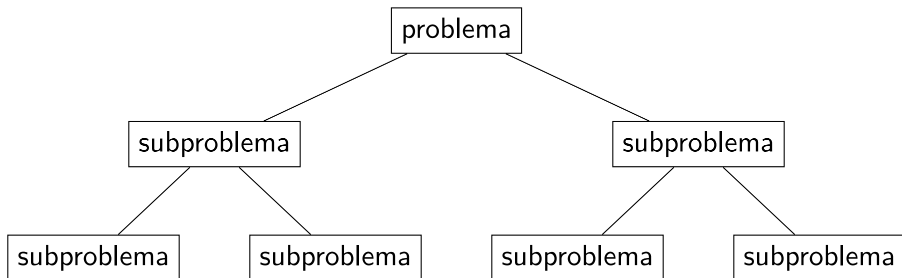
- L'esquema algorísmic de dividir i vèncer
- Mergesort
- Quicksort
- Problemes

- Dividir i vèncer és una técnica general de resolució de problemes mitjançant la seva divisió en subproblemes més petits i més fàcils de resoldre, la resolució recursiva d'aquests subproblemes més petits, i la combinació de les seves solucions en una solució del problema original.
- Els subproblemes de talla suficientment petita es resolen de manera directa.
- La talla dels subproblemes ha de ser una fracció de la talla del problema perquè l'algorisme de dividir i vèncer sigui més eficient, com es desprèn de la resolució de les equacions de recurrència que descriuen el cost temporal i espacial dels algorismes de dividir i vèncer.
  - La divisió d'un problema en subproblemes d'aproximadament la mateixa talla que el problema original dóna algorismes menys eficients.
- La tècnica de dividir i vèncer es pot aplicar a aquells problemes que mostren el *principi d'independència*, segons el qual una instància del problema es pot dividir en una sèrie d'instàncies més petites i independents entre sí.

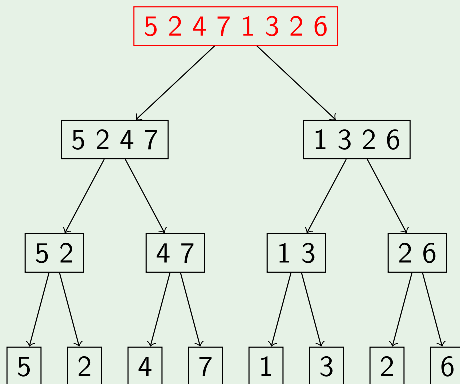
- L'esquema algorísmic de dividir i vèncer es basa en els tres passos següents a cada nivell de la recursió:
  - Dividir El problema es divideix en una sèrie de subproblemes.
  - Vèncer Els subproblemes es resolen recursivament. Els subproblemes de talla prou petita, però, es resolen de manera directa.
  - Combinar Les solucions dels subproblemes es combinen en una solució del problema original.



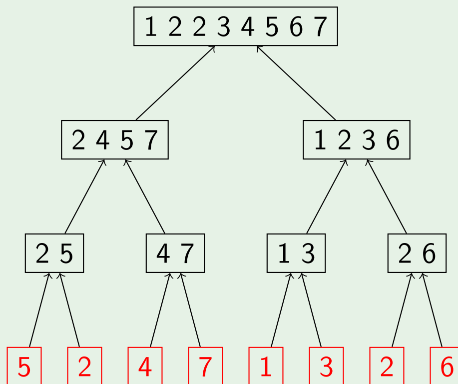
- Mergesort és un algorisme d'ordenació de cost temporal $\Theta(n \log n)$ en el cas pitjor, sobre una llista de n elements.
- Té l'avantatge d'ordenar sobre la mateixa llista, sense necessitat de cap llista auxiliar.
- La idea general de l'algorisme mergesort és la següent.

–
function mergesort (L : llista)
  triar la posició mitjana m de L
  particionar L en els m primers elements L_1 i els restants $n - m$ elements L_2
  return fusió de mergesort(L_1) i mergesort(L_2)
end function

Exemple (Ordenació de la llista $\langle 5, 2, 4, 7, 1, 3, 2, 6 \rangle$ amb mergesort.)



Exemple (Ordenació de la llista $\langle 5, 2, 4, 7, 1, 3, 2, 6 \rangle$ amb mergesort.)



- L'algorisme mergesort es basa en el paradigma dividir i vèncer.
- Amb aquest algorisme, es garanteix que la talla dels subproblemes serà una fracció (la meitat) de la talla del problema original.
- Dividir La llista $L[\ell..u]$ es particiona en dues subllistes no buides $L[\ell..m]$ i $L[m+1..u]$, on $m = \lfloor (u - \ell + 1)/2 \rfloor$.
- Vèncer Les dues subllistes $L[\ell..m]$ i $L[m+1..u]$ s'ordenen mitjançant crides recursives a mergesort.
- Combinar Les dues subllistes ordenades es fusionen per combinar-les.

– Tota la llista $L[\ell..u]$ és ara ordenada.

- El procediment següent implementa l'algorisme mergesort.
  - **procedure** mergesort (L : llista; n : enter)
    if $n > 1$ **then**
    enter $m := n$ **div** 2
    llista L'
    split (L, L', m)
    mergesort (L, m)
    mergesort ($L', n - m$)
    merge (L, L')
    end if
    end procedure

- Donada una llista $L = [x_1, \ldots, x_n]$ i un enter m amb $1 \leqslant m \leqslant n$, split (L, L', m) divideix L en dues llistes $L = [x_1, \ldots, x_m]$ i $L' = [x_{m+1}, \ldots, x_n]$.

```
procedure split (L, L' : llista; m : enter)
   pointer to node p := L
   for enter i := 1 to m − 1 do
      p := p → next
   end for
   L' := p → next
   p → next := null
end procedure
```

- Donades dues llistes L i L', merge (L, L') fusiona els elements de L i L' en L, tot buidant L'.

  - **procedure** merge $(L, L' :$ llista$)$
    if $L =$ **null then**
    $L := L'$
    $L' :=$ **null**
    end if
    if $L \neq$ **null and** $L' \neq$ **null then**
    if $L \rightarrow$ info $\leqslant L' \rightarrow$ info **then**
    merge $(L \rightarrow$ next$, L')$
    else
    merge $(L' \rightarrow$ next$, L)$
    $L := L'$
    $L' :=$ **null**
    end if
    end if
    end procedure

- El cost de l'algorisme mergesort és descrit per l'equació de recurrència

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 2\,T(n/2) + \Theta(n) & n > 1 \end{cases}$$

  amb solució $T(n) = \Theta(n \log n)$.

– El cost temporal de *split* sobre una subllista $L[\ell, u]$ és $\Theta(u - \ell + 1) = \Theta(n)$.

– El cost temporal de *merge* sobre dues subllistes $L[\ell, m]$ i $L[m + 1..u]$ també és $\Theta(u - \ell + 1) = \Theta(n)$.

- L'ordenació per fusió (mergesort) d'avall cap amunt (no recursiva) consisteix a dividir la llista en blocs de $m = 2^k$ elements i fusionar cada parell de blocs consecutius, amb $0 \leqslant k < \lceil \log n \rceil$.
- El cost de l'algorisme mergesort d'avall cap amunt també és descrit per l'equació de recurrència $T(n) = 2\,T(n/2) + \Theta(n)$, amb solució $T(n) = \Theta(n \log n)$.

Exemple (Ordenació de la llista $\langle 5, 2, 4, 7, 1, 3, 2, 6 \rangle$ amb mergesort d'avall cap amunt.)

$m = 1$	5	2	4	7	1	3	2	6
$m = 2$	2	5	4	7	1	3	2	6
$m = 4$	2	4	5	7	1	2	3	6
$m = 8$	1	2	2	3	4	5	6	7

- Els blocs consecutius són $L[i..m+i-1]$ i $L[m+i..2m+i-1]$.
  - $L[1..m]$ i $L[m+1..2m]$.
  - $L[2m+1..3m]$ i $L[3m+1..4m]$.
- El procediment següent implementa l'algorisme mergesort d'avall cap amunt.
  -

```
procedure mergesort (L : llista; n : enter)
  m := 1
  while m ⩽ n do
    i := 1
    while i ⩽ n − m do
      merge L[i..m + i − 1] i L[m + i..2 ∗ m + i − 1]
      i := i + 2 ∗ m
    end while
    m := 2 ∗ m
  end while
end procedure
```

- Quicksort és un algorisme d'ordenació de cost temporal $\Theta(n^2)$ en el cas pitjor, sobre un vector de n elements, però és un dels algorismes d'ordenació més usats a la pràctica degut a la seva eficiència en el cas mitjà.
- De fet, el seu cost temporal esperat és $\Theta(n \log n)$, i els factors constants amagats rere la notació asimptòtica són molt petits.
- Té també l'avantatge d'ordenar sobre el mateix vector, sense necessitat de cap vector auxiliar.
- La idea general de l'algorisme quicksort és la següent.

– **function** quicksort (S : conjunt)
  triar un element pivot p de S
  particionar S en $S_1 = \{x \in S \mid x \leqslant p\}$ i $S_2 = \{x \in S \mid x \geqslant p\}$,
  tot repartint-hi els elements iguals al pivot
  return quicksort(S_1) seguit de quicksort(S_2)
  end function

- L'algorisme quicksort es basa en el paradigma dividir i vèncer.
- A diferència d'altres algorismes dins aquest paradigma, com ara mergesort, però, no es garanteix que la talla dels subproblemes serà una fracció de la talla del problema original.
- Dividir El vector $A[\ell..u]$ es particiona (reorganitza) en dos subvectors no buits $A[\ell..p]$ i $A[p+1..u]$ tals que tot element de $A[\ell..p]$ és més petit o igual que tot element de $A[p+1..u]$.
- La posició p es calcula com a subproducte del procediment de particionat.
- Vèncer Els dos subvectors $A[\ell..p]$ i $A[p+1..u]$ s'ordenen mitjançant crides recursives a quicksort.
- Combinar Atès que els dos subvectors s'ordenen sobre el mateix vector, no cal cap esforç addicional per combinar-los.
- Tot el vector $A[\ell..u]$ és ara ordenat.

- Mentre que en l'algorisme mergesort la fase de divisió és molt senzilla i l'esforç es realitza durant la fase de combinació, en l'algorisme quicksort tot l'esforç es realitza durant la fase de divisió i aleshores la fase de combinació resulta trivial.
- El procediment següent implementa l'algorisme quicksort.

– **procedure** quicksort (**e/s** A : **array**$[1..n]; \ell, u : [1..n]$)
  if $\ell < u$ **then**
    partition(A, ℓ, u, p)
    quicksort(A, ℓ, p)
    quicksort$(A, p + 1, u)$
  end if
end procedure

- La crida inicial per ordenar un vector A de n elements és quicksort$(A, 1, n)$.

- La divisió del problema d'ordenació en subproblemes es realitza mentre $\ell < u$, és a dir, mentre $u - \ell + 1 > 1$.
- Òbviament, l'ordenació d'un vector de pocs elements es pot realitzar mitjançant qualque algorisme d'ordenació més senzill. El llindar òptim m de tall de la recursió oscil·la entre 20 i 25.

– **procedure** quicksort (**e/s** A : **array**$[1..n]; \ell, u : [1..n]$)
  if $u - \ell + 1 < m$ **then**
    sort(A, ℓ, u) {amb un algorisme d'ordenació més senzill}
  else
    partition(A, ℓ, u, p) $\{\ell \leqslant p < u$ i $A[\ell..p] \leqslant A[p+1..u]\}$
    quicksort(A, ℓ, p)
    quicksort$(A, p+1, u)$
  end if
  end procedure

- La clau de l'algorisme quicksort és el procediment *partition*, que particiona el subvector $A[\ell..u]$ i reorganitza els elements sobre el mateix vector.
- De fet, aquest procediment situa un element p, anomenat pivot, en la seva posició final del vector ordenat.
- El procediment següent calcula una posició de pivot p amb $\ell \leqslant p < u$ i divideix el vector A en $A[\ell..p]$ i $A[p+1..u]$ de manera tal que $A[\ell..p] \leqslant A[p+1..u]$.
- Es pren com a pivot p el primer element $A[\ell]$ de la part del vector $A[\ell..u]$ que està essent ordenada.

–

```
procedure partition (e/s A : array[1..n]; ℓ, u, p : [1..n])
   p := ℓ
   for i := ℓ + 1 to u do
      if A[i] < A[ℓ] then
         p := p + 1
         A[p] ↔ A[i]
      end if
   end for
   A[ℓ] ↔ A[p]
end procedure
```

- Si hi ha elements repetits, cal repartir els elements iguals que el pivot entre els dos subvectors.
- El cost temporal de *partition* sobre un subvector $A[\ell, u]$ és $\Theta(u - \ell + 1) = \Theta(n)$.

Els problemes marcats amb una estrelleta ($\star$) són més difícils. Si no es diu el contrari i us cal, considereu que tots els logaritmes són en base 2.

- 2.1
- 2.2
- 2.3
- 2.4
- 2.7
- 2.8
- 2.17
- 2.18
- 2.22 (a)
- 2.27

Problema (2.1)

Escriviu un algorisme de cost $\Theta(\log n)$ *que, donada una taula ordenada* T *amb* n *elements i un element* x*, retorni* -1 *si* x *no és en* T *o un índex* i *tal que* $T[i] = x$ *altrament. Si heu escrit el vostre algorisme recursivament, passeu-lo a iteratiu.*

Problema (2.2)

Si no heu aconseguit fer el problema anterior, potser n'haureu buscat una solució en alguna web o en algun llibre de text. Digueu què en penseu dels tres fragments de codi següents (el fragment 1 es troba en una web de programadors, el fragment 2 és una adaptació del codi del llibre Modern software development using Java de Tymann i Schneider i el fragment 3 és una adaptació del codi del llibre Developing Java software de Winder i Roberts):

Problema (2.2)

1.
```
int lookup1 (vector<int>& T, int x) {
  int l = 0;
  int r = T.size() - 1;
  while (l <= r) {
    int m = (l+r) / 2;
    if (x < T[m]) r = m;
    else if (x > T[m]) l = m + 1;
    else return m;
  }
  return -1;
}
```

Problema (2.2)

2.

```
int lookup2 (vector<int>& array, int target) {
  int start = 0;
  int end = array.size();
  int position = -1;
  while (start<=end and position==-1) {
    int middle = (start+end)/2;
    if (target<array[middle]) end = middle - 1;
    else if (target>array[middle]) start = middle + 1;
    else position = middle;
  }
  return position;
}
```

Problema (2.2)

3.
```
int lookup3 (vector<int>& v, int o) {
  int hi = v.size();
  int lo = 0;
  while (true) {
    int centre = (hi+lo)/2;
    if (centre==lo) {
      if (v[centre]==o) return centre;
      else if (v[centre+1]==o) return centre+1;
      else return -1;
    }
    if (v[centre]<o) lo = centre;
    else if (o<v[centre]) hi = centre;
    else return centre;
} }
```

Problema (2.3)

Escriviu un algorisme de cost $\Theta(\log n)$ *que, donada una taula ordenada* T *amb* n *elements i un element* x*, retorni* -1 *si* x *no és en* T *o l'índex de la primera posició de* T *que conté* x*.*

Problema (2.4)

Escriviu un algorisme de cost $\Theta(\log n)$ *que, donada una taula ordenada* T *amb* n *elements i dos elements* x *i* y *amb* $x \leqslant y$*, retorni el nombre d'elements en* T *que es troben entre* x *i* y *(*x *i* y *inclosos).*

Problema (2.7)

Ordeneu la taula $\langle 3, 8, 15, 7, 12, 6, 5, 4, 3, 7, 1 \rangle$ *amb l'algorisme d'ordenació per fusió recursiu i amb l'algorisme d'ordenació per fusió d'avall cap amunt.*

Problema (2.8)

És l'algorisme d'ordenació per fusió estable? De què depèn?

Problema (2.17)

Sigui $T[i..j]$ una taula amb $n = j - i + 1$ elements. Considereu l'algorisme d'ordenació anomenat l'ordenació boja d'en Quim:

- *Si $n \leqslant 2$, la taula s'ordena trivialment.*
- *Si $n \geqslant 3$, "dividim" la taula en tres intervals $T[i..k-1]$, $T[k..\ell]$ i $T[\ell+1..j]$, on $k = i + \lfloor n/3 \rfloor$, $\ell = j - \lfloor n/3 \rfloor$. L'algorisme ordena recursivament $T[i..\ell]$, després ordena $T[k..j]$, i finalment ordena de nou $T[i..\ell]$.*

Demostreu que aquest algorisme ordena correctament. Doneu, raonadament, una recurrència per al temps que triga aquest algorisme i resoleu-la.

Problema (2.18)

Escriviu un algorisme que compti el nombre d'inversions d'una taula amb n *elements en temps* $O(n \log n)$ *en el cas pitjor.*

Problema (2.22)

Dissenyeu algorismes de cost $O(n \log n)$ que comptin el nombre d'elements dels conjunts definits en els apartats següents:

a. $\{i \in 1 \dots n \ : \ |\{j \ : \ j \neq i \wedge t[j] < t[i]\}| \ = \ i \ \}$

Problema (2.27)

Aquest problema estudia un algorisme de dividir i vèncer per multiplicar dos números de $n = 2^k$ bits. Recordeu que l'algorisme escolar utilitza $\Theta(n^2)$ passos.

a. *Siguin x i y dos números de n bits cadascun de forma que $x = a2^{n/2} + b$ i que $y = c2^{n/2} + d$. Comproveu que $xy = (a2^{n/2} + b)(c2^{n/2} + d)$.*

   Utilitzant aquesta idea, dissenyeu un algorisme de dividir i vèncer per reduir la multiplicació de dos nombres de n bits a quatre multiplicacions de nombres de $n/2$ bits que es combinen mitjançant addicions i desplaçaments.

   Analitzeu el cost de l'algorisme resultant.

Problema (2.27)

Aquest problema estudia un algorisme de dividir i vèncer per multiplicar dos números de $n = 2^k$ bits. Recordeu que l'algorisme escolar utilitza $\Theta(n^2)$ passos.

- b) *L'any 1962, Karatsuba i Ofman van descobrir una forma molt astuta per reduir les anteriors quatre multiplicacions de nombres de $n/2$ bits a només tres: Comproveu que*
  $xy = ((a+b)(c+d) - ac - bd)2^{n/2} + ac2^n + bd.$

  Utilitzant aquesta idea, dissenyeu un nou algorisme de dividir i vèncer i analitzeu-ne el cost.

- c) *Com es generalitzen els algorismes anteriors quan n no és una potència de 2?*