## **Evaluación asincrónica 2** de Estrategias Algorítmicas, ITESO, Luis Gatica

### Fecha de entrega de la evaluación completa: lunes 11 de abril de 2022

Nombre: Rodríguez Castro, Carlos Eduardo; Cordero Hernández, Marco Ricardo

# Parte 1: análisis *divide y vencerás*

## Ejercicios más sencillos

### Potencia divide y vencerás

Use las siguientes propiedades aritméticas para programar un **método divide y vencerás** (llamado *pow*) que reciba *x*,*y* y calcule *x*^*y* (asuma que *x*,*y* no son muy grandes (no habrá *overflow* si usa *int* en Java)) y que *x*>=*1* y *y*>=*0*:

1. Si *y*=*0*, entonces *x*^*y* = *x*^*0* = *1*
2. Si *y* es par, entonces *x*^*y* se puede calcular como *x*^(*y/2*) multiplicado por sí mismo.
3. Si *y* es non, *x*^*y* se puede calcular como *x*^(*y*/*2*) multiplicado por sí mismo y luego por *x*.\*

\*Nota: en los puntos 2 y 3 el operador de división es el operador de división entera (sólo entrega la parte entera de la división; ej: *7*/*2* = *3*).

int **pow**(int *x*, int *y*) {

return (*y* <= 0) ? 1 : (*y* % 2 == 0) ?

**pow**(*x* \* *x*, *y* / 2) : *x* \* **pow**(*x* \* *x*, *y* / 2);

}

Haga un análisis de la complejidad temporal del método. Una justificación completa puede ser escrita, en este caso, en dos líneas (no es necesario que elabore árboles o reglas de recurrencia, aunque puede utilizar cualquiera de esas herramientas).

**R:** Para la solución proporcionada, la complejidad temporal es *log(n)*, ya que las potencias se van calculando de forma que el exponente se divide entre dos y se compensa con la *x* individual de la última línea como potencia impar.

### *Búsqueda binaria*

Escriba en Java los métodos necesarios (2) para implementar *binarySearch()* para enteros.

boolean **search**(int[] *arr*, int *n*) { *// Iterativo*

int t = *arr*.length - 1, b = 0, m;

while (b <= t) {

      m = b + ((t - b) / 2);

            if (*arr*[m] > *n*) t = m - 1;

            else if (*arr*[m] < *n*) b = m + 1;

            else return true;

      }

        return false;

}

public static boolean **searchR**(int[] *arr*, int *n*, int *b*, int *t*) { *// Recursivo*

if (*t* < *b*) return false;

int m = *b* + ((*t* - *b*) / 2);

if (*n* > *arr*[m]) return **searchR**(*arr*, *n*, m + 1, *t*);

else if (*n* < *arr*[m]) return **searchR**(*arr*, *n*, *b*, m - 1);

else return true;

}

### *Mergesort*

Escriba en Java los métodos necesarios para implementar *mergesort*. Recuerde que sólo tiene permitido reservar memoria adicional para otro arreglo del mismo tamaño que el original, más las variables (acotadas en un orden constante) que necesite: contadores, índices, temporales...

void **sort**(int[] *array*) {

int[] helper = new int[*array*.length];

**mergesort**(*array*, helper, 0, *array*.length - 1);

}

void **mergesort**(int[] *array*, int[] *helper*, int *low*, int *high*) {

if (*low* < *high*) {

int middle = (*low* + *high*) / 2;

**mergesort**(*array*, *helper*, *low*, middle); *// Ordenar mitad izquierda*

**mergesort**(*array*, *helper*, middle + 1, *high*); *// Ordenar mitad derecha*

**merge**(*array*, *helper*, *low*, middle, *high*); *// MEZCLARLOS*

}

}

void **merge**(int[] *array*, int[] *helper*, int *low*, int *middle*, int *high*) {

for (int i = *low*; i <= *high*; i++) {

*helper*[i] = *array*[i];

        }

        int helperLeft = *low*;

        int helperRight = *middle* + 1;

        int current = *low*;

        while (helperLeft <= *middle* && helperRight <= *high*) {

            if (*helper*[helperLeft] <= *helper*[helperRight]) {

*array*[current] = *helper*[helperLeft];

                helperLeft++;

            } else {

*array*[current] = *helper*[helperRight];

                helperRight++;

            } current++;

        }

        int remaining = *middle* - helperLeft;

        for (int i = 0; i <= remaining; i++) {

*array*[current + i] = *helper*[helperLeft + i];

}

}

### *Quicksort*

Escriba en Java los métodos necesarios para implementar *quicksort*, incluyendo un método recursivo principal y un método iterativo (no recursivo) *partition()* para calcular la posición final del pivote actual, colocando a ese pivote en ella, a los elementos menores o iguales a su izquierda y a los mayores a su derecha.

void **quickSort**(int[] *arr*, int *left*, int *right*) {

    int index = **partition**(*arr*, *left*, *right*);

    if (*left* < index - 1) **quickSort**(*arr*, *left*, index - 1);

    if (index < *right*) **quickSort**(*arr*, index, *right*);

}

void **swap**(int[] *arr*, int *left*, int *right*) {

    int holder = *arr*[*left*];

*arr*[*left*] = *arr*[*right*];

*arr*[*right*] = holder;

}

int **partition**(int[] *arr*, int *left*, int *right*) {

    int pivot = *arr*[*right* - 1];

    while (*left* < *right*) {

        while (*arr*[*left*] < pivot) *left*++;

        while (*arr*[*right*] > pivot) *right*--;

        if (*left* <= *right*) {

**swap**(*arr*, *left*, *right*);

*left*++;

*right*--;

        }

    }

    return *left*;

}

### Estadístico de orden *i*

Aprovechando el método *partition()* que ya escribió, agregue los métodos necesarios para calcular el estadístico de orden *i*: es decir, para devolver el dato que estaría en el índice i si el arreglo estuviera ordenado. Siga un enfoque divide y vencerás para resolver este problema, y no intente ordenar primero todo el arreglo para simplemente devolver lo que está en la posición *i*: eso sería demasiado trabajo computacional para resolver un problema mucho más sencillo (es decir: menos complejo, en términos del orden de crecimiento de la solución buscada).

int **findStatisticN**(int[] *arr*, int *n*, int *left*, int *right*) {

    int pivot = **partition**(*arr*, *left*, *right*);

    return (pivot == *n* - 1) ? *arr*[pivot] :

            (*n* - 1 < pivot) ? **findStatisticN**(*arr*, *n*, *left*, pivot - 1) :

**findStatisticN**(*arr*, *n*, pivot + 1, *right*);

}

## El problema del subarreglo de suma máxima: introducción

El profesor de Selena está convencido de la utilidad de su clase para las dinámicas del capitalismo tardío. Por ello, deja a Selena y a sus amigos una tarea sobre optimización de ganancias.

Selena es contratada por una compañía que comercia piñas y genera toda su utilidad revendiéndolas. La información de la cual ella dispone es el precio de la piña cada día durante los próximos *N* días, y como el mercado está en una competencia perfecta, ella no puede vender la piña a un precio mayor porque no se la comprarían (ni a un precio menor, porque ganaría menos). El problema de Selena consiste en decidir cuándo comprar la piña y cuándo revenderla para maximizar su ganancia, asumiendo que sólo puede comprar en un día y sólo puede revender en otro.

Aquí podemos observar el precio de la piña (en alguna moneda über-capitalista que desconocemos) en los próximos *N*=17 días:

Una forma de determinar qué día comprar y qué día revender depende de calcular las *diferencias de precio* de un día a otro, que se pueden observar (para los datos de la gráfica anterior) en la siguiente tabla:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Día** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** | **16** |
| Precio | 100 | 113 | 110 | 85 | 105 | 102 | 86 | 63 | 81 | 101 | 94 | 106 | 101 | 79 | 94 | 90 | 97 |
| Incremento | | 13 | -3 | -25 | 20 | -3 | -16 | -23 | 18 | 20 | -7 | 12 | -5 | -22 | 15 | -4 | 7 |

Visto así, el problema se reduce a **encontrar los índices de un sub-arreglo contiguo de suma máxima** (tomando como arreglo original la lista de incrementos). En este caso, el sub-arreglo de suma máxima es *{18, 20, -7, 12}*, y por lo tanto Selena debe comprar el día 7 y revender el día 11.

Si hay varios sub-arreglos de suma máxima, Selena dará prioridad al de índices más pequeños, porque entre más pronto ella compre y revenda, más contentos estarán sus jefes über-capitalistas.

En esta tarea, el profesor quiere que queden claros (o reforzados) los siguientes temas:

* La técnica de diseño de algoritmos divide y vencerás
* El problema del sub-arreglo de suma máxima
* El análisis a posteriori de caso *promedio*

Puede ayudar a Selena a resolver esta tarea utilizando Java, C o C++.

## Descripción

### 1. Solución del problema

La solución al problema del sub-arreglo de suma máxima debe implementarse siguiendo la estrategia *divide y vencerás*, y la complejidad de la solución, de esta manera, será cuasilineal.

La entrada estará compuesta por un renglón que incluya un número *N* de días (identificados del *0* al *N-1*). Luego, en el siguiente renglón, *N* enteros representando cada uno el precio de la piña cada día.

La salida estará compuesta por dos líneas con el siguiente formato:

"Selena debe comprar las piñas el día %d", inicio

"Selena debe revender las piñas el día %d", fin

### Ejemplo

#### Entrada

17

100 113 110 85 105 102 86 63 81 101 94 106 101 79 94 90 97

#### Salida

Selena debe comprar las piñas el día 7.

Selena debe revender las piñas el día 11.

import java.util.ArrayList;

import java.util.List;

public class maxSum {

    private static class Pair<K, V> {

        public K a; public V b;

        public **Pair**(K *a*, V *b*) {

            this.a = *a*;

            this.b = *b*;

        }

        public K **getA**() {return a;}

        public V **getB**() {return b;}

    }

    private static List<Integer> **parseInput**(String *values*) {

        List<Integer> parsed = new **ArrayList**<>();

        String[] newValues = *values*.**split**(" ");

        for(int i = 0; i < newValues.length; i++) {

            parsed.**add**(Integer.**parseInt**(newValues[i]));

        }

        return parsed;

    }

    private static List<Integer> **getDifferences**(List<Integer> *arr*) {

        List<Integer> differences = new **ArrayList**<>();

        for (int i = 1; i < *arr*.**size**(); i++) {

            differences.**add**(*arr*.**get**(i) - *arr*.**get**(i - 1));

        }

        return differences;

    }

    private static Pair<Integer, Integer> **getIndex**(List<Integer> *arr*) {

        int currMax = 0, prevMax = 0, start = 0, i1 = 0, i2 = 0;

        for (int i = 0; i < *arr*.**size**(); i++) {

            currMax += *arr*.**get**(i);

            if (currMax < 0) {

                start = i + 1;

                currMax = 0;

            } else if (currMax > prevMax) {

                i2 = i;

                i1 = start;

                prevMax = currMax;

            }

        }

        return new **Pair**(i1, i2);

    }

    private static void **printList**(List<Integer> *arr*) {

        for (int i = 0; i < *arr*.**size**(); i++) {

            System.out.**printf**("%d ", *arr*.**get**(i));

        }

        System.out.**println**();

    }

    public static void **main**(String[] *args*) {

*// Entrada de tamaño de datos (no se necesita para Java)*

        int n = 17;

*// Datos de entrada separados por espacios*

        String data = "100 113 110 85 105 102 86 63 81 101 94 106 101 79 94 90 97";

        Pair<Integer, Integer> res = **getIndex**(**getDifferences**(**parseInput**(data)));

        System.out.**printf**("Selena debe comprar las piñas el día %d.\n", res.a);

        System.out.**printf**("Selena debe revender las piñas el día %d.\n", res.b + 1);

    }

}

### 2. Análisis a posteriori

Modifique el código de su solución para contar *comparaciones* entre elementos de sus arreglos (o elementos derivados de ellos, como acumuladores).

Ahora corra el algoritmo para arreglos de tamaño *n=1000* hasta *n=4000* con incremento de *10*:

* Genere en cada caso *n/10* arreglos de enteros con datos entre ***-n/2*** y ***n/2***.
* Calcule el promedio de *comparaciones* para esos *n/10* arreglos.
* Despliegue la información con el siguiente formato:

"%d\t%.2f\n", n, promedioComparaciones

Copie esos datos, péguelos tal cual en Excel y genere una *gráfica de dispersión* donde aparezcan dos curvas:

1. Promedio de *comparaciones* en función de *n*.
2. *nlog2(n)*.

Agregue a la primera curva la ecuación de tendencia (potencial, no polinómica) y el valor de ajuste estadístico *R2*.

Utilizando las siguientes ecuaciones, tabule para *N* desde 10 millones hasta 200 millones, utilizando un incremento de 10 millones:

1. La ecuación de tendencia potencial obtenida en el paso anterior.
2. *nlog2(n)*.
3. *nlog10(n)*.

No incluya aquí su tabla, pero sí una *gráfica* donde aparezcan las tres curvas correspondientes.

Escriba a qué *conclusiones* llega, a partir de estos datos y gráficas, al respecto de la complejidad algorítmica de su solución.

Dado que la curva correspondiente a la solución proporcionada siempre está por debajo de las líneas de los logaritmos calculados, se puede decir que es una solución lineal.

## Entregables y rúbrica

1. Ejercicios previos (100%)
   * Ejercicio de potencia con divide y vencerás (20%)
     + Solución 15%
     + Análisis justificado 5%
   * Búsqueda binaria 20%
   * *Mergesort* 20%
   * *Quicksort* 20%
   * Estadístico de orden *i*  20%
2. Subarreglo de suma máxima (100%)
3. Solución al problema (código fuentecorrecto) (70%)
   * División en sub-problemas 10%
   * Conquista (solución de sub-problemas) 20%
   * Combinación de sub-soluciones 20%
   * Consideración del sub-arreglo central máximo 20%
4. Análisis a posteriori del código de subarreglo de suma máxima (30%)
   * Código modificado 5%
   * Conteos correctos 5%
   * Código invocador del algoritmo (*main*) 5%
   * Primera gráfica de dispersión con ecuación y *R2* 5%
   * Segunda gráfica de dispersión 5%
   * Conclusiones 5%
   * *R2* 10%