Estructuras de Datos y Algoritmos

Ordenamiento

Yerko Ortiz[1]

Clase 7: Ordenamiento, Agosto 2023

[1]Escuela de informática y telecomunicaciones
Universidad Diego Portales

Contenidos

Introducción

Introducción

- El ordenamiento de un conjunto de datos es uno de los problemas más estudiados en ciencias de computación.
- Muchos problemas en ciencias de computación se pueden reducir a un problema de ordenamiento o como mínimo el ordenamiento de los datos es uno de los pasos de la solución.
- Muchas de las técnicas de análisis y construcción de algoritmos aparecen en el contexto de algoritmos de ordenamiento.
    - Insertion Sort: Online Algorithms
    - Merge Sort: Divide and Conquer
    - Quick Sort: Randomized Algorithms
    - Heap Sort: Data Structures

Definición

Input

Sea una secuencia A de n enteros $\langle a_0, a_1, a_2, ..., a_{n-1} \rangle$

Output

Una permutación A' obtenida de la secuencia A de forma tal que $a'_0 \leq a'_1 \leq a'_2 \leq ... \leq a'_{n-1}$

Ordenamiento Interno y Externo

Ordenamiento Interno

Se le llama *internal sorting* a cualquier procedimiento de ordenamiento que ocurra de inicio a fin en la memoria principal(ram) del sistema.

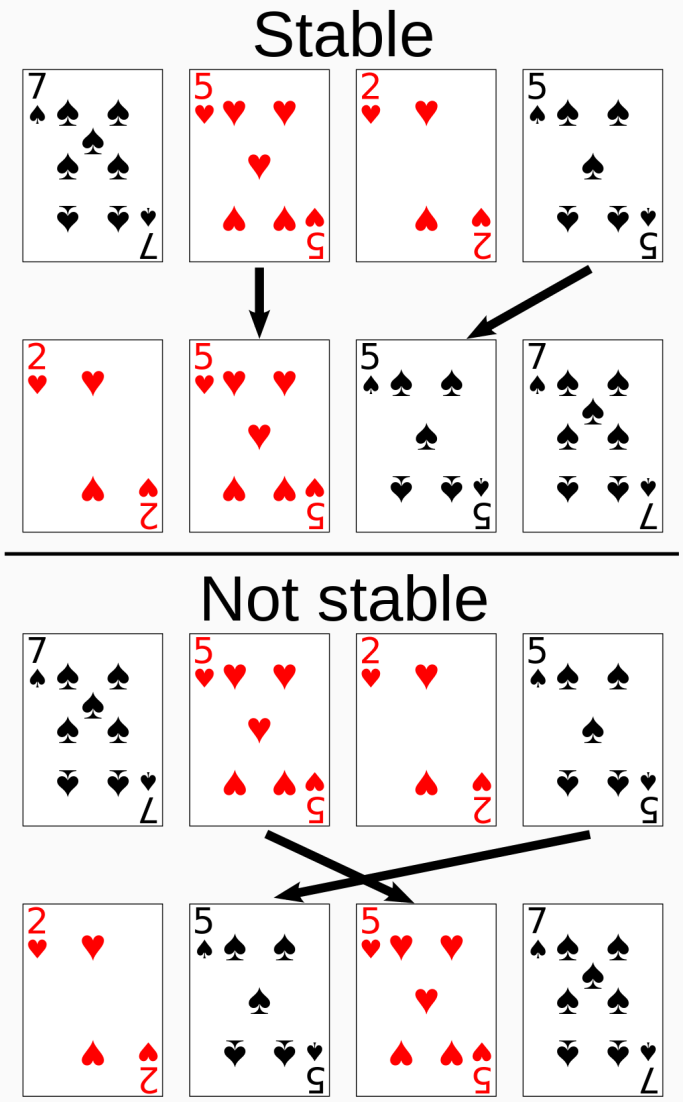
Ordenamiento Externo

Se le llama *external sorting* a cualquier procedimiento de ordenamiento que necesite recurrir a la unidad de almacenamiento secundaria del sistema(file system).

Ordenamiento Estable

Ordenamiento Estable

Se le llama *estable* a cualquier algoritmo de ordenamiento que ordene los elementos repetidos en el **mismo orden** que aparecen en la entrada.



Problemas Relacionados

- Búsqueda: buscar un elemento en un conjunto se puede realizar en $\mathcal{O}(lgn)$ si el conjunto está ordenado.
- Par más cercano: Encontrar un par de números enteros en un conjunto cuya diferencia es la mínima.
- Elementos duplicados: Encontrar elementos duplicados en un conjunto.
- Palabras que empiezan con un prefijo: Encontrar el subconjunto de palabras que empiezan con un determinado patrón.
- Selección: Encontrar el k-ésimo elemento más grande en un conjunto.

Bubble Sort

Bubble Sort

Bubble Sort es un algoritmo de ordenamiento que realiza swaps entre **elementos adyacentes** si no están en orden. En cada iteración i deja el elemento de mayor valor en la posición $N - i - 1$.

```
1   static void bubbleSort(int []A, int N) {
2       int temp;
3       for(int i = 0; i < N - 1; i++) {
4           for(int j = 0; j < N - i - 1; j++){
5               if(A[j] > A[j+1]) {
6                   temp = A[j];
7                   A[j] = A[j+1];
8                   A[j+1] = temp;
9               }
10          }
11      }
12  }
```

Análisis de Bubble Sort

En primer ciclo i llega hasta N - 1

El segundo ciclo(anidado) j llega hasta N - i - 1

Si hacemos un par de cálculos para el número de iteraciones del ciclo interior:

$(n-0-1)+(n-1-1)+(n-2-1)+...+2+1$

Que se traduce a:

$(n-1)+(n-2)+(n-3)+...+2+1$

Esta suma es conocida como serie aritmética:

$\sum_0^k = \frac{k(k+1)}{2}$

Substituimos k con n - 1 y el resultado es $\frac{n^2-n}{2}$, por lo que podemos decir que el tiempo de ejecución es en función de $\Theta(n^2)$

Selection Sort

Selection Sort

Imagine que usted quiere ordenar un librero horizontal con N libros por apellido del autor. Un algoritmo a usar en esta situación podría ser el siguiente:

- Buscar el libro con menor orden lexicográfico y ponerlo en la posición de más la izquierda.
- Buscar el libro con menor orden lexicográfico en los libros restantes y ponerlo a la derecha del libro anterior.
- Repetir el paso anterior hasta haber tener el librero ordenado.

Selection Sort es justamente ese algoritmo pero generalizado para ordenar cualquier tipo de *record* con sus respectiva *llave*(key).

Selection Sort

Si llevamos dicho algoritmo a código podría ser:

```
1   static void selectionSort(int []A, int N) {
2       int temp;
3       for(int i = 0; i < N - 1; i++) {
4           int min = i;
5           for(int j = i + 1; j < N; j++) {
6               if(A[j] < A[min]) {
7                   min = j;
8               }
9           }
10          temp = A[min];
11          A[min] = A[i];
12          A[i] = temp;
13      }
14  }
```

Análisis de Selection Sort

El primer ciclo itera desde 0 hasta N - 1.

El ciclo interno itera desde i hasta N.

Si hacemos un par de cálculos en papel podemos decir que el número de ciclos internos serán en la forma:

$(n-1)+(n-2)+(n-3)+...+2+1$

Esta suma es conocida como serie aritmética:

$\sum_0^k = \frac{k(k+1)}{2}$

Substituimos k con n - 1 y el resultado es $\frac{n^2-n}{2}$, por lo que podemos decir que el tiempo de ejecución es en función de $\Theta(n^2)$

Insertion Sort

Insertion Sort

Ahora imagine que quiere ordenar una mano de cartas. Un algoritmo podría ser el siguiente:

- Deje todas las cartas apiladas en una superficie
- Tome la primera carta y déjela en la mano. Puesto que su mano solo tiene una carta podemos afirmar que está ordenada.
- Tome la siguiente carta e insértela en su mano buscando su posición correspondiente. Puesto que la mano estaba ordenada a priori si se inserta de manera ordenada la mano seguirá ordenada.
- Repita el paso anterior hasta tener todas las cartas en la mano ordenadas.

Insertion Sort

```
1   static void insertionSort(int []A, int N) {
2       int key;
3       for(int i = 1; i < N; i++) {
4           key = A[i];
5           int j = i - 1;
6           while(j >= 0 && A[j] > key) {
7               A[j+1] = A[j];
8               j--;
9           }
10          A[j+1] = key;
11      }
12  }
```

Análisis de Insertion Sort

Antes de analizar el código conviene clarificar ciertas cosas:

- Key es el elemento a insertar en el subconjunto ordenado A[0:i]
- Dicho de otra forma, key es la carta a insertar y A[0:i] es la mano de cartas.

Volviendo al análisis podemos decir que el peor caso es siempre insertar al final de la mano la carta en cuestión.

Esto lo podemos describir como:

$1 + 2 + ... + (N - 2) + (N - 1)$

Lo cual como ya hemos visto es una serie aritmética.

Se puede afirmar que el peor caso de Insertion Sort es $\mathcal{O}(N^2)$

Merge Sort

- Tiempo de Ejecución $\Theta(Nlg(N))$
- Es un algoritmo Divide and Conquer

Input

Sea arreglo A de enteros: $\langle a_0, a_1, a_2, ..., a_{n-1} \rangle$

Sea p, r: índices del sub arreglo respectivo, donde $p \geq 0$ y $r \leq n - 1$

Output

Subarreglo A[p..r] que contiene los elementos ordenados en orden ascendente

Divide And Conquer

Divide and conquer es una técnica de diseño de algoritmos. Sus pasos son los siguientes:

- Divide: Dividir el problema en un conjunto de subproblemas que son **instancias reducidas** del problema original.
- Conquer: Resolver los subproblemas resolviendolos de manera **recursiva**. Si los subproblemas son lo suficientemente pequeños entonces pueden ser resueltos como *caso base*.
- Combine: Combinar las soluciones de los subproblemas para darle solución al problema original.

Divide And Conquer

En el caso de MergeSort los pasos podrían describirse así:

- Divide: Dividir el subarreglo A[p:r] por la mitad formando dos subarreglos A[p:q] y A[q + 1:r].
- Conquer: Ordenar cada subarreglo A[p:q], A[q + 1:r] de forma recursiva usando MergeSort
- Combinar: Mezclar cada subarreglo ya ordenado en A[p:q] para producir la solución esperada.

Merge Sort

MergeSort(A, p, r)

1. Si $p \geq r$ entonces el subarreglo A[p:r] tiene a lo más un elemento. Esto permite puede afirmar que el subarreglo está ordenado. Por lo que el algoritmo termina.
2. En caso contrario:
   A. Instanciar q con valor $\lfloor \frac{(p+r)}{2} \rfloor$
   B. MergeSort(A, p, q)
   C. MergeSort(A, q + 1, r)
   D. Merge(A, p, q, r)

Merge Sort

De la descripción anterior podemos denotar lo siguiente:

- El **caso base** de MergeSort(A, p, r) es cuando $p \geq r$. Este caso base tiene por premisa que todo arreglo de tamaño 0 o tamaño 1 está siempre vacío. Por lo tanto es la solución al subproblema.
- Puesto que las llamadas recursivas MergeSort(A, p, q) y MergeSort(A, q + 1, r) reducen el tamaño del problema inicial a la mitad ($q = (p + r)/2$). Eventualmente llegarán al caso base donde el subarreglo A[p:q] y A[q+1:r] está ordenado(caso base).
- El paso final es empezar a mezclar los subarreglos ordenados A[p:q] y A[q+1:r] de manera ordenada en el subarreglo A[p:r]. Una vez que todos los subarreglos son mezclados, el problema original MergeSort(A, p, r) ya habrá ordenado el subarreglo A[p:r],

Merge Sort

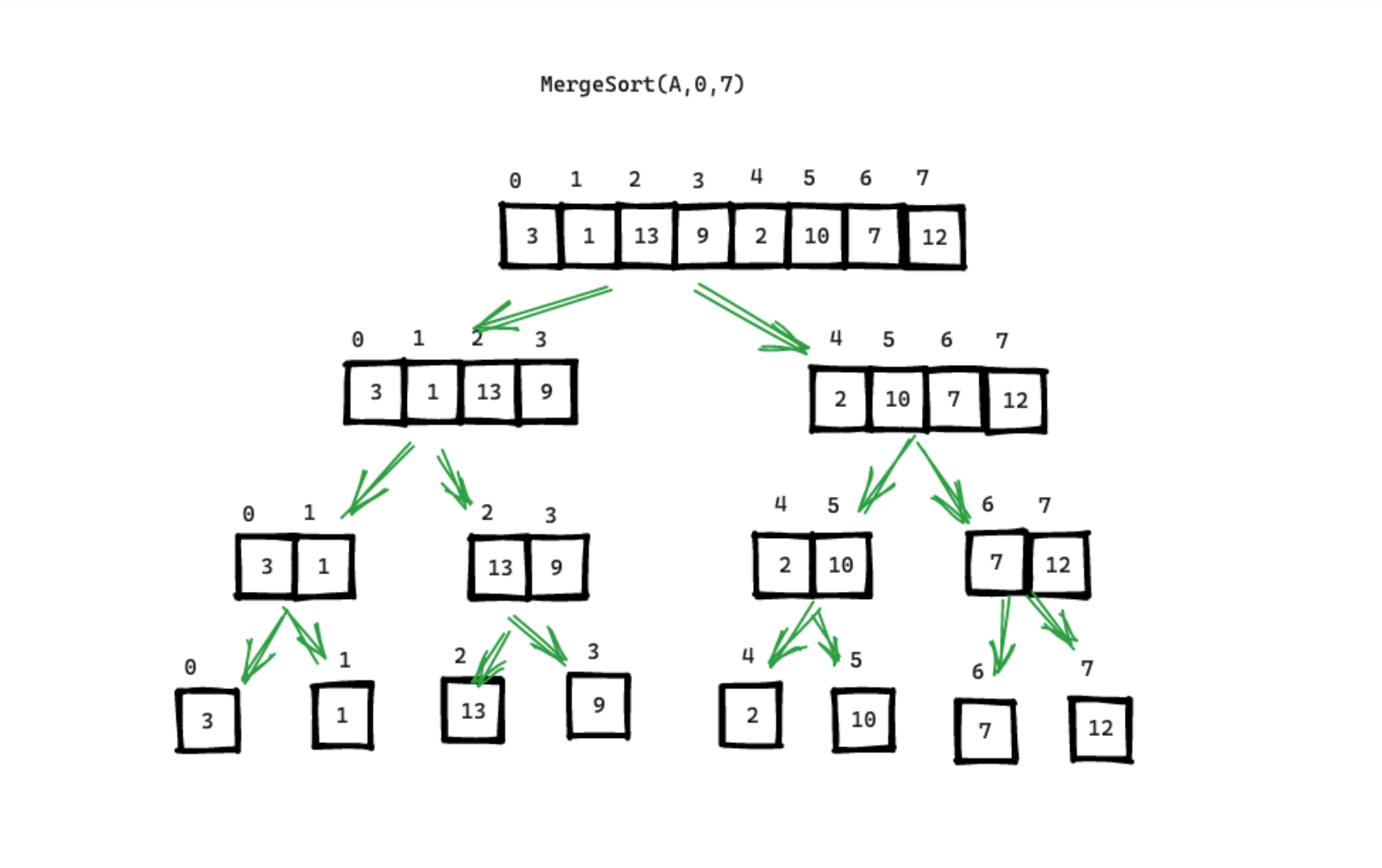


Figure 1: Ilustración de las llamadas recursivas de MergeSort y los respectivos subarreglos A[p:q], A[q+1:r].

Merge Sort

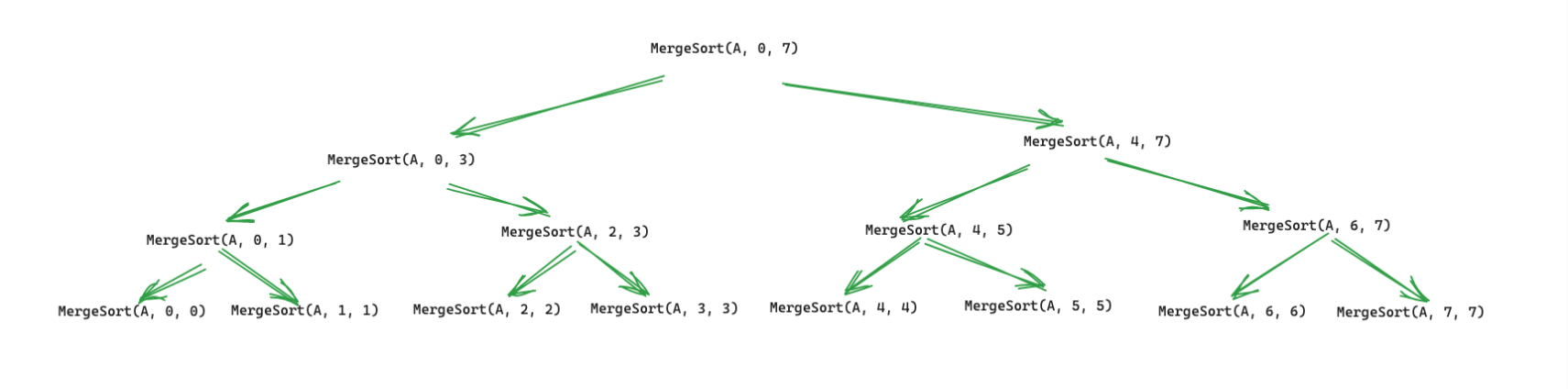


Figure 2: Ilustración de las llamadas recursivas de MergeSort y los respectivos valores de p y r en cada llamada. Se puede apreciar que cuando $p \geq r$ las llamadas de MergeSort llegan a su término porque convergen al caso base de la recursión.

Merge

Hasta el momento podemos afirmar que MergeSort divide el problema original hasta llegar a su caso base donde los subarreglos están ordenados.

Si pensamos en divide and conquer como una secuencia de pasos podemos afirmar que tenemos listo el divide y el conquer pero aun nos falta el combinar las soluciones para dar solución al problema original.

Merge

Para mezclar las soluciones tenemos que dar solución al siguiente problema:

Sea A un arreglo $\langle a_0, a_1, ...a_{n-1} \rangle$ donde $a_0 \leq a_1 \leq ... \leq a_{n-1}$

Sea B un arreglo $\langle b_0, b_1, ...b_{m-1} \rangle$ donde $b_0 \leq b_1 \leq ... \leq b_{m-1}$.

Diseñe un algoritmo que retorne un arreglo C $\langle c_0, c_1, ...c_{n+m-1} \rangle$ donde $c_0 \leq c_1 \leq ... \leq c_{n+m-1}$ y donde C contiene todos los elementos de A y B.

En otras palabras mezclar A y B en otro arreglo C de forma que este mantiene el orden ascendente que ya existía en A y B.

Merge

Con la premisa de que A y B ya están ordenados de manera ascendente este problema puede ser solucionado con la siguiente idea:

- En c_0 debe ir el mínimo de la unión de A y B es decir comparar a_0 con b_0 y quedarse con el menor.
- Con la posición c_1 escoger el mínimo de A y B que no haya sido incluido anteriormente.
- Repetir el paso anterior sucesivamente para todo c_i hasta haber agregado todos los elementos en A y B.

Merge

Input

Sea A un arreglo de enteros: $\langle a_0, a_1, ..., a_{n-1} \rangle$

Sean p,q,r los respectivos indices de cada subarreglo: A[p:q] y A[q+1:r]

Output

Ordenar el subarreglo A[p:r] de forma que

$a_p \leq a_{p+1} \leq a_{p+2} \leq ... \leq a_{r-1} \leq a_r$

Merge

Merge(A, p, q, r)

1. Instanciar n_1 con valor $q - p + 1$ e instanciar n_2 con valor $r - q$.
2. Instanciar los arreglos L y R con tamaño n_1 y n_2 respectivamente.
3. Copiar los elementos de A[p:q] en el arreglo L y copiar los elementos de A[q+1:r] en el arreglo R.
4. Instanciar i, j con valor 0 y k con valor p.
5. Mientras $i < n_1$, $j < n_2$ y $k < n_1 + n_2$
    A. Si $L[i] \leq R[j]$ asignar a A[k] el valor de L[i], luego incrementar i y k en uno.
    B. En caso contrario asignar a A[k] el valor de R[j], luego incrementar el valor de j y k en uno.

Merge

Merge(A, p, q, r)

6. Mientras $i < n_1$

   A. Asignar a A[k] el valor de L[i], luego incrementar i y k en uno.

7. Mientras $j < n_2$

   A. Asignar a A[k] el valor de R[j], luego incrementar j y k en uno.

MergeSort

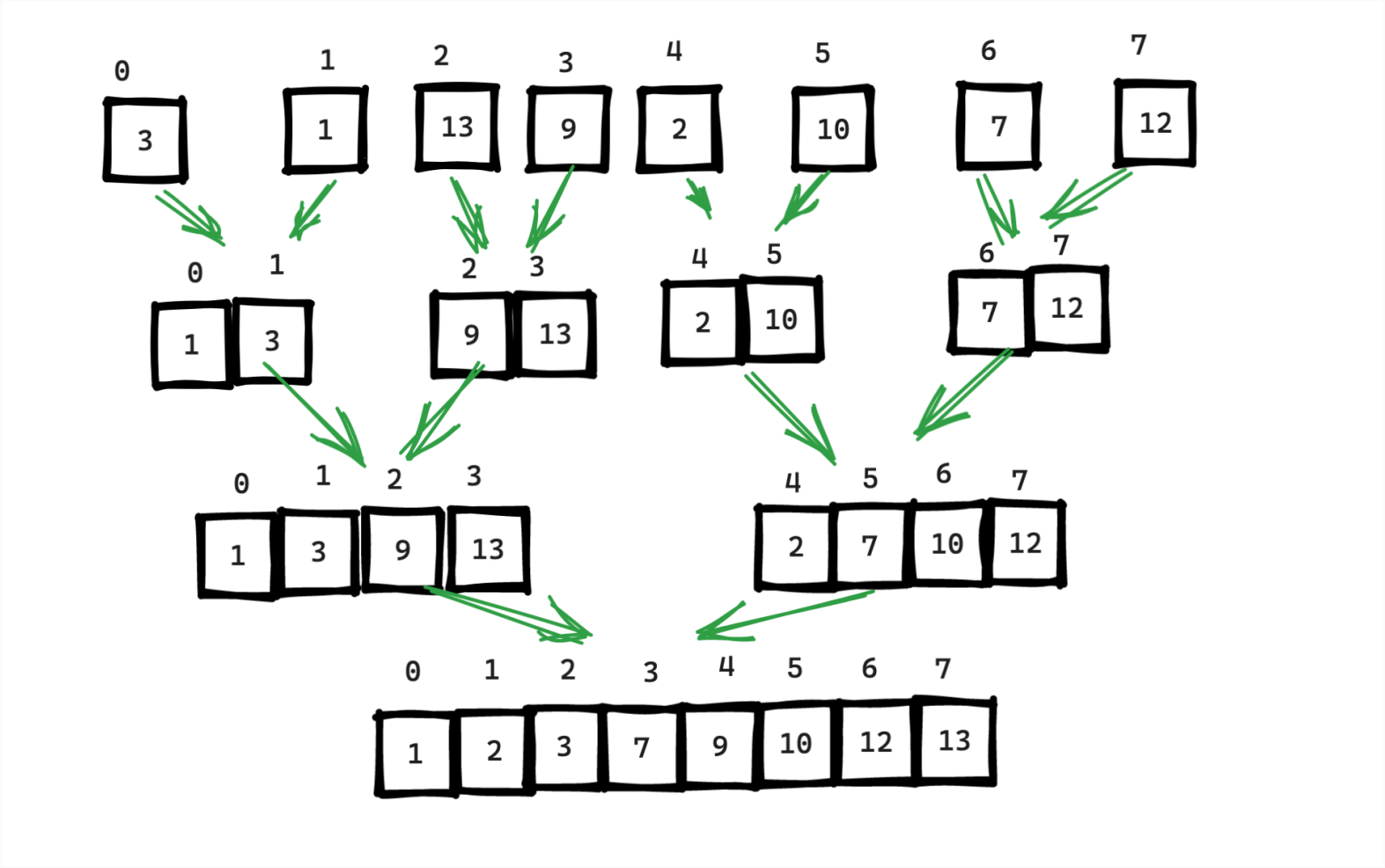


Figure 3: Al ejecutar Merge(A, p, q, r) para cada uno de los subarreglos ordenados previamente iremos será posible ordenar por completo el subarreglo A[p:r].

MergeSort

```
1    static void mergeSort(int[] arr, int lo, int hi) {
2        if(lo >= hi) {
3            return;
4        }
5        int q = (lo + hi)/2;
6        mergeSort(arr, lo, q);
7        mergeSort(arr, q + 1, hi);
8        merge(arr, lo, (lo + hi) / 2, hi);
9    }
```

MergeSort

```
1   static void merge(int A[], int p, int q, int r) {
2       int n1 = q - p + 1; // mid - lo + 1
3       int n2 = r - q; // hi - mid
4
5       int L[] = new int[n1];
6       int R[] = new int[n2];
7
8       for (int i = 0; i < n1; i++)
9         L[i] = A[p + i];
10      for (int j = 0; j < n2; j++)
11        R[j] = A[q + 1 + j];
12
13      int i = 0, j = 0, k = p;
14
15      while (i < n1 && j < n2) {
16        if (L[i] <= R[j]) {
17          A[k] = L[i];
18          i++;
19        } else {
20          A[k] = R[j];
21          j++;
22        }
23        k++;
24      }
25  // ... continua en la siguiente slide
```

MergeSort

```
1   static void merge(int A[], int p, int q, int r) {
2       //...
3       while (i < n1) {
4         array[k] = L[i];
5         i++;
6         k++;
7       }
8
9       while (j < n2) {
10        array[k] = M[j];
11        j++;
12        k++;
13      }
14    }
```

MergeSort

Para calcular el tiempo de ejecución tenemos que considerar que es una función recursiva es decir una función que se llama a sí misma.

1. Sabemos que en cada llamada el problema se reduce a la mitad porque $q = \lfloor \frac{p+q}{2} \rfloor$.
2. Por lo que podemos describir el tiempo de ejecución de la siguiente forma: $T(N) = T(N/2) + f(N)$

En la expresión $T(N)$ es una función que describe el tiempo de ejecución en función del tamaño de entrada N. Sabemos que en cada llamada recursiva N disminuye a la mitad por lo que de ahí viene el término $T(N/2)$. Por otro lado el término $f(N)$ viene de que en cada llamada recursiva se hace uso de Merge(A, p, q, r) por lo que $f(N)$ representa en tiempo de ejecución de Merge(A, p, q, r).

MergeSort

Antes de calcular el resultado de nuestra expresión $T(N) = T(N/2) + f(N)$, debemos analizar Merge(A, p, q, r) para obtener el tiempo de ejecución de este en términos asintóticos.

Este algoritmo es iterativo por lo que podemos observar la cantidad de iteraciones de cada loop.

1. Copiar los elementos en L y R tiene un comportamiento lineal puesto que se itera hasta n_1 y n_2 respectivamente por lo que sería $\mathcal{O}(N)$ respectivamente.

2. El siguiente es el ciclo while que itera mientras $i < n_1$ y $j < n_2$. Este ciclo mueve los elementos de L y R al arreglo A de forma ordenada. Puesto que itera a lo largo de L y R respectivamente realiza $n_1 + n_2$ iteraciones en su peor caso. Esto tiene un comportamiento lineal por lo que sería $\mathcal{O}(N)$ su tiempo de ejecución.

3. Solo queda ver los otros dos ciclos while que llenan A[p:r] con los elementos del arreglo L o R que aun falten. Ambos tienen un comportamiento lineal puesto que realizan a lo más n_1 o n_2 iteraciones respectivamente.

Por lo que podemos decir que Merge(A, p, q, r) tiene tiempo de ejecución $\mathcal{O}(N)$ en su peor caso.

MergeSort

Volviendo al análisis de MergeSort(A, p, r) nuestra expresión del tiempo de ejecución toma la siguiente forma:

$$T(N) = T(N/2) + \Theta(N)$$

Ahora solo nos falta resolver dicha expresión. Para nuestra conveniencia existe un teorema llamado el teorema maestro. Dicho teorema sirve para resolver recurrencias de la forma:

$$T(N) = aT(\frac{N}{b}) + f(N)$$

Si aplicamos dicho teorema(los fundamentos y demostración de este se deben estudiar en un curso superior) el resultado de nuestra expresión será:

$$T(N) = \Theta(Nlg(N))$$

QuickSort

- Tiempo de Ejecución $\Theta(NlgN)$
- Es un algoritmo Divide and Conquer
- No es estable
- Tiene peor caso $\mathcal{O}(N^2)$
- Tiene caso promedio $\Theta(NlgN)$

Input

Sea arreglo A de enteros: $\langle a_0, a_1, a_2, ..., a_{n-1} \rangle$

Sea p, r: índices del sub arreglo respectivo, donde $p \geq 0$ y $r \leq n - 1$

Output

Subarreglo A[p..r] que contiene los elementos ordenados en orden ascendente

QuickSort

Como algoritmo divide and conquer QuickSort podría ser descrito de la siguiente forma:

- Divide: Se escoge cualquier elemento en el subarreglo A[p:r] como pivote. Rearreglar el arreglo A[p:r] de forma que los elementos menores que el pivote queden a la izquierda de este y los mayores a la derecha.
- Conquer: Recursivamente ordenar los elementos a la izquierda y derecha del pivote.
- Combinar: Puesto que el paso anterior ya ordena, este paso no hace nada.

QuickSort

QuickSort(A, p, r)

1. if $p \geq r$
    A. return
2. q = Partition(A, p, r)
3. QuickSort(A, p, q - 1)
4. QuickSort(A, q + 1, r)

Partition

La clave para que QuickSort funcione es el algoritmo de partición. El algoritmo puede ser descrito de la siguiente forma:

Input

1. Sea A un arreglo de enteros
2. Sean p y r números enteros que representan los índices de inicios del subarreglo A[p:r] de forma tal que $0 \leq p \leq r \leq N - 1$, donde N es el tamaño del arreglo A.

Output

1. Rearreglo de los elementos de A, de forma que los elementos del subarreglo A[p:r] estarán ordenados de forma ascendente.

Para una versión determinista de QuickSort utilizaremos como pivote al último elemento del subarreglo A[p:r], es decir A[r].

Partition

Partition(A, p, r)

1. set q = p
2. for u = p to r - 1:
    A. if A[u] < A[r]:
        a. Swap(A[q], A[u])
        b. q = q + 1
3. Swap(A[q], A[r])
4. Return q

Esta versión de Partition utiliza como pivote A[r]. Al término del algoritmo todos los elementos menores que A[r] estarán a la izquierda de este, mientras que los mayores a la derecha.

Análisis

Partition

La primera observación a realizar es que Partition es un algoritmo que tiene un comportamiento lineal. Itera desde q hasta r - 1 por lo que se puede dicer que tiene tiempo de ejecución $\mathcal{O}(N)$.

QuickSort

Por el lado de QuickSort tenemos una expresión similar a la de MergeSort, pero con una sutil diferencia. La forma en que el problema se reduce en cada recursión depende del pivote que escojamos. Si el pivote resulta quedar en la posición p o posición r en cada llamada a Partition tendremos el peor caso $\mathcal{O}(N^2)$, por otro lado si el pivote tiende a estar en la posición $\lfloor r/2 \rfloor$ QuickSort tendrá tiempo de ejecución $\mathcal{O}(NlgN)$.

Análisis

- Tiempo de ejecución cuando el pivote tiende a estar en los extremos de A[p:r]: $T(N) = T(N-1) + \mathcal{O}(N) = \mathcal{O}(N^2)$

- Tiempo de ejecución cuando el pivote tiende a estar al medio de A[p:r]: $T(N) = 2T(N/2) + \mathcal{O}(N) = \mathcal{O}(NlgN)$

HeapSort

- Tiempo de Ejecución ($Nlg(N)$)
- Es un algoritmo basado en una estructura de datos llamada Heap.
- Funciona in place
- No es estable

Input

Sea arreglo A de enteros: $\langle a_0, a_1, a_2, ..., a_{n-1} \rangle$
Sea N el tamaño del arreglo A.

Output

Una permutación A' obtenida de la secuencia A de forma tal que $a'_0 \leq a'_1 \leq a'_2 \leq ... \leq a'_{n-1}$

Heap

Un Heap es una estructura de datos que si bien es representada mediante un árbol binario, es implementada con un arreglo.

Un arreglo A[1:N] representa al Heap y sus elementos. Donde N es el tamaño del arreglo, por su lado el Heap tiene un atributo que llamaremos heapSize que representa la cantidad de elementos que tiene el Heap. La raíz del Heap está en A[1].

Se puede aseverar lo siguiente:

- $0 \leq heapSize \leq N$.
- Si heapSize es igual a 0 entonces el heap está vacío.

Heap

Al ser representado mediante un árbol binario, cada elemento tiene un hijo izquierdo, un hijo derecho y un padre.

Parent(i)

1. return $\lfloor i/2 \rfloor$

Left(i)

1. return 2 * i

Right(i)

1. return 2 * i + 1

Heap

Los heap tienen una propiedad que les permite mantener una estructura donde los padres siempre tienen un valor mayor que los hijos o al contrario los padres siempre tienen menor valor que los hijos.

MaxHeap Property

En un MaxHeap se dice que los nodos padre siempre tienen mayor valor que los hijos. $A[Parent(i)] \geq A[i]$

MinHeap Property

En un MinHeap se dice que los nodos padre siempre tienen menor valor que los hijos. $A[Parent(i)] \leq A[i]$

Heap

Antes de continuar es importante hacer algunas observaciones y definiciones que serán útiles más tarde:

1. En un MaxHeap el elemento máximo estará en la raíz del Heap, es decir A[1].
2. En un MinHeap el elemento mínimo estará en la raíz del Heap, es decir A[1].
3. Una hoja en un árbol es un elemento que no tiene hijos.
4. La altura de un árbol se define como la distancia máxima entre la raíz y una de sus hojas.
5. Si un Heap tiene N elementos entonces, dicho Heap tiene altura $\lfloor lg(N) \rfloor$.

MaxHeapify

MaxHeapify es un algoritmo que aplicado a un elemento A[i] en el Heap, reordena los subárboles del elemento de forma que se cumpla la propiedad de MaxHeap. Este método asume que los subárboles del elemento A[i] cumplen la propiedad de MaxHeap.

Input

1. Un arreglo de enteros A que contiene al heap en el subarreglo A[1:heapSize].
2. Un indice i que corresponde al elemento A[i] desde el cual se evaluará la propiedad de MaxHeap hacía abajo(subárboles).

MaxHeapify

MaxHeapify(A,i)

1. l = Left(i)
2. r = Right(i)
3. largest = A[i]
4. if l $\leq$ heapSize and A[l] > A[i]
    A. largest = A[l]
5. if r $\leq$ heapSize and A[r] > A[i]
    A. largest = A[r]
6. if *largest* $\neq$ *i*
    A. Swap(A[i], A[largest])
    B. MaxHeapify(A, largest)

BuildMaxHeap

BuildMaxHeap es un algoritmo que transforma un arreglo con tamaño N en un MaxHeap. Este algoritmo tiene una ejecución bottom up, ejecutándose desde las hojas hacía la raíz. Cabe destacar que si se tiene un Heap con tamaño heapSize entonces las hojas estarán ubicadas en el subarreglo A[heapSize/2 + 1: heapSize].

Input

1. Un arreglo A.
2. Un entero N que denota el tamaño de A.

BuildMaxHeap

BuildMaxHeap(A, N)

1. heapSize = N
2. for i = $\lfloor N/2 \rfloor$ down to 1
    A. MaxHeapify(A, i)

HeapSort

HeapSort hace uso de los métodos que implementamos previamente.

Input

1. Un arreglo A.
2. Un entero N que denota la cantidad de elementos en el arreglo A.

Output

1. Una permutación A' obtenida de la secuencia A de forma tal que $a'_0 \leq a'_1 \leq a'_2 \leq ... \leq a'_{n-1}$

HeapSort

HeapSort(A,N)

1. BuildMaxHeap(A,N)
2. for i = N down to 2
    A. Swap(A[1], A[i])
    B. heapSize = heapSize - 1
    C. MaxHeapify(A, 1)

Tabla Resumen

Algoritmo	Tiempo Ejecución Promedio	¿Es estable?
InsertionSort	$\Theta(N^2)$	Si
SelectionSort	$\Theta(N^2)$	No
BubbleSort	$\Theta(N^2)$	Si
MergeSort	$\Theta(Nlg(N))$	Si
QuickSort	$\Theta(Nlg(N))$	No
HeapSort	$\Theta(Nlg(N))$	No

Referencias i


A. V. Aho, J. E. Hopcroft, and J. D. Ullman.
***Data Structures and Algorithms*.**
Pearson, 1th edition, 1983.


T. H. Cormen.
***Algorithms Unlocked*.**
The MIT Press, 1th edition, 2013.


T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein.
***Introduction to Algorithms*.**
The MIT Press, 4th edition, 2022.


S. S. Skiena.
***The Algorithm Design Manual*.**
Springer, 2nd edition, 2008.