

# Análisis Amortizado

## Ejercicio 1

Observemos que la cantidad de cuadrados perfectos desde 1 hasta  $n$  es  $\lfloor \sqrt{n} \rfloor$ , a lo que llamaremos  $m$ .

El objetivo es calcular el costo amortizado por operación, que está dado por  $C(n) = \frac{T(n)}{n}$ , donde  $T(n)$  es el tiempo total tras realizar  $n$  operaciones y  $n$  es el número total de operaciones.

El tiempo total  $T(n)$  está dado por:

$$T(n) = n - m + \sum_{i=1}^m i^2$$

Desarrollando esta expresión:

$$\begin{aligned} T(n) &= n - m + \frac{m(m+1)(2m+1)}{6} \\ &= n - m + \frac{2m^3 + 3m^2 + m}{6} \\ &= \frac{6n - 6m + 2m^3 + 3m^2 + m}{6} \\ &= \frac{6n + 2m^3 + 3m^2 - 5m}{6} \end{aligned}$$

Ahora calculamos  $C(n)$ :

$$\begin{aligned} C(n) &= \frac{T(n)}{n} \\ &= \frac{6n + 2m^3 + 3m^2 - 5m}{6n} \\ &= 1 + \frac{m^3}{3n} + \frac{m^2}{2n} - \frac{5m}{6n} \end{aligned}$$

Reemplazando  $m$  por  $\lfloor \sqrt{n} \rfloor$ :

$$C(n) = 1 + \frac{\lfloor \sqrt{n} \rfloor^3}{3n} + \frac{\lfloor \sqrt{n} \rfloor^2}{2n} - \frac{5\lfloor \sqrt{n} \rfloor}{6n}$$

Dado que es complicado trabajar con  $\lfloor \sqrt{n} \rfloor$ , aproximamos  $\lfloor \sqrt{n} \rfloor \approx \sqrt{n}$ :

$$\begin{aligned} C(n) &= 1 + \frac{n^{3/2}}{3n} + \frac{n^{2/2}}{2n} - \frac{5n^{1/2}}{6n} \\ &= 1 + \frac{n^{1/2}}{3} + \frac{1}{2} - \frac{5}{6n^{1/2}} \\ &= 1 + \frac{1}{2} + n^{1/2} - \frac{5}{6n^{1/2}} \in \Theta(\sqrt{n}) \end{aligned}$$

Por lo tanto, el costo amortizado por operación  $C(n)$  es del orden  $\Theta(\sqrt{n})$ .

## Ejercicio 2

### Minpila

La estructura **minpila** es una pila que permite obtener el elemento mínimo en tiempo constante. Se implementa utilizando una pila que almacena pares de enteros. Cada par consiste en:

1. **Elemento actual**: el valor del elemento que se inserta.
2. **Mínimo actual**: el valor mínimo de todos los elementos en la pila hasta ese momento.

### Implementación

- **Insertar (push)**: Cuando se inserta un nuevo elemento, se obtiene el mínimo actual de la pila. Luego, se compara el nuevo elemento con este mínimo. Se crea un par que contiene el nuevo elemento y el mínimo entre el nuevo elemento y el mínimo anterior, y este par se apila en la pila principal. Así, cada vez que se realiza una inserción, se mantiene un registro del mínimo hasta ese punto.
- **Sacar (pop)**: Para desapilar, se retira el par en la cima de la pila. El primer valor del par es el elemento que se ha desapilado, mientras que el segundo valor sigue representando el mínimo hasta ese momento.
- **Obtener el mínimo (getMin)**: Para acceder al mínimo, se puede obtener el segundo elemento del par en la cima de la pila, que siempre contendrá el mínimo actual.

### Mincola

La **mincola** es una estructura que simula el comportamiento de una cola utilizando dos pilas de tipo **minpila**: **entrada** y **salida**. Cada una de estas pilas actúa como un tipo de dato abstracto que mantiene su propio registro del mínimo.

Al insertar un elemento, se apila en **entrada** usando la funcionalidad de **minpila**. Para obtener o sacar el elemento al frente, si **salida** está vacía, se mueven todos los elementos de **entrada** a **salida**, invirtiendo su orden. Si **salida** no está vacía, simplemente se desapila el elemento en la cima de **salida**. Para obtener el valor mínimo, se comparan los mínimos de **entrada** y **salida**, devolviendo el menor.

### Complejidad Amortizada

El análisis de la complejidad amortizada indica que, aunque algunas operaciones como **dequeue** y **front** pueden ser  $O(n)$  en ciertos casos, el costo promedio de las operaciones a lo largo de múltiples interacciones es  $O(1)$ . Esto se debe a que cada elemento se mueve a **salida** solo una vez, y las operaciones de inserción y obtención del mínimo se realizan en  $O(1)$ . Por lo tanto, el costo amortizado por operación en la mincola es  $O(1)$ .

## Dividir y Conquistar

### Subarreglo con Suma Máxima

```
int maxSubArray(int A[], int izq, int der) {  
    if (der - izq + 1 == 1)  
        return A[izq];  
    else {  
        // Dividir  
        int m = (izq + der) / 2;
```

```

// Conquistar
int s1 = maxSubArray(A, izq, m);
int s2 = maxSubArray(A, m + 1, der);

// Combinar
int s = A[m];
int M1 = A[m];
for (int j = m - 1; j >= izq; j--) {
    s += A[j];
    if (s > M1) M1 = s;
}

s = A[m + 1];
int M2 = A[m + 1];
for (int j = m + 2; j <= der; j++) {
    s += A[j];
    if (s > M2) M2 = s;
}

return max({s1, s2, M1, M2, M1 + M2});
}
}

```

El tiempo de ejecución es  $O(n \log n)$ .

## Fibonacci

Asumiremos que la multiplicación de dos números tiene un tiempo constante. Aunque esto no siempre es cierto en todas las circunstancias, para este problema consideramos que la operación de multiplicar dos números es  $O(1)$ .

Siendo  $f_n$  el  $n$ -ésimo número de Fibonacci, tenemos:

$$\begin{pmatrix} f_{n-1} & f_n \\ f_n & f_{n+1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n$$

Como las multiplicaciones de matrices son siempre entre matrices  $2 \times 2$ , podemos considerar el tiempo de multiplicación como constante. Realizar las  $n$  multiplicaciones nos tomaría tiempo  $O(n)$ .

Hay que notar que la multiplicación de matrices es asociativa.

Para  $n = 4$ , por ejemplo, podemos considerar multiplicar la mitad de la derecha y luego la de la izquierda, esto reduce el número de multiplicaciones a 3:

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^4 = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} = \left( \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \right) \left( \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \right)$$

Notar ahora que ambas mitades son iguales, por lo que solo necesito multiplicar una mitad y multiplicar el resultado por sí mismo, terminando con un total de 2 multiplicaciones.

Generalizando, cuando  $n = 2^k$ ,  $k \in \mathbb{N}$  ( $n$  es potencia de 2), podemos definir recursivamente la matriz para  $n$ :

$$M(n) = \begin{cases} M, & n = 1 \\ (M(n/2))^2, & n > 1 \end{cases}$$

De esta forma se realizan  $\log(n)$  multiplicaciones de matrices.

Para el caso general, es decir, cuando  $n$  no es necesariamente una potencia de 2, en algun punto puede que  $n/2$  sea impar. En este caso se realiza division entera (i.e: funcion piso) y luego se multiplica por el resto:

$$M(n) = \begin{cases} M, & n = 1 \\ (M(n/2))^2, & n > 1 \text{ y } n \text{ par} \\ M \times (M(\lfloor n/2 \rfloor))^2, & n > 1 \text{ y } n \text{ impar} \end{cases}$$

Asi, tenemos un algoritmo que calcula el  $n$ -esimo numero de Fibonacci en tiempo  $O(\log(n))$

## Desafío ultra difícil

[Click aqui](#)

## Correctitud

### MergeSort

```
#include <bits/stdc++.h>
using namespace std;

// Merges two subarrays of arr[].
// First subarray is arr[left..mid]
// Second subarray is arr[mid+1..right]
void merge(vector<int>& arr, int left, int mid, int right){
    int n1 = mid - left + 1;
    int n2 = right - mid;

    // Create temp vectors
    vector<int> L(n1), R(n2);

    // Copy data to temp vectors L[] and R[]
    for (int i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];

    int i = 0, j = 0;
    int k = left;

    // Merge the temp vectors back
    // into arr[left..right]
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
```

```

        i++;
    }
    else {
        arr[k] = R[j];
        j++;
    }
    k++;
}

// Copy the remaining elements of L[],
// if there are any
while (i < n1) {
    arr[k] = L[i];
    i++;
    k++;
}

// Copy the remaining elements of R[],
// if there are any
while (j < n2) {
    arr[k] = R[j];
    j++;
    k++;
}
}

// begin is for left index and end is right index
// of the sub-array of arr to be sorted
void mergeSort(vector<int>& arr, int left, int right){
    if (left >= right)
        return;

    int mid = left + (right - left) / 2;
    mergeSort(arr, left, mid);
    mergeSort(arr, mid + 1, right);
    merge(arr, left, mid, right);
}

// Function to print a vector
void printVector(vector<int>& arr){
    for (int i = 0; i < arr.size(); i++)
        cout << arr[i] << " ";
    cout << endl;
}

```

Extraído de: <https://www.geeksforgeeks.org/merge-sort/>

**Correctitud:** Unir dos arreglos (merge)

En clase mencioné que se podía asumir la correctitud de la función merge o demostrarla usando un invariante de ciclo (en una prueba podrían solicitar la demostración formal de su correctitud).

Dado que contamos con dos subarreglos ordenados,  $L$  y  $R$ , queremos demostrar que la función `merge` combina correctamente ambos subarreglos en uno solo, también ordenado.

Para realizar esta demostración, utilizaremos un **invariante de ciclo**. Esta técnica es similar a la inducción matemática, donde demostramos que una condición específica se mantiene verdadera en cada iteración del ciclo.

Asumimos la notación del arreglo  $arr$ , donde  $arr[left...right]$  representa el subarreglo de  $arr$  que va desde la posición  $left$  hasta  $right$ . Notemos que si  $right < left$ , esto representa el conjunto vacío.

La **invariante** es que, en la iteración  $k$ -ésima del ciclo de combinación, el subarreglo  $arr[left...k]$  contiene los elementos ordenados más pequeños entre los que se han procesado de los subarreglos  $L$  y  $R$ .

- **Caso base:**

Antes de entrar al ciclo, ningún elemento ha sido procesado, es decir, el subarreglo  $arr[left...left-1]$  es vacío. Por definición, un subarreglo vacío está ordenado, por lo tanto, la invariante se cumple en este caso.

- **Hipótesis inductiva:**

Supongamos que en la iteración  $k$ -ésima, el subarreglo  $arr[left...k]$  contiene los primeros  $k$  elementos más pequeños de la combinación de  $L$  y  $R$ , y que los índices  $i$  y  $j$  (que recorren  $L$  y  $R$ , respectivamente) apuntan a las posiciones correctas de ambos subarreglos.

- **Paso inductivo:**

Vamos a demostrar que la invariante se sigue cumpliendo en la iteración  $k+1$ : Dado que  $arr[left...k]$  ya contiene los  $k$  elementos más pequeños de la unión de  $L$  y  $R$ , los elementos en las posiciones actuales de  $L[i]$  y  $R[j]$  deben ser mayores o iguales a  $arr[k]$ , debido a que ambos subarreglos están ordenados. Por lo tanto, el siguiente elemento a insertar,  $arr[k+1]$ , será el menor entre  $L[i]$  y  $R[j]$ .

- **Caso 1:** Si  $L[i] \leq R[j]$ , entonces  $L[i]$  es el menor de todos los elementos restantes en  $L[i...len(L)]$  y  $R[j...len(R)]$ . Por lo tanto,  $L[i]$  será el próximo elemento que se colocará en  $arr[k+1]$ , y la invariante se mantendrá.
- **Caso 2:** Si  $L[i] > R[j]$ , entonces  $R[j]$  es el menor de todos los elementos restantes en los dos subarreglos. Así,  $R[j]$  será el siguiente en insertarse en  $arr[k+1]$ , y el subarreglo seguirá ordenado, manteniendo la invariante.

En ambos casos, se demuestra que después de la iteración  $k+1$ , el subarreglo  $arr[left...k+1]$  estará ordenado y contendrá los  $k+1$  elementos más pequeños de la unión de  $L$  y  $R$ , cumpliendo la invariante.

Por lo tanto, la función `merge` es correcta, ya que al finalizar el ciclo, todo el subarreglo  $arr[left...right]$  estará ordenado como resultado de la combinación de los elementos de  $L$  y  $R$ .

### Correctitud: MergeSort(arr, left, right)

Vamos a demostrar la correctitud de `MergeSort` mediante inducción sobre el tamaño del arreglo.

Definimos  $n$  como  $n = right - left + 1$ .

- **Caso Base:**  $1 \leq n \leq 1$

Un arreglo que contiene un único elemento está ordenado por definición.

- **Hipótesis inductiva:**  $1 \leq n \leq k$

Supondremos que para un arreglo de tamaño  $n$  que cumple  $1 \leq n \leq k$ , `MergeSort` ordena el arreglo correctamente.

- **Paso inductivo:**  $1 \leq n \leq k + 1$

Queremos demostrar que `MergeSort` funciona para un tamaño  $n$  que cumple  $1 \leq n \leq k + 1$ . Dado que asumimos que funciona para  $n \leq k$ , solo necesitamos verificar el caso para  $n = k + 1$ .

Supongamos que llamamos a `MergeSort(arr, left, right)` donde  $n = \text{right} - \text{left} + 1 = k + 1$  (notamos que  $k + 1 > 1$ ). La función divide el arreglo en dos mitades, realizando las siguientes llamadas recursivas:

`MergeSort(arr, left,  $\lfloor \frac{\text{left} + \text{right}}{2} \rfloor$ )` y `MergeSort(arr,  $\lfloor \frac{\text{left} + \text{right}}{2} \rfloor + 1, \text{right}$ )`

Observamos que la longitud de la primera mitad,  $\lfloor \frac{\text{left} + \text{right}}{2} \rfloor - \text{left} + 1$ , será menor que  $n$ , ya que al menos quedará un elemento en la otra mitad (si esto no ocurre, la implementación es errónea, ya que podría generar un bucle infinito). Lo mismo aplica para la segunda mitad, donde la longitud será:  $\text{right} - \left( \lfloor \frac{\text{left} + \text{right}}{2} \rfloor + 1 \right) + 1$

Dado que ambas mitades son menores que  $n$ , sabemos que, por la hipótesis inductiva, esas llamadas recursivas ordenarán las mitades. A partir de la demostración anterior de la correctitud de `merge`, podemos concluir que `MergeSort` es correcto, ya que combinará adecuadamente las mitades ordenadas en un arreglo completamente ordenado.