

Tema 2. Algorismes de dividir i vèncer

Estructures de Dades i Algorismes

FIB

Transparències d' **Antoni Lozano**
(amb edicions menors d'altres professors)

Q1 2019 – 20

1 Ordenació per fusió

- Algorisme de fusió bàsic
- Variants

2 Ordenació ràpida

- Algorisme general
- Variants
- Anàlisi

3 Productes i exponents

- Algorisme de Karatsuba
- Exponenciació ràpida
- Algorisme de Strassen

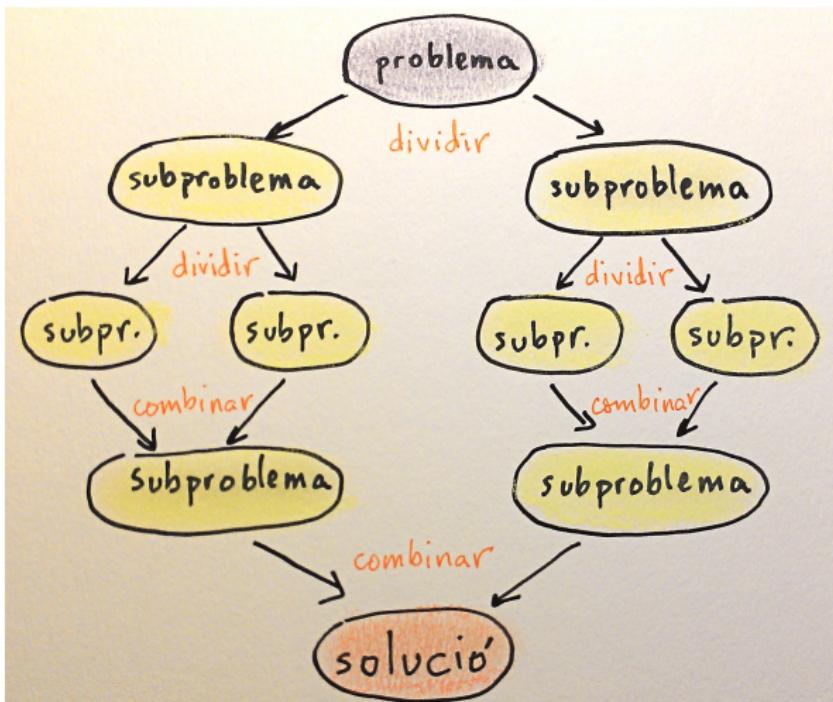
4 Altres algorismes

- Torres de Hanoi
- Mediana

L'estratègia de **dividir i vèncer** resol un problema en tres passos:

- ➊ dividint-lo en *subproblems*, casos més “petits” del mateix problema,
- ➋ resolent els subproblems recursivament i
- ➌ combinant les respostes de manera adequada.

Dividir i vèncer



La feina es fa, per tant, en tres parts: (1) en la divisió en subproblems, (2) al final de la recursió i (3) en la combinació de les solicions.

Els algorismes de *dividir i vèncer* segueixen sovint una mateixa estratègia:
ataquen un problema de mida n

- dividint-lo en a subproblemes de mida n/b ,
- resolent els subproblemes recursivament, i
- combinant les respostes,

on $a \geq 1, b > 1$, i el cost de dividir en subproblemes i combinar respostes és $\Theta(n^k)$ per a $k \geq 0$

Llavors el cost de l'algorisme es pot descriure mitjançant la recurrència

$$T(n) = a \cdot T(n/b) + \Theta(n^k)$$

que es pot resoldre aplicant el teorema mestre de recurrències divisores.

Teorema mestre de recurredències divisores

Sigui $T(n)$ la recurredència

$$T(n) = \begin{cases} f(n), & \text{si } 0 \leq n < n_0 \\ a \cdot T(n/b) + g(n), & \text{si } n \geq n_0 \end{cases}$$

on $n_0 \in \mathbb{N}$, $b > 1$, f és una funció arbitrària i $g \in \Theta(n^k)$ per a $k \geq 0$.

Sigui $\alpha = \log_b(a)$. Aleshores,

$$T(n) \in \begin{cases} \Theta(n^k), & \text{si } \alpha < k \\ \Theta(n^k \log n), & \text{si } \alpha = k \\ \Theta(n^\alpha), & \text{si } \alpha > k \end{cases}$$

```
int cerca_binaria(const vector<int>& a, int i, int j, int x)
{  if (i <= j) {
    int k = (i + j) / 2;
    if (x < a[k])
        return cerca_binaria(a, i, k-1, x);
    else if (x > a[k])
        return cerca_binaria(a, k+1, j, x);
    else
        return k;
}
else return -1;
}
```

El paràmetre de recursió és $n = j - i + 1$ i el cost $T(n) = T(n/2) + \Theta(1)$.
Pel teorema mestre de recurrències divisores, $T(n) \in \Theta(\log n)$.

Tema 2. Algorismes de dividir i vèncer

1 Ordenació per fusió

- Algorisme de fusió bàsic
- Variants

2 Ordenació ràpida

- Algorisme general
- Variants
- Anàlisi

3 Productes i exponents

- Algorisme de Karatsuba
- Exponenciació ràpida
- Algorisme de Strassen

4 Altres algorismes

- Torres de Hanoi
- Mediana

L'**ordenació per fusió (mergesort)** és un bon exemple de **dividir i vèncer** que fa servir un nombre de comparacions gairebé òptim.

És un algorisme estable en un doble sentit:

- preserva l'ordre entre valors iguals
- es comporta igual independentment de com d'ordenada estigui l'entrada

Mergesort va ser inventat per **John von Neumann** l'any 1945.

L'operació clau consisteix en combinar (**fusionar**) dos vectors ordenats en un.

Donat un vector T de talla ≥ 2 , l'esquema de l'algorisme és:

- 1 Partir el vector en dues meitats.
- 2 Ordenar recursivament la primera meitat de T .
- 3 Ordenar recursivament la segona meitat de T .
- 4 Retornar la fusió de les dues meitats.

Algorisme de fusió bàsic

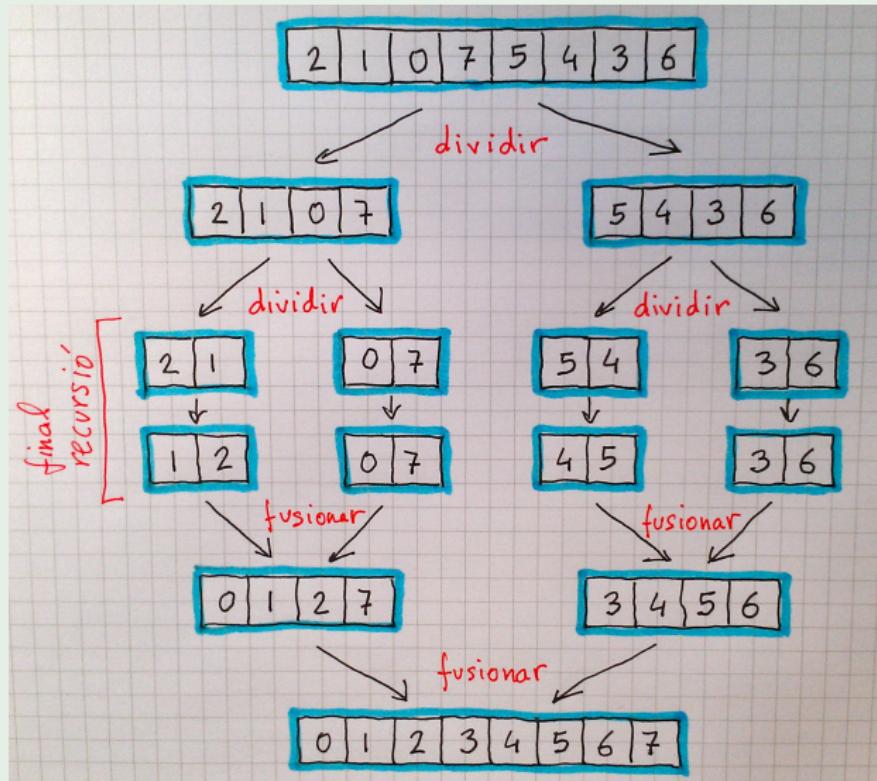
Ordenació per fusió (*Algoritmes en C++, EDA*)

```
template <typename elem>
void mergesort (vector<elem>& T) {
    mergesort(T, 0, T.size() - 1);
}

template <typename elem>
void mergesort (vector<elem>& T, int e, int d) {
    if (e < d) {
        int m = (e + d) / 2;
        mergesort(T, e, m);
        mergesort(T, m + 1, d);
        merge(T, e, m, d);
    }
}
```

Algorisme de fusió bàsic

Exemple (aquí, la recursió acaba per a talla 2, en l'algorisme és per a 1)



Algorisme de fusió bàsic

El cor de l'algorisme és fer la fusió de dos vectors ordenats.

Fusió (*Algorismes en C++, EDA*)

```
template <typename elem>
void merge (vector<elem>& T, int e, int m, int d) {
    vector<elem> B(d - e + 1);
    int i = e, j = m + 1, k = 0;
    while (i <= m and j <= d) {
        if (T[i] <= T[j]) B[k++] = T[i++];
        else                  B[k++] = T[j++];
    }
    while (i <= m) B[k++] = T[i++];
    while (j <= d) B[k++] = T[j++];
    for (k = 0; k <= d-e; ++k) T[e + k] = B[k];
}
```

Algorisme de fusió bàsic

```
template <typename elem>
void merge (vector<elem>& T, int e, int m, int d) {
    vector<elem> B(d - e + 1);
    int i = e, j = m + 1, k = 0;
    while (i <= m and j <= d) {
        if (T[i] <= T[j]) B[k++] = T[i++];
        else                  B[k++] = T[j++];
    }
    while (i <= m) B[k++] = T[i++];
    while (j <= d) B[k++] = T[j++];
    for (k = 0; k <= d-e; ++k) T[e + k] = B[k];
}
```

Exemple

0	1	2	7
---	---	---	---

3	4	5	6
---	---	---	---

0

Algorisme de fusió bàsic

```
template <typename elem>
void merge (vector<elem>& T, int e, int m, int d) {
    vector<elem> B(d - e + 1);
    int i = e, j = m + 1, k = 0;
    while (i <= m and j <= d) {
        if (T[i] <= T[j]) B[k++] = T[i++];
        else                  B[k++] = T[j++];
    }
    while (i <= m) B[k++] = T[i++];
    while (j <= d) B[k++] = T[j++];
    for (k = 0; k <= d-e; ++k) T[e + k] = B[k];
}
```

Exemple

0	1	2	7
---	---	---	---

3	4	5	6
---	---	---	---

0	1
---	---

Algorisme de fusió bàsic

```
template <typename elem>
void merge (vector<elem>& T, int e, int m, int d) {
    vector<elem> B(d - e + 1);
    int i = e, j = m + 1, k = 0;
    while (i <= m and j <= d) {
        if (T[i] <= T[j]) B[k++] = T[i++];
        else                  B[k++] = T[j++];
    }
    while (i <= m) B[k++] = T[i++];
    while (j <= d) B[k++] = T[j++];
    for (k = 0; k <= d-e; ++k) T[e + k] = B[k];
}
```

Exemple

0	1	2	7
---	---	---	---

3	4	5	6
---	---	---	---

0	1	2
---	---	---

Algorisme de fusió bàsic

```
template <typename elem>
void merge (vector<elem>& T, int e, int m, int d) {
    vector<elem> B(d - e + 1);
    int i = e, j = m + 1, k = 0;
    while (i <= m and j <= d) {
        if (T[i] <= T[j]) B[k++] = T[i++];
        else                  B[k++] = T[j++];
    }
    while (i <= m) B[k++] = T[i++];
    while (j <= d) B[k++] = T[j++];
    for (k = 0; k <= d-e; ++k) T[e + k] = B[k];
}
```

Exemple

0	1	2	7
---	---	---	---

3	4	5	6
---	---	---	---

0	1	2	3
---	---	---	---

Algorisme de fusió bàsic

```
template <typename elem>
void merge (vector<elem>& T, int e, int m, int d) {
    vector<elem> B(d - e + 1);
    int i = e, j = m + 1, k = 0;
    while (i <= m and j <= d) {
        if (T[i] <= T[j]) B[k++] = T[i++];
        else                  B[k++] = T[j++];
    }
    while (i <= m) B[k++] = T[i++];
    while (j <= d) B[k++] = T[j++];
    for (k = 0; k <= d-e; ++k) T[e + k] = B[k];
}
```

Exemple

0	1	2	7
---	---	---	---

3	4	5	6
---	---	---	---

0	1	2	3	4
---	---	---	---	---

Algorisme de fusió bàsic

```
template <typename elem>
void merge (vector<elem>& T, int e, int m, int d) {
    vector<elem> B(d - e + 1);
    int i = e, j = m + 1, k = 0;
    while (i <= m and j <= d) {
        if (T[i] <= T[j]) B[k++] = T[i++];
        else                  B[k++] = T[j++];
    }
    while (i <= m) B[k++] = T[i++];
    while (j <= d) B[k++] = T[j++];
    for (k = 0; k <= d-e; ++k) T[e + k] = B[k];
}
```

Exemple

0	1	2	7
---	---	---	---

3	4	5	6
---	---	---	---

0	1	2	3	4	5
---	---	---	---	---	---

Algorisme de fusió bàsic

```
template <typename elem>
void merge (vector<elem>& T, int e, int m, int d) {
    vector<elem> B(d - e + 1);
    int i = e, j = m + 1, k = 0;
    while (i <= m and j <= d) {
        if (T[i] <= T[j]) B[k++] = T[i++];
        else                  B[k++] = T[j++];
    }
    while (i <= m) B[k++] = T[i++];
    while (j <= d) B[k++] = T[j++];
    for (k = 0; k <= d-e; ++k) T[e + k] = B[k];
}
```

Exemple

0	1	2	7
---	---	---	---

3	4	5	6
---	---	---	---

0	1	2	3	4	5	6
---	---	---	---	---	---	---

Algorisme de fusió bàsic

```
template <typename elem>
void merge (vector<elem>& T, int e, int m, int d) {
    vector<elem> B(d - e + 1);
    int i = e, j = m + 1, k = 0;
    while (i <= m and j <= d) {
        if (T[i] <= T[j]) B[k++] = T[i++];
        else                  B[k++] = T[j++];
    }
    while (i <= m) B[k++] = T[i++];
    while (j <= d) B[k++] = T[j++];
    for (k = 0; k <= d-e; ++k) T[e + k] = B[k];
}
```

Exemple

0	1	2	7
---	---	---	---

3	4	5	6
---	---	---	---

0	1	2	3	4	5	6	7
---	---	---	---	---	---	---	---

```
template <typename elem>
void merge (vector<elem>& T, int e, int m, int d) {
    vector<elem> B(d - e + 1);
    int i = e, j = m + 1, k = 0;
    while (i <= m and j <= d) {
        if (T[i] <= T[j]) B[k++] = T[i++];
        else B[k++] = T[j++];
    }
    while (i <= m) B[k++] = T[i++];
    while (j <= d) B[k++] = T[j++];
    for (k = 0; k <= d-e; ++k) T[e + k] = B[k];
}
```

Observació

Després de cada comparació s'afegeix un element a la taula B llevat l'última, que n'afegeix almenys dos.

- Per tant, el nombre de **comparacions** de tipus elem és $n = d - e + 1$.
- El nombre d'**assignacions** de tipus elem és $2n$.
- El cost és **lineal** (assumint que assignar un elem és $\Theta(1)$).

Algorisme de fusió bàsic

```
template <typename elem>
void mergesort (vector<elem>& T, int e, int d) {
    if (e < d) {
        int m = (e + d) / 2;
        mergesort (T, e, m);
        mergesort (T, m + 1, d);
        merge (T, e, m, d);
    }
}
```

Donat que el procediment `merge` és lineal,
el cost de l'ordenació per fusió es pot expressar amb la recurrència

$$T(1) = \Theta(1)$$

$$T(n) = 2T(n/2) + \Theta(n) \text{ per a } n > 1$$

i, aplicant el teorema mestre de recurrències divisores, tenim que

$$T(n) \in \Theta(n \log n).$$

Ordenació per fusió amb inserció per a vectors petits (*Alg. en C++, EDA*)

```
template <typename elem>
void mergesort (vector<elem>& T, int e, int d) {
    const int talla_critica = 50;
    if (d-e < talla_critica)
        ordena_insercio(T, e, d);
    else {
        int m = (e + d) / 2;
        mergesort(T, e, m);
        mergesort(T, m + 1, d);
        merge(T, e, m, d);
    }
}
```

Les dues versions iteratives que veurem parteixen del fet que les fusions només comencen al final de la recursió, de manera que:

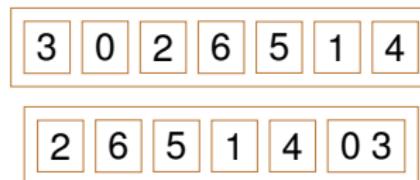
- comencen directament pels elements a ordenar i
- arriben al vector ordenat mitjançant fusions.

Ordenació per fusió iterativa 1

Versió del llibre *Algorithms* de Dasgupta/Papadimitriou/Vazirani en pseudocodi (pàg. 51). Es fa servir el TAD cua amb operacions

- **inject (Q, e)**: afegir l'element e a la cua Q i
- **eject (Q)**: funció que treu i retorna l'últim element de Q

```
function mergesort_queue(a[1...n])
    Q = [] (cua de vectors buida)
    for i=1 to n:
        inject(Q, a[i])
    while |Q| > 1:
        inject(Q, merge(eject(Q), eject(Q)))
    return eject(Q)
```

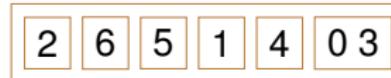


Ordenació per fusió iterativa 1

Versió del llibre *Algorithms* de Dasgupta/Papadimitriou/Vazirani en pseudocodi (pàg. 51). Es fa servir el TAD cua amb operacions

- `inject (Q , e)`: afegir l'element e a la cua Q i
- `eject (Q)`: funció que treu i retorna l'últim element de Q

```
function mergesort_queue(a[1...n])
    Q = [] (cua de vectors buida)
    for i=1 to n:
        inject(Q, a[i])
    while |Q| > 1:
        inject(Q, merge(eject(Q), eject(Q)))
    return eject(Q)
```

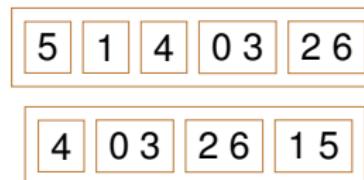


Ordenació per fusió iterativa 1

Versió del llibre *Algorithms* de Dasgupta/Papadimitriou/Vazirani en pseudocodi (pàg. 51). Es fa servir el TAD cua amb operacions

- **inject (Q, e)**: afegir l'element e a la cua Q i
- **eject (Q)**: funció que treu i retorna l'últim element de Q

```
function mergesort_queue(a[1...n])
    Q = [] (cua de vectors buida)
    for i=1 to n:
        inject(Q, a[i])
    while |Q| > 1:
        inject(Q, merge(eject(Q), eject(Q)))
    return eject(Q)
```



Ordenació per fusió iterativa 1

Versió del llibre *Algorithms* de Dasgupta/Papadimitriou/Vazirani en pseudocodi (pàg. 51). Es fa servir el TAD cua amb operacions

- `inject(Q, e)`: afegir l'element `e` a la cua `Q` i
- `eject(Q)`: funció que treu i retorna l'últim element de `Q`

```
function mergesort_queue(a[1...n])
    Q = [] (cua de vectors buida)
    for i=1 to n:
        inject(Q, a[i])
    while |Q| > 1:
        inject(Q, merge(eject(Q), eject(Q)))
    return eject(Q)
```

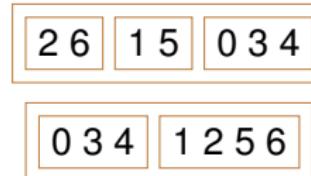


Ordenació per fusió iterativa 1

Versió del llibre *Algorithms* de Dasgupta/Papadimitriou/Vazirani en pseudocodi (pàg. 51). Es fa servir el TAD cua amb operacions

- `inject(Q, e)`: afegir l'element `e` a la cua `Q` i
- `eject(Q)`: funció que treu i retorna l'últim element de `Q`

```
function mergesort_queue(a[1...n])
    Q = [] (cua de vectors buida)
    for i=1 to n:
        inject(Q, a[i])
    while |Q| > 1:
        inject(Q, merge(eject(Q), eject(Q)))
    return eject(Q)
```



Ordenació per fusió iterativa 1

Versió del llibre *Algorithms* de Dasgupta/Papadimitriou/Vazirani en pseudocodi (pàg. 51). Es fa servir el TAD cua amb operacions

- `inject(Q, e)`: afegir l'element `e` a la cua `Q` i
- `eject(Q)`: funció que treu i retorna l'últim element de `Q`

```
function mergesort_queue(a[1...n])
    Q = [] (cua de vectors buida)
    for i=1 to n:
        inject(Q, a[i])
    while |Q| > 1:
        inject(Q, merge(eject(Q), eject(Q)))
    return eject(Q)
```

0 1 2 3 4 5 6

0 3 4 1 2 5 6

Ordenació per fusió iterativa 2 (Algorismes en C++, EDA)

```
template <typename elem>
void mergesort_bottom_up (vector<elem>& T) {
    int n = T.size();
    for (int m = 1; m < n; m *= 2) {
        for (int i = 0; i < n-m; i += 2*m) {
            merge(T, i, i+m-1, min(i+2*m-1, n-1));
    } } }
```



Ordenació per fusió iterativa 2 (Algorismes en C++, EDA)

```
template <typename elem>
void mergesort_bottom_up (vector<elem>& T) {
    int n = T.size();
    for (int m = 1; m < n; m *= 2) {
        for (int i = 0; i < n-m; i += 2*m) {
            merge(T, i, i+m-1, min(i+2*m-1, n-1));
    } } }
```



Ordenació per fusió iterativa 2 (Algorismes en C++, EDA)

```
template <typename elem>
void mergesort_bottom_up (vector<elem>& T) {
    int n = T.size();
    for (int m = 1; m < n; m *= 2) {
        for (int i = 0; i < n-m; i += 2*m) {
            merge(T, i, i+m-1, min(i+2*m-1, n-1));
    } } }
```



Ordenació per fusió iterativa 2 (Algorismes en C++, EDA)

```
template <typename elem>
void mergesort_bottom_up (vector<elem>& T) {
    int n = T.size();
    for (int m = 1; m < n; m *= 2) {
        for (int i = 0; i < n-m; i += 2*m) {
            merge(T, i, i+m-1, min(i+2*m-1, n-1));
    } } }
```



Ordenació per fusió iterativa 2 (Algorismes en C++, EDA)

```
template <typename elem>
void mergesort_bottom_up (vector<elem>& T) {
    int n = T.size();
    for (int m = 1; m < n; m *= 2) {
        for (int i = 0; i < n-m; i += 2*m) {
            merge(T, i, i+m-1, min(i+2*m-1, n-1));
    } } }
```



Ordenació per fusió iterativa 2 (Algorismes en C++, EDA)

```
template <typename elem>
void mergesort_bottom_up (vector<elem>& T) {
    int n = T.size();
    for (int m = 1; m < n; m *= 2) {
        for (int i = 0; i < n-m; i += 2*m) {
            merge(T, i, i+m-1, min(i+2*m-1, n-1));
    } } }
```

0 1 2 3 4 5 6

0 2 3 6 1 4 5

Tema 2. Algorismes de dividir i vèncer

1 Ordenació per fusió

- Algorisme de fusió bàsic
- Variants

2 Ordenació ràpida

- Algorisme general
- Variants
- Anàlisi

3 Productes i exponents

- Algorisme de Karatsuba
- Exponenciació ràpida
- Algorisme de Strassen

4 Altres algorismes

- Torres de Hanoi
- Mediana

L'**ordenació ràpida**, o *QuickSort*, és l'algorisme d'ordenació genèric més ràpid, com el seu nom indica.

Tot i que el seu cost en el cas pitjor és $\Theta(n^2)$, el cas mitjà és $\Theta(n \log n)$, i l'eficiència del seu bucle intern fa que sigui el millor algorisme a la pràctica.

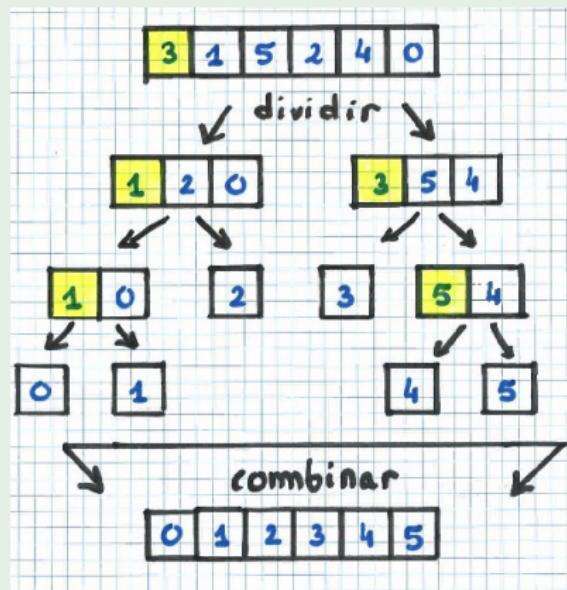
QuickSort va ser inventat per **C. A. R. Hoare** l'any 1960.
Ha estat molt estudiat des de llavors.

Donat un vector T d'almenys 2 elements, l'**algorisme bàsic** fa el següent:

- 1 Triar un element x de T .
- 2 Dividir T en dos grups disjunts T_1 i T_2 de forma que:
 - T_1 conté elements $\leq x$ de T , i
 - T_2 conté elements $\geq x$ de T .
- 3 Ordenar T_1 i T_2 recursivament.
- 4 Retornar T_1 seguit de T_2 .

Algorisme general

Exemple



Algorisme general

Ordenació ràpida (*Algoritmes en C++, EDA*)

```
void quicksort (vector<elem>& T) {  
    quicksort (T, 0, T.size() - 1);  
}  
  
template <typename elem>  
void quicksort (vector<elem>& T, int e, int d) {  
    if (e < d) {  
        int q = partition (T, e, d);  
        quicksort (T, e, q);  
        quicksort (T, q + 1, d);  
    }    }
```

- Precondició de $q = \text{partition}(T, e, d)$: $0 \leq e \leq d \leq T.size() - 1$
- Postcondició de $q = \text{partition}(T, e, d)$: $\exists x$ tal que $\forall i$
 - si $e \leq i \leq q$, tenim que $T[i] \leq x$
 - si $q < i \leq d$, tenim que $T[i] \geq x$

- Paral·lelismes amb l'ordenació per fusió:

- resol dos subproblemes i
- fa un treball addicional lineal.

- Estratègies oposades:

- **Ordenació per fusió:**

divisió en subproblemes directa, fusió dels vectors feta amb cura

- **Ordenació ràpida:**

divisió en subproblemes feta amb cura, combinació directa.

Partició original de Hoare amb el primer element com a pivot.

Partició de Hoare (*Algorismes en C++, EDA*)

```
template <typename elem>
int partition (vector<elem>& T, int e, int d) {
    elem x = T[e];
    int i = e - 1;
    int j = d + 1;
    for (;;) {
        while (x < T[--j]);
        while (T[++i] < x);
        if (i >= j) return j;
        swap(T[i], T[j]);
    }
}
```

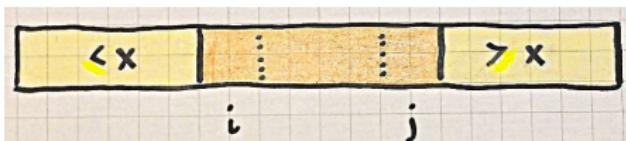
Partició de Hoare

- Inici de la funció `partition(T, e, d)`:

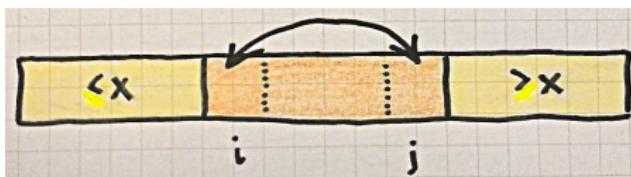


- Bucle principal:

- es troben els valors de i, j més centrals tals que



- s'intercanvien els continguts de les posicions i, j



- Una altra diferència amb l'ordenació per fusió és que els subproblemes no sempre tenen la mateixa mida.
- La mida dels subproblemes depèn de la tria de l'element inicial, anomenat pivot.

Considerem tres estratègies per a l'elecció del pivot:

1 Tria el **primer** element:

- és acceptable si l'entrada és aleatòria
- si l'entrada està ordenada (en ordre creixent o decreixent), l'algorisme pren temps $\Theta(n^2)$ per no fer res!

2 Tria un element **aleatori**

- en mitjana divideix el problema en subproblemes semblants
- no sempre fa l'algorisme més ràpid
(cost de la generació de nombres aleatoris, etc.)

3 Tria la **mediana** de tres elements

- la millor opció seria triar la mediana del vector, però és massa cara
- una bona estimació és fer la mediana de 3, normalment el primer, l'element mig i l'últim

Considerem tres estratègies per a l'elecció del pivot:

1 Tria el **primer** element:

- és acceptable si l'entrada és aleatòria
- si l'entrada està ordenada (en ordre creixent o decreixent), l'algorisme pren temps $\Theta(n^2)$ per no fer res!

2 Tria un element **aleatori**

- en mitjana divideix el problema en subproblemes semblants
- no sempre fa l'algorisme més ràpid
(cost de la generació de nombres aleatoris, etc.)

3 Tria la **mediana** de tres elements

- la millor opció seria triar la mediana del vector, però és massa cara
- una bona estimació és fer la mediana de 3, normalment el primer, l'element mig i l'últim

Considerem tres estratègies per a l'elecció del pivot:

1 Tria el **primer** element:

- és acceptable si l'entrada és aleatòria
- si l'entrada està ordenada (en ordre creixent o decreixent), l'algorisme pren temps $\Theta(n^2)$ per no fer res!

2 Tria un element **aleatori**

- en mitjana divideix el problema en subproblemes semblants
- no sempre fa l'algorisme més ràpid
(cost de la generació de nombres aleatoris, etc.)

3 Tria la **mediana** de tres elements

- la millor opció seria triar la mediana del vector, però és massa cara
- una bona estimació és fer la mediana de 3,
normalment el primer, l'element mig i l'últim

Ordenació ràpida per a pivot aleatori (*Algorismes en C++, EDA*)

```
template <typename elem>
void quicksort (vector<elem>& T, int e, int d) {
    if (e < d) {
        int p = randint(e, d);
        swap(T[e], T[p]);
        int q = partition(T, e, d);
        quicksort(T, e, q);
        quicksort(T, q + 1, d);
    }
}
```

Ordenació ràpida amb la mediana com a pivot

```
template <typename elem>
void quicksort (vector<elem>& T, int e, int d) {
    if (e < d) {
        int centre = (e + d) / 2;
1       if (T[e] < T[centre]) swap(T[centre], T[e]);
2       if (T[d] < T[centre]) swap(T[centre], T[d]);
3       if (T[d] < T[e])      swap(T[e], T[d]);
        // el pivot és a la posició e
        int q = partition(T, e, d);
        quicksort(T, e, q);
        quicksort(T, q + 1, d);
    }
}
```

Després de les línies 1, 2 i 3:

$$T[\text{centre}] \leq T[e], \quad T[\text{centre}] \leq T[d], \quad T[e] \leq T[d].$$

Per tant, $T[\text{centre}] \leq T[e] \leq T[d]$ i la mediana és $T[e]$.

Per a vectors molt petits, l'ordenació per inserció va millor que *QuickSort*. Una bona solució, doncs, és tallar la recursió quan el vector és més petit que una certa mida (normalment, entre 5 i 20).

Ordenació ràpida amb inserció per a vectors petits

```
template <typename elem>
void quicksort (vector<elem>& T, int e, int d) {
    const int talla_critica = 20;
    if (d - e < talla_critica)
        ordena_insercio(T, e, d);
    else {
        int q = partition(T, e, d);
        quicksort(T, e, q);
        quicksort(T, q + 1, d);
    }
}
```

Recurrència

Sigui $T(n)$ el cost de l'algorisme d'ordenació ràpida amb n elements.

Llavors, per a $n \geq 2$,

$$T(n) = T(i) + T(n - i) + \Theta(n),$$

on i és el nombre d'elements de la primera meitat, i $n - i$ de la segona.

Cas general

$$T(n) = T(i) + T(n - i) + \Theta(n)$$

En el cas pitjor, el pivot és sempre l'element més petit o el més gran. Com a resultat, una crida recursiva té cost constant i l'altra $T(n - 1)$.

Cas pitjor

$$T(n) = T(n - 1) + \Theta(n)$$

Resolent la recurrència (directament, o amb el teorema mestre), s'obté:

$$T(n) \in \Theta(n^2).$$

Cas general

$$T(n) = T(i) + T(n - i) + \Theta(n)$$

En el cas pitjor, el pivot és sempre l'element més petit o el més gran. Com a resultat, una crida recursiva té cost constant i l'altra $T(n - 1)$.

Cas pitjor

$$T(n) = T(n - 1) + \Theta(n)$$

Resolent la recurrència (directament, o amb el teorema mestre), s'obté:

$$T(n) \in \Theta(n^2).$$

Cas general

$$T(n) = T(i) + T(n - i) + \Theta(n)$$

En el cas millor, el pivot és la mediana del vector i, per tant, els dos subvectors tenen la mateixa mida.

Cas millor

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

És la mateixa recurrència de l'ordenació per fusió que, pel teorema mestre de recurrències divisores, dóna el cost:

$$T(n) \in \Theta(n \log n).$$

Cas general

$$T(n) = T(i) + T(n - i) + \Theta(n)$$

En el cas millor, el pivot és la mediana del vector i, per tant, els dos subvectors tenen la mateixa mida.

Cas millor

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

És la mateixa recurrència de l'ordenació per fusió que, pel teorema mestre de recurrències divisores, dóna el cost:

$$T(n) \in \Theta(n \log n).$$

Anàlisi: cas mitjà

Cas general

$$T(n) = T(i) + T(n - i) + \Theta(n)$$

Per al cas mitjà, suposarem que la partició és aleatòria.

Hi ha una probabilitat $\frac{1}{n-1}$ de tenir una meitat de mida i per a $1 \leq i \leq n-1$.

El valor mitjà de $T(i)$ i de $T(n - i)$ serà de $\frac{1}{n-1} \sum_{j=1}^{n-1} T(j)$.

Cas mitjà

Per a alguna constant c ,

$$T(n) = \frac{2}{n-1} \left[\sum_{j=1}^{n-1} T(j) \right] + cn \quad (1)$$

Anàlisi: cas mitjà

Cas general

$$T(n) = T(i) + T(n - i) + \Theta(n)$$

Per al cas mitjà, suposarem que la partició és aleatòria.

Hi ha una probabilitat $\frac{1}{n-1}$ de tenir una meitat de mida i per a $1 \leq i \leq n-1$.

El valor mitjà de $T(i)$ i de $T(n - i)$ serà de $\frac{1}{n-1} \sum_{j=1}^{n-1} T(j)$.

Cas mitjà

Per a alguna constant c ,

$$T(n) = \frac{2}{n-1} \left[\sum_{j=1}^{n-1} T(j) \right] + cn \quad (1)$$

Anàlisi: cas mitjà

Multipliquem l'equació 1 per $n - 1$:

$$(n - 1)T(n) = 2 \left[\sum_{j=1}^{n-1} T(j) \right] + cn(n - 1). \quad (2)$$

Per eliminar el sumatori, escrivim el cas $n + 1$

$$nT(n+1) = 2 \left[\sum_{j=1}^n T(j) \right] + c(n+1)n \quad (3)$$

i restem l'equació (2) de la (3)

$$nT(n+1) - (n - 1)T(n) = 2T(n) + 2cn. \quad (4)$$

Reordenant els termes,

$$nT(n+1) = (n + 1)T(n) + 2cn. \quad (5)$$

Anàlisi: cas mitjà

Dividim l'equació (5) per $n(n+1)$:

$$\frac{T(n+1)}{n+1} = \frac{T(n)}{n} + \frac{2c}{n+1}. \quad (6)$$

Substituem n per tots els valors de $n - 1$ fins a 1:

$$\frac{T(n)}{n} = \frac{T(n-1)}{n-1} + \frac{2c}{n}. \quad (7)$$

$$\frac{T(n-1)}{n-1} = \frac{T(n-2)}{n-2} + \frac{2c}{n-1}. \quad (8)$$

⋮

$$\frac{T(2)}{2} = \frac{T(1)}{1} + \frac{2c}{2}. \quad (9)$$

La suma de totes les equacions (7)–(9) dóna

$$\frac{T(n)}{n} = \frac{T(1)}{1} + 2c \sum_{i=2}^n \frac{1}{i}. \quad (10)$$

D'altra banda, se sap que

$$\sum_{i=2}^n \frac{1}{i} = \ln(n) + \gamma - 1,$$

on $\gamma \approx 0.577$ és la constant d'Euler.

Substituint aquest valor en l'equació (10) obtinguda abans

$$\frac{T(n)}{n} = \frac{T(1)}{1} + 2c \sum_{i=2}^n \frac{1}{i}$$

es dedueix que

$$\frac{T(n)}{n} \in \Theta(\log n)$$

i, per tant,

$$T(n) \in \Theta(n \log n).$$

Tema 2. Algorismes de dividir i vèncer

1 Ordenació per fusió

- Algorisme de fusió bàsic
- Variants

2 Ordenació ràpida

- Algorisme general
- Variants
- Anàlisi

3 Productes i exponents

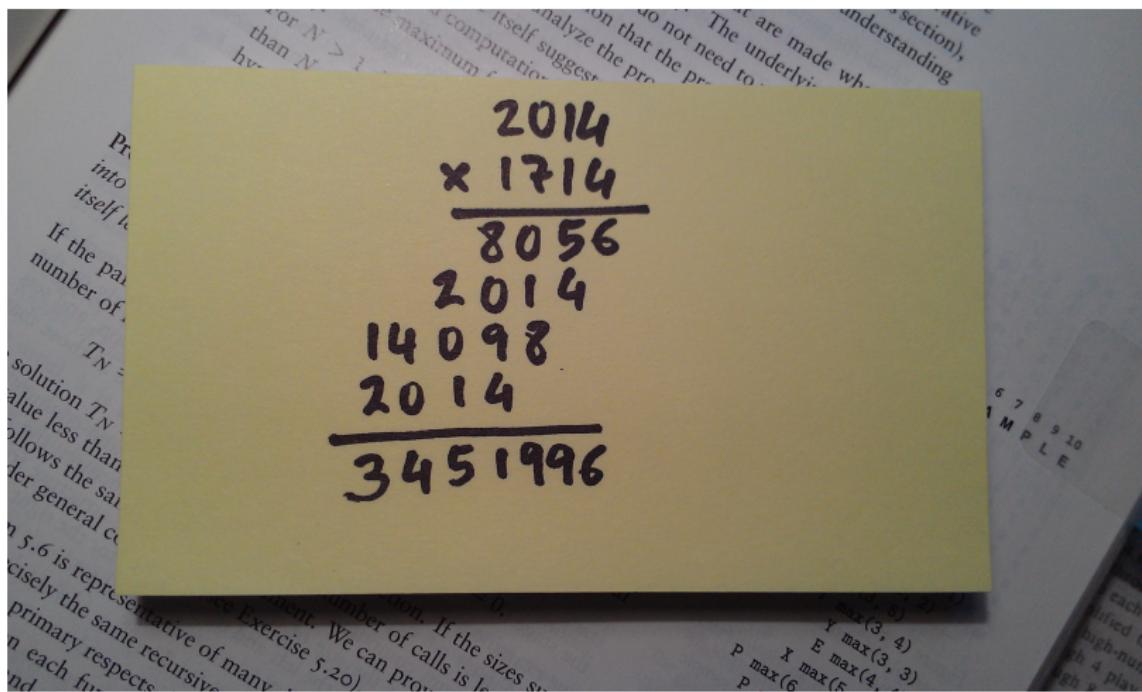
- Algorisme de Karatsuba
- Exponenciació ràpida
- Algorisme de Strassen

4 Altres algorismes

- Torres de Hanoi
- Mediana

Algorisme de Karatsuba

Quin cost té l'algorisme escolar de multiplicació?



Algorisme de Karatsuba

$\Theta(n^2)$, si n és el nombre de díigits.

A handwritten multiplication diagram for 2014×1714 . The numbers are arranged vertically:

$$\begin{array}{r} 2014 \\ \times 1714 \\ \hline 8056 \\ 2014 \\ 14098 \\ 2014 \\ \hline 3451996 \end{array}$$

The first two digits of each number (2 and 1) are highlighted with a green rectangular box. The result of multiplying these digits, 8056, is written inside the box. To the right of the box, the label "n x n" is written in green.

Algorisme de Karatsuba

El matemàtic rus A. Kolmogorov va conjecturar que l'algorisme escolar de multiplicació és òptim.

Conjectura (Kolmogorov, 1952)

Qualsevol algorisme per multiplicar dos nombres de n díigits té cost $\Omega(n^2)$.

Un estudiant de 23 anys de Kolmogorov, Anatolii Alexeevitch Karatsuba, va trobar un algorisme de cost $\Theta(n^{1.585})$.

Refutació (Karatsuba, 1960)

Hi ha un algorisme que multiplica dos nombres de n díigits en temps $\Theta(n^{\log_2 3}) \approx \Theta(n^{1.585})$.

Algorisme de Karatsuba

El matemàtic rus A. Kolmogorov va conjecturar que l'algorisme escolar de multiplicació és òptim.

Conjectura (Kolmogorov, 1952)

Qualsevol algorisme per multiplicar dos nombres de n díigits té cost $\Omega(n^2)$.

Un estudiant de 23 anys de Kolmogorov, Anatolii Alexeevitch Karatsuba, va trobar un algorisme de cost $\Theta(n^{1.585})$.

Refutació (Karatsuba, 1960)

Hi ha un algorisme que multiplica dos nombres de n díigits en temps $\Theta(n^{\log_2 3}) \approx \Theta(n^{1.585})$.

Algorisme de Karatsuba

L'origen de la idea de l'algorisme de Karatsuba es remunta al s. XVIII.

Observació

El matemàtic Carl Friedrich Gauss (1777-1855) va observar que malgrat que el producte de dos complexos

$$(a + bi)(c + di) = ac - bd + (bc + ad)i$$

sembla requerir 4 productes de reals, es pot obtenir amb 3. La raó és que

$$bc + ad = (a + b)(c + d) - ac - bd.$$

Algorisme de Karatsuba

Avancem ara al s. XX.

Suposem que x i y són dos naturals de n bits, amb n parell.

Expressem x , y en dues parts:

$$x = \boxed{x_E} \quad \boxed{x_D} = 2^{n/2}x_E + x_D$$

$$y = \boxed{y_E} \quad \boxed{y_D} = 2^{n/2}y_E + y_D$$

Exemple

Si $x = 10010111_2$ i $y = 11001010_2$ (el subíndex 2 vol dir "en binari"), llavors

$$x = \boxed{x_E} \quad \boxed{x_D} = \boxed{1001}_2 \quad \boxed{0111}_2$$

$$y = \boxed{y_E} \quad \boxed{y_D} = \boxed{1100}_2 \quad \boxed{1010}_2$$

Algorisme de Karatsuba

Ara, el producte

$$xy = (2^{n/2}x_E + x_D)(2^{n/2}y_E + y_D)$$

es pot reescriure com

$$xy = 2^n x_E y_E + 2^{n/2}(x_E y_D + x_D y_E) + x_D y_D$$

El cas senar és similar.

Un algorisme basat en aquesta expressió que calculés recursivament els productes tindria un cost

$$T(n) = 4T(n/2) + \Theta(n).$$

Algorisme de Karatsuba

Ara, el producte

$$xy = (2^{n/2}x_E + x_D)(2^{n/2}y_E + y_D)$$

es pot reescriure com

$$xy = 2^n x_E y_E + 2^{n/2}(x_E y_D + x_D y_E) + x_D y_D$$

El cas senar és similar.

Un algorisme basat en aquesta expressió que calculés recursivament els productes tindria un cost

$$T(n) = 4T(n/2) + \Theta(n).$$

$$T(n) = 4T(n/2) + \Theta(n)$$

Pel teorema mestre de recurredades divisores, sabem que $T(n) \in \Theta(n^2)$.

Però si apliquem el truc de Gauss, podem obtenir el producte xy fent servir 3 subproductes en lloc de 4 i, així, rebaixar el cost quadràtic.

Algorisme de Karatsuba

Com abans, observem que

$$x_E y_D + x_D y_E = (x_E + x_D)(y_E + y_D) - x_E y_E - x_D y_D.$$

Si ara anomenem

$$\textcolor{red}{a} = x_E y_E, \quad \textcolor{red}{b} = x_D y_D, \quad \textcolor{red}{c} = (x_E + x_D)(y_E + y_D),$$

llavors l'equació

$$xy = 2^n x_E y_E + 2^{n/2}(x_E y_D + x_D y_E) + x_D y_D$$

es pot reescrivre com

$$2^n \textcolor{red}{a} + 2^{n/2}(\textcolor{red}{c} - \textcolor{red}{a} - \textcolor{red}{b}) + \textcolor{red}{b}$$

que només depèn de 3 subproductes.

Algorisme de Karatsuba

Com abans, observem que

$$x_E y_D + x_D y_E = (x_E + x_D)(y_E + y_D) - x_E y_E - x_D y_D.$$

Si ara anomenem

$$\textcolor{red}{a} = x_E y_E, \quad \textcolor{red}{b} = x_D y_D, \quad \textcolor{red}{c} = (x_E + x_D)(y_E + y_D),$$

llavors l'equació

$$xy = 2^n x_E y_E + 2^{n/2}(x_E y_D + x_D y_E) + x_D y_D$$

es pot reescriure com

$$2^n \textcolor{red}{a} + 2^{n/2}(\textcolor{red}{c} - \textcolor{red}{a} - \textcolor{red}{b}) + \textcolor{red}{b}$$

que només depèn de 3 subproductes.

Algorisme de Karatsuba

La nova expressió dóna lloc a un algorisme de cost

$$T(n) = 3T(n/2) + \Theta(n)$$

i, pel teorema mestre de recurredències divisores, sabem que

$$T(n) \in \Theta(n^{\log_2 3}) \approx \Theta(n^{1.585}).$$

Exponenciació ràpida

- L'algorisme iteratiu evident per calcular x^n faria servir la descomposició

$$x^n = \underbrace{x \cdot x \cdot \dots \cdot x}_{n \text{ factors}}$$

que faria $\Theta(n - 1) = \Theta(n)$ multiplicacions.

- Però amb un enfocament recursiu, tenim

$$x^n = (x^{n/2})^2$$

quan n és parell i

$$x^n = x^{n-1} \cdot x = (x^{(n-1)/2})^2 \cdot x$$

quan n és senar.

- Cas base: $x^0 = 1$

Exponenciació ràpida

- L'algorisme iteratiu evident per calcular x^n faria servir la descomposició

$$x^n = \underbrace{x \cdot x \cdot \dots \cdot x}_{n \text{ factors}}$$

que faria $\Theta(n - 1) = \Theta(n)$ multiplicacions.

- Però amb un enfocament recursiu, tenim

$$x^n = (x^{n/2})^2$$

quan n és parell i

$$x^n = x^{n-1} \cdot x = (x^{(n-1)/2})^2 \cdot x$$

quan n és senar.

- Cas base: $x^0 = 1$

Exponenciació ràpida

Exemple

Per calcular x^{62} , l'algorisme faria els càlculs

$$\begin{aligned}x^{62} &= (\textcolor{brown}{x}^{31})^2, \quad x^{31} = (\textcolor{brown}{x}^{15})^2 \cdot x, \quad x^{15} = (\textcolor{brown}{x}^7)^2 \cdot x \\x^7 &= (\textcolor{brown}{x}^3)^2 \cdot x, \quad x^3 = \textcolor{brown}{x}^2 \cdot x, \quad x = 1 \cdot x\end{aligned}$$

(en groc, els valors calculats recursivament)

Exponenciació ràpida

Exponenciació ràpida

```
double potencia (double x, int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        double y = potencia (x, n / 2);  
        if (n % 2 == 0) return y * y;  
        else return y * y * x;  
    } }
```

El càlcul del cost és directe i ve donat per la recurrència

$$T(n) = T(n/2) + \Theta(1)$$

que, segons el teorema mestre de recurrències divisores, implica

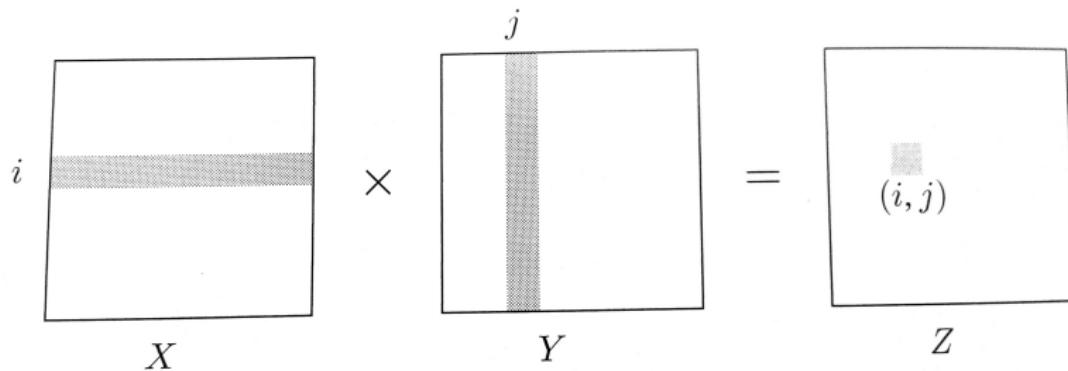
$$T(n) \in \Theta(\log n).$$

Algorisme de Strassen

El producte de dues matrius X i Y de mida $n \times n$ és una matriu Z de mida $n \times n$ tal que

$$Z_{ij} = \sum_{k=1}^n X_{ik} Y_{kj}.$$

És a dir, Z_{ij} és el producte de la fila i -èsima de X per la columna j -èsima de Y .



Algorisme de Strassen

La fórmula

$$Z_{ij} = \sum_{k=1}^n X_{ik} Y_{kj}.$$

calculada per a cada i, j amb valors $i, j = 1 \dots n$ implica un algorisme $\Theta(n^3)$.

Es va pensar que el cost $\Theta(n^3)$ era òptim fins al 1969,
quan Volker Strassen va anunciar un algorisme de cost $\Theta(n^{\log_2 7}) \approx \Theta(n^{2.81})$.

Algorisme de Strassen

La fórmula

$$Z_{ij} = \sum_{k=1}^n X_{ik} Y_{kj}.$$

calculada per a cada i, j amb valors $i, j = 1 \dots n$ implica un algorisme $\Theta(n^3)$.

Es va pensar que el cost $\Theta(n^3)$ era òptim fins al 1969,
quan Volker Strassen va anunciar un algorisme de cost $\Theta(n^{\log_2 7}) \approx \Theta(n^{2.81})$.

Algorisme de Strassen

Producte estàndard de matrius

Algorisme $\Theta(n^3)$, adaptat de *Data Structure and Alg. Analysis in C++, Weiss*.

```
matrix<int> producte_matrius
    (const matrix<int> & a, const matrix<int> & b)
{
    int n = a.numrows();
    matrix<int> c(n, n);
    int i;
    for (i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            c[i][j] = 0;
    for (i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            for (int k = 0; k < n; k++)
                c[i][j] += a[i][k] * b[k][j];
    return c;
}
```

Algorisme de Strassen

Una primera idea és que el producte de matrius es pot fer *per blocs*.

Dividim X i Y en quatre quadrants cadascuna:

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix}, \quad Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}.$$

Aleshores, es pot veure que

$$XY = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}.$$

Els productes de matrius es poden calcular aleshores recursivament.

Exemple

Per fer el producte XY

$$XY = \begin{bmatrix} 4 & 3 & 1 & 6 \\ 1 & 5 & 2 & 7 \\ 2 & 1 & 5 & 9 \\ 3 & 4 & 2 & 6 \end{bmatrix} \begin{bmatrix} 2 & 6 & 9 & 4 \\ 3 & 2 & 4 & 1 \\ 1 & 1 & 8 & 3 \\ 2 & 1 & 3 & 1 \end{bmatrix}$$

definim les vuit matrius 2×2

$$A = \begin{bmatrix} 4 & 3 \\ 1 & 5 \end{bmatrix}, B = \begin{bmatrix} 1 & 6 \\ 2 & 7 \end{bmatrix}, E = \begin{bmatrix} 2 & 6 \\ 3 & 2 \end{bmatrix}, F = \begin{bmatrix} 9 & 4 \\ 4 & 1 \end{bmatrix},$$
$$C = \begin{bmatrix} 2 & 1 \\ 3 & 4 \end{bmatrix}, D = \begin{bmatrix} 5 & 9 \\ 2 & 6 \end{bmatrix}, G = \begin{bmatrix} 1 & 1 \\ 2 & 1 \end{bmatrix}, H = \begin{bmatrix} 8 & 3 \\ 3 & 1 \end{bmatrix}.$$

Algorisme de Strassen

Quin cost té el nou algorisme? Recordem que es basa en el producte

$$XY = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}.$$

El cost $T(n)$ és la suma de fer:

- vuit productes de matrius de mida $n/2$: $8T(n/2)$
- quatre sumes de matrius de mida $n/2$: $\Theta(n^2)$

Per tant, tenim la recurrència

$$T(n) = 8T(n/2) + \Theta(n^2)$$

que, pel teorema mestre de recurrències divisores, dóna

$$T(n) \in \Theta(n^{\log_2 8}) = \Theta(n^3).$$

Algorisme de Strassen

Però el nombre de productes es pot reduir a 7.

Volker Strassen (1969)

$$XY = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix}$$

on

$$P_1 = A(F - H), \quad P_4 = D(G - E)$$

$$P_2 = (A + B)H, \quad P_5 = (A + D)(E + H)$$

$$P_3 = (C + D)E, \quad P_6 = (B - D)(G + H)$$

$$P_7 = (A - C)(E + F)$$

Fent servir la descomposició de Strassen, obtenim un algorisme amb cost

$$T(n) = 7T(n/2) + \Theta(n^2)$$

que, pel teorema mestre de recurrències divisores, dóna

$$T(n) \in \Theta(n^{\log_2 7}) \approx \Theta(n^{2.81}).$$

Algorisme de Strassen

Però el nombre de productes es pot reduir a 7.

Volker Strassen (1969)

$$XY = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix}$$

on

$$P_1 = A(F - H), \quad P_4 = D(G - E)$$

$$P_2 = (A + B)H, \quad P_5 = (A + D)(E + H)$$

$$P_3 = (C + D)E, \quad P_6 = (B - D)(G + H)$$

$$P_7 = (A - C)(E + F)$$

Fent servir la descomposició de Strassen, obtenim un algorisme amb cost

$$T(n) = 7T(n/2) + \Theta(n^2)$$

que, pel teorema mestre de recurrències divisores, dóna

$$T(n) \in \Theta(n^{\log_2 7}) \approx \Theta(n^{2.81}).$$

Tema 2. Algorismes de dividir i vèncer

1 Ordenació per fusió

- Algorisme de fusió bàsic
- Variants

2 Ordenació ràpida

- Algorisme general
- Variants
- Anàlisi

3 Productes i exponents

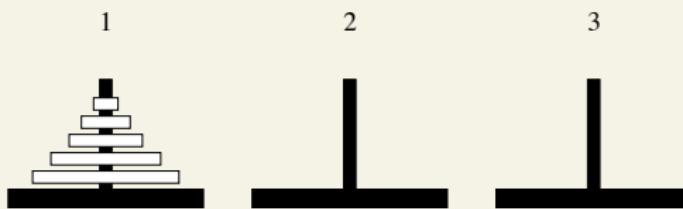
- Algorisme de Karatsuba
- Exponenciació ràpida
- Algorisme de Strassen

4 Altres algorismes

- Torres de Hanoi
- Mediana

Joc de les torres de Hanoi

Donades tres varetes 1, 2 i 3 i una sèrie de discs col·locats en la primera com indica el dibuix



cal traslladar-los a la vareta 2 amb dues restriccions:

- només es pot moure un disc a la vegada i
- mai es pot col·locar un disc a sobre d'un de més petit.

Solució

Per traslladar n discs de la vareta 1 a la 2,

- si $n = 1$, moure l'únic disc de 1 a 2.
- si $n > 1$,
 - 1 traslladar $n - 1$ discs de 1 a 3
 - 2 moure el disc restant (el més gran) de 1 a 2
 - 3 traslladar $n - 1$ discs de 3 a 2

Algorisme

```
void hanoi(int n, int a, int b, int c){  
    if (n > 0) {  
        hanoi(n-1, a, c, b);  
        cout << a << ' ' << b << endl;  
        hanoi(n-1, c, b, a);  
    }    }  
}
```

Cost

El cost creix com el nombre de moviments. Definim

$$T(n) = \text{nombre de moviments que fa } \text{hanoi}(n, 1, 2, 3).$$

Llavors,

$$T(0) = 0$$

$$T(n) = 2T(n-1) + 1, \quad \text{si } n > 0.$$

Recurrència

$$T(0) = 0$$

$$T(n) = 2T(n-1) + 1, \quad \text{si } n > 0.$$

Solució asimptòtica

Aplicant el teorema mestre de recurrències subtractives, obtenim $T \in \Theta(2^n)$.

Solució exacta

Definim $S(n) = T(n) + 1$ i l'escrivим sense dependre de $T(n)$:

$$S(0) = 1$$

$$S(n) = 2T(n-1) + 2 = 2(S(n-1) - 1) + 2 = 2S(n-1), \quad \text{si } n > 0.$$

Ara, $S(n)$ es resol directament i dóna: $S(n) = 2^n$ per a tot $n \geq 0$.
Per tant,

$$T(n) = 2^n - 1 \text{ per a tot } n \geq 0.$$

Recurrència

$$T(0) = 0$$

$$T(n) = 2T(n-1) + 1, \quad \text{si } n > 0.$$

Solució asimptòtica

Aplicant el teorema mestre de recurrències subtractives, obtenim $T \in \Theta(2^n)$.

Solució exacta

Definim $S(n) = T(n) + 1$ i l'escrivim sense dependre de $T(n)$:

$$S(0) = 1$$

$$S(n) = 2T(n-1) + 2 = 2(S(n-1) - 1) + 2 = 2S(n-1), \quad \text{si } n > 0.$$

Ara, $S(n)$ es resol directament i dóna: $S(n) = 2^n$ per a tot $n \geq 0$.
Per tant,

$$T(n) = 2^n - 1 \text{ per a tot } n \geq 0.$$

Recurrència

$$T(0) = 0$$

$$T(n) = 2T(n - 1) + 1, \quad \text{si } n > 0.$$

Solució asimptòtica

Aplicant el teorema mestre de recurrències subtractives, obtenim $T \in \Theta(2^n)$.

Solució exacta

Definim $S(n) = T(n) + 1$ i l'escrivим sense dependre de $T(n)$:

$$S(0) = 1$$

$$S(n) = 2T(n - 1) + 2 = 2(S(n - 1) - 1) + 2 = 2S(n - 1), \quad \text{si } n > 0.$$

Ara, $S(n)$ es resol directament i dóna: $S(n) = 2^n$ per a tot $n \geq 0$.

Per tant,

$$T(n) = 2^n - 1 \text{ per a tot } n \geq 0.$$

Definició

La **mediana** d'una llista de nombres és l'element per al qual n'hi tants de més petits com de més grans.

Per exemple, la mediana de (15, 3, 34, 5, 10) és 10 perquè quan s'escriuen en ordre, és el que queda al mig:

3 5 10 15 34

Si la llista té llargària parell, hi ha dues opcions. Llavors prenem, per exemple, el més petit.

Característiques de la mediana:

- Sempre és **un dels valors** del conjunt de dades
- És **menys sensible a les observacions atípiques (outliers)**. Per exemple:
 - 1 Donats els nombres 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 100
(10 vegades 1 i una vegada 100),
 - la mitjana és 10
 - la mediana és 1
 - 2 Donats 2, 4, 6, 8, 10.000,
 - la mitjana és 1002
 - la mediana és 6

Mediana

Per **calcular** la mediana, n'hi ha prou a ordenar els elements.

El problema és que ordenar pren temps $\Theta(n \log n)$

[8, 0, 3, 10, 5, 7, 12, 3, 7, 2, 9, 1, 6]



[0, 1, 2, 3, 3, 5, 6, 7, 8, 9, 10, 12]

Però es fa més feina de la necessària:

només volem l'element del mig i no caldria ordenar la resta

{8, 0, 3, 10, 5, 7}, 6, {12, 3, 7, 2, 9, 1}

Mediana

Per **calcular** la mediana, n'hi ha prou a ordenar els elements.

El problema és que ordenar pren temps $\Theta(n \log n)$

[8, 0, 3, 10, 5, 7, 12, 3, 7, 2, 9, 1, 6]



[0, 1, 2, 3, 3, 5, 6, 7, 8, 9, 10, 12]

Però es fa més feina de la necessària:

només volem l'element del mig i no caldria ordenar la resta

{8, 0, 3, 10, 5, 7}, 6, {12, 3, 7, 2, 9, 1}

Mediana

Sovint és més fàcil treballar amb una versió més general del problema.
En aquest cas, considerem el problema de selecció.

Definició

Si S és un vector i $k \geq 1$, anomenem

$\text{selecció}(S, k)$

al k -èsim element més petit de S .

Problema de selecció

Donat un vector S i un natural k , determinar $\text{selecció}(S, k)$.

La mediana d'un vector de n nombres és
la solució del problema de selecció per a $k = \lfloor (n + 1)/2 \rfloor$.

Sovint és més fàcil treballar amb una versió més general del problema.
En aquest cas, considerem **el problema de selecció**.

Definició

Si S és un vector i $k \geq 1$, anomenem

$$\text{selecció}(S, k)$$

al k -èsim element més petit de S .

Problema de selecció

Donat un vector S i un natural k , determinar $\text{selecció}(S, k)$.

La mediana d'un vector de n nombres és
la solució del problema de selecció per a $k = \lfloor (n + 1)/2 \rfloor$.

Problema de selecció

Donat un vector S i un natural k , determinar selecció(S, k).

Idea per a un algorisme

Prenem un nombre x i partim el vector en 2 parts:

- elements més petits o iguals que x
- elements més grans o iguals que x

Si tenim el vector

$$S : [\begin{array}{cccccccccccc} 2 & 36 & 5 & 21 & 8 & 13 & 11 & 20 & 5 & 4 & 1 \end{array}]$$

per a $x = 5$ el podem dividir en

$$S_E : [\begin{array}{cccc} 2 & 4 & 1 & 5 \end{array}] \quad S_D : [\begin{array}{cccccccccc} 5 & 36 & 21 & 8 & 13 & 11 & 20 \end{array}]$$

Mediana

Idea per a un algorisme

Suposem ara que volem el 9è element de S

$$S : [\ 2 \ 36 \ 5 \ 21 \ 8 \ 13 \ 11 \ 20 \ 5 \ 4 \ 1 \]$$

Sabem que serà el 5è element de S_D perquè $|S_E| = 4$.

$$S_E : [\ 2 \ 4 \ 1 \ 5 \] \quad S_D : [\ 5 \ 36 \ 21 \ 8 \ 13 \ 11 \ 20 \]$$

Podem definir l'operador selecció(S, k) de manera recursiva:

$$\text{selecció}(S, k) = \begin{cases} \text{selecció}(S_E, k), & \text{si } k \leq |S_E| \\ \text{selecció}(S_D, k - |S_E|), & \text{si } k > |S_E| \end{cases}$$

Mediana

Idea per a un algorisme

Suposem ara que volem el 9è element de S

$$S : [\ 2 \ 36 \ 5 \ 21 \ 8 \ 13 \ 11 \ 20 \ 5 \ 4 \ 1 \]$$

Sabem que serà el 5è element de S_D perquè $|S_E| = 4$.

$$S_E : [\ 2 \ 4 \ 1 \ 5 \] \quad S_D : [\ 5 \ 36 \ 21 \ 8 \ 13 \ 11 \ 20 \]$$

Podem definir l'operador selecció(S, k) de manera recursiva:

$$\text{selecció}(S, k) = \begin{cases} \text{selecció}(S_E, k), & \text{si } k \leq |S_E| \\ \text{selecció}(S_D, k - |S_E|), & \text{si } k > |S_E| \end{cases}$$

Algorisme QuickSelect [Hoare, 1962]

$$\text{selecció}(S, k) = \begin{cases} \text{selecció}(S_E, k), & \text{si } k \leq |S_E| \\ \text{selecció}(S_D, k - |S_E|), & \text{si } k > |S_E| \end{cases}$$

Passos:

- 1 Triar un element x de S i partir S en S_E i S_D (partition del QuickSort)
- 2 Calcular selecció(S, k) recursivament

- Variants: partir S en elements més petits, iguals, i més grans que x
- En el **cas pitjor** el cost de QuickSelect és $\Theta(n^2)$.
- En el **cas mig** el cost de QuickSelect és $\Theta(n)$.

- Amb una estratègia adequada per escollir el pivot, es pot aconseguir que el cost en el cas pitjor sigui $\Theta(n)$
- Usarem el mateix algorisme de selecció per a escollir el pivot!

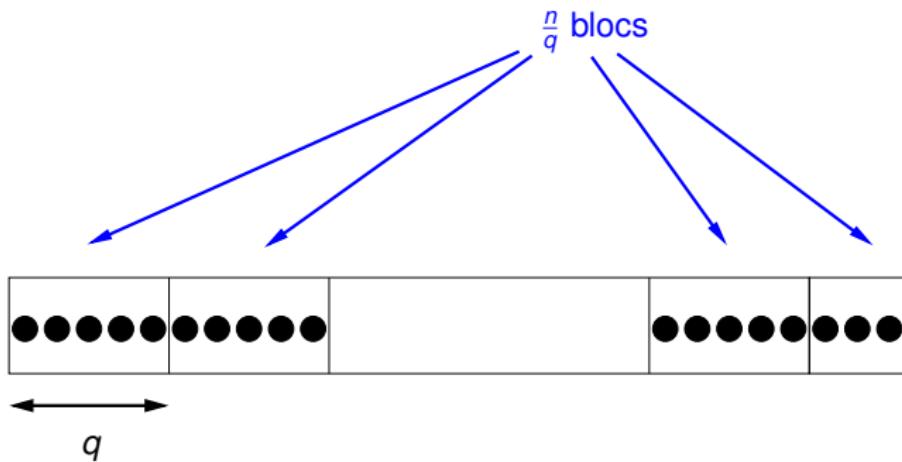
- 1 Per a un cert q , dividim el vector $A[l..u]$ de mida $n = u - l + 1$ en blocs de mida q (excepte potser l'últim)
- 2 Per cada bloc calculem la mediana de q elements (de qualsevol forma)
- 3 L'algorisme de selecció s'aplica recursivament per trobar la mediana de les $\lceil \frac{n}{q} \rceil$ medianes obtingudes en el pas anterior.
- 4 La mediana de les medianes (la **pseudo mediana**) s'usa com a pivot per partir el vector original, i es crida recursivament sobre el subvector esquerre o dret que pertoqui.

- 1 Per a un cert q , dividim el vector $A[l..u]$ de mida $n = u - l + 1$ en blocs de mida q (excepte potser l'últim)
- 2 Per cada bloc calculem la mediana de q elements (de qualsevol forma)
- 3 L'algorisme de selecció s'aplica recursivament per trobar la mediana de les $\lceil \frac{n}{q} \rceil$ medianes obtingudes en el pas anterior.
- 4 La mediana de les medianes (la **pseudo mediana**) s'usa com a pivot per partir el vector original, i es crida recursivament sobre el subvector esquerre o dret que pertoqui.

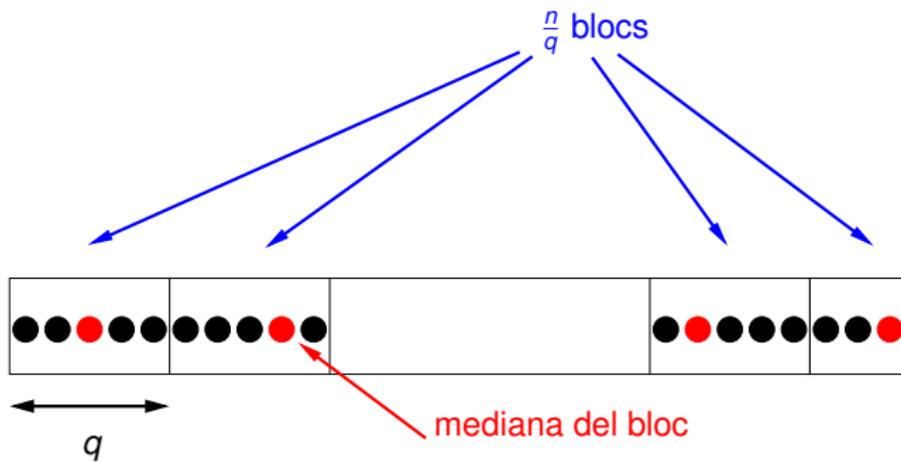
- 1 Per a un cert q , dividim el vector $A[l..u]$ de mida $n = u - l + 1$ en blocs de mida q (excepte potser l'últim)
- 2 Per cada bloc calculem la mediana de q elements (de qualsevol forma)
- 3 L'algorisme de selecció s'aplica recursivament per trobar la mediana de les $\lceil \frac{n}{q} \rceil$ medianes obtingudes en el pas anterior.
- 4 La mediana de les medianes (la **pseudo mediana**) s'usa com a pivot per partir el vector original, i es crida recursivament sobre el subvector esquerre o dret que pertoqui.

- 1 Per a un cert q , dividim el vector $A[l..u]$ de mida $n = u - l + 1$ en blocs de mida q (excepte potser l'últim)
- 2 Per cada bloc calculem la mediana de q elements (de qualsevol forma)
- 3 L'algorisme de selecció s'aplica recursivament per trobar la mediana de les $\lceil \frac{n}{q} \rceil$ medianes obtingudes en el pas anterior.
- 4 La mediana de les medianes (la **pseudo mediana**) s'usa com a pivot per partir el vector original, i es crida recursivament sobre el subvector esquerre o dret que pertoqui.

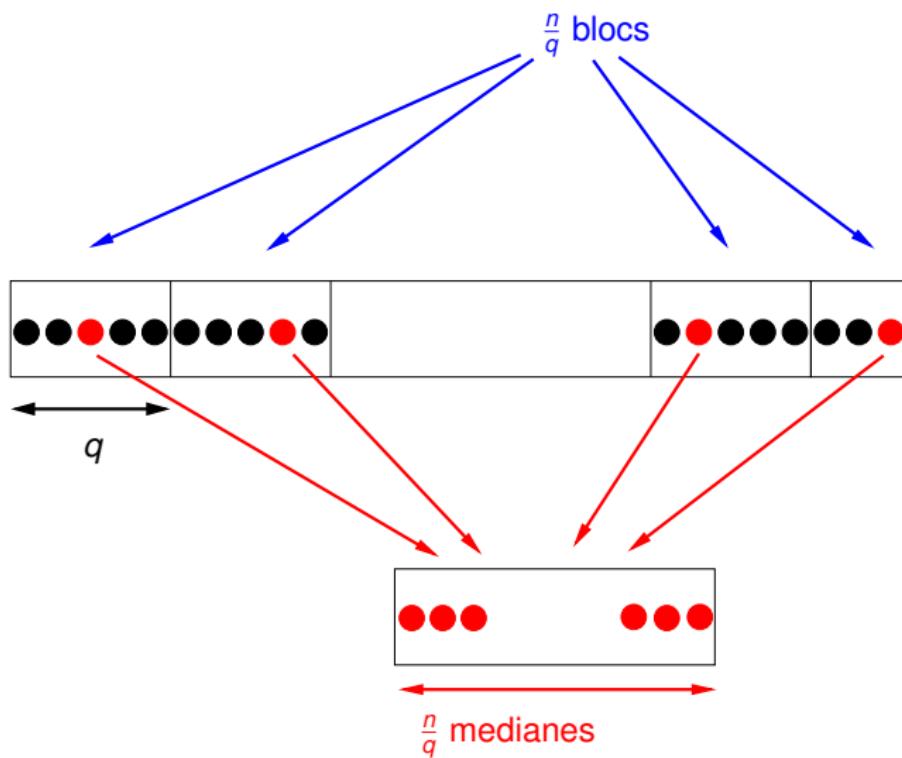
Mediana



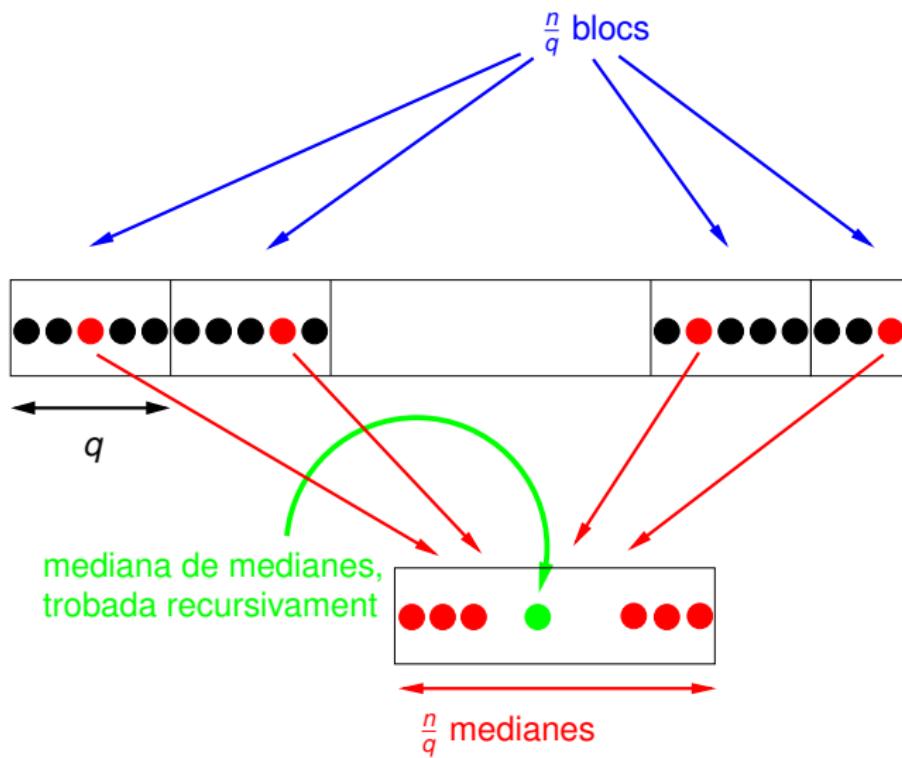
Mediana



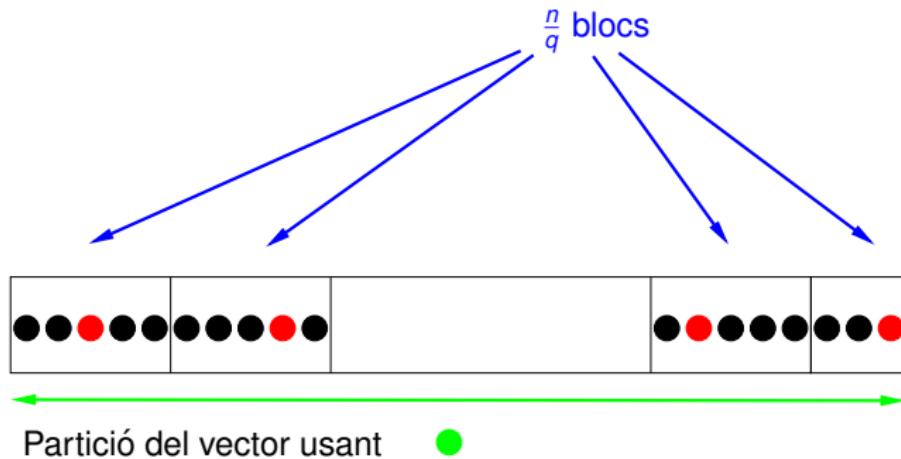
Mediana



Mediana



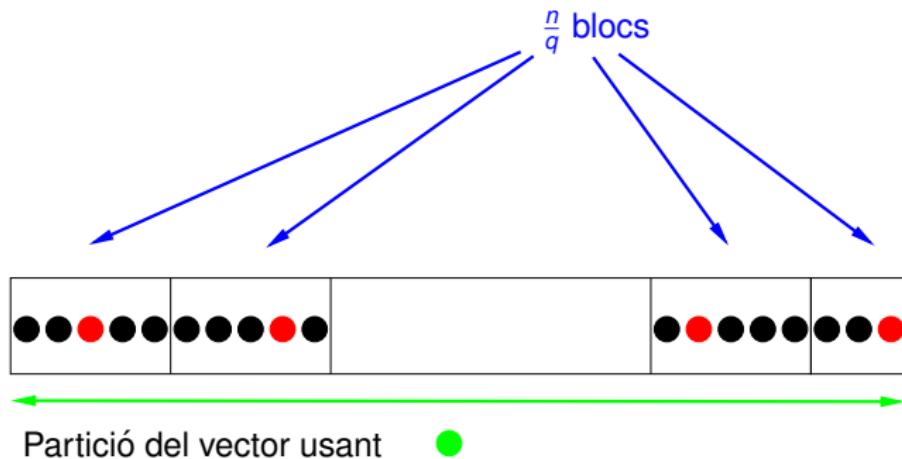
Mediana



$$\textcolor{red}{\bullet} \geq \frac{n}{2q} \textcolor{red}{\bullet}$$

$$\textcolor{red}{\bullet} \geq \frac{q}{2} \textcolor{black}{\bullet}$$

Mediana



$$\left. \begin{array}{l} \textcolor{red}{\bullet} \geq \frac{n}{2q} \quad \textcolor{red}{\bullet} \\ \textcolor{red}{\bullet} \geq \frac{q}{2} \quad \bullet \end{array} \right\} \quad \textcolor{red}{\bullet} \geq \frac{n}{4} \quad \bullet$$

- Anem a calcular $C(n)$, el cost en el cas pitjor de l'algorisme
- Trobar la mediana de cada bloc costa $\Theta(1)$, perquè q és una constant
- En total trobar les medianes dels blocs costa $\Theta(n)$, perquè hi ha $\frac{n}{q}$ blocs
- El cost de la crida recursiva per trobar la pseudo mediana és $C(\frac{n}{q})$
- Tenim la garantia que, usant la pseudo mediana com a pivot x :
 - almenys $\frac{n}{4}$ elements són $\leq x$, i
 - almenys $\frac{n}{4}$ elements són $\geq x$
- El subvector més petit de la partició tindrà mida com a mínim $\frac{n}{4}$
- El subvector més gran de la partició tindrà mida com a màxim $\frac{3n}{4}$
- La crida recursiva es fa sobre un vector de mida com a molt $\frac{3n}{4}$
- Per tant

$$C(n) = C\left(\frac{n}{q}\right) + C\left(\frac{3n}{4}\right) + \Theta(n)$$

- Es pot demostrar que la solució és $\Theta(n)$ si $\frac{1}{q} + \frac{3}{4} < 1$ (per ex., si $q = 5$)

Mediana

- Anem a calcular $C(n)$, el cost en el cas pitjor de l'algorisme
- Trobar la mediana de cada bloc costa $\Theta(1)$, perquè q és una constant
- En total trobar les medianes dels blocs costa $\Theta(n)$, perquè hi ha $\frac{n}{q}$ blocs
- El cost de la crida recursiva per trobar la pseudo mediana és $C(\frac{n}{q})$
- Tenim la garantia que, usant la pseudo mediana com a pivot x :
 - almenys $\frac{n}{4}$ elements són $\leq x$, i
 - almenys $\frac{n}{4}$ elements són $\geq x$
- El subvector més petit de la partició tindrà mida com a mínim $\frac{n}{4}$
- El subvector més gran de la partició tindrà mida com a màxim $\frac{3n}{4}$
- La crida recursiva es fa sobre un vector de mida com a molt $\frac{3n}{4}$
- Per tant

$$C(n) = C\left(\frac{n}{q}\right) + C\left(\frac{3n}{4}\right) + \Theta(n)$$

- Es pot demostrar que la solució és $\Theta(n)$ si $\frac{1}{q} + \frac{3}{4} < 1$ (per ex., si $q = 5$)

Mediana

- Anem a calcular $C(n)$, el cost en el cas pitjor de l'algorisme
- Trobar la mediana de cada bloc costa $\Theta(1)$, perquè q és una constant
- En total trobar les medianes dels blocs costa $\Theta(n)$, perquè hi ha $\frac{n}{q}$ blocs
- El cost de la crida recursiva per trobar la pseudo mediana és $C(\frac{n}{q})$
- Tenim la garantia que, usant la pseudo mediana com a pivot x :
 - almenys $\frac{n}{4}$ elements són $\leq x$, i
 - almenys $\frac{n}{4}$ elements són $\geq x$
- El subvector més petit de la partició tindrà mida com a mínim $\frac{n}{4}$
- El subvector més gran de la partició tindrà mida com a màxim $\frac{3n}{4}$
- La crida recursiva es fa sobre un vector de mida com a molt $\frac{3n}{4}$
- Per tant

$$C(n) = C\left(\frac{n}{q}\right) + C\left(\frac{3n}{4}\right) + \Theta(n)$$

- Es pot demostrar que la solució és $\Theta(n)$ si $\frac{1}{q} + \frac{3}{4} < 1$ (per ex., si $q = 5$)

Mediana

- Anem a calcular $C(n)$, el cost en el cas pitjor de l'algorisme
- Trobar la mediana de cada bloc costa $\Theta(1)$, perquè q és una constant
- En total trobar les medianes dels blocs costa $\Theta(n)$, perquè hi ha $\frac{n}{q}$ blocs
- El cost de la crida recursiva per trobar la pseudo mediana és $C(\frac{n}{q})$
- Tenim la garantia que, usant la pseudo mediana com a pivot x :
 - almenys $\frac{n}{4}$ elements són $\leq x$, i
 - almenys $\frac{n}{4}$ elements són $\geq x$
- El subvector més petit de la partició tindrà mida com a mínim $\frac{n}{4}$
- El subvector més gran de la partició tindrà mida com a màxim $\frac{3n}{4}$
- La crida recursiva es fa sobre un vector de mida com a molt $\frac{3n}{4}$
- Per tant

$$C(n) = C\left(\frac{n}{q}\right) + C\left(\frac{3n}{4}\right) + \Theta(n)$$

- Es pot demostrar que la solució és $\Theta(n)$ si $\frac{1}{q} + \frac{3}{4} < 1$ (per ex., si $q = 5$)

- Anem a calcular $C(n)$, el cost en el cas pitjor de l'algorisme
- Trobar la mediana de cada bloc costa $\Theta(1)$, perquè q és una constant
- En total trobar les medianes dels blocs costa $\Theta(n)$, perquè hi ha $\frac{n}{q}$ blocs
- El cost de la crida recursiva per trobar la pseudo mediana és $C(\frac{n}{q})$
- Tenim la garantia que, usant la pseudo mediana com a pivot x :
 - almenys $\frac{n}{4}$ elements són $\leq x$, i
 - almenys $\frac{n}{4}$ elements són $\geq x$
- El subvector més petit de la partició tindrà mida com a mínim $\frac{n}{4}$
- El subvector més gran de la partició tindrà mida com a màxim $\frac{3n}{4}$
- La crida recursiva es fa sobre un vector de mida com a molt $\frac{3n}{4}$
- Per tant

$$C(n) = C\left(\frac{n}{q}\right) + C\left(\frac{3n}{4}\right) + \Theta(n)$$

- Es pot demostrar que la solució és $\Theta(n)$ si $\frac{1}{q} + \frac{3}{4} < 1$ (per ex., si $q = 5$)