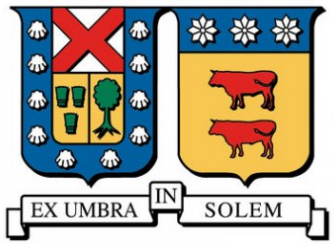
INF-221 Algoritmos y Complejidad

Unidad 2: Análisis de Algoritmos

Diego Arroyuelo

Departamento de Informática
Universidad Técnica Federico Santa María
darroyue@inf.utfsm.cl



17 de agosto de 2016

Outline

Outline

Modelo de Computación RAM

Usaremos el modelo de computación RAM (Random Access Machine).

Definición

Una RAM es un computador hipotético compuesto por:

- Unidad de procesamiento (CPU)
- Un número limitado de registros del procesador, $R_1, R_2, \ldots, R_k$.
- Un conjunto de instrucciones que son ejecutadas por la CPU.
- Memoria Principal compuesta por una cantidad infinita de celdas, $\mathrm{M}[0], \mathrm{M}[1], \ldots$, las cuales pueden ser accedidas de manera aleatoria.

Asumimos el modelo de John von Neumann:

Los programas son almacenados en la memoria principal de la RAM, junto con los datos.

Modelo de Computación RAM

En una RAM:

- Cada instrucción del procesador se ejecuta en 1 ciclo del procesador.
- Cada celda de la memoria (palabra de memoria) puede almacenar un entero (e.g., de 32 bits).
- Sólo una cantidad finita de celdas de memoria está en uso en cada momento.
- Una celda de memoria es accedida en 1 ciclo del procesador.

Outline

Modelo de Computación RAM

- Al medir tiempos de ejecución de un algoritmo, nos independizamos de las implementaciones del mismo.
- Nos centramos en las operaciones elementales que realiza (normalmente coinciden con las instrucciones de la RAM).
  - **Asignación** (=, +=, *=, ++, –, etc.).
  - **Operaciones aritméticas y lógicas** (+, *, -, /, &&, ||, !).
  - **Operaciones relacionales** (==, <=, >=, <, >, !=).
  - **Lectura o escritura**.
  - **Saltos (gotos) implícitos o explícitos** (return, break, etc.).
- A menudo también se suele medir sólo cierto tipo de operaciones ejecutadas por un algoritmo.
  - En particular cuando hay operaciones que son más costosas que otras: **comparaciones (algoritmos de ordenamiento), multiplicaciones o divisiones, accesos a disco, etc.**

Modelo de Computación RAM

Ejemplo 1

```
int buscarMaximo(int *A, int n) {
    int posMax = 0;
    int i;
    for (i=1; i<n; i++)
        if (A[posMax] < A[i])
            posMax = i;
    return posMax;
}
```

Este algoritmo procesa arreglos de tamaño n, y produce una posición del arreglo como resultado El algoritmo compara valores de tipo **int**.

Modelo de Computación RAM

Definición (Tiempo de ejecución de un algoritmo)

Cantidad de operaciones básicas ejecutadas como función del tamaño de entrada del algoritmo.

Notar cómo, con esta definición, nos independizamos de las implementaciones.

- Si n es el tamaño de la entrada de un algoritmo, $T(n)$ denota el tiempo de ejecución del algoritmo
- ¿Tiempo de ejecución del algoritmo de Ejemplo 1?
  - n iteraciones.
  - En cada iteración el costo es c operaciones básicas.
    (Queremos una aproximación al tiempo del algoritmo, no interesan las constantes).
  - $T(n) = c \cdot n$.
    (Esta ecuación describe la velocidad de crecimiento del tiempo de ejecución del algoritmo).

Modelo de Computación RAM

Velocidad de crecimiento

- Al analizar y comparar algoritmos, lo importante es la velocidad de crecimiento de las funciones de tiempo correspondientes.
- Es decir, importa cómo el tiempo de ejecución de un algoritmo crece a medida que crece el tamaño de la entrada.
- El análisis asintótico es adecuado en estos casos.
- Estudiamos las notaciones asintóticas más importantes a continuación.

Outline

Análisis Asintótico

Técnicas de Análisis Asintótico

- Ya sabemos cómo calcular el tiempo de ejecución de un algoritmo.
- Ahora clasificamos las funciones de tiempo con el objetivo de compararlas.
- Estudiamos a continuación el concepto de cota asintótica.

Análisis Asintótico

Notación O ("O" grande)

- Queremos estudiar las funciones f que crecen a lo sumo (cota superior) tan rápido como otra función g.
- Sea

$$g : \mathbb{N} \mapsto [0, +\infty].$$

- Definimos el conjunto de funciones de orden $O(g)$ como:

$$\begin{aligned} O(g(n)) \quad = \{ \quad & f : \mathbb{N} \mapsto [0, +\infty), \\ & \exists c \in \mathbb{R}, \\ & \exists n_0 \in \mathbb{N}, \\ & 0 \leq f(n) \leq c \cdot g(n), \ \forall n \geq n_0 \}. \end{aligned}$$

Análisis Asintótico

Notación O ("O" grande)

- Si una función $f \in O(g)$, entonces decimos que f es de orden $O(g)$.
- $f \in O(g)$ indica que f está acotada superiormente por algún múltiplo de g.
- Estamos interesados en la menor función g tal que $f \in O(g)$.
  - $2n + 3 \in O(n^2)$ es correcto, pero no es una cota ajustada.
  - $2n + 3 \in O(n)$ sí es una cota ajustada.
- Por ejemplo, el tiempo de ejecución del algoritmo `buscarMaximo` visto anteriormente es $O(n)$.

Análisis Asintótico

Notación Ω ("Omega" grande)

- Queremos estudiar las funciones f que crecen al menos (cota inferior) tan rápido como otra función g.
- Sea

$$g : \mathbb{N} \mapsto [0, +\infty].$$

- Definimos el conjunto de funciones $\Omega(g)$ como:

$$\begin{aligned} \Omega(g(n)) \quad = \{ \quad & f : \mathbb{N} \mapsto [0, +\infty), \\ & \exists c \in \mathbb{R}, \\ & \exists n_0 \in \mathbb{N}, \\ & f(n) \geq c \cdot g(n) \geq 0, \ \forall n \geq n_0 \}. \end{aligned}$$

Análisis Asintótico

Notación Θ ("Theta")

- En ocasiones es importante estudiar las funciones f que tienen un crecimiento asintótico exactamente igual a otra función g.
- Sea

$$g : \mathbb{N} \mapsto [0, +\infty].$$

- Definimos el conjunto de funciones $\Omega(g)$ como:

$$\begin{aligned}\Theta(g(n)) \quad = \{ \quad & f : \mathbb{N} \mapsto [0, +\infty), \\ & \exists c_1, c_2 \in \mathbb{R}, \\ & \exists n_0 \in \mathbb{N}, \\ & 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n), \ \forall n \geq n_0\}.\end{aligned}$$

Análisis Asintótico

Notación o ("o" chica)

- En ocasiones es importante estudiar las funciones f que tienen un crecimiento asintótico estrictamente menor a otra función g.
- Sea

$$g : \mathbb{N} \mapsto [0, +\infty].$$

- Definimos el conjunto de funciones $o(g)$ como:

$$\begin{aligned} o(g(n)) = \{ \quad & f : \mathbb{N} \mapsto [0, +\infty), \\ & \forall c \in \mathbb{R}, \\ & \exists n_0 \in \mathbb{N}, \\ & 0 \leq f(n) < c \cdot g(n), \ \forall n \geq n_0 \}. \end{aligned}$$

Análisis Asintótico

Notación ω ("omega" chica)

- En ocasiones es importante estudiar las funciones f que tienen un crecimiento asintótico estrictamente mayor a otra función g.
- Sea

$$g : \mathbb{N} \mapsto [0, +\infty].$$

- Definimos el conjunto de funciones $\omega(g)$ como:

$$\begin{aligned} \omega(g(n)) \quad = \{ \quad & f : \mathbb{N} \mapsto [0, +\infty), \\ & \forall c \in \mathbb{R}, \\ & \exists n_0 \in \mathbb{N}, \\ & f(n) > c \cdot g(n) \geq 0, \ \forall n \geq n_0 \}. \end{aligned}$$

Análisis Asintótico

Propiedades de las Notaciones Asintóticas

1. $\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$.
2. $o(g(n)) = O(g(n)) - \Theta(g(n))$.
3. $\omega(g(n)) = \Omega(g(n)) - \Theta(g(n))$.
4. Ejemplo:

$$O(n) = \{\underbrace{1, 2, \ldots, \log\log n, \ldots, \log n, \ldots, \sqrt{n}, n^{1/3}, \ldots}_{o(n)}, \underbrace{n, n+1, 2n, \ldots}_{\Theta(n)}\}$$

$$\Omega(n) = \{\underbrace{n, n+1, 2n, \ldots}_{\Theta(n)}, \underbrace{n\log\log n, \ldots, n\log n, \ldots, n\sqrt{n}, \ldots, n^2, \ldots}_{\omega(n)}\}$$

Análisis Asintótico

Propiedades de las Notaciones Asintóticas

1. Si $f(n) \in O(kg(n))$ para cualquier constante $k > 0$, entonces

$$f(n) \in O(g(n)).$$

2. Si $f_1(n) \in O(g_1(n))$ y $f_2(n) \in O(g_2(n))$, entonces

$$f_1(n) + f_2(n) \in O(\text{máx}\,(g_1(n), g_2(n))).$$

3. Si $f_1(n) \in O(g_1(n))$ y $f_2(n) \in O(g_2(n))$, entonces

$$f_1(n) \cdot f_2(n) \in O(g_1(n) \cdot g_2(n)).$$

Análisis Asintótico

Propiedades de las Notaciones Asintóticas

Muchas de las propiedades de números reales son válidas también para las notaciones asintóticas (asumiendo que $f(n)$ y $g(n)$ son asimptóticamente positivas).

Transitividad

1. $f(n) \in \Theta(g(n))$ y $g(n) \in \Theta(h(n))$ implica $f(n) \in \Theta(h(n))$.
2. $f(n) \in O(g(n))$ y $g(n) \in O(h(n))$ implica $f(n) \in O(h(n))$.
3. $f(n) \in \Omega(g(n))$ y $g(n) \in \Omega(h(n))$ implica $f(n) \in \Omega(h(n))$.
4. $f(n) \in o(g(n))$ y $g(n) \in o(h(n))$ implica $f(n) \in o(h(n))$.
5. $f(n) \in \omega(g(n))$ y $g(n) \in \omega(h(n))$ implica $f(n) \in \omega(h(n))$.

Análisis Asintótico

Propiedades de las Notaciones Asintóticas

Reflexividad

1. $f(n) \in \Theta(f(n))$.
2. $f(n) \in O(f(n))$.
3. $f(n) \in \Omega(f(n))$.

Simetría

1. $f(n) \in \Theta(g(n))$ si y sólo si $g(n) \in \Theta(f(n))$.

Simetría Transpuesta

1. $f(n) \in O(g(n))$ si y sólo si $g(n) \in \Omega(f(n))$.
2. $f(n) \in o(g(n))$ si y sólo si $g(n) \in \omega(f(n))$.

Análisis Asintótico

Notaciones Asintóticas versus Operadores Relacionales

Al cumplirse esas propiedades, podemos hacer una analogía entre la comparación asintótica de funciones f y g y números reales a y b:

O	Ω	Θ	o	ω
$\leq$	$\geq$	$=$	$<$	$>$

Sin embargo, hay una propiedad que no se cumple para las notaciones asintóticas:

Tricotomía: Para dos números reales a y b, exactamente una de las siguientes se cumple: $a < b$, $a = b$, o $a > b$.

Aunque dos números reales cualquiera pueden ser comparados, no todas las funciones son asintóticamente comparables.

Puede pasar que $f(n) \notin O(g(n))$ y $f(n) \notin \Omega(g(n))$, como es el caso de n y $n^{1+\sin n}$.

Análisis Asintótico

Clasificando Funciones

- Dadas dos funciones $f(n)$ y $g(n)$, usualmente uno quiere saber si una crece más rápido que la otra.
- La mejor manera de averiguar esto es mediante el límite:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = k,$$

distinguiendo los siguientes casos:

1. $k = \infty$, entonces $f(n) \in \Omega(g(n))$.
2. $k = 0$, entonces $f(n) \in O(g(n)) \ \wedge \ g(n) \notin O(f(n))$.
3. $k \neq 0$ y $k \neq \infty$, entonces $f(n) \in \Theta(g(n))$.

Análisis Asintótico

Cuándo usar una Notación u Otra

- Varios autores (por ejemplo, Knuth) reportan el uso indebido de las notaciones asintóticas.
- Es importante usar la notación adecuada en cada caso, de manera que exprese exactamente lo que se quiere decir.
- Al acotar el tiempo de ejecución de un algoritmo, Θ es en general preferible por sobre O, ya que entrega más información.
  - Sin embargo, no siempre es posible acotar usando Θ, como veremos en los siguientes ejemplos.

Análisis Asintótico

Cuándo usar una Notación u Otra

El siguiente algoritmo busca un elemento `x` dentro de un arreglo desordenado `A` de n elementos:

```
int buscarDesordenado(int *A, int n, int x) {
    int i;
    for (i=0; i < n; i++)
        if (A[i] == x)
            break;
    return i;
}
```

- El tiempo de búsqueda exitosa es $O(n)$.
- El tiempo de búsqueda infructuosa es $\Theta(n)$.

Análisis Asintótico

Cuándo usar una Notación u Otra

El siguiente algoritmo busca un elemento `x` dentro de un arreglo ordenado `A` de n elementos:

```
int buscarOrdenado(int *A, int n, int x) {
    int i;
    for (i=0; i < n; i++)
        if (A[i] >= x) break;
    if (i < n && A[i] == x) return i;
    else return n;
}
```

- El tiempo de búsqueda exitosa es $O(n)$.
- El tiempo de búsqueda infructuosa es $O(n)$.

Análisis Asintótico

Uso de Notaciones Asintóticas para Simplificar Partes de una Función

También se usan las notaciones asintóticas para simplificar sólo una parte de una función:

- En general cuando estamos interesados en las constantes de la parte principal de la función, mientras que otra parte de la misma puede ser simplificada.
- Por ejemplo, el polinomio $3n^3 + 5n^2 - 2n + 4$ puede escribirse como $3n^3 + O(n^2)$, o más precisamente $3n^3 + \Theta(n^2)$.

Análisis Asintótico

Uso de Notaciones Asintóticas para Simplificar Partes de una Función

- Otro ejemplo es $n + \frac{n}{\log n}$, que puede escribirse como $n + o(n)$, para destacar que la función es n más un término sublineal.
- Decir $n + \Theta(\frac{n}{\log n})$ es más preciso en ese caso, pero a veces uno quiere recalcar que el termino de orden inferior es sublineal y conviene usar la notacion o.

Análisis Asintótico

Uso de Notaciones Asintóticas para Simplificar Partes de una Función

- Por ejemplo, la expansión en series de Taylor de e^x es:

$$
\begin{aligned}
e^x = \sum_{n \geq 0} \frac{x^n}{n!} &= \frac{x^0}{0!} + \frac{x^1}{1!} + \frac{x^2}{2!} + \cdots + \frac{x^n}{n!} + \cdots \\
&= 1 + x + \frac{x^2}{2!} + \cdots + \frac{x^n}{n!} + \Theta(x^{n+1}).
\end{aligned}
$$

Outline

Análisis de Caso Promedio

Formas de Analizar Algoritmos

- Existen al menos 3 formas de analizar un algoritmo:
  1. Análisis de mejor caso: costo del caso que ejecuta menos instrucciones.
  2. Análisis de peor caso: costo del caso que ejecuta más instrucciones.
  3. Análisis de caso promedio: promedio del costo de todos los casos posibles.

- Si un algoritmo tiene costo de mejor caso $f(n)$ y peor caso $f(n)$, entonces decimos que el tiempo de ejecución del algoritmo (sin importar el caso) es $\Theta(f(n))$.

Análisis de Caso Promedio

- En general, un análisis de peor caso es adecuado y da suficiente información respecto al comportamiento de un algoritmo.
  - Es el tipo de análisis a elegir cuando uno quiere estar seguro de lo peor que puede ocurrir al ejecutar un algoritmo.
  - Hay aplicaciones para las que esto es imprescindible.
- Sin embargo, existen algoritmos que tienen un mal comportamiento de peor caso, pero que funcionan bien en la práctica (por ej., el peor caso es muy poco probable).
  - En esos casos, un análisis de caso promedio es más adecuado, y da una perspectiva de cómo se va a comportar un algoritmo en la práctica.
  - Un análisis de caso promedio suele ser un poco más complicado de realizar, ya que hay que considerar todos los posibles casos para luego promediarlos.

Análisis de Caso Promedio

Ejemplo 1: Análisis de caso promedio para el algoritmo de búsqueda de un elemento en un arreglo desordenado.

- Es importante considerar dos tipos de búsqueda: búsqueda exitosa y búsqueda no existosa.
- El caso de búsqueda no exitosa es simple, ya que toda búsqueda que falla debe recorrer todo el arreglo, con costo $\Theta(n)$.
- El costo promedio de búsqueda no exitosa es, por lo tanto, $\Theta(n)$.
- La búsqueda exitosa es más interesante de analizar, como vemos a continuación.

Análisis de Caso Promedio

- Considerar cada uno de los posibles casos de búsqueda exitosa.
- Son n en total (buscar cada uno de los elementos del arreglo).
- Medimos el costo en cantidad de celdas del arreglo consultadas (si medimos en cantidad de comparaciones, el análisis es similar).
- Notar que buscar el elemento que está en la posición i del arreglo cuesta i celdas consultadas.
- Entonces el costo promedio de búsqueda exitosa puede calcularse mediante:

$$\frac{1}{n}\sum_{i=1}^{n} i = \frac{1}{n} \cdot \frac{n(n+1)}{2} = \frac{n+1}{2} \in \Theta(n).$$

- Esto significa que en promedio debemos revisar la mitad del arreglo para una búsqueda exitosa.

Análisis de Caso Promedio

Ejercicio: ¿Qué ocurre con los costos promedio si ahora ordenamos el arreglo, y seguimos aplicando búsqueda secuencial?

Hacer el análisis para búsqueda exitosa y no exitosa, y concluir.

Análisis de Caso Promedio

- Hay otros ejemplos en donde hacer análisis de caso promedio no es tan simple.
- En general, debido a que la cantidad de casos a considerar para el promedio es muy grande.
- Es el caso de los árboles o estructuras de hashing.
- Nos dedicamos a continuación a estudiar el costo de búsqueda promedio en árboles.

Análisis de Caso Promedio

- Consideremos árboles binarios de búsqueda (ABB) de n nodos internos.
- Notar que la cantidad de nodos externos es $n + 1$.
- Sean $x_1, x_2, \ldots, x_n$ los nodos internos.
- Sean $\ell_1, \ell_2, \ldots, \ell_{n+1}$ los nodos externos.
- **Ejercicio:** Demostrar que la cantidad de nodos externos es $n + 1$. *Hint*: use inducción sobre el número de nodos internos.

Análisis de Caso Promedio

- Estudiaremos el costo promedio de búsqueda en un ABB.
- El costo de búsqueda exitosa en un ABB depende de la profundidad a la que se encuentre el nodo interno correspondiente.
- Por lo tanto, es $\Theta(n)$ en el peor caso.
  - Y el costo de búsqueda es $O(n)$ en general.

Análisis de Caso Promedio

- Para el costo promedio, hay que calcular cuál es la profundidad promedio de un nodo en un ABB.
- Sin embargo, no es tan sencillo, dado que hay que considerar todos los árboles binarios de n nodos, los cuales son (para $n \geq 0$):

$$C(n) = \frac{1}{n+1}\binom{2n}{n} \approx 4^n.$$

- Corresponde a los números de Catalan, cuya secuencia es: $1, 1, 2, 5, 14, 42, 132, \ldots$

Análisis de Caso Promedio

- Notar que hay $n!$ posibles formas de insertar los elementos en un árbol.
- ¿Por qué hay menos árboles resultantes?
- Definimos dos conceptos importantes para realizar el análisis de caso promedio en ABB T.
- **Longitud de caminos internos (internal path length)**

$$I_n = \sum_{i=1}^{n} \mathsf{depth}(x_i).$$

- **Longitud de caminos externos (external path length)**

$$E_n = \sum_{i=1}^{n+1} \mathsf{depth}(\ell_i).$$

Análisis de Caso Promedio

Teorema

Dado un árbol binario T de $n \geq 0$ nodos, se cumple que

$$E_n = I_n + 2n.$$

Demostración: Por inducción.
(**Caso base.**) $n = 0$, por lo tanto tenemos el árbol vacío, con 0 nodos internos y 1 nodo externo. En ese caso, $E_n = 0$, $I_n = 0$, y la relación propuesta se cumple.

Análisis de Caso Promedio

(**Inducción.**) Asumimos que el teorema es cierto para n. Sean T_1 y T_2 dos árboles binarios de hasta n nodos cada uno, y por ende para los que se cumple la HI.
Digamos que T_1 tiene $l \leq n$ nodos internos y T_2 tiene $r \leq n$ nodos internos. Entonces se cumple

$$E_l = I_l + 2l$$

y

$$E_r = I_r + 2r.$$

Análisis de Caso Promedio

Construimos a continuación un árbol binario T' agregando un nodo raíz como padre de T_1 y T_2, con $n' = l + r + 1$ nodos internos. Entonces tenemos:

$$\begin{aligned} E_{n'} - I_{n'} &= (E_l + l + 1 + E_r + r + 1) \\ &\quad -(I_l + l + I_r + r) \\ &= E_l + 1 + E_r + 1 - I_l - I_r \\ &\overset{h.i.}{=} I_l + 2l + 1 + I_r + 2r + 1 - I_l - I_r \\ &= 2l + 2r + 2 \\ &= 2(l + r + 1) = 2n'. \end{aligned}$$

Por lo tanto,

$$E_{n'} = I_{n'} + 2n'.$$

Análisis de Caso Promedio

- I_n y E_n son clave para calcular el costo promedio de búsqueda en ABB.
- **Costo Promedio de Búsqueda Exitosa**

$$C_n = 1 + \frac{I_n}{n}$$

- **Costo Promedio de Búsqueda Infructuosa**

$$C'_n = \frac{E_n}{n+1}.$$

- Dado que $E_n = I_n + 2n$, tenemos:

$$\begin{aligned} C_n &= 1 + \frac{E_n - 2n}{n} = \frac{E_n}{n} - 1 \\ &= \frac{n+1}{n+1}\frac{E_n}{n} - 1 \\ &= \frac{n+1}{n} C'_n - 1. \end{aligned}$$

Análisis de Caso Promedio

Por lo tanto, la ecuación que relaciona las búsquedas exitosas con las búsquedas infructuosas es:

$$C_n = \left(1 + \frac{1}{n}\right) C'_n - 1.$$

- Notar que a medida que se insertan más elementos en el árbol (n es más grande), los costos de búsqueda exitosa e infructuosa son cada vez más parecidos.
- Vamos a demostrar que si los elementos son insertados de formar aleatoria en un ABB, entonces los árboles balanceados son comunes, y los árboles de peor caso son raros.

Análisis de Caso Promedio

- Asumimos que cada una de las $n!$ formas distintas posibles de insertar los elementos en el ABB son igualmente probables.
- El número de comparaciones necesarias para encontrar un elemento es exactamente uno más que el número de comparaciones necesarios para insertar ese elemento en el árbol.
- En otras palabras, el costo de búsqueda de un elemento en un ABB es igual al costo de búsqueda infructuosa justo antes de insertarlo más 1.
- Esto quiere decir que si ya habían k elementos en el árbol y se inserta uno más, el costo esperado de búsqueda para este último es $1 + C'_k$.

Análisis de Caso Promedio

- Si se busca (de forma exitosa) un elemento en un árbol de n nodos, tenemos que considerar los siguientes casos:
  - Que el elemento puede haber sido insertado con costo promedio $(1 + C'_0)$ en un árbol vacío.
  - O con costo promedio $(1 + C'_1)$ en un árbol con 1 nodo.
  - $\vdots$
  - O con costo promedio $(1 + C'_{n-1})$ en un árbol con $n - 1$ nodos.
- Para calcular el costo promedio de búsqueda exitosa, debemos considerar esos n posibles casos, y promediarlos, tal como en la siguiente ecuación.

Análisis de Caso Promedio

Por lo tanto:

$$\begin{aligned} C_n &= \frac{(1+C'_0)+(1+C'_1)+\cdots+(1+C'_{n-1})}{n} \\ &= 1+\frac{C'_0+C'_1+\cdots+C'_{n-1}}{n}. \end{aligned} \tag{1}$$

Pero la relación entre caminos internos y externos (ya demostrada) nos dice que:

$$C_n = \left(1+\frac{1}{n}\right) C'_n - 1. \tag{2}$$

Igualando las ecuaciones (1) y (2) produce:

$$(n+1)C'_n = 2n + C'_0 + C'_1 + \cdots + C'_{n-1}. \tag{3}$$

Análisis de Caso Promedio

Esta recurrencia es fácil de resolver, restando:

$$nC'_{n-1} = 2(n-1) + C'_0 + C'_1 + \cdots + C'_{n-2}, \tag{4}$$

obteniendo:

$$(n+1)C'_n - nC'_{n-1} = 2 + C'_{n-1}. \tag{5}$$

Por lo tanto

$$C'_n = C'_{n-1} + \frac{2}{n+1}. \tag{6}$$

Notar cómo el costo de búsqueda infructuosa crece cada vez más lentamente a medida que crece n.

Análisis de Caso Promedio

Dado que $C'_0 = 0$, tenemos la siguiente ecuación de recurrencia:

$$C'_n = \begin{cases} C'_{n-1} + \frac{2}{n+1} & \text{si } n > 0 \\ 0 & \text{si } n = 0 \end{cases}$$

Esta recurrencia es fácil de resolver, ya que

$$\begin{aligned} C'_n &= C'_{n-1} + \frac{2}{n+1} \\ &= C'_{n-2} + \frac{2}{n} + \frac{2}{n+1} \\ &\vdots \\ &= C'_0 + \frac{2}{2} + \cdots + \frac{2}{n+1} \\ &= 2\sum_{k=2}^{n+1} \frac{1}{k} = 2(H_{n+1} - 1). \end{aligned} \tag{7}$$

Análisis de Caso Promedio

En donde

$$H_n = \sum_{k=1}^{n} \frac{1}{k}$$

es el n-ésimo número armónico.

Se sabe que

$$\ln(n+1) \leq H_n \leq \ln(n) + 1.$$

Entonces tenemos que el costo promedio de una búsqueda infructuosa es

$$C'_n = 1{,}386 \lg(n+1) \in O(\lg n)$$

.

Análisis de Caso Promedio

Para las búsquedas exitosas, reemplazamos el resultado de (7) en (2):

$$C_n = 2\left(1 + \frac{1}{n}\right) H_n - 3. \tag{8}$$

Entonces, tenemos que

$$C_n = 1{,}386 \lg n \in O(\lg n).$$

Análisis de Caso Promedio

Ejercicio: acotar I_n y E_n, usando notación asintótica y los resultados anteriores.

Outline

Análisis Amortizado

- Análisis de peor caso es pesimista en ciertas ocasiones.
- Análisis de caso promedio ayuda cuando el comportamiento de peor caso es poco probable.
  - Se hacen suposiciones sobre el algoritmo.
- En un *análisis amortizado*, el tiempo requerido para llevar a cabo una secuencia de operaciones sobre una estructura de datos es promediado sobre todas las operaciones realizadas.
- Se usa para mostrar que el costo promedio de una operación es pequeño, aún cuando una única ejecución de la operación podría ser costosa.
- Se diferencia del análisis de caso promedio en que no entran en juego las distribuciones de probabilidad.
  - Garantiza el comportamiento promedio de cada operación en el peor caso.

Análisis Amortizado: Método de Agregación

- Primero se muestra que, para todo n, una secuencia de n operaciones sobre una estructura de datos toma tiempo de peor caso $T(n)$.
- Luego, podemos concluir que en el peor caso el costo amortizado por operación es $T(n)/n$.
- Este costo amortizado aplica para todas las operaciones realizadas en la secuencia, aún cuando puede haber diferentes tipos de operaciones realizadas.

Análisis Amortizado: Método de Agregación

Ejemplo

Operaciones sobre un stack: Supongamos que tenemos stacks con las operaciones típicas Push y Pop. Cada una de esas operaciones toma tiempo $O(1)$, asi que asumimos que su costo es simplemente 1.

Definimos además la operación:

MultiPop(S, k)

While not empty(S) and $k \neq 0$ **do**

Pop(S)

$k \leftarrow k - 1$

El tiempo de ejecución de MultiPop en un stack de s elementos es $\min(s, k)$.

Análisis Amortizado: Método de Agregación

- Supongamos ahora una secuencia de n operaciones Push, Pop, y MultiPop.
- Ya que el tamaño de la pila será a lo sumo n, el costo de peor caso de MultiPop es $O(n)$.
- Entonces, una secuencia de n operaciones tiene costo total de peor caso $O(n^2)$.
- Aunque el análisis es correcto, no es ajustado.

Análisis Amortizado: Método de Agregación

- Para un análisis ajustado, notar que cada elemento puede ser sacado de la pila a lo más una única vez después que es colocado en ésta.
- El número de veces que Pop puede invocarse sobre una pila no vacía (incluyendo las invocaciones desde MultiPop) es a lo sumo el número que operaciones Push ejecutadas (a lo más n).
- Es decir, cualquier secuencia de n operaciones Push, Pop y MultiPop tiene costo total $O(n)$.
- Por lo tanto, el costo amortizado por operación es $O(n)/n = O(1)$.
- Esto significa que aunque MultiPop puede ser una operación costosa, al ser usada en una secuencia de operaciones su costo amortizado es bajo.

Análisis Amortizado: Método de Agregación

Ejemplo (Incrementando un contador binario)

- Consideremos incrementar un contador binario de k bits, inicializado en 0.
- Nuestro contador será el arreglo $A[0..k-1]$, inicializado todo en 0.
- $A[0]$ es el bit menos significativo, $A[k-1]$ es el bit más significativo.

Increment$(A[0..k-1])$
  $i \leftarrow 0$
  While $i < k$ and $A[i] = 1$ **do**
    $A[i] \leftarrow 0$
    $i \leftarrow i + 1$
  if $i < k$ **then** $A[i] \leftarrow 1$

El costo del algoritmo está dado por la cantidad de bits del contador que cambian su valor (flips) en el incremento.

Análisis Amortizado: Método de Agregación

- Ejemplo con un contador de 8 bits.
- En el peor caso un incremento toma tiempo $O(k)$.
- Por lo tanto, el costo total de peor caso para n incrementos sería $O(nk)$.
- Nuevamente, el análisis es adecuado, pero no es ajustado.

Counter value	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Total cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

Análisis Amortizado: Método de Agregación

- Para ajustar nuestro análisis, hay que notar que no todos los bits hacen flip cuando se hace un incremento.
- En particular:
  - $A[0]$ cambia de valor siempre (n en total)
  - $A[1]$ cambia de valor cada 2 incrementos ($\lfloor n/2 \rfloor$ en total)
  - $A[2]$ cambia de valor cada 4 incrementos ($\lfloor n/4 \rfloor$ en total)
  - En general $A[i]$ se cambia $\lfloor n/2^i \rfloor$ veces en total, para $i = 0, \ldots, \lfloor \lg n \rfloor$.

Análisis Amortizado: Método de Agregación

- El número total de flips en la secuencia de n incrementos es

$$\sum_{i=0}^{\lfloor \lg n \rfloor} \left\lfloor \frac{n}{2^i} \right\rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n.$$

- Entonces, el costo total de peor caso para los n incrementos es $O(n)$, lo que da costo total amortizado de $O(n)/n = O(1)$ por incremento.

Análisis Amortizado: Método de Agregación

- Suponga que se debe recibir y almacenar un conjunto de datos por ráfagas (streaming).
- Los datos deben almacenarse recordando el orden en el que van llegando. Para eso se usa un arreglo.
- Sin embargo, no se conoce de antemano la cantidad total de elementos que deben almacenarse, por lo que el arreglo debe crecer dinámicamente (realloc).
- Suponiendo que la operación realloc se puede ejecutar en tiempo de peor caso O(1) por elemento del arreglo actual:

¿Qué estrategia usaría para lograr tiempo amortizado O(1) por operación?

Outline

Cotas Inferiores

- En la mayoría de las clases del curso, estaremos preocupados en encontrar algoritmos eficientes para resolver un problema en particular.
- También, en analizar y comparar dichos algoritmos.
- Sin embargo, en la algoritmia es también importante decir que ciertos problemas no pueden ser resueltos tan eficientemente como uno quisiera.
  - En muchos casos permite entender que ciertos algoritmos son lo más eficiente que se puede hacer para el problema.

Cotas Inferiores

- Sea $T_A(X)$ el tiempo de ejecución de un algoritmo A sobre una entrada X.
- El tiempo de ejecución de peor caso para A se define como:

$$T_A(n) \equiv \max_{|X|=n} T_A(X).$$

- Definimos la complejidad de peor caso de un problema Π como el tiempo de ejecución de peor caso del algoritmo más eficiente que lo resuelve:

$$T_\Pi(n) \equiv \min_{A \text{ resuelve } \Pi} T_A(n) = \min_{A \text{ resuelve } \Pi} \max_{|X|=n} T_A(X)$$

- Cualquier algoritmo A que resuelva Π inmediatamente implica una cota superior para la complejidad de Π.

Cotas Inferiores

- Algoritmos más rápidos nos dan mejores cotas superiores a un problema.
- El enfoque es que estamos mostrando que un problema es cada vez más fácil de resolver (dado que bajamos su cota superior).
- Ahora veremos el enfoque inverso: argumentaremos que ciertos problemas son difíciles, demostrando cotas inferiores para su complejidad.

Cotas Inferiores

- Lamentablemente es un poco más complejo que para el caso de las cotas superiores.
  - Determinar cotas inferiores para un problema consiste en obtener cuál es la eficiencia mínima que debe tener **cualquier** algoritmo que resuelva el problema.

- Para probar que

$$T_{\Pi}(n) \in \Omega(f(n))$$

debemos probar que todo algoritmo que resuelve Π tiene un tiempo de ejecución de peor caso que es $\Omega(f(n))$.

Cotas Inferiores

- Conocer la cota inferior para un problema, permite saber cuál es la mejora que podemos esperar hacer a los algoritmos para resolver el problema.
- Decimos que la cota inferior es ajustada cuando coincide con el costo de peor caso de uno de los algoritmos que resuelve el problema.
  - El hecho de existir un algoritmo cuya eficiencia coincide con esa cota nos dice que la cota inferior no puede mejorarse (de otra manera, el tiempo de ejecución del algoritmo sería menor que la cota, lo que es una contradicción).

Cotas Inferiores

- Siempre que exista una brecha entre la cota inferior de un problema y el más eficiente de los algoritmos que lo resuelva, puede significar dos cosas:
  - Que quizás el mejor de los algoritmos aún puede ser mejorado para obtener una solución más eficiente.
  - Que quizás la cota inferior puede aún ser mejorada.
- La búsqueda concluye cuando ambas cotas son iguales (la cota es ajustada, y ya no se pueden introducir mejoras, sólo mejoras constantes).

Cotas Inferiores: Árboles de Decisión

- Desafortunadamente, no es fácil ni práctico considerar todos los posibles algoritmos para resolver un problema dado.
- Al probar cotas inferiores, primero debemos especificar precisamente qué clase de algoritmos consideraremos, y precisamente cómo medir el tiempo de ejecución.
- Esta especificación es llamada el **modelo de computación**.
- En particular, estudiaremos el modelo de *árbol de decisión*.

Cotas Inferiores: Árboles de Decisión

- Muchos algoritmos importantes funcionan comparando elementos de la entrada:
  - Algoritmos de búsqueda y de ordenamiento.
- Un árbol de decisión binario es un modelo de computación en el cual un *algoritmo particular* es representado por un árbol con las siguientes características:
  - Nodos internos: representan comparaciones de elementos hechas por el algoritmo.
  - Subárbol izquierdo: representa las comparaciones a realizar si la comparación del nodo actual resulta ser falsa.
  - Subárbol derecho: idem al anterior, pero si la comparación del nodo actual resulta ser verdadera.
  - Nodos externos: representan posibles salidas del algoritmo.

Cotas Inferiores: Árboles de Decisión

- Notar que la cantidad de nodos externos del árbol debe ser al menos el número de posibles salidas del algoritmo (pero pueden ser más, ya que una misma salida puede ser derivada por múltiples caminos en el árbol).
- La ejecución del algoritmo sobre una entrada en particular corresponde a un camino desde la raíz del árbol hasta el nodo externo correspondiente.
- Se define el tiempo de ejecución de un algoritmo representado como árbol de decisión para una entrada dada, como el largo del camino realizado desde la raíz del árbol hasta el nodo externo resultante.
- El tiempo de ejecución de peor caso es, por lo tanto, la altura del árbol de decisión.

Cotas Inferiores: Árboles de Decisión

- La idea central detrás del modelo de árboles de decisión es la siguiente:
  - Determinar el número de nodos externos l que debe tener cualquier árbol que resuelve el problema en cuestión.
  - Un árbol con l nodos externos (posibles salidas) tiene que ser lo suficientemente alto para tener ese número nodos externos.
- En particular, se cumple que para cualquier árbol binario con l nodos externos y altura h,

$$h \geq \lceil \log_2 l \rceil.$$

- Esto es, el número de nodos externos define una cota inferior respecto a la altura del árbol.
  - Esto define una cota inferior para el número de comparaciones en peor caso para cualquier algoritmo para el problema en cuestión.

Cotas Inferiores: el Problema de Ordenamiento

- Dado un arreglo $A[1..n]$ de números enteros, se necesita ordenarlo de manera tal que:

$$A[1] < A[2] < \ldots < A[n].$$

- La mayoría de los algoritmos de ordenamiento están basados en comparaciones entre los elementos.
- Usaremos árboles de decisión para modelarlos y obtener cotas inferiores para el problema de ordenamiento.
- Un algoritmo de ordenamiento debe encontrar una permutación de los elementos del arreglo en la cual los elementos estén ordenados ascendentemente.

Cotas Inferiores: el Problema de Ordenamiento

- Consideremos, como ejemplo, un arreglo de tres elementos, a, b, y c, sobre los que se ha definido una relación de orden (por ejemplo, números enteros y el orden $<$).
- Si la salida es $a < c < b$, la permutación resultante es 1, 3, 2.
- Un posible algoritmo de ordenamiento para tres elementos usando comparaciones es representado por el árbol de decisión binario:

Cotas Inferiores: el Problema de Ordenamiento

- En general, para n elementos, hay $n!$ posibles permutaciones.
- Eso significa que el árbol de decisión debe tener altura

$$h \geq \log_2 n!.$$

- Esa es la cota inferior para el problema de ordenamiento.
- Usando la fórmula de Stirling para $n!$, tenemos que:

$$\begin{aligned} \lceil \log_2 n! \rceil &\approx \log_2 \left(\sqrt{2\pi n} \left(\frac{n}{e} \right)^n \right) \\ &= n \log_2 n - n \log_2 e + \frac{\log_2 n}{2} + \frac{\log_2 2\pi}{2} \\ &\approx n \log_2 n. \end{aligned}$$

Cotas Inferiores: el Problema de Ordenamiento

- Esto significa que en el peor caso, un algoritmo de ordenamiento basado en comparaciones de pares de elementos no puede realizar menos de (aproximadamente) $n \log_2 n$ comparaciones.
- Es decir, la complejidad de peor caso del **problema** de ordenamiento es $\Omega(n \log_2 n)$.
- Veremos en el curso algoritmos que logran esta complejidad, por lo cual esta cota inferior es ajustada (no puede ser mejorada).

Cotas Inferiores: el Problema de Ordenamiento

- También se podría calcular la cota inferior en promedio, calculando la profundidad promedio de los nodos externos del árbol de decisión
  - Coincide con la profundidad promedio de un nodo externo en árboles binarios, ya estudiada anteriormente, el cual es $O(\log_2 n)$.
- Se debe asumir que cualquiera de las $n!$ posibles permutaciones son igualmente probables.
- Se llega a que el la cota inferior en caso promedio es también $\Omega(n \log_2 n)$.

Cotas Inferiores: el Problema de Búsqueda en Arreglo Ordenado

- Otro problema importante es el de buscar un elemento x en un arreglo ordenado de n elementos.
- Cualquier algoritmo de búsqueda sobre un arreglo ordenado puede ser modelado como un árbol de decisión binario.
- Nodos internos representan comparación de x con $A[i]$.
- Se sigue la rama izquierda del árbol si $x < A[i]$.
- Se sigue la rama derecha del árbol si $A[i] < x$.
- La búsqueda termina en nodo interno si $A[i] = x$.
- Búsqueda infructuosa termina en un nodo externo del árbol.

Cotas Inferiores: el Problema de Búsqueda en Arreglo Ordenado

- En base a esto, ¿Cuántos algoritmos de búsqueda sobre arreglo ordenado distintos tenemos?
- Es igual a la cantidad de árboles binarios distintos con n nodos (el número de Catalan).
- Notar que la cantidad de nodos externos que tiene nuestro árbol de decisión es $n+1$ (los posibles casos de búsqueda infructuosa).
- Entonces, la altura del árbol es al menos $\lceil \log_2 (n+1) \rceil$.
- Esto significa que cualquier algoritmo de búsqueda sobre un arreglo ordenado no puede hacer menos de esa cantidad de comparaciones.
- ¿Será esta cota ajustada?

Cotas Inferiores: el Problema de Búsqueda en Arreglo Ordenado

- Se puede probar que en promedio la cota inferior también es $\Omega(\log_2 n)$.
- ¿Por qué las técnicas de hashing permiten buscar en tiempo promedio $\Theta(1)$?
  - ¿Es esto una contradicción?

Outline

Ecuaciones de Recurrencia

- Aparecen al momento de analizar algoritmos recursivos.
  - El tiempo de ejecución es expresado de la forma

$$T(n) = E(n)$$

  - En la expresión $E(n)$ aparece la propia función T.
- Es decir, $T(n)$ es definida en términos de sí misma.
- Además de la definición recurrente, se deben especificar los casos conocidos para T
  - Son los casos de borde, o condiciones iniciales.

Ecuaciones de Recurrencia

- Por ejemplo

$$T(n) = \begin{cases} T(n-1) + T(n-2) & \text{si } n \geq 2 \\ 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \end{cases}$$

- Una ecuación de recurrencia define una secuencia de números enteros $\langle a_0, a_1, \ldots a_i, \ldots \rangle$ tal que $a_0 = T(0)$, $a_1 = T(1)$, . . ., $a_i = T(i)$, . . .
- La ecuación del ejemplo anterior define la secuencia de Fibonacci.
- Resolver una ecuación de recurrencia consiste en encontrar una expresión no recursiva de T.

Ecuaciones de Recurrencia Lineales Homogéneas

- Tienen la forma

$$a_0T(n) + a_1T(n-1) + \cdots + a_kT(n-k) = 0.$$

- En donde $a_0, \ldots, a_k \in \mathbb{R}$ son constantes, $a_i \neq 0$, y $1 \leq k \leq n$.
- Para resolverlas se hace el cambio:

$$T(n) = x^k,$$

obteniendo la *ecuación característica* asociada

$$a_0x^k + a_1x^{k-1} + \cdots + a_k = 0.$$

- Sean $r_1, \ldots, r_k$ las raíces del polinomio.
- Dependiendo de su multiplicidad tenemos dos casos.

Ecuaciones de Recurrencia Lineales Homogéneas

Caso 1: Raíces Distintas

- Si todas las raíces de la ecuación característica son distintas, la solución de la ecuación de recurrencia es

$$T(n) = c_1 r_1^n + c_2 r_2^n + \cdots + c_k r_k^n,$$

en donde los coeficientes c_i se determinan a partir de las condiciones iniciales de la definición.

- Por ejemplo

$$T(n) = \begin{cases} T(n-1) + T(n-2) & \text{si } n \geq 2 \\ 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \end{cases}$$

Ecuaciones de Recurrencia Lineales Homogéneas

Caso 2: Raíces con Multiplicidad Mayor a 1

- Si $r_1, r_2, \ldots, r_s$ $(s \leq k)$ son las raíces distintas de la ecuación característica de una ecuanción de recurrencia homogénea lineal, cada una con multiplicidad $m_1, m_2, \ldots, m_s$, entonces la ecuación característica puede expresarse como

$$(x - r_1)^{m_1} \times (x - r_2)^{m_2} \times \cdots \times (x - r_s)^{m_s} = 0.$$

- La solución a la ecuación de recurrencia es

$$T(n) = r_1^n \sum_{i=1}^{m_1} c_{1i} n^{i-1} + \cdots + r_s^n \sum_{i=1}^{m_s} c_{si} n^{i-1}.$$

- Nuevamente los coeficientes c_{ij} se obtienen mediante las condiciones iniciales.

Ecuaciones de Recurrencia Lineales Homogéneas

Caso 2: Raíces con Multiplicidad Mayor a 1

- Por ejemplo

$$T(n) = \begin{cases} 5T(n-1) - 8T(n-2) + 4T(n-3), & n \geq 2 \\ 0 & n = 0 \\ 1 & n = 1 \\ 2 & n = 2 \end{cases}$$

- Otro ejemplo

$$T(n) = \begin{cases} 4T(n-1) - 4T(n-2), & n \geq 2 \\ 0 & n = 1 \\ 1 & n = 1 \end{cases}$$

Ecuaciones de Recurrencia Lineales No Homogéneas

- Tienen la forma

$$a_0T(n) + a_1T(n-1) + \cdots + a_kT(n-k) = b^n p(n).$$

- En donde $a_0, \ldots, a_k, b \in \mathbb{R}$ son constantes, $a_i \neq 0$, $1 \leq k \leq n$, $p(n)$ es un polinomio en n de grado d.
- Para esta ecuación de recurrencia se define la ecuación característica:

$$\left(a_0x^k + a_1x^{k-1} + \cdots + a_k\right)(x-b)^{d+1} = 0.$$

- Se procede de forma similar a los casos anteriores, encontrando las raices $r_1, \ldots, r_s$ del polinomio $a_0x^k + a_1x^{k-1} + \cdots + a_k$.

Ecuaciones de Recurrencia Lineales Homogéneas

- Por ejemplo, la recurrencia para el algoritmo de las torres de Hanoi.

$$T(n) = \begin{cases} 2T(n-1)+1, & n \geq 2 \\ \\ 1 & n = 1 \end{cases}$$