Pontificia Universidad Católica de Chile
Facultad de Ingenieria
Departamento de Ciencias de la Computación
IIC2283 - Diseño y Análisis de Algoritmos

Ayudantía 3

31 de agosto de 2018

Ayudante: Andrés Ríos

Problema 1

Se decide reemplazar a una persona en cierto cargo en una empresa. Para esto se entrevista a n candidatos uno a uno, y si el candidato es mejor que el empleado contratado en ese momento, entonces el empleado se despide y se contrata al candidato. En otras palabras, el algoritmo es:

```
function Contratar(n)
    for i = 1...n do
        Entrevistar candidato i
        if Candidato i es mejor que empleado actual then
            Despedir empleado actual
            Contratar candidato i
```

El proceso de despedir al empleado actual y contratar a uno nuevo tiene costo c. Se busca analizar el costo del algoritmo `Contratar` en el mejor caso, peor caso y en el caso promedio.

Asuma que para todo par de empleados i y j, el empleado i es mejor a j o viceversa (no hay dos candidatos iguales). Además, el empleado inicial es peor que cualquier candidato y los candidatos están ordenados de manera aleatoria.

Solución:

El mejor caso es que el primer candidato sea el mejor, en cual caso el costo total es c. El peor caso es que vengan ordenados de peor a mejor, en cual caso se contratan todos los candidatos y el costo total es cn.

Si X personas son contratadas, entonces la cantidad promedio de personas contratadas es

$$\mathbb{E}[X] = \sum_{x=1}^{n} x \cdot Pr[X = x]$$

Resolver esto no sería trivial. Definamos variables auxiliares para simplificar el cálculo:

$$X_i = \begin{cases} 1 & \text{si el candidato } i \text{ es contratado} \\ 0 & \text{en caso contrario} \end{cases}$$

Así, tenemos que $X = X_1 + X_2 + ... + X_n$. Con esto, tenemos que

$$\begin{aligned}\mathbb{E}[X] &= \mathbb{E}\left[\sum_{i=1}^{n} X_i\right] \\ &= \sum_{i=1}^{n} \mathbb{E}[X_i] \\ &= \sum_{i=1}^{n} Pr[X_i = 1]\end{aligned}$$

Notar que si el candidato i es contratado, eso significa que es mejor que los candidatos $1, ..., i-1$. En otras palabras, el candidato i es el mejor entre los candidatos $1, ..., i$. Dado que los candidatos llegan en orden aleatorio, cualquiera de los primeros i candidatos es igualmente probable de ser el mejor entre los primeros i candidatos. Por lo tanto, la probabilidad de que el candidato i sea el mejor es $\frac{1}{i}$. Con esto tenemos que

$$\begin{aligned}\mathbb{E}[X] &= \sum_{i=1}^{n} Pr[X_i = 1] \\ &= \sum_{i=1}^{n} \frac{1}{i} \\ &= \log(n) + O(1)\end{aligned}$$

Por lo tanto, en promedio se contratan $O(\log(n))$ candidatos. Dado que contratar a cada candidato cuesta c, el costo promedio del algoritmo es $O(c\log(n))$.

Problema 2

Se busca analizar la complejidad de BucketSort. El arreglo A de entrada contiene números entre el 0 y el 1.

function BucketSort(A[0...n-1])
  for $i = 0...n$ **do**
    $B[i] \leftarrow \emptyset$
  for $i = 0...n$ **do**
    Insertar $A[i]$ en la lista $B[\lfloor A[i] \cdot n \rfloor]$
  for $i = 0...n$ **do**
    Ordenar $B[i]$ con $InsertionSort$
  return Concatenación de B en orden

Demuestre que si los valores de A distribuyen uniforme entre 0 y 1, entonces BucketSort funciona en $\Theta(n)$ en promedio.

Solución:

Sea n_i una variable aleatoria que indica la cantidad de elementos en el bucket i. Como $InsertionSort$ funciona en tiempo $O(n^2)$, entonces el tiempo de $BucketSort$ está dado por (relajando la notación):

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)$$

Luego, el tiempo promedio del algoritmo es

$$\begin{aligned}\mathbb{E}\left[T(n)\right] &= \mathbb{E}\left[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)\right] \\ &= \Theta(n) + \sum_{i=0}^{n-1} \mathbb{E}\left[O(n_i^2)\right] \\ &= \Theta(n) + \sum_{i=0}^{n-1} O(\mathbb{E}\left[n_i^2\right])\end{aligned}$$

Necesitamos resolver $\mathbb{E}\left[n_i^2\right]$. Para esto se define el conjunto de variables aleatorias X_{ij}, donde

$$X_{ij} = \begin{cases} 1 & A[j] \text{ se asignó al bucket } B[i] \\ 0 & \text{en otro caso} \end{cases}$$

Se tiene que

$$\begin{aligned}\mathbb{E}\left[n_i^2\right] &= \mathbb{E}\left[\left(\sum_{j=0}^{n-1} X_{ij}\right)^2\right] \\ &= \mathbb{E}\left[\sum_{j=0}^{n-1} X_{ij}^2 + \sum_{j=0}^{n-1}\sum_{\substack{k=0 \\ k\neq j}}^{n-1} X_{ij}X_{ik}\right] \\ &= \sum_{j=0}^{n-1} \mathbb{E}\left[X_{ij}^2\right] + \sum_{j=0}^{n-1}\sum_{\substack{k=0 \\ k\neq j}}^{n-1} \mathbb{E}\left[X_{ij}X_{ik}\right]\end{aligned}$$

Luego,

$$\mathbb{E}\left[X_{ij}^2\right] = 1^2 \cdot Pr[X_{ij} = 1] + 0^2 \cdot Pr[X_{ij} = 0] = \frac{1}{n}$$

Por el otro lado, si $j \neq k$ entonces X_{ij} es independiente de X_{ik}, por lo que

$$\mathbb{E}\left[X_{ij}X_{ik}\right] = \mathbb{E}\left[X_{ij}\right]\mathbb{E}\left[X_{ik}\right] = \frac{1}{n^2}$$

Reemplazamos:

$$\begin{aligned}\mathbb{E}\left[n_i^2\right] &= \sum_{j=0}^{n-1} \mathbb{E}\left[X_{ij}^2\right] + \sum_{j=0}^{n-1}\sum_{\substack{k=0 \\ k\neq j}}^{n-1} \mathbb{E}\left[X_{ij}X_{ik}\right] \\ &= \sum_{j=0}^{n-1} \frac{1}{n} + \sum_{j=0}^{n-1}\sum_{\substack{k=0 \\ k\neq j}}^{n-1} \frac{1}{n^2} \\ &= \frac{1}{n}\sum_{j=0}^{n-1} 1 + \frac{1}{n^2}\sum_{j=0}^{n-1}\sum_{\substack{k=0 \\ k\neq j}}^{n-1} 1 \\ &= \frac{n}{n} + \frac{n(n-1)}{n^2} \\ &= 2 - \frac{1}{n}\end{aligned}$$

Por lo tanto,

$$\mathbb{E}\left[T(n)\right] = \Theta(n) + O(n(2 - 1/n)) \in \Theta(n)$$

Problema 3

Sea $\mathbb{M}$ la clase de matrices cuadradas $A_{n\times n}$ de números enteros tales que:

- para cada $i \in \{1, ..., n-1\}$ y $j \in \{1, ..., n\}$, se tiene que $A[i,j] \leq A[i+1,j]$
- para cada $i \in \{1, ..., n\}$ y $j \in \{1, ..., n-1\}$, se tiene que $A[i,j] \leq A[i,j+1]$

El problema es: dada una matriz $A \in \mathbb{M}$ y un número entero a, determinar la posición (i,j) tal que $A[i,j] = 1$ o retornar no si tal posición no existe. La operación básica a contar es el acceso a una posición de la matriz A.

- Encuentre una cota inferior para el número de accesos a las posiciones de A que debe realizar un algoritmo que resuelve el problema
- Construya un algoritmo que resuelva el problema y alcance la cota inferior anterior.

Solución:

- Para encontrar una cota inferior se puede utilizar la técnica del adversario. Se construye la matriz $A[1...n, 1...n]$ de la siguiente forma:

$$A[i,j] = \begin{cases} a-1 & i+j < n+1 \\ a+1 & i+j \geq n+1 \end{cases}$$

  Luego si el algoritmo revisa menos de $2n-1$ posiciones, entonces no revisó todas las posiciones (i,j) tales que $i+j = n$ o $i+j = n+1$. Luego si el algoritmo responde que a está en alguna posición, el adversario entrega la matriz A como está descrita anteriormente, con lo que el algoritmo se equivoca. Si el algoritmo responde que a no está en la matriz, entonces el adversario puede asignar a en alguna posición (i,j) no consultada por el algoritmo tal que $n \leq i+j \leq n+1$, generando una matriz M consistente con todas las observaciones del algoritmo pero que hace que la respuesta entregada sea incorrecta. Luego, todo algoritmo correcto debe revisar por lo menos $2n-1$ posiciones en alguna instancia del problema.

- El algoritmo es simple:

```
function ENCONTRAR_A(A)
    i = 1
    j = n
    while i ≤ n ∧ j ≥ 1 do
        x = A[i, j]
        if x == a then return (i, j)
        if x < a then j = j − 1
        else i = i + 1
    return None
```

  Si se revisa la posición (i',j') y el elemento encontrado es menor a a, entonces a no puede estar en una posición con $j \geq j'$. Similarmente, si el elemento es mayor a a entonces a no puede estar en una posición con $i \leq i'$. Por lo tanto, el algoritmo es correcto. Además, el loop se repite a lo más $2n-1$ veces, por lo que se alcanza la cota inferior encontrada en el punto anterior.

Problema 4

Los árboles de decisión se pueden generalizar en árboles de decisión lineales, que representan funciones de $\mathbb{R}^n$ a $\{0,1\}$. Estos árboles son ternarios, donde cada nodo interno está etiquetado con un vector $v_i \in \mathbb{R}^n$, cada arco de salida está etiquetado con un elemento del conjunto $\{-1,0,1\}$, y dos arcos distintos que salen del mismo nodo tienen etiquetas distintas. Además, cada hoja está etiquetada con un valor en $\{0,1\}$.

Dado un árbol y un vector x, se calcula el valor de la función comenzando desde la raíz y siguiendo el arco de salida que corresponda según:

$$\begin{cases} -1 & \langle x, v_i \rangle < 0 \\ 0 & \langle x, v_i \rangle = 0 \\ 1 & \langle x, v_i \rangle > 0 \end{cases}$$

hasta alcanzar una hoja. El valor de la función con ese input es la etiqueta de la hoja.

Además, definimos que un conjunto $C \subseteq \mathbb{R}^n$ es convexo, si para todos $a, b \in C$ y $\lambda \in [0, 1]$, se tiene que $\lambda a + (1 - \lambda)b \in C$.

1. Muestre que la intersección de conjuntos convexos es un conjunto convexo

   Solución:

   Sean C_1 y C_2 conjuntos convexos.

   p.d. $C_1 \wedge C_2$ es convexo

   Sean $x_1, x_2 \in C_1 \wedge C_2$. Tenemos que $\lambda x_1 + (1 - \lambda)x_2 \in C_1$ pues $x_1, x_2 \in C_1$ y C_1 es convexo. Además, $\lambda x_1 + (1 - \lambda)x_2 \in C_2$ pues $x_1, x_2 \in C_2$ y C_2 también es convexo. Por lo tanto, $\lambda x_1 + (1 - \lambda)x_2 \in C_1 \wedge C_2$. De esto se concluye que $C_1 \wedge C_2$ es convexo.

2. Dado un nodo v del árbol, denotamos por $P(v)$ al conjunto de vectores cuya evaluación pasa por el nodo v. Demuestre que para todo v, $P(v)$ es un conjunto convexo.

   Idea de Solución:

   Se hace inducción en la altura. Para la raiz r, $P(r)$ es igual a $\mathbb{R}^n$, lo que es convexo. Para otros casos, si el padre de v es t, entonces $P(v)$ es igual a la intersección entre $P(t)$ y $\{s | \langle t, s \rangle \star 0\}$, donde $\star$ es $<$,$>$ o $=$ dependiendo del caso. $P(t)$ es convexo por la hipótesis de inducción, y es fácil demostrar que $\{s | \langle t, s \rangle \star 0\}$ también es convexo. Por el resultado de la parte 1, tenemos que $P(v)$ también es convexo.

3. Muestre que para cada hoja, los vectores que llegan a ella forman un conjunto convexo.

   Solución:

   Se desprende directamente del resultado anterior: para toda hoja h, tenemos que $P(h)$ (conjunto de vectores cuya valuacion termina en h) es un conjunto convexo.

4. Si el árbol calcula una función F, denominamos $\#F_0$ al número de hojas donde la función tiene valor 0, y $\#F_1$ al número de hojas donde tiene valor 1. Demuestre que todo árbol que calcula F tiene altura mayor o igual a $\log_3(\#F_0 + \#F_1)$

   Idea de Solución:

   El número de hojas del árbol es $\#F_0 + \#F_1$. Luego, como el árbol es ternario, se puede demostrar fácilmente por inducción que su altura es mayor o igual a $\log_3$(numero de hojas) $= \log_3(\#F_0 + \#F_1)$.

5. Demuestre que todo árbol lineal que calcula F:

   $$F(x) = \begin{cases} 1 & x \text{ tiene elementos repetidos} \\ 0 & x \text{ no tiene elementos repetidos} \end{cases}$$

   tiene altura al menos $(n/2)\log_3(n/2)$. Para esto, considere un vector $z = (z_1, z_2, z_3, ..., z_n)$ con todas sus componentes distintas, junto con $z' = (z_2, z_1, z_3, ..., z_n)$. En particular, analice si es posible que z y z' lleguen a la misma hoja del árbol.

   Idea de Solución:

Por definicion $F(z) = F(z') = 0$. Sin embargo, sea $y = \frac{1}{2}z + \frac{1}{2}z'$. Tenemos que $y_1 = y_2 = \frac{1}{2}z_1 + \frac{1}{2}z_2$. Por lo tanto, $F(y) = 1$. Esto implica que y no llega a la misma hoja que z y z'. Dado que y es de la forma $\lambda z + (1 - \lambda)z'$, se concluye que z y z' no llegan ambos a una misma hoja (si llegaran a una misma hoja, entonces y llegaría a la misma hoja, lo que es una contradicción).

Con este argumento se puede demostrar que cada una de las permutaciones de z termina en una hoja distinta (hay que encontrar el λ apropiado), por lo que hay por lo menos $n!$ hojas. Luego se puede utilizar la cota $n! \geq (n/2)^{(n/2)}$ y lo demostrado en la parte 4 para mostrar que la altura es por lo menos $(n/2)\log_3(n/2)$.

6. Se considera el problema de encontrar elementos repetidos en una lista de largo n. Demuestre que todo algoritmo basado en comparaciones para este problema requiere $\Omega(n\log(n))$ operaciones en el peor caso.

Idea de Solución:

Se consideran algoritmos que puedan ser representados con árboles donde en cada nodo se comparan posiciones fijas del arreglo. Dado un árbol de este tipo, se puede construir un árbol lineal de la misma altura, transformando cada nodo donde se compare x_i con x_j por un nodo con vector $v = e_i - e_j$. Luego, si existiera un árbol de comparaciones de altura menor a $(n/2)\log_3(n/2)$, entonces existiría un árbol lineal de la misma altura, lo que es una contradicción con el resultado de la parte 5. Por lo tanto, todo algoritmo basado en comparaciones requiere como mínimo $(n/2)\log_3(n/2) \in \Omega(n\log(n))$ comparaciones en el peor caso.