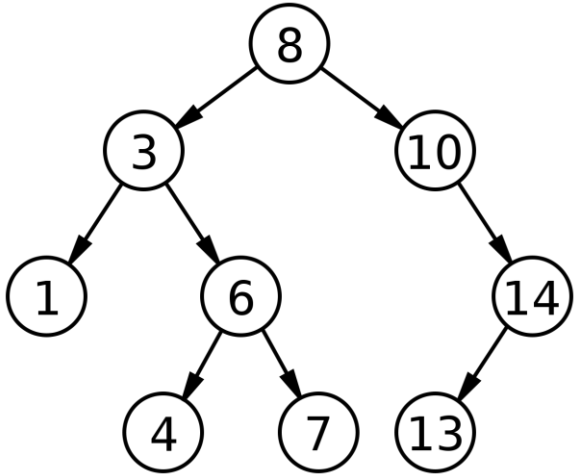
Complejidad Computacional

Introducción al mundo de los algoritmos

Oscar Burga

Propósito

- Introducción al mundo de la complejidad computacional, algoritmos y estructuras de datos.
- Ejemplos, temas relacionados y más.



Eficiencia y escalabilidad de un algoritmo

- ¿Cómo saber qué tan eficiente es mi algoritmo?
- ¿Cómo estimar los recursos computacionales necesarios para ejecutar mi algoritmo?
- ¿Cómo varía el rendimiento de mi algoritmo según los datos de entrada que recibe?

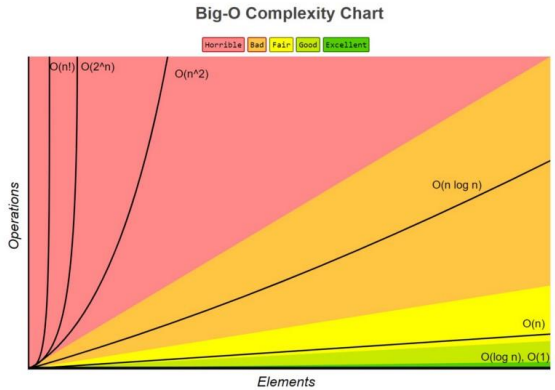
Análisis de algoritmos

Análisis de algoritmos

- Determinar la complejidad computacional de un algoritmo.
- Estimar los recursos computacionales que requiere la ejecución de un algoritmo, en términos de los datos de entrada que recibe.
- Recursos computacionales principales:
    - Tiempo (complejidad temporal)
    - Memoria (complejidad espacial)

Notación Big-O

- Formalmente, describe la cota superior asintótica de una función matemática.
- Estimar el rendimiento de un algoritmo en el peor caso posible.
- No es el único tipo de notación asintótica:
  - Big-Ω: Mejor caso posible (cota inferior de una función).
  - Big-Θ: Ambos (cota inferior y superior de una función).



Análisis de algoritmos

- Consiste en medir un algoritmo según su eficiencia para manejar los datos de entrada que recibe
  - Complejidad temporal: Cantidad de operaciones a realizar para procesar un conjunto de n datos
  - Complejidad espacial: Cantidad de memoria a utilizar para procesar un conjunto de n datos
- Expresar esta medida como una función $f(n)$.
- Nos importa el término de mayor orden de la función (crece más rápido). Las constantes no son importantes.

Análisis de algoritmos

n	$f(n) = n^3$	$f(n) = n^3 + n^2$	$f(n) = n^3 - 8n^2 + 20n$
1	1	2	13
10	1’000	1’100	400
1’000	1’000’000’000	1’001’000’000	992’020’000
1’000’000	1.0×10^{18}	1.000001×10^{18}	9.99992×10^{17}

Fuente: Harvard University – CS50

Ejemplo: Complejidad temporal

```
1  Function SumOfArray(array):            // array tiene N elementos
2      TotalSum = 0                       // +1 op.
3      For Each number in array:          // N veces
4          TotalSum = TotalSum + number   // +2 op.
5      EndFor
6      Return TotalSum
```

- Las 2 operaciones dentro del For Each se ejecutan N veces
- 1 operación extra fuera del For Each
- Total operaciones: $f(N) = 2N + 1$
- Término de mayor orden: $2N$
- Término de mayor orden, sin constantes: N
- Complejidad temporal: $O(N)$

Ejemplo: Complejidad temporal

```
1 Function SumOfArrays(A, B):                    // Arreglos (N y M elementos)
2     TotalSum = 0                               // +1 op.
3     For Each a in A:                           // N veces
4         For Each b in B:                       // M veces
5             TotalSum = TotalSum + (a * b)      // +3 op.
6     EndFor
7     Return TotalSum
```

- Dentro del segundo For Each: 3 operaciones
- Segundo For Each se ejecuta M veces $\rightarrow 3M$ operaciones
- Primer For Each se ejecuta N veces $\rightarrow N \times 3M = 3NM$ operaciones
- Cantidad de operaciones: $f(N, M) = 3NM + 1$
- Término de mayor orden: $3NM \rightarrow NM$ sin constantes
- Complejidad temporal: $O(NM)$

Tabla de complejidades	
$O(1)$	Constante
$O(\log N)$	Logarítmico
$O(N)$	Lineal
$O(N \log N)$	Tiempo logarítmico-lineal (log-linear, linearithmic)
$O(N^2)$	Cuadrático
$O(N^c)$	Polinomial
$O(c^N)$	Exponencial
$O(N!)$	Factorial
$O(\infty)$	Infinito

Fuente: Harvard University – CS50

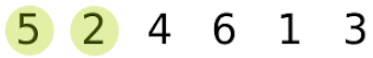
Ordenamientos

Ordenamientos y algoritmos de ordenamiento

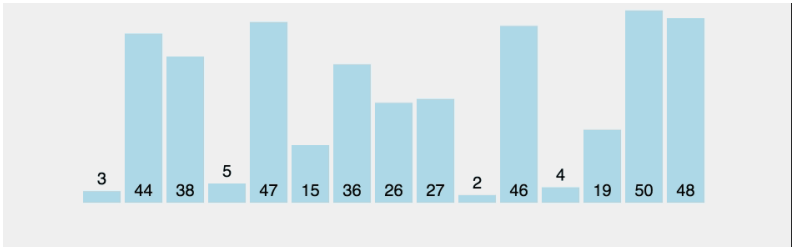
- Ordenamiento: Problema absolutamente fundamental en el campo de los algoritmos.
  - Algoritmos importantes y complejos usan ordenamientos como subrutinas
  - Muchas aplicaciones del mundo real requieren tener datos ordenados
  - Otros problemas pueden reducirse a problemas de ordenamiento
- Cota inferior para algoritmos de ordenamientos: $\Omega(N \log N)$

Ejemplo: Ordenamiento Burbuja

- Algoritmo de ordenamiento básico.
- Iterativamente revisa pares de elementos adyacentes y los intercambia si no están ordenados.



Fuente: Xybernetics



Fuente: Gfycat

Ejemplo: Ordenamiento Burbuja

```
0 void bubblesort(int a[], int n) {
1     for (int i = 0; i < n; i++) {        // N veces
2         for (int j = 0; j+1 < n; j++) {  // N-1 veces
3             // Todo este bloque interior hace una
4             // cantidad constante C de operaciones
5             if (a[j] > a[j+1]) {
6                 int temp = a[j];
7                 a[j] = a[j+1];
8                 a[j+1] = temp;
9             }
10        }
11    }
12 }
```

Se realizan N iteraciones, y en cada iteración se intercambian elementos adyacentes que estén fuera de orden.

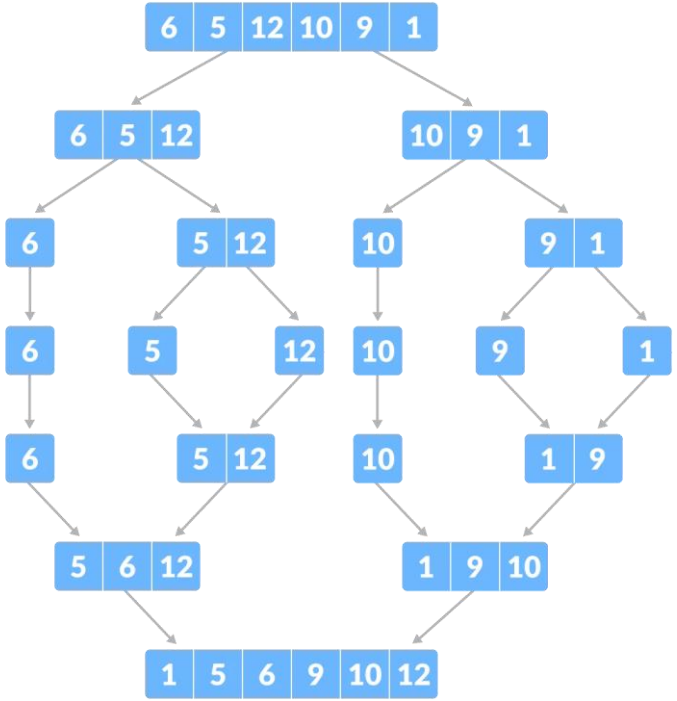
$$f(N) = N \times (N-1) \times C$$
$$f(N) = CN^2 - CN$$

Se eliminan las constantes y se toma el término de mayor orden → N^2

Complejidad resultante: $O(N^2)$

Ejemplo: Merge Sort

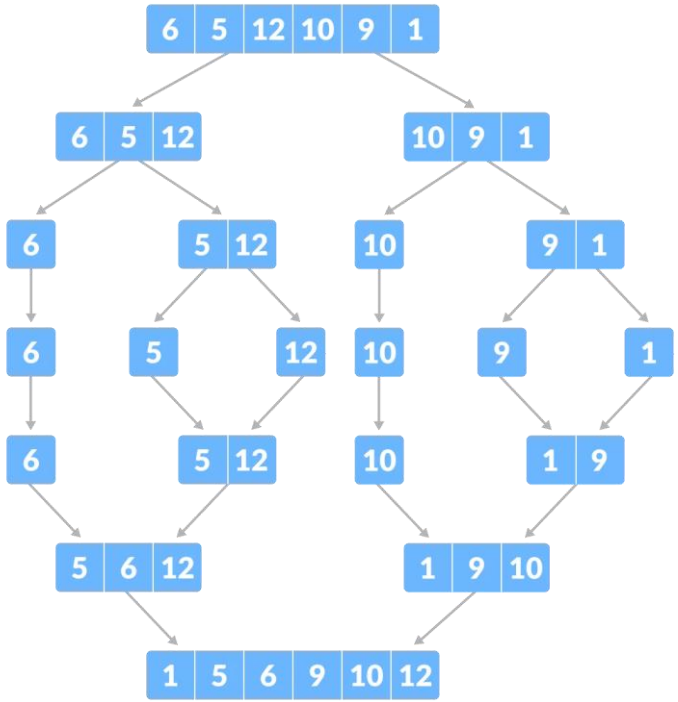
- Concepto nuevo: Divide y Vencerás (Divide and Conquer)
- Dividir recursivamente el conjunto en mitades, ordenarlas independientemente, y luego mezclarlas.
- Mezclar 2 arreglos ordenados puede hacerse en tiempo lineal $O(N)$
- Algoritmo con complejidad temporal $O(N \log N)$ y complejidad espacial $O(N)$



Fuente: Programiz

Ejemplo: Merge Sort

```
0 void mergeSort(int a[], int l, int r) {
1     if (l >= r)
2         return;
3     int mid = (l+r) / 2;
4     mergeSort(a, l, mid);   // Ordenar la mitad izquierda
5     mergeSort(a, mid+1, r); // Ordenar la mitad derecha
6     // Mezclar los rangos a[l:mid] y a[mid+1:r]
7     // en tiempo y espacio lineal O(R-L+1)
8     merge(a, l, mid, r);
9 }
```



Algoritmos de búsqueda

Búsqueda lineal

- Método más simple de búsqueda en un conjunto de datos.
- Se recorren todos los elementos del conjunto hasta que se encuentre el que se busca (o se determine que no existe).
- En el peor caso se realizan $O(N)$ comparaciones.

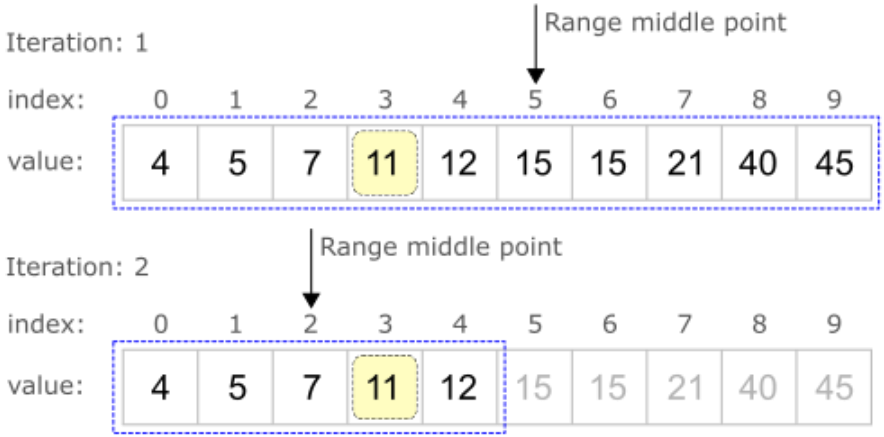
```
0 bool searchInt(int a[], int n, int x) {
1     for (int i = 0; i<n; i++) {    // N iteraciones
2         if (a[i] == x) {           // 1 comparacion
3             return true;
4         }
5     }
6     return false;
7 }
```

Búsqueda binaria

- Aplicación de divide y vencerás
- Idea principal: Partir sucesivamente el espacio de búsqueda a la mitad hasta encontrar el elemento que buscamos o agotar el espacio de búsqueda.
- Permite buscar elementos eficientemente dentro de conjuntos de datos ordenados (complejidad $O(\log N)$)
- Aplicaciones más avanzadas: Búsqueda de puntos de corte en funciones monótonas, búsqueda binaria en espacios continuos
    - Ejemplo: Calcular la raíz cuadrada (no necesariamente entera) de un número.
    - Otras aplicaciones avanzadas: Máxima circunferencia inscribible dentro de un polígono convexo.

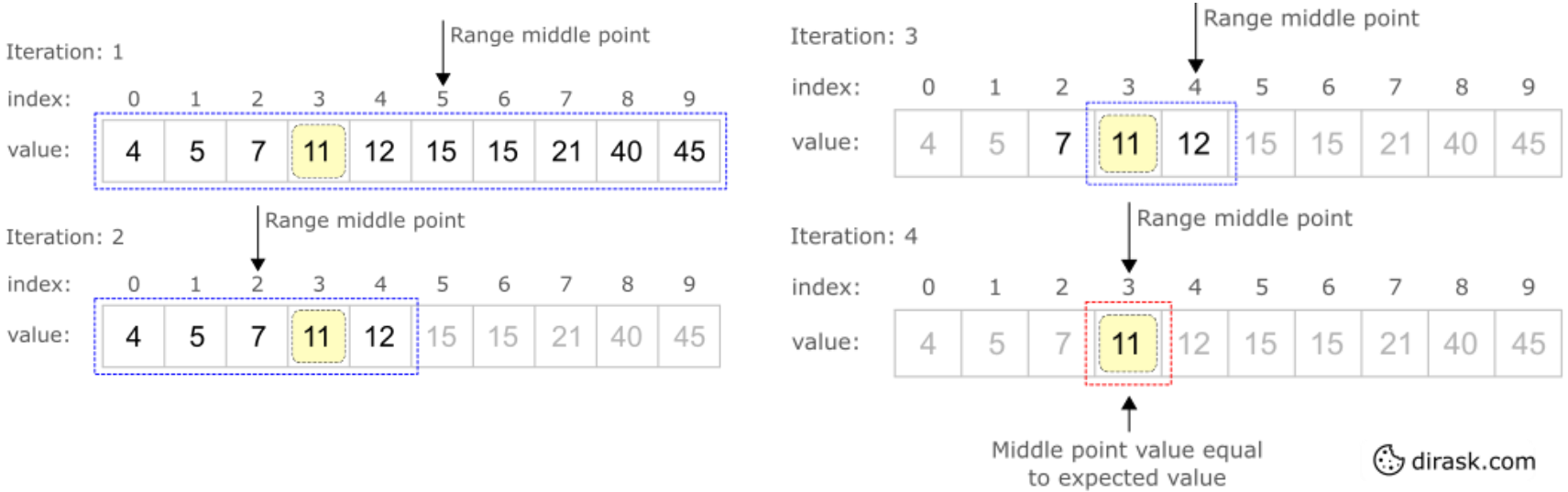
Búsqueda binaria en arreglos ordenados

- Iniciamos con el rango completo de elementos del arreglo ordenado, y supongamos que buscamos el valor x
- En cada iteración, se toma el elemento del medio del rango actual y se compara con el valor que se está buscando
  - Si el elemento del medio es menor a x, tomamos como nuevo rango a la mitad derecha.
  - Si el elemento del medio es mayor a x, tomamos como nuevo rango a la mitad izquierda.
  - Si el elemento es igual a x, finaliza el algoritmo.



Fuente: FullStack.Cafe

Búsqueda binaria en arreglos ordenados



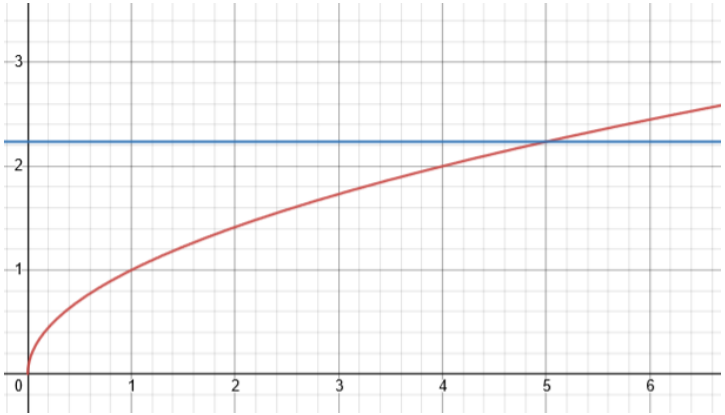
Búsqueda binaria en arreglos ordenados

- En cada iteración de la búsqueda binaria se realiza una cantidad constante de operaciones (únicamente una comparación y actualizar los índices del nuevo rango).
- En cada iteración, el tamaño del rango por evaluar disminuye a la mitad.
- Sabemos que un número solo puede dividirse sucesivamente una cantidad logarítmica de veces.
- De igual manera, el rango solo puede dividir su tamaño una cantidad logarítmica de veces.
- Por lo tanto, la complejidad total es $O(\log N)$.

Búsqueda binaria en espacios continuos

- La búsqueda binaria también puede ser utilizada en espacios continuos para hallar puntos de corte de funciones monótonas.
- Ejemplo: Hallar la raíz cuadrada de un número real

Min	Max	Mid	$(Mid)^2$	$Objetivo$
0.0000	5.0000	2.5000	6.2500	5
0.0000	2.5000	1.2500	1.5625	5
1.2500	2.5000	1.8750	3.5156	5
1.8750	2.5000	2.1875	4.7852	5
2.1875	2.5000	2.3437	5.4932	5
2.1875	2.3437	2.2656	5.1331	5
2.1875	2.2656	2.2266	4.9576	5



En rojo: $f(x) = \sqrt{x}$
En azul: $y = \sqrt{5}$

Búsqueda binaria en espacios continuos

```
0 x = 5                    # Numero del cual se busca la raiz cuadrada
1 l, r = 0.0, 5.0          # Limites inferior (minimo) y superior (maximo)
2 eps = 1e-9               # Epsilon auxiliar
3
4 for i in range(50):      # Cantidad fija de iteraciones por simplicidad
5     mid = (l+r) / 2      # tomar punto medio
6     if mid * mid <= x:   # mid es menor o igual a la raiz de X
7         l = mid;
8     else:                # mid es mayor a la raiz de X
9         r = mid - eps
10
11 print(l)                # Tomar el ultimo limite inferior como respuesta
```

¿Qué hay más allá?

¿Qué hay más allá?

- Complejidad computacional es un campo gigantesco, apenas hemos tocado la superficie.
- Teoría de la complejidad computacional (clases de complejidad, P = NP, etc.)
- Muchos más temas interesantes en el mundo de los algoritmos
  - Grafos, búsquedas en grafos y otros algoritmos de grafos
  - Programación dinámica
  - Estructuras de datos
  - Algoritmos y estructuras de cadenas
  - Algoritmos numéricos
  - Geometría computacional
  - ... y muchos más.

Material de consulta recomendado

- Libro: Introduction to Algorithms, 3rd. Edition (Cormen, Leiserson, Rivest, Stein)
- Libro: Algorithms in C (Robert Sedgewick)
- Página Web: CP-Algorithms (compilación de descripciones y tutoriales de algoritmos)
- Página Web: Codeforces-Edu (temas avanzados)
- Videos: Pavel Mavrin (Youtube, clases de algoritmos de la Universidad ITMO de Rusia)

Gracias por su atención