



Tecnológico
de Monterrey

Algoritmos de ordenamiento

Estructura de Datos

Víctor de la Cueva

vcueva@itesm.mx

Arreglo ordenado

- La búsqueda binaria es tan solo una aplicación en donde se requiere tener un **arreglo ordenado**.
- En realidad, hay un **gran número de aplicaciones** donde el tener un arreglo ordenado nos facilita mucho la operación deseada:
 - Buscar
 - Regresar el menor o el mayor
 - Insertar nuevos elementos
 - etc.
- El ordenamiento, al igual que la búsqueda, no es una estructura de datos, pero es una de las **operaciones más importantes** a realizar sobre los datos guardados en una estructura.

Métodos de ordenamiento

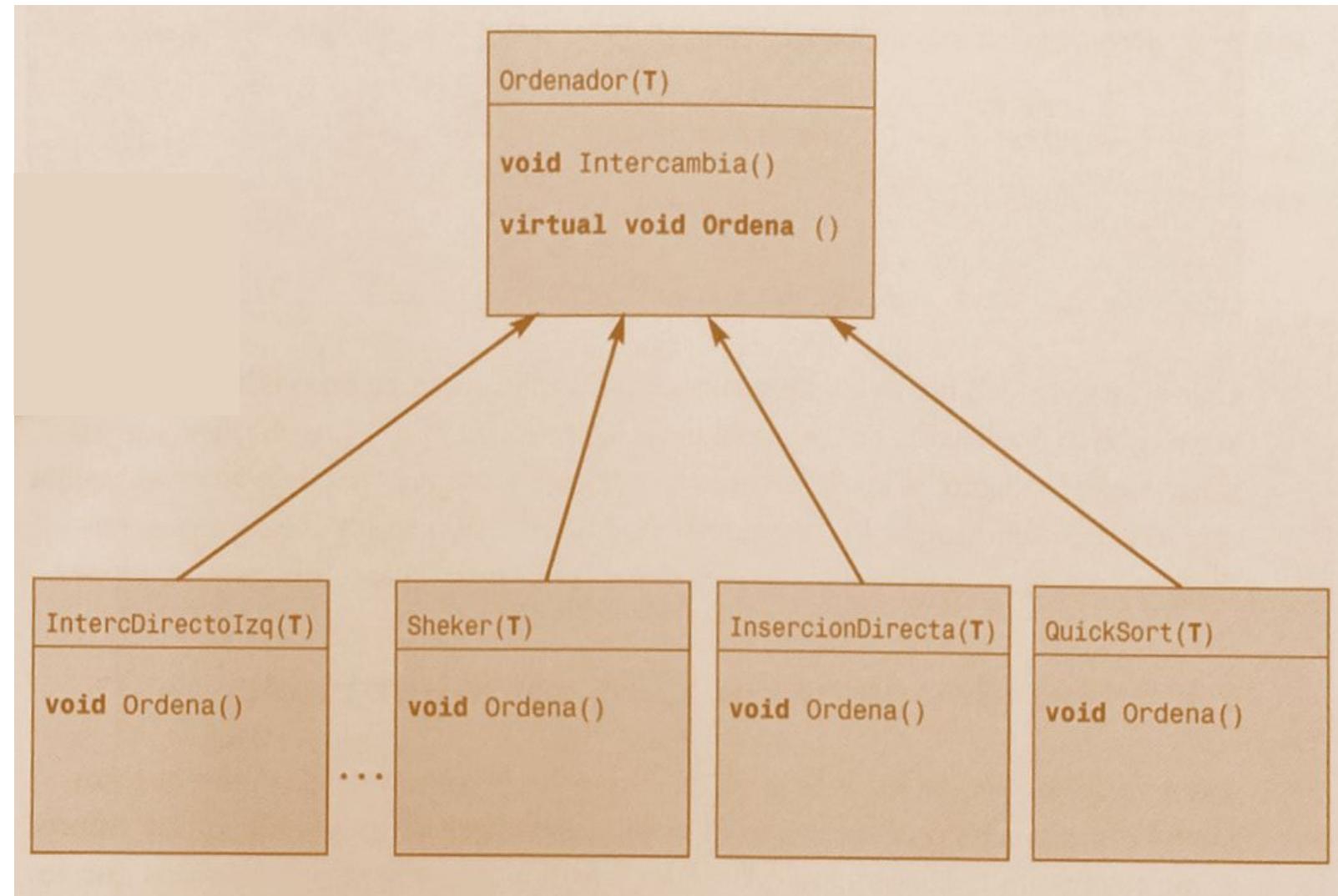
- Existe un **gran número** de métodos de ordenamiento para un arreglo.
- Cada algoritmo tiene diferentes **ventajas y desventajas**
 - NO existe **EL ALGORITMO** de ordenamiento que sea el más eficiente en todos los casos
 - La eficiencia **depende del problema** al que se aplique.
- La selección del algoritmo depende del **análisis** del problema
 - Obtener características de los datos y decidir cuál es la mejor opción.

Ordenamiento en C++

- Una forma muy común de programar los métodos de ordenamiento en C++ es definir:
 - Una clase base abstracta: **Ordenador(T)**
 - Un conjunto de clases derivadas: nombre del algoritmo implementado con un método **ordena()**.
- Cada una de las clases derivadas representa uno de los métodos que se estudiarán en este capítulo.

NOTA: En los proyectos no lo haremos así.

Gráficamente



Operaciones básicas

- Todos los métodos que veremos utilizan dos operaciones básicas para llevar a cabo el ordenamiento de los elementos de un arreglo:
 - La **comparación**
 - El movimiento o **intercambio** de los mismos
- Por esta razón, en la clase abstracta se incluye el método **intercambia()**, que será común a todas las clases derivadas y que tiene por objetivo intercambiar los valores de dos posiciones del arreglo.

Ejemplo de clase abstracta

- La clase abstracta tendrá un **método virtual puro** que se redefinirá en cada subclase, dependiendo del método de ordenamiento que se esté implementando.
- Además, tiene un método auxiliar **intercambia()**, para generalizar la operación de intercambio que será usada por las subclases.
- Si lo que se quiere ordenar son objetos, hay que tener en cuenta que se deben **sobrecargar los operadores de comparación** en las clases correspondientes, para hacer uso de los métodos que vamos a programar.

Clase abstracta

```
/* Clase abstracta que se utiliza para definir clases derivadas que re-
➥presentan cada uno de los métodos de ordenación interna. */
template <class T>
class Ordenador
{
public:
    void Intercambia (int, int, Arreglo<T> *);
    virtual void Ordena (Arreglo<T> *) = 0;
};

/* Método auxiliar que intercambia los contenidos de dos elementos del
➥arreglo que se está ordenando. */
template <class T>
void Ordenador<T>::Intercambia(int Ind1, int Ind2, Arreglo<T> *Arre)
{
    T Auxiliar;
    Auxiliar= Arre->RegresaValor(Ind1);
    Arre->AsignaValor(Ind1, Arre->RegresaValor(Ind2));
    Arre->AsignaValor(Ind2, Auxiliar);
}
```

Otra forma de implementación

- Otra forma de implementar los métodos de ordenamiento es la misma que seguimos para los métodos de búsqueda, es decir, haciendo que los métodos de ordenamiento sean **métodos de la clase Arreglo**.
- Esta es la forma en la que lo vamos a hacer en los proyectos.

Métodos de ordenamiento

- Los métodos de ordenamiento que vamos a ver son:
 - Métodos de ordenamiento simple:
 - Selection Sort (Selección)
 - Bubble sort (Burbuja)
 - Insertion Sort (Inserción)
 - Merge Sort
 - Quick Sort

Selection Sort

```
function A* start
    closedset := []
    openset := []
    came_from := {}

    g_score[st]
    f_score[st]

    while openset:
        current = min(openset, key=f_score.get)
        if current == goal:
            path = reconstruct_path(came_from, current)
            return path

        remove(current, openset)
        for neighbor in neighbors(current):
            if neighbor in closedset:
                continue
            tentative_g_score = g_score[current] + distance(current, neighbor)

            if neighbor not in openset or tentative_g_score <= g_score[neighbor]:
                came_from[neighbor] = current
                g_score[neighbor] = tentative_g_score
                f_score[neighbor] = g_score[neighbor] + h_score[neighbor]
                if neighbor not in openset:
                    openset.append(neighbor)

    return fail

function reconstruct_path(came_from, current):
    if current not in came_from:
        return None
    p := current
    while p != start:
        p = came_from[p]
    return p
```

Selection sort

- Es uno de los métodos más simples que existen pero también es uno de los más ineficientes.
- Se trata de seleccionar el menor (mayor) en cada paso y colocarlo en la parte superior (izquierda) del arreglo, disminuir en uno de longitud y repetir el proceso.

Algoritmo Selection Sort

```
for i=0 hasta n-2 // para arreglos base 0
    mínimo = i;
    for j=i+1 hasta n-1
        if lista[j] < lista[mínimo] entonces
            mínimo = j      /* posición del menor */
        fin if
    fin for
    intercambiar(i, mínimo)
fin for
```

Bubble Sort

```
function A* state
closedset := []
openset := []
came_from := {}

g_score[state] = 0
f_score[state] = g_score[state] + h(state)

while openset:
    current = openset.pop(0)
    if current == goal:
        path = []
        while current != start:
            path.append(current)
            current = came_from[current]
        path.append(start)
        return path[::-1]

        remove
        add cur
        fop ex
        if
        ter
        if
        return fail

function reconstruct_path(came_from, start, goal):
    if goal not in came_from:
        return None
    else:
        p = [goal]
        p.append(goal)
        current = goal
        while current != start:
            current = came_from[current]
            p.append(current)
        p.append(start)
        return p[::-1]
```

Ordenamiento de burbuja

- Es otro ordenamiento muy simple pero muy ineficiente, aunque es menos ineficiente que el anterior si se coloca una bandera.
- La idea es que los elementos menos densos (los menores) “floten” hacia la superficie.
- Para lograrlo, se comparan los elementos de dos en dos yendo hacia abajo.
 - Si el elemento de abajo es menor que el de arriba, se intercambian.
- Al final de la primera pasada, el elemento más denso estará en la parte inferior.
- El proceso se repite con un elemento menos.

Algoritmo Bubble Sort

procedimiento DeLaBurbuja ($a_0, a_1, a_2, \dots, a_{(n-1)}$)

para $i \leftarrow 1$ hasta n hacer

para $j \leftarrow 0$ hasta $n - i$ hacer

si $a_{(j)} > a_{(j+1)}$ entonces

aux $\leftarrow a_{(j)}$

$a_{(j)} \leftarrow a_{(j+1)}$

$a_{(j+1)} \leftarrow aux$

fin si

fin para

fin para

fin procedimiento

n-1

n-i-1

Insertion Sort

```
function A* state
closedset := []
openset := []
came_from := {}

g_score[state] = 0
f_score[state] = g_score[state] + h(state)

while openset:
    current = openset.pop(0)
    if came_from[current] == None:
        remove(current, openset)
        for neighbor in neighbors(current):
            if neighbor not in closedset:
                f_score[neighbor] = g_score[neighbor] + h(neighbor)
                if neighbor not in openset:
                    openset.append(neighbor)
                    came_from[neighbor] = current
                else:
                    if f_score[neighbor] > f_score[neighbor]:
                        f_score[neighbor] = f_score[neighbor]
                        came_from[neighbor] = current
    if current == goal:
        return came_from
    else:
        remove(current, openset)
        for neighbor in neighbors(current):
            if neighbor not in closedset:
                f_score[neighbor] = g_score[neighbor] + h(neighbor)
                if neighbor not in openset:
                    openset.append(neighbor)
                    came_from[neighbor] = current
                else:
                    if f_score[neighbor] > f_score[neighbor]:
                        f_score[neighbor] = f_score[neighbor]
                        came_from[neighbor] = current
    if len(openset) == 0:
        return None

def reconstruct_path(came_from, start, end):
    if end not in came_from:
        return None
    path = [end]
    while path[-1] != start:
        path.append(came_from[path[-1]])
    path.reverse()
    return path
```

Insertion Sort

- En cada paso, el elemento en cuestión queda colocado en el lugar correcto del arreglo, de acuerdo a todos los elementos que están antes que él.
- Cada elemento se compara con los que están arriba de él:
 - Si es mayor, ahí se queda.
 - Si es menor se mueve uno hacia arriba y se repite la operación hasta llegar a su posición correcta.

Algoritmo Insertion Sort

- Hacerlo

```
for i = 1 to n
    for j = i to 1
        if a[j] < a[j-1]
            swap(j,j-1)
        else
            break
```

Merge Sort

```
function A* state
closedset := []
openset := []
came_from := {}

g_score[state] = 0
f_score[state] = g_score[state] + h(state)

while openset:
    current = openset.pop(0)
    if current == goal:
        path = []
        while current != start:
            path.append(current)
            current = came_from[current]
        path.append(start)
        return path[::-1]

    remove(current, openset)
    for neighbor in neighbors(current):
        if neighbor in closedset:
            continue
        tentative_g_score = g_score[current] + distance(current, neighbor)
        if neighbor not in openset or tentative_g_score < g_score[neighbor]:
            came_from[neighbor] = current
            g_score[neighbor] = tentative_g_score
            f_score[neighbor] = g_score[neighbor] + h(neighbor)
            if neighbor not in openset:
                openset.append(neighbor)

    closedset.append(current)
    if len(openset) == 0:
        return None

function reconstruct_path(came_from, start, goal):
    if goal not in came_from:
        return None
    else:
        path = [goal]
        current = goal
        while current != start:
            current = came_from[current]
            path.append(current)
        path.append(start)
        path.reverse()
        return path
```



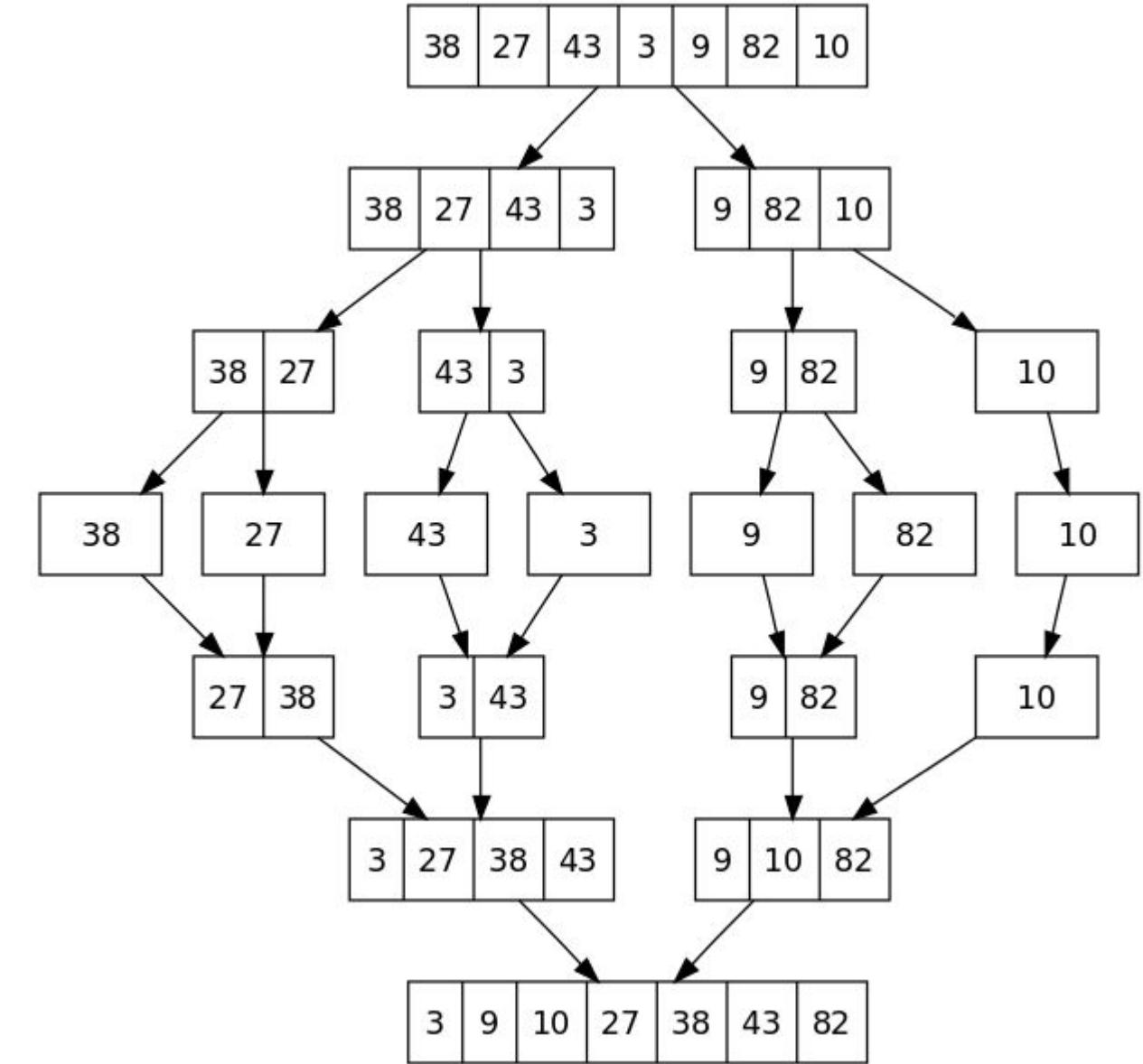
Merge Sort

- Fue inventado por **John von Neumann** en 1945.
- Es un algoritmo muy eficiente basado en comparación, de hecho, óptimo en trabajo realizado: $O(n \log n)$.
- Requiere una memoria adicional igual a **n** (número de datos) para trabajar.
- Es uno de los algoritmos llamados “**divide y vencerás**”, ya que explota la idea de dividir el problema mayor en problemas más pequeños, los cuales tardan menos en resolverse, para después usar esas soluciones para encontrar la solución del problema mayor.

Trabajo del merge sort

- Merge sort trabaja con una idea muy simple en tres pasos:
 - Divide el arreglo desordenado en dos partes.
 - Ordena cada una de las partes recursivamente (llamando a merge sort)
 - Combina (merge) las partes resultantes ya ordenadas.
- El método hace una copia del arreglo de entrada **a**, a un arreglo auxiliar **aux** y luego combina (*merge*) de regreso al arreglo de entrada.
- Para ordenar el arreglo **a[lo..hi]**, se divide en dos partes **a[lo..mid]** y **a[mid+1,hi]**, se ordenan independientemente y se mezclan (*merge*).

Ejemplo gráfico



Fuente: https://en.wikipedia.org/wiki/Merge_sort#/media/File:Merge_sort_algorithm_diagram.svg

Algoritmo del merge sort

```
int aux[n];

void sort(int a[], int lo, int hi)
{
    // ordenar a[lo..hi].
    if (hi <= lo) return;
    int mid = lo + (hi - lo)/2;
    sort(a, lo, mid); // ordenar mitad izquierda
    sort(a, mid+1, hi); // ordenar mitad derecha
    merge(a, lo, mid, hi); // Merge las dos mitades
}
```

```
void merge(int a[], int lo, int mid, int hi)
{
    // Merge a[lo..mid] con a[mid+1..hi]
    // copiar a[lo..hi] a aux[lo..hi]
    // Merge de regreso en a[lo..hi]
}
```

Proceso de Merging y ejemplo

- La clave del algoritmo es un proceso de mezcla llamado **merging**, que toma dos arreglos ya ordenados para formar un solo arreglo ordenado.



4



Ejemplo de Merge Sort

	a[]																
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
merge(a, 0, 0, 1)	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E	
merge(a, 2, 2, 3)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E	
merge(a, 0, 1, 3)	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E	
merge(a, 4, 4, 5)	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E	
merge(a, 6, 6, 7)	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E	
merge(a, 4, 5, 7)	E	G	M	R	E	S	O	R	S	T	E	X	A	M	P	L	E
merge(a, 0, 3, 7)	E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E	
merge(a, 8, 8, 9)	E	E	G	M	O	R	R	S	E	T	X	A	M	P	L	E	
merge(a, 10, 10, 11)	E	E	G	M	O	R	R	S	E	T	A	X	M	P	L	E	
merge(a, 8, 9, 11)	E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E	
merge(a, 12, 12, 13)	E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E	
merge(a, 14, 14, 15)	E	E	G	M	O	R	R	S	A	E	T	X	M	P	E	L	
merge(a, 12, 13, 15)	E	E	G	M	O	R	R	S	A	E	T	X	E	L	M	P	
merge(a, 8, 11, 15)	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X	
merge(a, 0, 7, 15)	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X	

Quick Sort

```
function A* state
closedset := []
openset := []
came_from := {}

g_score[state] = 0
f_score[state] = g_score[state] + h(state)

while openset:
    current = openset.pop(0)
    if current == goal:
        path = came_from[current]
        break
    remove(current, openset)
    for neighbor in neighbors(current):
        if neighbor in closedset:
            continue
        tentative_g_score = g_score[current] + distance(current, neighbor)
        if neighbor not in openset or tentative_g_score <= g_score[neighbor]:
            came_from[neighbor] = current
            g_score[neighbor] = tentative_g_score
            f_score[neighbor] = g_score[neighbor] + h(neighbor)
            if neighbor not in openset:
                openset.append(neighbor)
        else:
            if tentative_g_score > g_score[neighbor]:
                came_from[neighbor] = current
                g_score[neighbor] = tentative_g_score
                f_score[neighbor] = g_score[neighbor] + h(neighbor)

    if len(openset) == 0:
        return fail

function recache(state):
    if current == state:
        p := parent
        return p
    else:
        return recache(parent)
```



Quick Sort

- Fue inventado por el científico británico Sir Charles Antony “Tony” Richard Hoare (Premio Turing 1980), en 1961.
- Probablemente, es el algoritmo de ordenamiento más utilizado.
- Es un algoritmo del enfoque “Divide y Vencerás”.
- Es sumamente eficiente en promedio: $O(n \log n)$, aunque su eficiencia en el peor de los casos es $O(n^2)$.
- Se puede programar elegantemente en forma recursiva.

Algoritmo Quick Sort

- El algoritmo trabaja de la siguiente forma:
 - Elegir un elemento de la lista de elementos a ordenar, al que llamaremos **pivote**.
 - **Partición**: resituar los demás elementos de la lista a cada lado del pivote, de manera que a un lado queden todos los menores que él, y al otro los mayores. En este momento, el pivote ocupa exactamente el lugar que le corresponderá en la lista ordenada.
 - La lista queda separada en dos sublistas, una formada por los elementos a la izquierda del pivote, y otra por los elementos a su derecha.
 - Repetir este proceso de forma recursiva para cada sublista mientras éstas contengan más de un elemento. Una vez terminado este proceso todos los elementos estarán ordenados.

Proceso de partición y ejemplo

- Fase I: Repetir hasta que los apuntadores i y j se crucen:
 - Mover i de izquierda a derecha mientras que $a[i] < a[lo]$.
 - Mover j de derecha a izquierda mientras que $a[j] > a[lo]$.
 - Cambiar $a[i]$ con $a[j]$.
- Fase II: Cuando los apuntadores se cruzan
 - Cambiar $a[lo]$ con $a[j]$.



Algoritmo Quick Sort

```
void sort(int a[], int lo, int hi)
{
    if (hi <= lo) return;
    int j = partition(a, lo, hi);
    // Ordena parte izquierda a[lo .. j-1]
    sort(a, lo, j-1);
    // Ordena parte derecha a[j+1 .. hi]
    sort(a, j+1, hi);
}
```

```
int partition(int a[], int lo, int hi)
{ // Partir en a[lo..i-1] y a[i], a[i+1..hi]
    int i = lo, j = hi+1; // iniciar índices
    int v = a[lo]; // pivote
    while (true) {
        // Mueve i a la derecho mientras a[i] < v
        // Mueve j a la izquierda mientras a[j] > v
        if (i >= j) break; // si se cruzan, termina
        // si no, intercambia los elementos a[i] y a[j]
    }
    v = a[j]; // Poner pivote en su posición
    return j; // con a[lo..j-1] <= a[j] <= a[j+1..hi].
}
```

Ejemplo Quick Sort

lo	j	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Initial values			Q	U	I	C	K	S	O	R	T	E	X	A	M	P	L	E
random shuffle			K	R	A	T	E	L	E	P	T	I	M	Q	C	X	O	S
	0	5	15	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O
	0	3	4	E	C	A	E	I	K	L	P	U	T	M	Q	R	X	O
	0	2	2	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O
	0	0	1	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O
	1		1	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O
	4		4	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O
no partition for subarrays of size 1	6	6	15	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O
	7	9	15	A	C	E	E	I	K	L	M	O	P	T	Q	R	X	U
	7	7	8	A	C	E	E	I	K	L	M	O	P	T	Q	R	X	U
	8		8	A	C	E	E	I	K	L	M	O	P	T	Q	R	X	U
	10	13	15	A	C	E	E	I	K	L	M	O	P	S	Q	R	T	U
	10	12	12	A	C	E	E	I	K	L	M	O	P	R	Q	S	T	U
	10	11	11	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U
	10		10	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U
	14	14	15	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U
	15		15	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U
result			A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X

Referencias

- [1] Elliot B. Koffman. Objects, abstraction, data structures, and design: using C++. John Wiley & Sons Inc. (2005).
- [2] Silvia Guardati. Estructura de datos orientada a objetos: Algoritmos con C++. Pearson (2007).
- [3] Wikipedia. Algoritmos de Ordenamiento.
https://es.wikipedia.org/wiki/Algoritmo_de_ordenamiento. Consultado el 13 de febrero de 2017.
- [4] R. Sedgewick and K. Wayne. Algorithms. 4th ed, Adison-Wesley (2011).