D & C

TDA

Facultad de ciencias Exactas

September 10, 2024

Divide and Conquer

Es una estrategia algorítmica que consiste en

- **Dividir** el problema en k subproblemas del mismo tipo, pero más chicos
- **Conquistar** resolviendo los subproblemas, recursivamente (para los casos grandes) o directamente (para los casos chicos)
- **Combinar** las soluciones obtenidas para resolver el problema original. Esto no siempre es necesario.

Teorema Maestro

Sea un problema que puede dividirse en a subproblemas de un tamaño $\frac{n}{b}$ y cuyo costo de división y combinación es $f(n)$, su complejidad puede expresarse como

$$T(n) = \begin{cases} aT\left(\frac{n}{b}\right) + f(n) & \text{si } n > 1 \\ 1 & \text{si } n = 1 \end{cases}$$

donde

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) & \text{si } f(n) = O(n^{\log_b a - \epsilon}) \text{ para algún } \epsilon > 0, \\ \Theta(n^{\log_b a} \log n) & \text{si } f(n) = \Theta(n^{\log_b a}), \\ \Theta(n^{\log_b a} \log^{k+1} n) & \text{si } f(n) = \Theta(n^{\log_b a} \log^k n) \text{ para algún } k \geq 0, \\ \Theta(f(n)) & \text{si } f(n) = \Omega(n^{\log_b a + \epsilon}) \text{ para algún } \epsilon > 0. \end{cases}$$

Teorema Maestro

Dado un problema recursivo que puede expresarse como:

$$T(n) = \begin{cases} aT\left(\frac{n}{b}\right) + f(n) & \text{si } n > 1 \\ 1 & \text{si } n = 1 \end{cases}$$

donde $a \geq 1$, $b > 1$. La solución de la recurrencia se clasifica en tres casos:

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) & \text{si } n^{\log_b a} > f(n), \\ \Theta(n^{\log_b a} \log n) & \text{si } n^{\log_b a} = f(n). \\ \Theta(n^{\log_b a} \log^{k+1} n) & \text{si } (n^{\log_b a} \log^k n) = f(n). \\ \Theta(n) & \text{si } n^{\log_b a} < f(n), \end{cases}$$

Desfile de moda

Julieta, una organizadora de eventos de Balenciaga, está preparando una gran pasarela. La empresa tiene una lista con los nombres de las modelos y ella necesita ordenarlas alfabéticamente por su nombre. Antes de trabajar en la industria de la moda cursó y fue ayudante de algoritmos 3, entonces sabe de una técnica algorítmica eficiente para realizar la tarea.

Enfoque de D&C

- Dividir el vector en 2 mitades.
- Ordenar cada mitad.
- Una lista vacía ya está ordenada.

Análisis de Complejidad

La recurrencia para el enfoque es:

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

Donde:

- $2T\left(\frac{n}{2}\right)$ es la complejidad del subproblema.
- n que es $O(n)$ es el tiempo que lleva combinar dos vectores.

Merge Sort

Algorithm 1: MergeSort

Input: V
Output: V

```
1 if |V| < 1 then
2     return V
3 else
4     V1 ← MergeSort(V[0 ... n/2])        // T(n/2)
5     V2 ← MergeSort(V[n/2 ... n])        // T(n/2)
6     V ← Merge(V1, V2)                   // Θ(n)
7     return V
```

Complejidad: $T(n) = 2T(\frac{n}{2}) + \Theta(n)$

Merge Sort

Algorithm 2: Merge

Input: $V1$, $V2$
Output: *Merged*

```
1  Merged ← [];
2  i ← 0;
3  j ← 0;
4  while i < |V1| and j < |V2| do
5      if V1[i] ≤ V2[j] then
6          Merged.append(V1[i]);
7          i ← i + 1;
8      else
9          Merged.append(V2[j]);
10         j ← j + 1;
11 while i < |V1| do
12     Merged.append(V1[i]);
13     i ← i + 1;
14 while j < |V2| do
15     Merged.append(V2[j]);
16     j ← j + 1;
17 return Merged
```

Código en C++ de Merge Sort

```cpp
#include <iostream>
#include <vector>

using namespace std;

vector<int> merge(vector<int>& ladoIzq, vector<int>& ladoDer) {
    vector<int> res;
    int i = 0, j = 0;
    while (i < ladoIzq.size() && j < ladoDer.size()) {
        if (ladoIzq[i] <= ladoDer[j]) {
            res.push_back(ladoIzq[i]);
            i++;
        } else {
            res.push_back(ladoDer[j]);
            j++;
        }
    }
    while (i < ladoIzq.size()) {
        res.push_back(ladoIzq[i]);
        i++;
    }
    while (j < ladoDer.size()) {
        res.push_back(ladoDer[j]);
        j++;
    }
    return res;
}
```

```cpp
vector<int> mergeSort(vector<int>& arr) {
    if (arr.size() <= 1) {
        return arr;
    }
    int medio = arr.size() / 2;
    vector<int> izq( first: arr.begin(),  last: arr.begin() + medio);
    vector<int> der( first: arr.begin() + medio,  last: arr.end());
    izq = mergeSort( &: izq);
    der = mergeSort( &: der);
    return merge( &: izq,  &: der);
}

int main() {
    int n;
    cin >> n;
    vector<int> vec(n);
    for (int i = 0; i < n; i++) {
        cin >> vec[i];
    }
    vector<int> ordenado = mergeSort( &: vec);
    for (int num : ordenado) {
        cout << num << " ";
    }
    cout << endl;

    return 0;
}
```

Aplicación del Teorema Maestro

Comparación de $f(n)$ con $n^{\log_b a}$:

- $a = 2$, $c = 2$, y $f(n) = n$.
- $\log_b a = \log_2 2 = 1$.
- $f(n) = n$, por lo tanto, estamos en el Caso 2 del Teorema Maestro.

La solución a la recurrencia es:

$$T(n) = \Theta(n \log n)$$

El desfile más grande de la historia

Para el próximo evento, Balenciaga decide romper el record de mayor cantidad de modelos en un show y contrata a más de 1000 modelos para desfilar.
Julieta decide que esto es demasiado y se va del proyecto, dejando a su asistente Ayelen a cargo de la organización.
Ayelen mantiene la lista de modelos ordenada, pero como no puede recordar los nombres y datos de tantas personas, necesita estar revisándola constantemente.
Revisar una por una las entradas de la lista resulta ser altamente ineficiente, con lo que necesita una mejor técnica de búsqueda. Decide aprovechar que está ordenada por nombre para buscar más rápidamente.

El desfile más grande de la historia

¿Cómo debería buscar Ayelen a cualquier modelo en la lista para aprovechar que están ordenadas?

El desfile más grande de la historia

¿Cómo debería buscar Ayelen a cualquier modelo en la lista para aprovechar que están ordenadas
¡Utilizando Binary Search, obviamente!

Binary Search

Repaso de Binary Search :

- tenemos una lista ordenada y buscamos donde está un elemento que sabemos existe dentro de la lista.
- nos paramos en la mitad de la lista y revisamos ese elemento, comparándolo con el que buscamos.
- Si el que buscamos es menor, estará en la primera mitad de nuestra lista
- Si es mayor, buscamos en la segunda mitad. .

Binary Search

Algorithm 3: Binary Search

Input: $list : ss, element : a$

```
1  i ← 0
2  d ← ss.size() − 1,
3  while i ≤ d do
4      m ← ⌊(i + d)/2⌋
5      if ss[m] < a then
6          i ← m + 1
7      if ss[m] > a then
8          d ← m − 1
9      else
10         return m
```

Binary Search

Pero
¿Cuánto tarda Binary Search?
Probemos que sea más rápido que la búsqueda lineal

Binary Search

Algorithm 4: Binary Search

Input: $list : ss, element : a$

```
1  i ← 0                                              // O(1)
2  d ← ss.size() − 1                                  // O(1)
3  while i ≤ d do                                     // O(??)
4      m ← ⌊(i + d)/2⌋                                // O(1)
5      if ss[m] < a then
6          i ← m + 1                                  // O(1)
7      if ss[m] > a then
8          d ← m − 1                                  // O(1)
9      else
10         return m                                   // O(1)
```

Binary Search

Apliquemos el Teorema maestro:

- $b = 2$, dividimos el problema en dos subproblemas al dividir a la mitad la lista.
- $a = 1$,de los dos subproblemas solo resolvemos uno.
- $f(n) = O(1)$, en cada iteración solo hacemos operaciones atómicas.

Binary Search

Apliquemos el Teorema maestro:

- $b = 2$, dividimos el problema en dos subproblemas al dividir a la mitad la lista.
- $a = 1$,de los dos subproblemas solo resolvemos uno.
- $f(n) = O(1)$, en cada iteración solo hacemos operaciones atómicas.

Entonces nos queda:
$T(n) = 1 * T(n/2) + O(1)$

Binary Search

- Como $n^{log_b a} = n^{log_2 1} = n^0 = 1$.
- Entonces cumplimos el segundo caso del Teorema Maestro: $f(n) = O(n^{log_b a})$
- Con lo que: $T(n) = O(n^{log_b a} * log(n)) = O(1 * log(n)) = O(log(n))$
- ¡Entonces la complejidad de Binary Search es O(log(n))!

Jera la especuladora

En el último desfile, Jera, una de las modelos de Balenciaga, se quejó porque la habían hecho participar solamente de un desfile primaveral, a pesar de que éstos eran sus eventos favoritos. Muy amablemente, se le explicó que si dos desfiles eran **equivalentes** no iba a poder participar en ambos porque no era una buena estrategia de marketing. Todos los desfiles están identificados con un código [string] de la misma longitud predeterminada. Por ejemplo "aaba", "abaa".
Definamos qué es que dos códigos sean equivalentes.

Jera la especuladora

Se dice que dos desfiles son **equivalentes** si, dados los códigos de ambos desfiles **a** y **b** pasa alguna de estas:

- Son iguales **a** y **b**
- Si partimos al string a en dos mitades del mismo tamaño **a1** y **a2**, y al string b en dos mitades del mismo tamaño **b1** y **b2**, entonces alguna de las siguientes es verdadera:
  - **a1** es equivalente a **b1** y **a2** es equivalente a **b2**
  - **a1** es equivalente a **b2** y **a2** es equivalente a **b1**

Hay que ayudar a Jera a chequear si dos desfiles son equivalentes. Asumimos que miramos 2 códigos de la misma longitud.
Ejemplos:

- aaba y abaa son equivalentes
- aabb y abab NO son equivalentes

Jera la especuladora

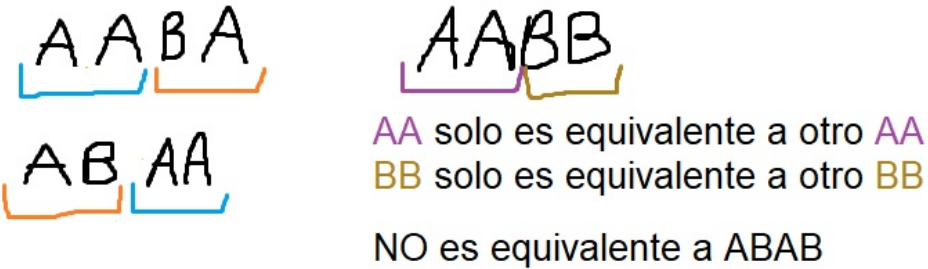

Figure: Ejemplo desglosado de equivalencias

Enfoque del problema

- Si no puedo dividir al código en dos partes iguales, tengo que comparar si son iguales (luego, son equivalentes)
- Si puedo dividir al código en dos partes iguales, tengo que mirar si puedo armar equivalencias entre sí con sus mitades.

Los códigos equivalentes forman parte de una misma **clase de equivalencia**.

- Sería útil considerar a un "representante" de la clase de equivalencia (y que encontremos siempre al mismo) y ver si ambos códigos se condicen con el mismo representante.
- Arbitrariamente, elegimos como el representante al que **está ordenado lexicográficamente de menor a mayor**.

Enfoque de D&C para encontrar al representante de la clase de equivalencia

- Dado un código de desfile, si no lo puedo dividir en 2 partes iguales, hago la comparación con ese código.
- Si puedo partirlo en 2 partes iguales, voy partiendo las subcadenas para encontrar "sub-representantes" de esas subcadenas
- En cada iteración, unimos las 2 subcadenas poniendo primero a la que es menor.

Llamo al algoritmos que encuentra representantes para ambos códigos y, si tienen al mismo representante, pertenecen a la misma clase de equivalencia. Luego, son equivalentes.

Algorithm 5: Main

```
1 stringa, stringb ← input();
2 smalla ← smallest(stringa);
3 smallb ← smallest(stringb);
4 if smalla == smallb then
5 |   print("YES");
6 else
7 |   print("NO");
8 print("\n");
```

Algorithm 6: Smallest

Input: S

Output: S

```
1 if S tiene longitud impar then
2     return s;
3 s1 ← smallest(primera mitad de S);
4 s2 ← smallest(segunda mitad de S);
5 if s1 < s2 then
6     return s1 + s2;
7 else
8     return s2 + s1;
```

Código en C++ de Jera la especuladora (créditos a Pablo Terlisky)

```
/* CodeForces 559B - Equivalent Strings
  url vjudge: https://vjudge.net/problem/CodeForces-559B
  Entrega 4 de Técnicas de Diseño de Algoritmos - 1C2024
  Solución usando el código en el Tutorial (requiere leve adaptación)
  ~Pablo Terlisky
*/

#include<iostream>

using namespace std;

//Adaptado del tutorial para funcionar en C++
string smallest(string s) {
  if (s.length() % 2 == 1) return s;
  string s1 = smallest(s.substr(0, s.length()/2));
  string s2 = smallest(s.substr(s.length()/2, s.length()));
  if (s1 < s2) return s1 + s2;
  else return s2 + s1;
}

int main() {
  string stringa, stringb; cin>>stringa>>stringb;
  string smalla = smallest(stringa);
  string smallb = smallest(stringb);
  if (smalla == smallb){
    cout << "YES";
  } else {
    cout << "NO";
  }
```

Complejidad Jera la especuladora

¿Cuál es la complejidad?
Spoiler: es la misma que una que ya vimos en esta clase

Parcialmente eficiente

En un parcial de TDA suele haber n^2 alumnos sentados en un aula cuadrada en forma de grilla. Todos los años entre los alumnos se ponen de acuerdo y solo asisten alumnos de manera que n una potencia de 2.

Parcialmente eficiente

Este año Echu se puso la gorra y quiere asegurarse de buscar de manera eficiente alumnos que se estén copiando en el parcial. Para esto Echu desarrolló el siguiente súper-poder. Dadas las posiciones de las esquinas superior izquierda (x_0, y_0) e inferior derecha (x_1, y_1) de cualquier sector de la grilla, él es capaz de descifrar si hay alguien copiándose dentro de este rectángulo.

Parcialmente eficiente

Sin embargo, esto no le dice ni quién es, ni en qué posición está el alumno deshonesto. Además, no es gratis, si bien es un súper-poder, a Echu le cuesta $\Theta(log((x_1 - x_0) * (y_1 - y_0)))$ operaciones donde $(x_1 - x_0) * (y_1 - y_0)$ es la cantidad de alumnos que hay sentados en ese rectángulo.

Descripción del problema formal

Formalmente:

- Se tiene una matriz cuadrada de booleanos A de tamaño $n \times n$.
- Necesitamos encontrar un elemento 'True' en la matriz con un algoritmo eficiente.
- Asumimos que existe al menos un elemento 'True' en la matriz.
- Tenemos una operación que dado un rectángulo de la matriz, nos dice en tiempo logarítmico en el tamaño del rectángulo si hay algún 'True' en ese sector.

Enfoque de D&C

- Dividir la matriz en 4 cuadrantes.
- Verificar la presencia de un ‘True‘ en cada cuadrante.
- Si se encuentra un ‘True‘, solo se sigue la búsqueda en ese cuadrante.
- Se repite el proceso hasta que se localiza el ‘True‘.

Análisis de Complejidad

La recurrencia para el enfoque es:

$$T(n) = T\left(\frac{n}{2}\right) + \log((\frac{n}{2})^2)$$

Donde:

- $T\left(\frac{n}{2}\right)$ es la complejidad del subproblema.
- $\log((\frac{n}{2})^2) = 2\log(\frac{n}{2})$ que es $O(log(n))$ es el tiempo para verificar la presencia de un 'True' en un cuadrante.

Aplicación del Teorema Maestro

Comparación de $f(n)$ con $n^{\log_b a}$:

- $a = 1$, $b = 2$, y $f(n) = \log(n)$.
- $\log_b a = \log_2 1 = 0$.
- $f(n) = \log(n)$, por lo tanto, estamos en el Caso 2 del Teorema Maestro.

La solución a la recurrencia es:

$$T(n) = \Theta(\log^2 n)$$

The Meeting Place Cannot Be Changed

Enunciado[1]

En una calle hay n amigos, cada uno en una posición x_i a lo largo de la misma. Para cada amigo se conoce la máxima velocidad v_i a la que puede moverse. Queremos averiguar el menor tiempo posible en el que podrían encontrarse todos.
Constraints: $2 \leq n \leq 10^5$, $1 \leq x_i, v_i \leq 10^9$.

[1] https://codeforces.com/contest/782/problem/B

The Meeting Place Cannot Be Changed

- Notemos que si t es el menor tiempo en cual se pueden encontrar, para todo $t' \geq t$ también les alcanzan t' segundos para encontrarse. Aparte, si $t'' < t$ entonces no les alcanza el tiempo.

The Meeting Place Cannot Be Changed

- Notemos que si t es el menor tiempo en cual se pueden encontrar, para todo $t' \geq t$ también les alcanzan t' segundos para encontrarse. Aparte, si $t'' < t$ entonces no les alcanza el tiempo.
- Entonces que si me alcanza el tiempo para que se encuentren, puedo buscar en tiempos menores y no vale la pena buscar en mayores. En caso contrario, tengo que buscar en tiempos mayores para que me alcance.

The Meeting Place Cannot Be Changed

- Notemos que si t es el menor tiempo en cual se pueden encontrar, para todo $t' \geq t$ también les alcanzan t' segundos para encontrarse. Aparte, si $t'' < t$ entonces no les alcanza el tiempo.
- Entonces que si me alcanza el tiempo para que se encuentren, puedo buscar en tiempos menores y no vale la pena buscar en mayores. En caso contrario, tengo que buscar en tiempos mayores para que me alcance.
- O sea, podemos hacer búsqueda binaria sobre este parámetro t.

The Meeting Place Cannot Be Changed

- Notemos que si t es el menor tiempo en cual se pueden encontrar, para todo $t' \geq t$ también les alcanzan t' segundos para encontrarse. Aparte, si $t'' < t$ entonces no les alcanza el tiempo.
- Entonces que si me alcanza el tiempo para que se encuentren, puedo buscar en tiempos menores y no vale la pena buscar en mayores. En caso contrario, tengo que buscar en tiempos mayores para que me alcance.
- O sea, podemos hacer búsqueda binaria sobre este parámetro t.
- ¿Cotas superiores e inferiores para t?

The Meeting Place Cannot Be Changed

- Dado un t_0 fijo. ¿Cómo podemos descubrir si es posible que los amigos se encuentren?

The Meeting Place Cannot Be Changed

- Dado un t_0 fijo. ¿Cómo podemos descubrir si es posible que los amigos se encuentren?
- En t_0 segundos el amigo i puede cubrir el rango de posiciones $[x_i - t * v_i, x_i + t * v_i]$.

The Meeting Place Cannot Be Changed

- Dado un t_0 fijo. ¿Cómo podemos descubrir si es posible que los amigos se encuentren?
- En t_0 segundos el amigo i puede cubrir el rango de posiciones $[x_i - t * v_i, x_i + t * v_i]$.
- Dos amigos llegan a encontrarse si sus intervalos intersectan. En general, todos se pueden encontrar en un tiempo t_0 si $\bigcap_{i=1}^{n} [x_i - t_0 * v_i, x_i + t_0 * v_i] \neq \emptyset$.

The Meeting Place Cannot Be Changed

- Dado un t_0 fijo. ¿Cómo podemos descubrir si es posible que los amigos se encuentren?
- En t_0 segundos el amigo i puede cubrir el rango de posiciones $[x_i - t * v_i, x_i + t * v_i]$.
- Dos amigos llegan a encontrarse si sus intervalos intersectan. En general, todos se pueden encontrar en un tiempo t_0 si $\bigcap_{i=1}^{n} [x_i - t_0 * v_i, x_i + t_0 * v_i] \neq \emptyset$.
- ¿En qué complejidad podemos calcular esa intersección?

The Meeting Place Cannot Be Changed

- Dado un t_0 fijo. ¿Cómo podemos descubrir si es posible que los amigos se encuentren?
- En t_0 segundos el amigo i puede cubrir el rango de posiciones $[x_i - t * v_i, x_i + t * v_i]$.
- Dos amigos llegan a encontrarse si sus intervalos intersectan. En general, todos se pueden encontrar en un tiempo t_0 si $\bigcap_{i=1}^{n} [x_i - t_0 * v_i, x_i + t_0 * v_i] \neq \emptyset$.
- ¿En qué complejidad podemos calcular esa intersección? $O(n)$. Luego, la complejidad final queda $O(n \log(cotaAlTiempo))$.

The Meeting Place Cannot Be Changed

- ¿Y cómo termina nuestra búsqueda? ¿Qué nos queda?

The Meeting Place Cannot Be Changed

- ¿Y cómo termina nuestra búsqueda? ¿Qué nos queda?
- Al estar buscando el mínimo, vamos a ver todos los candidatos de los subrangos de búsqueda que vaya encontrando binary search, con lo que termina al quedarnos con un subrango cruzado (empieza en un tiempo posterior a donde terminar. Ver el código en el campus para entenderlo mejor). Nos quedamos con el menor tiempo posible de los que encontramos que los amigos pueden encontrarse.

Mediana

Enunciado

Dado un vector v de n elementos, debemos encontrar su mediana. Más precisamente:

- Dar un algoritmo que la encuentre en $O(n \log n)$.
- Dar un algoritmo randomizado que la encuentre en $O(n)$.
- Dar un algoritmo sin aleatoriedad que la encuentre en $O(n)$.

Mediana

- El primer algoritmo es simplemente ordernar el vector y acceder al elemento de índice $\frac{n}{2}$.

[2] La demo de que esto es $O(n)$ en complejidad promedio la pueden leer en *Computational Complexity: A Modern Approach*, Claim 7.4

Mediana

- El primer algoritmo es simplemente ordernar el vector y acceder al elemento de índice $\frac{n}{2}$.
- El segundo consiste en una búsqueda a lo *QuickSort*: elegimos un índice al azar, y lo usamos para pivotear a todo el vector. Luego seguimos la búsqueda recursivamente en el lado que quedó[2].

[2] La demo de que esto es $O(n)$ en complejidad promedio la pueden leer en *Computational Complexity: A Modern Approach*, Claim 7.4

Mediana

- El primer algoritmo es simplemente ordernar el vector y acceder al elemento de índice $\frac{n}{2}$.
- El segundo consiste en una búsqueda a lo *QuickSort*: elegimos un índice al azar, y lo usamos para pivotear a todo el vector. Luego seguimos la búsqueda recursivamente en el lado que quedó[2].
- Para el último algoritmo, nos gustaría repetir esta estrategia, pero tomando un pivote de forma determinística.

[2] La demo de que esto es $O(n)$ en complejidad promedio la pueden leer en *Computational Complexity: A Modern Approach*, Claim 7.4

Mediana

- El truco está en buscar la mediana de un vector más chico recursivamente, y que esa mediana nos sirva como aproximación a la mediana de nuestro vector más grande.

Mediana

- El truco está en buscar la mediana de un vector más chico recursivamente, y que esa mediana nos sirva como aproximación a la mediana de nuestro vector más grande.
- Puntualmente, la idea es dividir al vector v de n elementos en $\frac{n}{5}$ vectores de tamaño 5. Para cada uno de ellos calculamos su mediana (¿En qué complejidad?), y luego buscamos la mediana de esas medianas haciendo recursión.

Mediana

- El algoritmo entonces quedaría como:
  1. Si $|v| \leq 5$, buscamos la mediana a mano.
  2. Sino, separamos v en $\frac{n}{5}$ vectores w_i de tamaño 5.
  3. Sea $\mathbf{w}$ un vector con las medianas de cada w_i. Calculamos recursivamente la medianda de $\mathbf{w}$, y la llamamos x.
  4. Usamos x para pivotear.

Mediana

- El algoritmo entonces quedaría como:
  1. Si $|v| \leq 5$, buscamos la mediana a mano.
  2. Sino, separamos v en $\frac{n}{5}$ vectores w_i de tamaño 5.
  3. Sea $\mathbf{w}$ un vector con las medianas de cada w_i. Calculamos recursivamente la medianda de $\mathbf{w}$, y la llamamos x.
  4. Usamos x para pivotear.
- ¿Complejidad de esto?

Mediana

- El algoritmo entonces quedaría como:
  1. Si $|v| \leq 5$, buscamos la mediana a mano.
  2. Sino, separamos v en $\frac{n}{5}$ vectores w_i de tamaño 5.
  3. Sea $\mathbf{w}$ un vector con las medianas de cada w_i. Calculamos recursivamente la medianda de $\mathbf{w}$, y la llamamos x.
  4. Usamos x para pivotear.
- ¿Complejidad de esto? Depende de qué tan bien salgan los pivoteos.

Mediana

- El algoritmo entonces quedaría como:
  1. Si $|v| \leq 5$, buscamos la mediana a mano.
  2. Sino, separamos v en $\frac{n}{5}$ vectores w_i de tamaño 5.
  3. Sea $\mathbf{w}$ un vector con las medianas de cada w_i. Calculamos recursivamente la medianda de $\mathbf{w}$, y la llamamos x.
  4. Usamos x para pivotear.
- ¿Complejidad de esto? Depende de qué tan bien salgan los pivoteos.
- Lema: x cumple que es mayor a por lo menos $\frac{3n}{10}$ elementos y menor que $\frac{3n}{10}$. Es decir, al usarlo como pivote, en peor caso nos quedamos con un caso recursivo de tamaño $\frac{7n}{10}$.

Mediana

Demostración del Lema:

Median_of_medians_time_complexity.png

Mediana

- ¿Cuál es la recursión que sigue la complejidad de este algoritmo?

[3] Aunque se pueden usar teoremas mas fuertes, como Akra-Bazzi
https://en.wikipedia.org/wiki/Akra%E2%80%93Bazzi_method

Mediana

- ¿Cuál es la recursión que sigue la complejidad de este algoritmo?

$$T(n) = T(\frac{n}{5}) + T(\frac{7n}{10}) + O(n)$$

[3] Aunque se pueden usar teoremas mas fuertes, como Akra-Bazzi
https://en.wikipedia.org/wiki/Akra%E2%80%93Bazzi_method

Mediana

- ¿Cuál es la recursión que sigue la complejidad de este algoritmo?

$$T(n) = T(\frac{n}{5}) + T(\frac{7n}{10}) + O(n)$$

- No podemos usar el teorema maestro porque no tiene la "forma estándar"[3].

[3] Aunque se pueden usar teoremas mas fuertes, como Akra-Bazzi
https://en.wikipedia.org/wiki/Akra%E2%80%93Bazzi_method

Mediana

- ¿Cuál es la recursión que sigue la complejidad de este algoritmo?

$$T(n) = T(\frac{n}{5}) + T(\frac{7n}{10}) + O(n)$$

- No podemos usar el teorema maestro porque no tiene la "forma estándar"[3].
- Aún asi, podemos probar a mano que la complejidad está acotada por una función lineal en n.

[3] Aunque se pueden usar teoremas mas fuertes, como Akra-Bazzi
https://en.wikipedia.org/wiki/Akra%E2%80%93Bazzi_method

Mediana

$$T(n) = T(\frac{n}{5}) + T(\frac{7n}{10}) + cn$$

- Llamemos c a alguna constante que acote la complejidad de la parte lineal de la función. Probemos que $T(n) \leq 10cn$ por inducción.

Mediana

$$T(n) = T(\frac{n}{5}) + T(\frac{7n}{10}) + cn$$

- Llamemos c a alguna constante que acote la complejidad de la parte lineal de la función. Probemos que $T(n) \leq 10cn$ por inducción.
- Los casos base se pueden resolver eligiendo un c lo suficientemente grande. Para el paso recursivo, notemos que:

$$\begin{aligned} T(n) &= T(n/5) + T(7n/10) + cn \\ &\leq 10c\frac{n}{5} + 10c\frac{7n}{10} + cn \\ &= 2cn + 7cn + cn = 10cn \end{aligned}$$

Mediana

- Con esto probamos que el algoritmo es lineal.

Mediana

- Con esto probamos que el algoritmo es lineal.
- Notar que también se puede emplear para encontrar el k-ésimo elemento más chico del vector.

Mediana

- Con esto probamos que el algoritmo es lineal.
- Notar que también se puede emplear para encontrar el k-ésimo elemento más chico del vector.
- En particular, este algoritmo permite *derandomizar* Quicksort.