.
- LOAD $*i$: Carga en el acumulador el contenido del registro indexado por el valor del registro i.

Máquina RAM - Ejemplos

$a \leftarrow b$		LOAD	2
		STORE	1
$a \leftarrow b - 3$		LOAD	2
		SUB	=3
		STORE	1
mientras $x > 0$ **hacer**	guarda	LOAD	1
$x \leftarrow x - 2$		JGTZ	mientras
fin mientras		JUMP	finmientras
	mientras	LOAD	1
		SUB	=2
		STORE	1
		JUMP	guarda
	finmientras	....	

Máquina RAM - Ejemplos

si $x \leq 0$ **entonces**
  $y \leftarrow x + y$
sino
  $y \leftarrow x$
fin si

```
guarda  LOAD   1
        JGTZ   sino
        LOAD   1
        ADD    2
        STORE  2
        JUMP   finsi
sino    LOAD   1
        STORE  2
finsi   ...
```

Programa para calcular k^k - Pseudocódigo

algoritmo $kalak(k)$
  entrada: un entero k
  salida: k^k si $k > 0$, 0 caso contrario

  si $n \leq 0$ **entonces**
    $x \leftarrow 0$
  sino
    $x \leftarrow k$
    $y \leftarrow k - 1$
    mientras $y > 0$ **hacer**
      $x \leftarrow x \cdot k$
      $y \leftarrow y - 1$
    fin mientras
  fin si
  retornar x

Programa para calcular k^k - máquina RAM

	READ	1	carga en R1 la primera celda de la unidad de entrada
	LOAD	1	carga en el acumulador el valor de R1
	JGTZ	sino	si el valor del aculmulador es ≥ 0 *salta a* sino
	LOAD	$= 0$	carga 0 en el acumulador
	STORE	2	escribe el valor del acumulador en R2
	JUMP	finsi	*salta a* finsi
sino	LOAD	1	carga en el acumulador el valor de R1
	STORE	2	escribe el valor del acumulador en R2
	LOAD	1	carga en el acumulador el valor de R1
	SUB	$= 1$	resta 1 al valor del acumulador
	STORE	3	escribe el valor del acumulador en R3
guarda	LOAD	3	carga en el acumulador el valor de R3
	JGTZ	mientras	si el valor del aculmulador es ≥ 0 *salta a* mientras
	JUMP	finmientras	*salta a* finmientras
mientras	LOAD	2	carga en el acumulador el valor de R2
	MULT	1	multiplica el valor del acumulador por el valor de R1
	STORE	2	escribe el valor del acumulador en R2
	LOAD	3	carga en el acumulador el valor de R3
	SUB	$= 1$	resta 1 al valor del acumulador
	STORE	3	escribe el valor del aculmulador en R3
	JUMP	guarda	*salta a* guarda
finmientras finsi	WRITE	2	escribe el valor de R2 en la unidad de salida
	HALT		para la ejecución

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 $|I|$!).

Operaciones básicas: tiempo de ejecución

- **Modelo uniforme:** Cada operación básica tiene un tiempo de ejecución constante.
  - Apropiado cuando los operandos entran en una palabra.
- **Modelo logarítmico:** El tiempo de ejecución de cada operación es una función del tamaño de los operandos.
  - Apropiado cuando los operandos pueden crecer arbitrariamente.

Tamaño de una instancia

Definición: Dada una instancia I, se define $|I|$ como el número de símbolos de un alfabeto finito necesarios para codificar I.

- Depende del alfabeto y de la base.
- 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 (notar que se puede mejorar!).
- etc.

Tamaño de una instancia

- Depende del problema que se esté analizando.
- En general, para problemas de ordenamiento, problemas sobre grafos, etc., utilizaremos como tamaño de la entrada la cantidad de elementos de la instancia de entrada. Esto modela de forma suficientemente precisa la realidad y facilita el análisis de los algoritmos.
- Para problemas sobre números, como cáculo del factorial, es más apropiado utilizar como tamaño de la entrada la cantidad de bits necesarios para representar la instancia de entrada en notación binaria.

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 $f = \Omega(g(n))$.

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 $O(n \log(n))$ es la complejidad **óptima** para algoritmos de ordenamiento basados en comparaciones.

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.

- 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?