
UNRN
Universidad Nacional
de Río Negro
| unrionegro

Complejidad

14

2023

UNRN
Universidad Nacional
de Río Negro

P1

Especificación

¿Por qué especificar?

-> Permite comprender ***qué*** problema debemos resolver de forma independiente a ***cómo*** hacerlo.

Por ejemplo:

- **“qué”:** ordenar una lista de menor a mayor.
- **“cómo”:** algoritmo a emplear.

¿Por qué especificar?

-> Posibilita el diseño de una solución más abstracta y, en consecuencia, más general.

-> Permite modificar la implementación según lo requiera el contexto de uso, que puede cambiar, al mismo tiempo que se preserva el problema resuelto.

Ejemplo: Especificación

Tomemos el algoritmo que computa el mínimo de una lista no vacía y observemos si esos estados sirven a los efectos de especificar el problema.

```
int min_arreglo(int capacidad, int arreglo[])


precondición: { l == l0 and len(l) >= 1 }
poscondición: { (exists i: int)(0 <= i < len(l) and ret_value == l[i])
          and (all x: int)(0 <= x < len(l) implies ret_value <= l[x]) }
```

Ejemplo: Motivación

Desde un punto de vista práctico, no todo algoritmo que satisface la especificación da lo mismo.

```
int min_arr(int capacidad, int arreglo[]){
    int i;
    int min = arreglo[0];
    for (i = 0; i < capacidad; i++){
        if (min > arreglo[i]){
            min = arreglo[i];
        }
    }
    return min;
}
```

```
int minimo_en_arreglo (int arreglo[], int capacidad)
{
    int minimo = 10000;

    for (int i = 0; i < capacidad; i++)
    {
        if (arreglo[i] < minimo)
        {
            minimo = arreglo[i];

        }
    }

    return minimo;
}
```

```
int minimo_en_arreglo (int arreglo[], int capacidad)
{
    ordenar_arreglo (arreglo[], capacidad);
    return arreglo[0];
}
```

¿Cuál es el “mejor”?

Ejemplo: Especificación, pero algo más complejo

```
procedimiento DeLaBurbuja (a_0, a_1, a_2, ..., a_(n-1))
    para i ← 1 hasta n − 1 hacer
        para j ← 0 hasta n − i hacer
            si a_(j) > a_(j+1) entonces
                aux ← a_(j)
                a_(j) ← a_(j+1)
                a_(j+1) ← aux
            fin si
        fin para
    fin para
fin procedimiento
```

6 5 3 1 8 7 2 4

Ejemplo: Ordenamiento por mezcla

Conceptualmente, el ordenamiento por mezcla funciona de la siguiente manera:

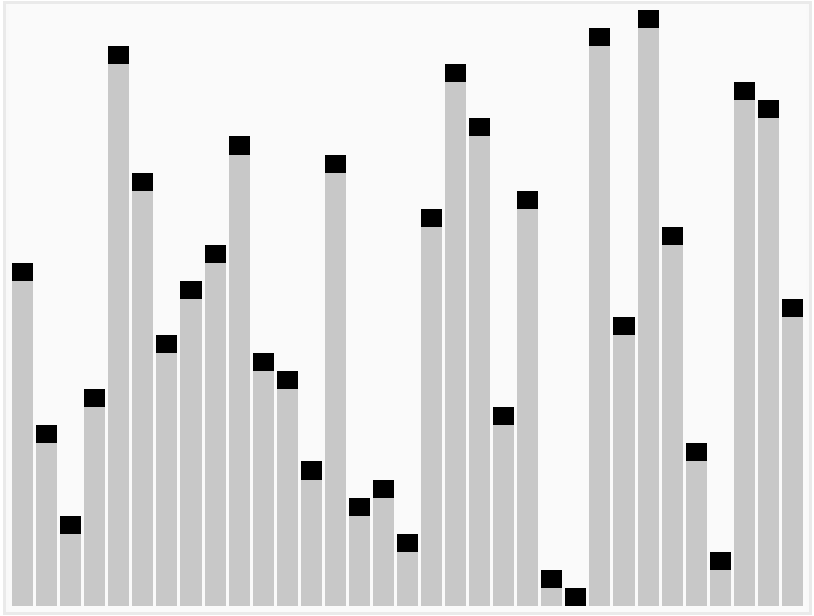
1. Si la longitud de la lista es 0 o 1, entonces ya está ordenada. En otro caso:
2. Dividir la lista desordenada en dos sublistas de aproximadamente la mitad del tamaño.
3. Ordenar cada sublista recursivamente aplicando el ordenamiento por mezcla.
4. Mezclar las dos sublistas en una sola lista ordenada.


```
function mergesort(m)
  var list left, right, result
  if length(m) ≤ 1
      return m
  else
      var middle = length(m) / 2
      for each x in m up to middle - 1
          add x to left
      for each x in m at and after middle
          add x to right
      left = mergesort(left)
      right = mergesort(right)
      if last(left) ≤ first(right)
         append right to left
         return left
      result = merge(left, right)
      return result
```

```
function merge(left, right)
  var list result
  while length(left) > 0 and length(right) > 0
      if first(left) ≤ first(right)
          append first(left) to result
          left = rest(left)
      else
          append first(right) to result
          right = rest(right)
  if length(left) > 0
      append rest(left) to result
  if length(right) > 0
      append rest(right) to result
  return result
```

Ejemplo: Ordenamiento quicksort

```
Algoritmo quicksort(A,inf,sup)
i<-inf
j<-sup
x<-A[(inf+sup)div 2]
mientras i=<j hacer
  mientras A[i]< x hacer
     i<-i+1
  fin_mientras
  mientras A[j]>x hacer
     j<- j-1
  fin_mientras
  si i=<j entonces
    tam<-A[i]
    A[i]<-A[j]
    A[j]<-tam
    i=i+1
    j=j-1
  fin_si
fin_mientras
si inf<j
   llamar_a quicksort(A,inf,j)
fin_si
si i<sup
   llamar_a quicksort(A,i,sup)
fin_si
```



¿Cuál es el “mejor”?


¿Preguntas?

Complejidad

Complejidad

➔ La complejidad de un algoritmo mide el consumo que hace de un recurso particular.

Los casos más comunes son:

➔ **Complejidad temporal** (cantidad de tiempo empleado)
➔ **Complejidad espacial** (cantidad de memoria empleado)

Ejemplo: Buscar 2 elementos iguales

Algoritmo REVISA:

```
for i ← 1 to n − 1 do
    for j ← i + 1 to n do
        if A[i] = A[j] then output (i, j)
            exit
```

Algoritmo ALMACENA:

```
for i ← 1 to n do
    if B[A[i]] != 0 then output A[i]
        exit
    else B[A[i]] ← 1
```

¿Cuál es “mejor”?

¿Qué ”costo” tienen?

Complejidad

- ➔ Nosotros nos limitaremos a analizar solo la complejidad temporal de los algoritmos
- ➔ No siempre podemos abstraernos de la complejidad espacial ya que muchas veces la memoria es un factor limitante del hardware del que disponemos.

Complejidad temporal

- **Con un cronómetro**: Se suele llamar **wall time**.
- **Bueno:** nos dice objetivamente cuánto tarda.
- **Malo:** depende de factores completamente ajenos al programa y los datos.
- Ni es confiable entre 2 ejecuciones consecutivas.

Complejidad temporal

➔ Con un **medidor de recursos (CPU time)**.

➔ **Bueno**: nos dice cuántos recursos utilizamos.

➔ **Malo**: es que depende de la computadora específica; luego si mañana cambio de computadora cambia el comportamiento.

¿Cómo hacemos?

Contando operaciones elementales!!!

Se suele llamar **complejidad algorítmica**.

Se trata de acotar la cantidad de operaciones elementales que toma resolver un problema en función del tamaño de la entrada.

complejidad algorítmica

- Las **operaciones elementales** de un lenguaje de programación son aquellas que, independientemente del nivel de abstracción del lenguaje, de si es compilado o interpretado, etc. son resueltas en tiempo **constante** por la arquitectura de la computadora

Operaciones elementales

- Todas las **operaciones aritméticas** sobre enteros y punto flotante
- Las **operaciones de listas que no requieren recorrido** (i.e. longitud, acceso, splits, etc.)
- La **lectura** o **escritura** de una variable

- Para abstraernos de arquitecturas particulares de cómputo les asignaremos costo 1

complejidad de programas

- La **complejidad de los programas** se computa a partir de asignarles complejidad a las construcciones:

- T (x = E) = 1 + T(E): E podría implicar la ejecución de alguna función con costo no unitario así que debemos contar las operaciones elementales involucradas en E y adicionar el costo asociado a las funciones ejecutadas para evaluar E.

Ejemplo: complejidad de programas

```
def esta (l: List[int], n: int) -> bool :
    res: bool = False
    i: int = 1
    while i < len (l) and not res :
        if l[i] == n :
            res = True
        else :
            pass
        i = i + 1
    return res
```

```
1
1
7
5
1
0
3
```

len (l)

$$T = 1+1+\text{len}(l)*(7+\underbrace{5+\max(1,0)}_{T\_if}+3)$$

$$= 2+\text{len}(l)*(7+5+1+3)$$

$$= 2+\text{len}(l)*16$$

Ejemplo: Buscar 2 elementos iguales

Algoritmo REVISA:

```
for i ← 1 to n − 1 do
    for j ← i + 1 to n do
        if A[i] = A[j] then output (i, j)
            exit
```

Algoritmo ALMACENA:

```
for i ← 1 to n do
    if B[A[i]] != 0 then output A[i]
        exit
    else B[A[i]] ← 1
```

Ejemplo: Ordenamiento por burbuja

```
procedimiento DeLaBurbuja (a_0, a_1, a_2, ..., a_(n-1))
    para i ← 1 hasta n − 1 hacer
        para j ← 0 hasta n − i hacer
            si a_(j) > a_(j+1) entonces
                aux ← a_(j)
                a_(j) ← a_(j+1)
                a_(j+1) ← aux
            fin si
        fin para
    fin para
fin procedimiento
```


¿Preguntas?

Clases de funciones

- Deseamos abstraernos de las constantes como mecanismo para identificar la "forma" de la función que determina el costo, en tiempo, de ejecutar algoritmo

- Para ello identificaremos **clases de funciones** que crecen en forma "similar", es decir, módulo constantes aditivas o multiplicativas

- En las clases de funciones están agrupadas pues todas ellas se acotan entre sí asintóticamente

complejidad de programas

- La **complejidad de los programas** se computa a partir de asignarles complejidad a las construcciones:
  - T (P;Q) = T(P)+T(Q)
  - T (**if** C : P **else** : **Q**) = T (C)+max(T(P), T(Q)): al no saber qué rama del if va a ser ejecutada, si queremos tener una estimación conservadora del costo debemos asumir que será la más costosa
  - T (**while** C : P) = Sum(1<=i<I : T(C_i)+T(P_i))+T(C_I): I es la cantidad de iteraciones del ciclo; nuevamente, para tener una estimación conservadora debemos calcular una cota superior para ese número. El costo asociado a testear la condición y ejecutar el cuerpo puede variar con cada iteración

complejidad de programas

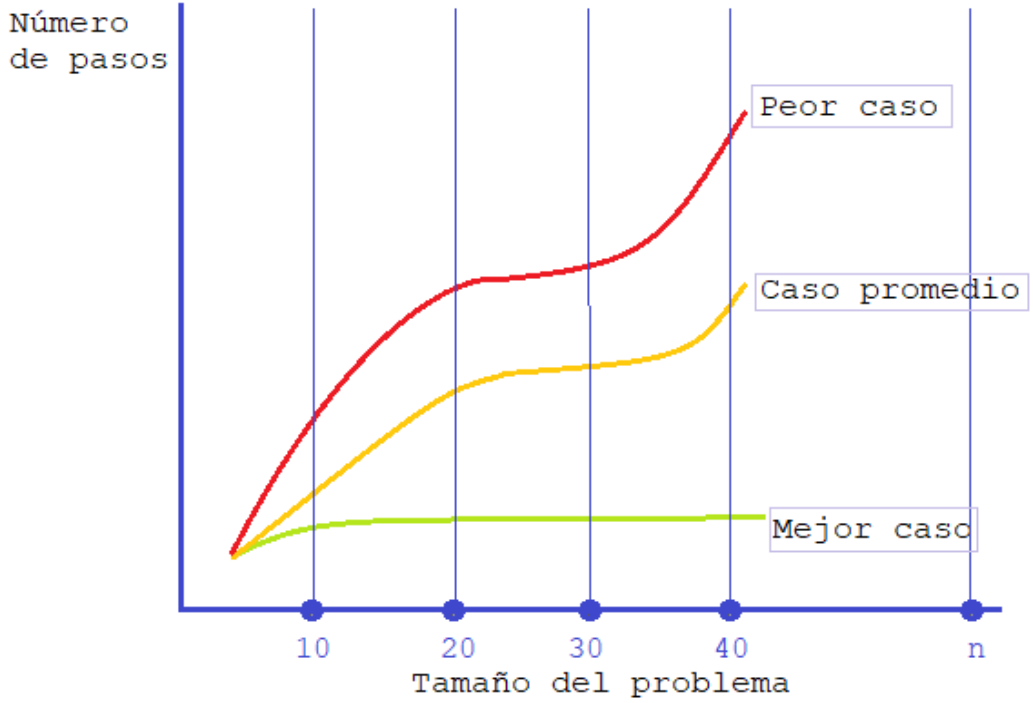
- La **complejidad de los programas** se computa a partir de asignarles complejidad a las construcciones:
  - T (P;Q) = T(P)+T(Q)
  - T (**if** C : P **else** : **Q**) = T (C)+max(T(P), T(Q)): al no saber qué rama del if va a ser ejecutada, si queremos tener una estimación conservadora del costo debemos asumir que será la más costosa
  - T (**while** C : P) = Sum(1<=i<I : T(C_i)+T(P_i))+T(C_I): I es la cantidad de iteraciones del ciclo; nuevamente, para tener una estimación conservadora debemos calcular una cota superior para ese número. El costo asociado a testear la condición y ejecutar el cuerpo puede variar con cada iteración

Clases de funciones

• **Mejor caso**: es la función definida por el número mínimo de pasos dados en cualquier instancia de tamaño n. Representa la curva más baja en el gráfico (verde) y se denomina cota inferior.

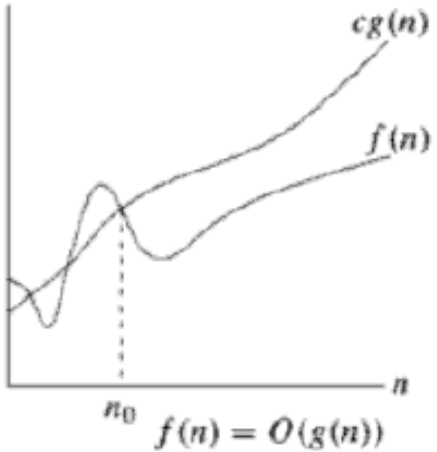
• **Caso promedio:** es la función definida por el número promedio de pasos dados en cualquier instancia de tamaño n.

• **Peor caso**: es la función definida por el número máximo de pasos dados en cualquier instancia de tamaño n. Esto representa la curva que pasa por el punto más alto en el gráfico (rojo) y se denomina cota superior.



Complejidad de peor caso

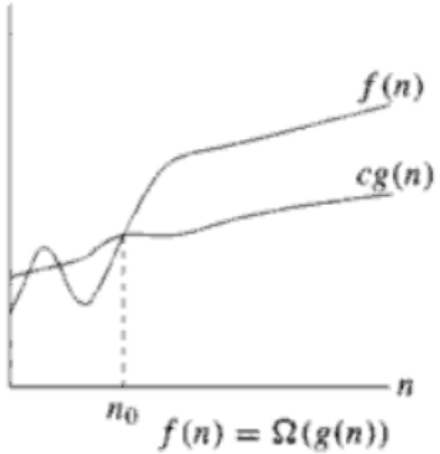
La clase de funciones *O(g(n))*



$$O(g(n)) = \{f(n) | (\exists c, x_o)(\forall x_0 \leq x)(f(x) < c * g(x))\}$$

Complejidad de mejor caso

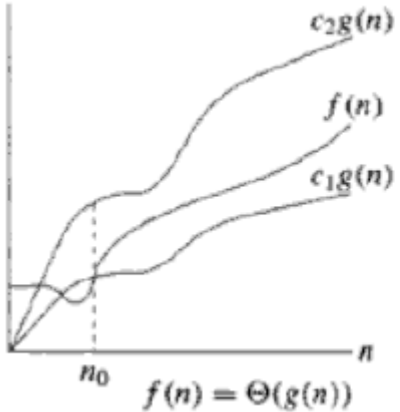
La clase de funciones *Ω(g(n))*



$$\Omega(g(n)) = \{f(n) | (\exists c, x_o)(\forall x_0 \leq x)(c * g(x)) < f(x)\}$$

Complejidad de ajustada

La clase de funciones *Θ(g(n))*



$$\Theta(g(n)) = \{f(n) | (\exists c_1, c_2, x_o)(\forall x_0 \leq x)(c_1 * g(x)) < f(x) \leq c_2 * g(x))\}$$

Ejemplo: Complejidad de peor caso

Sea f(n) = 3*n^3+n+5; ¿f(n) está en O(n^3)?

La definición exige que demostremos que existen c y n0 tales que:
para todo n >= n0 vale que 3*n^3+n+5 <= c*n^3.

Esto es equivalente a encontrar c y n0 tales que:
para todo n >= n0 vale que n+5 <= (c-3)*n^3.

Tomemos c = 4 y veamos que:

1) Ambas funciones son monótonas crecientes,
2) Observando ambas derivadas, la tasa de crecimiento de n+5 es menor que las de n^3 (de hecho la de la 1ra es cte mientras que la de la 2da es cuadrática).
3) Tomando n0 = 2 ya vale que 2+5 = 7 <= 8 = 2^3.

Clases de funciones

Propiedades algebraicas

- Las operaciones que toman tiempo constante (aquellas que identificamos como tomando una unidad de tiempo) se encuentran en O(1).
- Las clases de funciones se combinan aritméticamente a partir de la combinación de sus miembros:

  - f(m) + O(g(n)) = O(f(m)+g(n))
  - f(m) * O(g(n)) = O(f(m)*g(n))
  - O(f(m)) + O(g(n)) = O(f(m)+g(n))
  - O(f(m)) * O(g(n)) = O(f(m)*g(n))

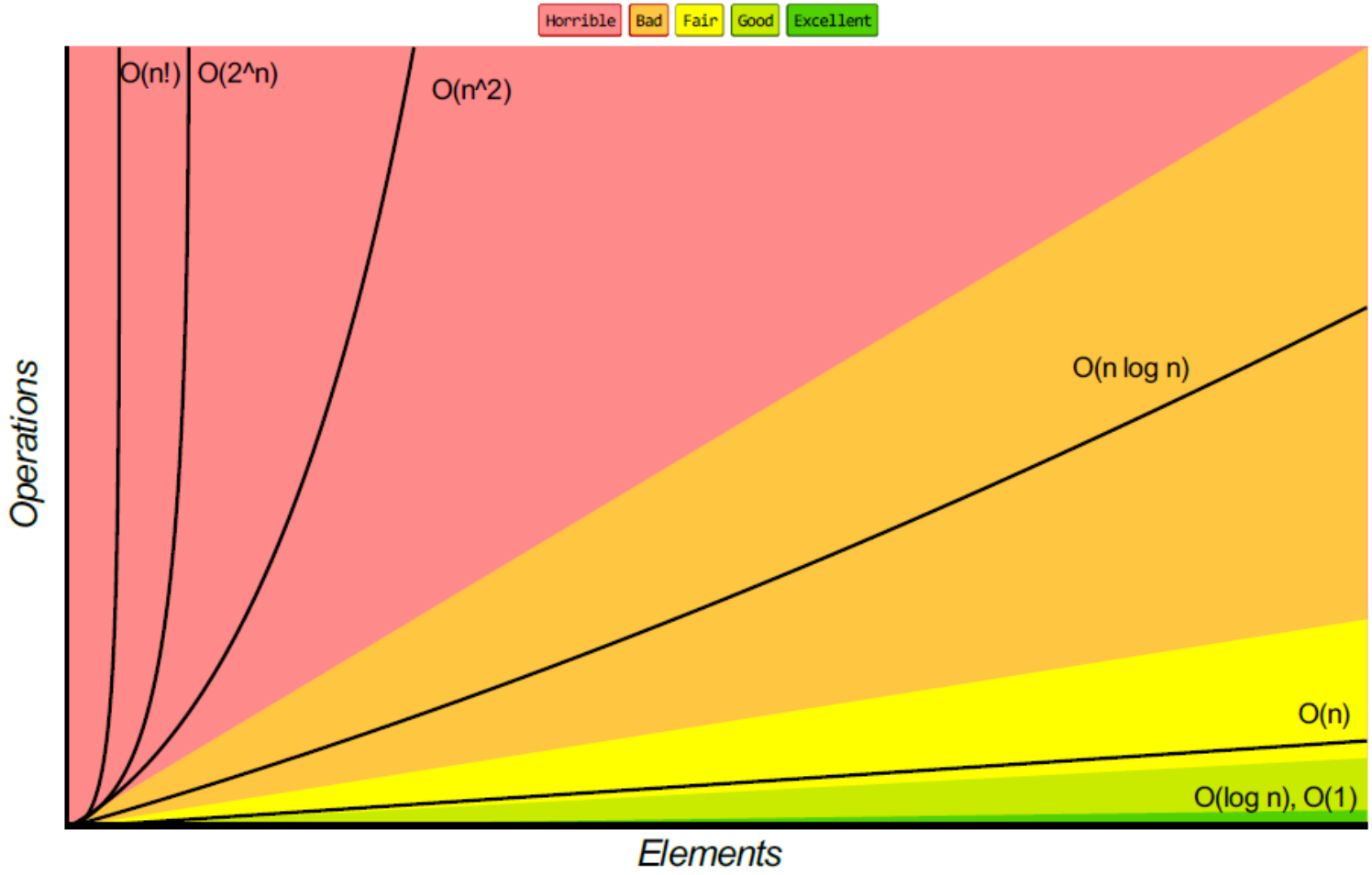
Clases de funciones

Propiedades algebraicas

Las funciones logarítmicas resultan todas iguales a los efectos del cómputo de la complejidad algorítmica pues solo se diferencian por una constante multiplicativa.

$$O(\log_a(n)) = O(\log_b(n)/\log_b(a)) = O(1/\log_b(a) * \log_b(n)) = O(\log_b(n))$$

Big-O Complexity Chart



notación	nombre
$O(1)$	orden constante (función acotada)
$O(\log \log n)$	orden sublogarítmica
$O(\log n)$	orden logarítmica
$O(\sqrt{n})$	orden sublineal
$O(n)$	orden lineal o de primer orden
$O(n \cdot \log n)$	orden lineal logarítmica
$O(n^2)$	orden cuadrática o de segundo orden
$O(n^3)$, ...	orden cúbica o de tercer orden, ...
$O(n^c)$	orden potencial fija (o polinomial)
$O(c^n)$, n > 1	orden exponencial
$O(n!)$	orden factorial
$O(n^n)$	orden potencial exponencial

Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

Estables				
Nombre traducido	Nombre original	Complejidad	Memoria	Método
Ordenamiento de burbuja	Bubblesort	$O(n^2)$	$O(1)$	Intercambio
Ordenamiento de burbuja bidireccional	Cocktail sort	$O(n^2)$	$O(1)$	Intercambio
Ordenamiento por inserción	Insertion sort	$O(n^2)$	$O(1)$	Inserción
Ordenamiento por casilleros	Bucket sort	$O(n)$	$O(n)$	No comparativo
Ordenamiento por cuentas	Counting sort	$O(n+k)$	$O(n+k)$	No comparativo
Ordenamiento por mezcla	Merge sort	$O(n \log n)$	$O(n)$	Mezcla
Ordenamiento con árbol binario	Binary tree sort	$O(n \log n)$	$O(n)$	Inserción
	Pigeonhole sort	$O(n+k)$	$O(k)$	
Ordenamiento Radix	Radix sort	$O(nk)$	$O(n)$	No comparativo
	Distribution sort	$O(n^3)$ versión recursiva	$O(n^2)$	
	Gnome sort	$O(n^2)$	$O(1)$	

Inestables				
Nombre traducido	Nombre original	Complejidad	Memoria	Método
Ordenamiento Shell	Shell sort	$O(n^{1.25})$	$O(1)$	Inserción
	Comb sort	$O(n \log n)$	$O(1)$	Intercambio
Ordenamiento por selección	Selection sort	$O(n^2)$	$O(1)$	Selección
Ordenamiento por montículos	Heapsort	$O(n \log n)$	$O(1)$	Selección
	Smoothsort	$O(n \log n)$	$O(1)$	Selección
Ordenamiento rápido	Quicksort	Promedio: $O(n \log n)$, peor caso: $O(n^2)$	$O(\log n)$	Partición
	Several Unique Sort	Promedio: $O(n\ u)$, peor caso: $O(n^2)$; u=n; u = número único de registros		
Cuestionables, imprácticos				
Nombre traducido	Nombre original	Complejidad	Memoria	Método
	Bogosort	$O(n \times n!)$, peor: no termina		
Ordenamiento de panqueques	Pancake sorting	$O(n)$, excepto en máquinas de Von Neumann		
Ordenamiento Aleatorio	Randomsort	Promedio: O(n!) Peor: No termina		


¿Preguntas?

unrn.edu.ar

UNRN Universidad Nacional de Río Negro

| unrionegro