TD3: Algoritmos y Estructuras de Datos

Prof. Agustín Garassino, Gervasio Pérez

Segundo Semestre de 2024

Clase Teórica 7
Ordenamiento

Resumen

En la clase de hoy veremos:

- Repaso ordenamiento en $O(n^2)$.
- Ordenamiento en $O(n \log n)$.
- Otros algoritmos de ordenamiento.

Problema de ordenamiento

Dado un vector de números enteros ordenar sus elementos en forma creciente.

59	7	388	41	2	280	50	123

$\downarrow$

2	7	41	50	59	123	280	388

Algoritmos $O(n^2)$

- Selection sort
  - Buscar el mínimo de la parte que falta ordenar y colocarlo al final de la parte que ya está ordenada.

59	7	388	41	2	280	50	123
2	7	388	41	59	280	50	123
2	7	41	388	59	280	50	123
2	7	41	388	59	280	50	123

...

- Insertion sort
  - Tomar el próximo elemento de la parte que falta ordenar e insertarlo de manera ordenada en la parte que ya está ordenada.

59	7	388	41	2	280	50	123
7	59	388	41	2	280	50	123
7	59	388	41	2	280	50	123
7	41	59	388	2	280	50	123

...

Algoritmos $O(n^2)$ (cont.)

- Bubble sort
  - De izquierda a derecha comparar pares de elementos vecinos e intercambiarlos si corresponde. Repetir n veces.

59	7	388	41	2
7	59	388	41	2
7	59	388	41	2
7	59	41	388	2
7	59	41	2	388
7	59	41	2	388
7	41	59	2	388
7	41	2	59	388
7	41	2	59	388
7	2	41	59	388
7	2	41	59	388

Merge sort

Es un algoritmo recursivo para resolver el problema en $O(n \log n)$ en peor caso.

59	7	388	41	2	280	50	123

- Dividir la secuencia en dos mitades.

59	7	388	41

2	280	50	123

- Ordenar cada mitad recursivamente.

7	41	59	388

2	50	123	280

- Combinar ambas mitades de manera ordenada.

2	7	41	50	59	123	280	388

Mergesort es Divide & Conquer

Divide & Conquer es una técnica de diseño de algoritmos que estructura la solución de un problema de tamaño N así:

1. Caso base: Si el problema es suficientemente pequeño, resolverlo directamente; si no...
2. Divide: Partir el problema a resolver en a subproblemas de tamaño N/b
3. Resolver los subproblemas (en general recursivamente)
4. Conquer: combinar los resultados de los subproblemas para resolver el problema original

En Mergesort: Caso base si el arreglo es de tamaño 1 (está ordenado); si no, realizamos divide y partimos el arreglo en dos mitades iguales (a=2, b=2), luego ordenamos cada mitad por separado, y finalmente hacemos conquer al combinar los dos subarreglos ordenados.

Merge sort (pseudocógido)

vector<int> *MergeSort*(**vector<int>** V)

1. Si $|V| \leq 1$:
   Devolver V
2. $med = |V|/2$
3. $izq = MergeSort(V[0 \dots med-1])$
4. $der = MergeSort(V[med \dots |V|-1])$
5. Devolver $Merge(izq, der)$

Donde $Merge(V1, V2)$ es una función que toma dos vectores ordenados y devuelve la union ordenada de ambos.

Merge sort en C++

```cpp
vector<int> mergesort(const vector<int> & v, int d, int h){
    // Ordena los elementos desde d hasta h (sin incluir)

    // Si hay 0 o 1 elementos, ya están ordenados
    if(h - d == 0)
        return {};
    if(h - d == 1)
        return {v[d]};

    // Divide y ordena cada mitad
    int med = (d+h)/2;
    vector<int> izq = mergesort(v, d, med);
    vector<int> der = mergesort(v, med, h);

    // Devuelve la unión ordenada de ambas mitades
    return merge(izq, der);
}
```

Merge sort – función **merge**

```
vector<int> merge(const vector<int> & v1, const vector<int> & v2){
    vector<int> res; int i = 0; int j = 0;

    while(i < v1.size() && j < v2.size()){
        if(v1[i] < v2[j]){
            res.push_back(v1[i]);
            i++;
        } else {
            res.push_back(v2[j]);
            j++;
        }
    }
    while(i < v1.size()){
        res.push_back(v1[i]);
        i++;
    }
    while(j < v2.size()){
        res.push_back(v2[j]);
        j++;
    }
    return res;
}
```

Merge sort

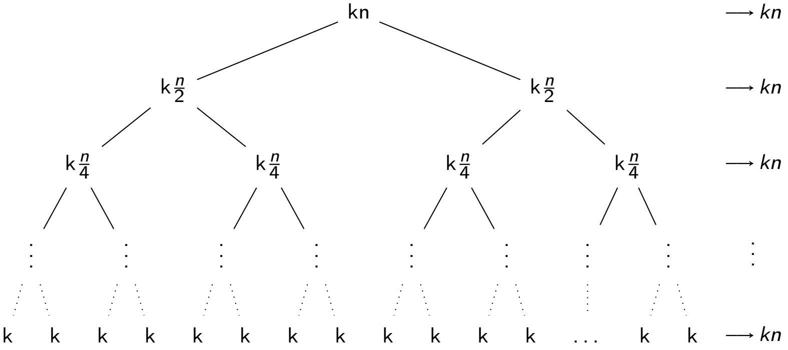
Calculemos el órden de complejidad:

```
vector<int> mergesort(const vector<int> & v, int d, int h){
    if(h - d == 0)          // 4 ops. constantes
        return {};          // 1 ops. constantes
    if(h - d == 1)          // 4 ops. constantes
        return {v[i]};      // 4 ops. constantes

    int med = (d+h)/2;      // 4 ops. constantes
    vector<int> izq = mergesort(v, d, med); // T(n/2)
    vector<int> der = mergesort(v, med, h); // T(n/2)

    return merge(izq, der); // k*n
}
```

El tamaño de entrada n es $h-d$.

$$T(n) = \begin{cases} k & \text{si } n=0, n=1, \\ 2T(n/2)+kn & \text{si } n>1 \end{cases}$$

Árbol de recursion



- Cantidad de niveles: $\log n + 1$.
- Suma en cada nivel: kn.
- Total: $\sum_{i=0}^{\log n+1} kn = kn \log n + kn$
- Propuesta: $T(n) \in O(n \log n)$

Demostración por inducción

$$T(n) = \begin{cases} k & \text{si } n = 0, n = 1, \\ 2T(n/2) + kn & \text{si } n > 1 \end{cases}$$

- Queremos ver que $T(n) \leq c \cdot (n \log n)$, para algún c.
- Caso base:
  - Con $n = 0$,
    $T(0) \leq c \cdot (0 \log 0)$. No sirve, $\log 0$ no está definido.
  - Con $n = 1$,
    $T(1) \leq c \cdot (1 \log 1)$
    $k \leq c \cdot 0$, no se cumple.
  - Con $n = 2$,
    $T(2) \leq c \cdot (2 \log 2)$
    $2T(1) + 2k \leq 2c$
    $2k + 2k \leq 2c$
    $2k \leq c$
  - Tomamos $c = 2k$ y $n_0 = 2$

Demostración por inducción

$$T(n) = \begin{cases} k & \text{si } n = 0, n = 1, \\ 2T(n/2) + kn & \text{si } n > 1 \end{cases}$$

- Caso inductivo: queremos ver que $T(n) \leq c \cdot (n \log n)$
- Hipótesis inductiva: vale $T(x) \leq c \cdot (x \log x)$ para $x < n$.
- Sabemos que $T(n) = 2T(n/2) + kn$
- $\leq 2c(n/2)\log(n/2) + kn$, por hipótesis inductiva
- $= cn \log(n/2) + kn$
- $= cn \log n - cn \log 2 + kn$
- $= cn \log n - cn + kn$, ya que $\log 2 = 1$.
- $\leq cn \log n$, siempre que $c \geq k$.
- Vale, ya que habíamos tomado $c = 2k$.
- Queda demostrado que $T(n) \in O(n \log n)$

Quicksort

123	7	388	41	2	280	50	59

- Elegir un elemento pivot y dividir el vector entre los menores y los mayores al pivot.

7	41	2	50		59		123	388	280

- Ordenar cada partición recursivamente.

2	7	41	50		59		123	280	388

- Por cómo se hicieron las particiones, el vector ya queda ordenado:

2	7	41	50	59	123	280	388

Quicksort en C++

```
void quicksort(vector<int> & v, int d, int h){
    if(d < h - 1){
        int pos = dividir(v, d, h);
        quicksort(v,d,pos);
        quicksort(v,pos+1,h);
    }
}
```

La función **dividir**:

- elige un pivot,
- particiona el subvector $[d \dots h]$ y
- devuelve la posición del pivot.

Quicksort – función **dividir**

```
int dividir(vector<int> & v, int d, int h){
    int pivot = v[h-1];
    int i = d;
    for(int j = d; j < h - 1; j++){
        if(v[j] <= pivot){
            swap(v[i], v[j]);
            i = i + 1;
        }
    }
    swap(v[i], v[h-1]);
    return i;
}
```

La función **swap(v[i],v[j])** toma **v[i]** y **v[j]** por referencia e intercambia los elementos de las posiciones **i** y **j** de **v**.

Quicksort – función **dividir** – demo

i = Primera posición *luego* de la *primera* partición

j = Próximo elemento a clasificar

123	7	388	41	2	280	50	59	`i=0, j=0`
123	7	388	41	2	280	50	59	`i=0, j=1`
7	123	388	41	2	280	50	59	`i=1, j=2`
7	123	388	41	2	280	50	59	`i=1, j=3`
7	41	388	123	2	280	50	59	`i=2, j=4`
7	41	2	123	388	280	50	59	`i=3, j=5`
7	41	2	123	388	280	50	59	`i=3, j=6`
7	41	2	50	388	280	123	59	`i=4, j=7`

Fin del ciclo

7	41	2	50	59	280	123	388	`swap(v[4],v[7])`

Quicksort

- En peor caso es $O(n^2)$.
  - Las particiones pueden ser desbalanceadas
  - ¿Qué pasa si quedan todos los elementos en una sola partición?
- En caso promedio es $O(n \log n)$.
  - Requiere asumir que todas las permutaciones del vector de entrada son equiprobables,
  - o requiere una implementación aleatorizada que elija el pivot de manera aleatoria.
- Los factores constantes ocultos en $O(n \log n)$ son muy bajos y, en la práctica puede ser mucho más eficiente que otros algoritmos $n \log n$.

Problema de ordenamiento

Nos piden ordenar una secuencia de enteros que almacena edades de personas. Sabemos que las edades nunca pueden ser mayores a 100. **¿Se puede resolver más rápido que O(n log n)?**

Idea:

- Armamos un vector auxiliar *contadores* de 101 posiciones (del 0 al 100) inicializadas con el valor 0.
- El elemento *contadores*[*i*] queremos que sea un contador que almacene la cantidad de veces que aparece el valor *i* en el vector de entrada.
- Recorremos el vector de entrada y llenamos los contadores de *contadores*.
- Recorremos el vector *contadores* y construimos el vector de salida agregando *i* la cantidad de veces que indica *contadores*[*i*].

Counting sort

```
vector<int> ordenar_edades(const vector<int> & v){
    vector<int> contadores(101);
    vector<int> res;

    // Inicializo los contadores en 0;
    for(int i = 0; i < 101; i++)
        contadores[i] = 0;

    // Recorro v y actualizo el contador correspondiente.
    for(int i = 0; i < v.size(); i++)
        contadores[v[i]] = contadores[v[i]] + 1;

    // Recorro los contadores y armo el vector ordenado.
    for(int i = 0; i < 101; i++)
        for(int j = 0; j < contadores[i]; j++)
            res.push_back(i);

    return res;
}
```

Counting sort

```
vector<int> ordenar_edades(const vector<int> & v){
    vector<int> contadores(101);        // O(101)
    vector<int> res;                    // O(1)

    // Inicializo los contadores en 0;
    for(int i = 0; i < 101; i++)        // 101 repeticiones
        contadores[i] = 0;              // O(1)

    // Recorro v y actualizo el contador correspondiente.
    for(int i = 0; i < v.size(); i++) // n repeticiones
        contadores[v[i]] = contadores[v[i]] + 1;  // O(1)

    // Recorro los contadores y armo el vector ordenado.
    for(int i = 0; i < 101; i++)        // Total = 101 + O(n)
        for(int j = 0; j < contadores[i]; j++)
            res.push_back(i);

    return res;
} // T(n) es O(n)
```

Counting sort

```cpp
vector<int> counting_sort(const vector<int> & v, int maximo){
    vector<int> contadores(maximo + 1);
    vector<int> res;

    for(int i = 0; i <= maximo; i++)
        contadores[i] = 0;
    for(int i = 0; i < v.size(); i++)
        contadores[v[i]] = contadores[v[i]] + 1;
    for(int i = 0; i <= maximo; i++)
        for(int j = 0; j < contadores[i]; j++)
            res.push_back(i);

    return res;
}
```

- $T(n, maximo) \in O(n + maximo)$, donde $n = |v|$.
- Si *maximo* es un valor constante, entonces $T(n, maximo) \in O(n)$

Cuidado: si *maximo* no es constante entonces el algoritmo **no es O(n)**

Repaso de la clase

Hoy vimos

- Mergesort
- Complejidad de algoritmos recursivos
  - Árbol de recursion,
  - Demostración por inducción.
- Quicksort
- Counting sort

Ya pueden hacer la Guía 5.

Bibliografía:

- "Introduction to Algorithms, Fourth Edition" por Thomas Cormen
  - Mergesort: Capítulo 2.3.1.
  - Complejidad de algoritmos recursivos: Capítulo 4.
  - Quicksort: Capítulo 7.1.
  - Counting sort: Capítulo 8.2.