Fri 1/2/19

# Code 1

[ COMPUTATIONAL COST ]

$\boxed{2+5n}$ linear on n

```
$j=n$
$k=1$
while $j \geq 1$ do
    $k=k+1$
    $j=n/k$
fin_while
```

n times

# Code 2

$1 + 3\log_5(n)$

```
j=n
while $j \geq 1$ do
    j=j/5
fin_while
```

# Code 3

$$2 + n(5 + 5n) \approx 2 + 5n + 5n^2$$

```
j=n
k=1
while j>=1 do
    k=k+1
    j=n/k
    for l=1 to n
        a++
    fin_for
fin_while
```

2

5

n times

1 1 1

2

5 · n

$n(5+5n)$

# Code 4

```
j=n
k=1
while j>=1 do
    k=k+1
    j=n/k
    for l=1 to j
        a++
    fin_for
fin_while
```

1

2 + 2

3

2

$2+5j$

while (cond)
code
- - -
for ( )
code

$2 + t(code)$

$3 + t(code)$

$3 + t(code)$

- - -

$3 + t(no)$

$2 + \#(yes)\,[3 + t(code)]$

while condition

$$[1 + t(code)] \cdot \underbrace{\#(yes)}_{\text{(times that the condition is fulfilled)}} + \overset{\#(no)}{1}$$

do while = repeat until

$$1 + n(7+5j) = 1 + n\underbrace{\left(7 + 5\frac{n}{k}\right)} = 1 + n\left(7 + 5\boxed{n\log_e(n)}\right)$$

$Ln(n)$

K is a variable which moves in a consecutive way → SUMMATORY

$$\sum_{k=1}^{n} 5\frac{n}{k} = 5n\underbrace{\boxed{\sum_{k=1}^{n}\frac{1}{k}}} = 5n\log_e(n)$$

$$\int \frac{1}{k}\,dk = \log_e k$$

$$T_4(n) = 3 + \sum_{k=1}^{n}\left(7 + \sum_{\ell=1}^{j} 5\right) = 3 + \sum_{k=1}^{n}\left(7 + \sum_{\ell=1}^{n/k} 5\right) =$$

$$= 3 + \sum_{k=1}^{n}\left(7 + 5\frac{n}{k}\right) = 3 + \sum_{k=1}^{n} 7 + \sum_{k=1}^{n} 5\frac{n}{k} = 3 + 7n + 5n\boxed{\sum_{k=1}^{n}\frac{1}{k}}$$

$\log_e(n)$

---

## Arithmetic Sucesion

$a_0$ → first term

$a_1 = a_0 + d$

$\dots$

$a_n = a_0 + nd$

$$S_n = \frac{(2a_0 + nd)(n+1)}{2}$$

## Geometric Sucesion

$a_0$ → first term

$a_1 = r \cdot a_0$

$\dots$

$a_n = r^n \cdot a_0$

$$S_n = \frac{1 - r^{n+1}}{1 - r}\,a_0$$

# Code 5

Mon 4/2/19

```
Procedure LF (n)
if n<1 then s=1
else       x=1
           for i=2 to n do
                  x=x·i
           fin_for
           y=0
           repeat
                  y=y+1
                  x=x/4
           until x=0
           s=y
fin_if
```

n - 2 + 1 loops

$\boxed{n - 1}$

factorial de n

$\log_4 !n$

upper bound $n \cdot \log_4 n$

small distance

lower bound $n$

- a factorial should appear only in the base not as an argument
- very accurate aproximation

$$\boxed{\log_{\square} n < n^{\square}}$$

→ whatever almost 0 power

# Code 6

```
function Jul91(n:int)
while n>1 do
    x=1
    y=n
    while $n>x^2$ do
        x=x+1
        y=y-x
    fin_while
    n=n/2
fin_while
Jul91=x+y
```

$\sqrt{n}$ iterations

$\log_2 n$

## Selection structures

Code 5

$$T_5(n) = \begin{cases} 2 & n < 1 \\ \underset{\text{not important in the end}}{\underbrace{7}} + \sum_{i=2}^{n} 5 + \underset{x = \frac{x}{4}}{\underbrace{\sum_{x!}^{x=0}}} 5 = 7 + 5(n-1) + 5\log_4(n!) \leq & \\ \leq 7 + 5n - 5 + 5n\, lg(n) = 2 + 5n + 5n\, lg(n) \in O(n \cdot lg(n)) & o.w. \end{cases}$$

## Upper bound and lower bound

$$lg_4(n!) = lg_4(n \cdot (n-1) \cdots 2 \cdot 1) = lg_4(n) + \overset{lg_4(n)}{\cancel{lg_4(n-1)}} + \cdots + \underset{1}{lg_4(4)} +$$

$$+ \underset{0'7}{lg_4(3)} + \underset{1/2}{lg_4(2)} + \overset{0 \;\; lg_4(n)}{\cancel{lg_4(1)}} \implies \boxed{n \cdot lg_4\, n}$$

## Variable change when it is limitating the outter loops

$$\boxed{\begin{aligned} \sum_{x=1}^{limit} f(x) &= F(limit) \\ \int f(x)\, dx &= F(x) \end{aligned}}$$

Code 6

$$T_6(n) = \begin{cases} O(1) & n \leq 1 \\ \sum_{i=lg_2(n)}^{0} \left( 5 + \sum_{x=1}^{\sqrt{2^i}} 6 \right) = 5\log_2(n) + \sum_{i=lg_2(n)}^{0} 6\sqrt{2^i} = & \end{cases}$$

$$n = 2^i$$

$$= 5\, lg_2(n) + 6 \sum_{i=0}^{lg_2(n)} (\sqrt{2})^i = 5\, lg_2(n) + 6K(\sqrt{2})^{lg_2(n)} =$$

$$= 5\, lg_2(n) + 6K\sqrt{n} \in O(\sqrt{n})$$

$$(\sqrt{2})^{lg_2(n)} = (2^{1/2})^{lg_2(n)} = (2^{lg_2(n)})^{1/2} = n^{1/2} = \sqrt{n}$$

# Code 7

Which is the fastest?

O(n)

every type of array

```
proc busqueda_sec(S[1..n],x,pos)
  pos ← 1
  mientras pos ≤ n Y S[pos] < x
  hacer
    pos ← pos + 1
  fin mientras
fin proc
```

n -> array size

O(log2(n))

already ordered array

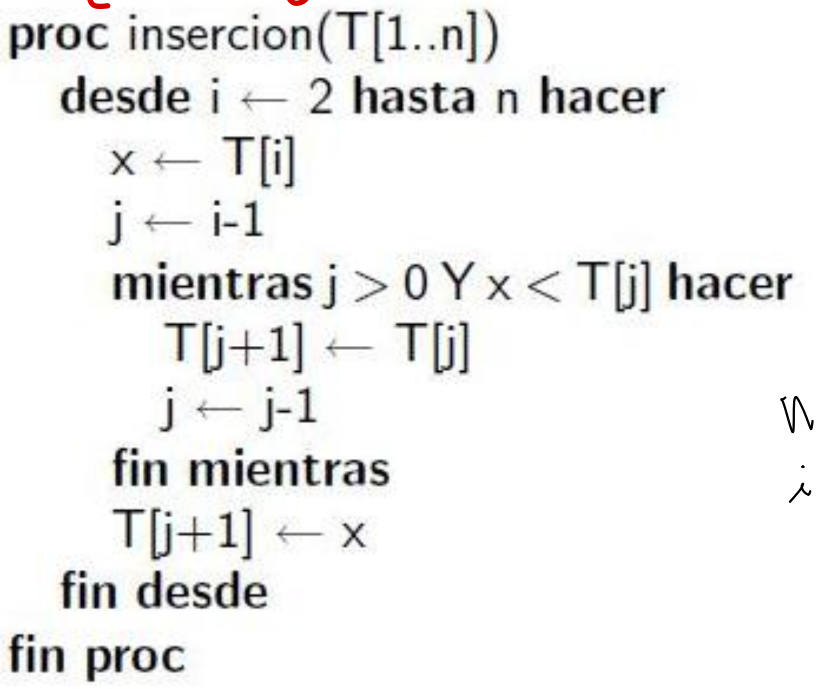
```
proc busqueda_bin(S[1..n],x,pos)
  inf ← 1
  sup ← n
  pos ← 0
  mientras inf ≤ sup Y pos = 0 hacer
    mitad ← ⌊ (inf+sup) / 2 ⌋
    si x = S[mitad] entonces
      pos ← mitad
    si no si x < S[mitad] entonces
      sup ← mitad - 1
    si no
      inf ← mitad + 1
    fin si
  fin mientras
fin proc
```

[Bipartition search]

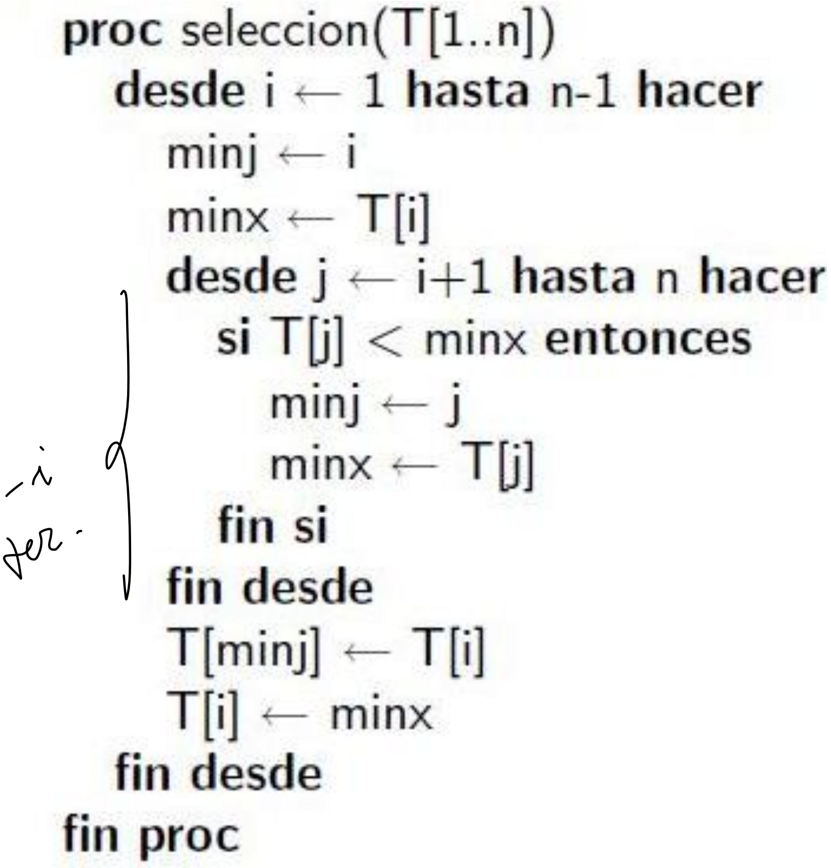
# Code 8

## Which is the fastest?

[ ordering an array ]



```
proc insercion(T[1..n])
   desde i ← 2 hasta n hacer
      x ← T[i]
      j ← i-1
      mientras j > 0 Y x < T[j] hacer
         T[j+1] ← T[j]
         j ← j-1
      fin mientras
      T[j+1] ← x
   fin desde
fin proc
```



```
proc seleccion(T[1..n])
   desde i ← 1 hasta n-1 hacer
      minj ← i
      minx ← T[i]
      desde j ← i+1 hasta n hacer
         si T[j] < minx entonces
            minj ← j
            minx ← T[j]
         fin si
      fin desde
      T[minj] ← T[i]
      T[i] ← minx
   fin desde
fin proc
```

n−i iter.

## Code 8

- left code

$$\sum_{i=2}^{n}\left(\boxed{\times}\ \sum_{j=i-1}^{0}\ \boxed{\phantom{xx}}\right) = kn^{2} \in O(n^{2})$$

$$k(i-1)$$

- right code

$$\sum_{i=1}^{n-1}\left(\text{“}8\text{”} + \text{“}5\text{”}\,(n-i)\right) = 2'5n^{2} \in O(n^{2})$$

# Case 1

```
procedure F(a,b,c : integer):integer
var x,y,i : integer        $P(x) \in O((a+b)^2)$
{   if a+b <20 then return a+b;
    else {        y=a+b
            x=0
            for i=1 to y do  { x=x+y }
            while y>1 do { P(x)              log3(a+b)
                                 y=y/3 }      recursive call
            for i=1 to 3 do { y=4·F(a-2,b,c) }
            return c*y+x
            }
}
```

the size of the problem will be (a) or (a+b)

$P(x) \in O((a+b)^2)$
$P(x) \in O(n^2)$

Size of the problem : $n = a+b$

$$T_{1,\,EX3}(n) = \begin{cases} 4 & n < 20 \\ 7 + \underset{\hookrightarrow \text{ line } 6}{5n} + \overset{\nearrow \sum_{n}^{1}}{\sum_{i=\lg_3(n)}^{0}} \Big(3 + \underset{\hookrightarrow \text{ line } 7 \text{ (while)}}{T(P)}\Big) \overset{\text{execution time of itself}}{\longrightarrow} \in O(n^2) + 3\big(8 + T_{1EX3}(\overset{a+b}{n}-2)\big) + 3 = \end{cases}$$

$$= \text{“}34\text{”} + 5n + 3\lg_3(n) + \lg_3(n) \cdot T(P) + 3T_{31}(n-2) \overset{\in \Omega(n)}{} =$$

$$= \text{“}34\text{”} + 5n + 3\lg_3(n) + \lg_3(n) \cdot \underset{\hookrightarrow \text{ constant}}{c} \cdot n^2 + 3T_{31}(n-2) \geqslant \text{“}4\text{”} \cdot n$$

$P \in O(n^2)$
$P(n) \leq c \cdot n^2$

$g(n) \in \Omega(f) \Longleftrightarrow \exists c > 0 \,/\, \forall n > n_0$
$g(n) \geqslant c \cdot f(n)$

Fri 8/2/19

# Case 2

```
function F(x,y)
if  y>x then
    c=F(2·x, y)
    while y>x do
        i=0
        repeat
            c=c+n·x·y
            i=i+1
        until i·x>y
        x=2·x
    fin_while
fin_if
```

x puede incrementar $\log_2(y)$ hasta que se deje de cumplir la condición } $x \cdot \log_2(y)$

Size of the problem $\Rightarrow n = y/x$

- $x$ is the only changing value it should be included

1) Selection statement ? Line 2
2) Recursive calls
3) Iterative structures

inner loop => repeat $\sum_{i=0}^{i=y/x}$   equivalent condition => while $\sum_{y/x>1}$   $\left.\begin{array}{l} n = y/x \\ \frac{y}{2x} = \frac{n}{2} \end{array}\right.$

$$T_{32}(n) = \begin{cases} 2 & n \geq 1 \\ \text{"5"} + T_{32}\left(\frac{n}{2}\right) + \sum_{j=\log_2(n)}^{0} \left(\text{"6"} + \sum_{i=1}^{2^j} 8\right) = \end{cases}$$

$n = 2^j$ variable change

$$= 5 + T_{32}\left(\frac{n}{2}\right) + 6\log_2(n) + \sum_{j=\log_2(n)}^{0} 8 \cdot 2^j =$$

$$= 5 + T_{32}\left(\frac{n}{2}\right) + 6\log_2(n) + 8\sum_{j=0}^{\log_2(n)} 2^j =$$

$\rightarrow \int 2^j \approx 2^j \cdot \ln 2 \simeq$
$\simeq 2^j$

$\rightarrow 2^{\log_2(n)} \approx n$

$$= 5 + T_{32}\left(\frac{n}{2}\right) + 6\log_2(n) + \text{"7"} \cdot n$$

constant $\neq 0$

↳ not sure if recursive call will contribute w/ value $\leq n$

$\in \Omega(n)$ lower bound

$\in O(?)$ upper bound

Mon 18/2/19

$$t(n) - t\left(\frac{n}{2}\right) = 5 + \underbrace{6\log_2(n)} + 7n$$

$n = 2^i$

$$t(2^i) - t(2^{i-1}) = \boxed{\boxed{5 + i} + \boxed{7 \cdot 2^i}} \rightarrow b^i p(i)$$

$\boxed{r = 1 \quad \alpha = 2}$   $\boxed{b = 2 \quad \alpha = 1}$

$r^i - r^{i-1} = 0$

$\div r^{min}$

$r - 1 = 0$

$r = 1$

$\boxed{r = 1 \quad \alpha = 1}$

$\left\{\begin{array}{l} r = 1 \quad \alpha = 3 \\ r = 2 \quad \alpha = 1 \end{array}\right.$

$$t(2^i) = \underbrace{A + Bi + Ci^2}_{r=1 \quad \alpha=3} + \underbrace{D2^i}_{r=2 \quad \alpha=1}$$

$$t(n) = A + B \log_2(n) + C \left(\log_2(n)\right)^2 + \underbrace{D \cdot n}_{\text{max. term}} \in \left\{\begin{matrix}\Omega(n)\\ O(n)\end{matrix}\right\} \Theta(n)$$

Fri 8/12/19

# Case 3

1º) a=b

2º) a≠b

```
function Q(a,b,n)
    x=1
    y=0
    for i=1 to n do { $x=x \cdot a$ }  → $x = a^n$
    while x>1 do { y=y+1            $\log_a(a^n) \Rightarrow n$
                   x=x/b }
                                      always fulfilled in the 1st scenario a = b
    if (a=b) or (n≤a) then { $y=2 \cdot a \cdot y$ }
    else {$y=a \cdot Q(a,b,n/a)+b \cdot Q(b,a,n/a)$}
    ~~Q=y~~ → return y
```

$1^\circ$) $a = b$

$\in \Omega(n)$

$\in O(?)$

Size of the problem => $n$

1) Selection statement line 7

$$T_{33}(n) = \begin{cases} \text{"11"} + \overset{\to\text{line 4}}{5n} + 5\,\overset{\to\text{lines 5 and 6}}{\cancel{\log_a(a^n)} \to n} = \text{"11"} + 5n + 5n & (a = b) \\ \text{"11"} + 5n + 5\log_b(a)\cdot n & (a \neq b) \wedge \overset{n \leq \text{smallest}(a,b)}{(n \leq a)} \\ \text{"11"} + 5n + \underbrace{5\log_b(a)}_{\text{constant}}\cdot n + 2\,T_{33}\left(\frac{n}{a}\right) + \text{"4"} & \overset{\text{otherwhise}}{(\text{o.w})} \end{cases}$$

Mon 18/2/19

$$t(n) = 10 + 6n + 2\,t(n/a)$$

$$t(n) - 2\,t(n/a) = 10 + 6n$$

$$t(a^i) - 2\,t(a^{i-1}) = \boxed{10 + 6a^i} \longrightarrow b^i\,p(i)$$

$$r^i - 2r^{i-1} = 0 \qquad\qquad r = a \quad \alpha = 1$$

$$r - 2 = 0 \qquad\qquad r = 1 \quad \alpha = 1$$

$$r = 2 \quad \alpha = 1$$

$$t(a^i) = \overbrace{A}^{r=1\ \alpha=1} + \overbrace{B2^i}^{r=2\ \alpha=1} + \overbrace{Ca^i}^{r=a\ \alpha=1}$$

$\boxed{n = a^i}$ => $\log_n(a) = i$

$$2^i = \left(a^{\log_a(2)}\right)^i$$

$$t(n) = A + B\left(a^{\log_a(2)}\right)^i + Cn$$

$$t(n) = A + B\left(a^i\right)^{\log_a(2)} + Cn$$

$$t(n) = A + B\,n^{\log_a(2)} + Cn$$

$$\begin{cases} a \geq 2 \Rightarrow \Theta(n) & \longrightarrow \begin{cases} \log_2(2) = 1 \\ \log_4(2) = 0{,}5 \end{cases} \\ a < 2 \Rightarrow \Theta\left(n^{\log_a(2)}\right) & \longrightarrow \left\{ \log_{\sqrt{2}}(2) = 2 \right. \end{cases}$$

# Case 4

```
function Jun91(n:int)
x=1
if n>2 then
    y=Jun91(n/2)
    while (x<n) do
        x=x+1
        y=y+x
    fin_while
    y=~~2~~*Jun91(n/2)+x
fin_if
Jun91=x+y
```

$2t(n/2)$

$\sum_{x=1}^{n}$ "5"

$\neq 2t(n/2)$

Size of the problem => n

$$t(n) = \begin{cases} 4 & n \le 2 \\ 11 - 2t(n/2) + \overset{\text{while}}{\sum_{x<n}} 5 \end{cases} = 11 + 2t(n/2) + \sum_{x=1}^{n} 5 =$$

$$= 11 + \underline{2t(n/2)} + 5n \quad \in \begin{cases} \Omega(n) \\ O(?) \end{cases}$$

Mon 18/2/19

$$t(n) = 11 + 2t(n/2) + 5n$$

$$t(n) - 2t(n/2) = 11 + 5n$$

$$\boxed{n = 2^i}$$

$$t(2^i) - 2t(2^{i-1}) = 11 + 5n \Rightarrow b^i p(i)$$

$$r^i - 2r^{i-1} = 0 \qquad\qquad \underbrace{11}_{\boxed{r=1 \quad \alpha=1}} + \underbrace{5 \cdot 2^i}_{\boxed{r=2 \quad \alpha=1}}$$

$$r - 2 = 0$$

$$\boxed{r = 2 \qquad \alpha = 1} \qquad \begin{cases} r = 2 & \alpha = 2 \\ r = 1 & \alpha = 1 \end{cases}$$

$$t(2^i) = \overbrace{A + B2^i}^{r=2 \quad \alpha=2} + \overbrace{Ci2^i}^{r=1 \quad \alpha=1}$$

$$t(n) = A + \underline{\underline{Bn}} + C \cdot \log_2(n) \cdot n \quad \in \begin{cases} O(n \cdot \overbrace{\log(n)}^{\log(n) > 1}) \downarrow \text{prove } O(n) \\ \Omega\,\underline{\underline{(n)}} \end{cases}$$

$$C \to 3 \Rightarrow \Theta(n\log(n))$$

$$C \to 0 \Rightarrow \Theta(n)$$

zero

# Case 5

size of the problem -> n

```
procedure F(n:integer):integer
{ if n>4 then {
      j:=n;  k:=1;  h:=0;
      while j>1 do { k:=k+1;
                    j:=n DIV k;
                    h:= h +G(k,j);
                    if k<4 then
                        h:=9* F(n DIV 3) +h
                    }
                }
      return h
}
```

$G(n,m) \in O(n*m)$

whole integer division :)

n>1

n-1>0

loop iters n-1

2 iters

k=2
k=3

not in 4th iter

Size of the problem $\boxed{n}$

$$t(n) = \begin{cases} 2 & n <= 4 \\ \leq \text{"}5\text{"} + \left(\frac{n}{2}+1\right)\left(\text{"}12\text{"} + Kn\right) + 2t(n/3) & \end{cases} \in \begin{cases} \Omega(n) \\ O(?) \end{cases}$$

# Case 6

```
procedure P(l,m : integer):integer
var a,i,y : integer;
{  if l<m+1 then return 0
          else { y:= l-m; a:=P(y/3,0);
                    for i:=m to l do a:= a*F(y);
                    while y>1 do
                              y:=y/3
                              a:=a*Q(l-m,m)
                    end;
                    return a
                    }
}  $F(n) \in O(n^2)$ y $Q(n,m) \in O(\log n)$
```

size of the problem $n = \ell - m$

$$t(n) = \begin{cases} \text{"3"} & n < 1 \\ \leq \text{"10"} + t(n/3) + \underbrace{\underset{\text{iters of for loop}}{(n+1)}\left(6 + K_1 \overset{\text{upper bounded}}{n^2}\right)}_{\text{¿}O(n^3)?} + \underset{\text{while loop iter}}{\log_3(n)}\left(\text{"6"} + K_2 \log(n)\right) \end{cases}$$

$$\in \begin{cases} \Omega(\underline{n}) \\ O(?) \end{cases}$$

Mon 18/2/19

$t(n) = 2t(n-1) + n$

$t(n) - 2t(n-1) = n \to \boxed{\begin{matrix} r = 1 \\ \alpha = 2 \end{matrix}}$

$r^n - 2r^{n-1} = 0$

$r - 2 = 0$

$\boxed{r = 2 \; ; \; \alpha = 1}$

$$\boxed{t(n) = A \cdot 2^n + B + Cn}$$

n is size of the problem

$t(n) = 2t\left(\frac{n}{2}\right) + n \to n = 2^i$

$t(n) - 2t\left(\frac{n}{2}\right) = n$

$r^n - 2r^{n/2} = 0$

$r^{n/2} - 2 = 0$

$n \longrightarrow n/2$

$2^i \longrightarrow 2^{i-1} = \frac{2^i}{2}$

$t(2^i) - 2t(2^{i-1}) = 2^i$

$r^i - 2r^{i-1} = 0$

$r - 2 = 0$

$\boxed{r = 2 \; ; \; \alpha = 1}$

$\to \boxed{r = 2 \quad \alpha = 2}$

- non homogeneous part

$b^i p(i)$

$2^i \cdot 1 \longrightarrow \boxed{r = 2 \; ; \; \alpha = 1}$

$$\begin{cases} t(2^i) = A2^i + B\,i\,2^i \\ t(n) = An + B\log_2(n) \cdot n \end{cases}$$

Dudas

size of the problem

# Case 7

```
function XPromedio(var k: Integ;i,d:Integer):Real;
begin
k:=k+1;
if d<= i+2 then
    XPromedio:=1
else
    XPromedio:=XPromedio(k,i+2,d)
    + XPromedio(k,i,d-2)
   - k*XPromedio(k,i+1,d);
end;
```

$n = d-(i+2)$

$n = d-i-2$

$n = n-2$

## [CASE 7]

size of the problem $\overset{\frown}{d-1}$ $n = d-(i+2) = d-i-2$ $\quad$ $n = d-(i+1)$

$n = n-2$ $\quad$ $n = d-i-1$

$n = n-1$

$$t(n) = \begin{cases} \text{"4"} & n \leq 2 \\ \underset{\Omega(1)}{\text{"10"}} + 2t(\overset{\downarrow}{n-2}) + t(\overset{\downarrow}{n-1}) & \end{cases}$$

$t(n) - 2t(n-2) - t(n-1) = 10 \;\rightarrow\; b^i p(i)$

$r^n - 2r^{n-2} - r^{n-1} = 0$ $\quad$ $\{\, r = 1 \quad \alpha = 1$

$r^2 - r - 2 = 0$

$$\begin{cases} r_1 = 2 & \alpha = 1 \\ r_2 = -1 & \alpha = 1 \end{cases}$$

$$t(n) = A(-1)^n + \underset{\Omega(1)}{\overline{B}} + \underline{C} \cdot 2^n \in \begin{cases} O(2^n) \\ \Omega(1) \end{cases} \qquad \begin{matrix} \Theta(2^n) \\ \Omega(1) \end{matrix}$$

$t(1) = t(2) = 4$ $\;\rightarrow t(1) = 4 \rightarrow 4 = -A + B + 2C$

$t(3) = 22$ $\;\rightarrow t(2) = 4 \rightarrow 4 = A + B + 4C$

$\rightarrow 22 = -A + B + 8C$ $\qquad -A + \cancel{B} + 2C = A + \cancel{B} + 4C$

$4 = +C + B + 2C$ $\qquad -2A = -2C + 4C$

$4 = B + 3C$ $\qquad -\cancel{2}A = \cancel{2}C$

$B = 4 - 3C$ $\qquad \boxed{C = -A}$

$C = \frac{18}{4} = 9/2 \neq 0$

# Case 8

*a) G(a,b) ∈ O(1)*   *b) G(a,b) ∈ O(a²)*

```
Function Suerte(X,Y,Z)
    Si (X<Y+5) entonces Devolver 0
    Sino {
            Z=1
3n —> Para I=Y hasta X-1 { Z=Z*(X-I)}        } laps —> X-Y=n
nlog5n —> Mientras(Z>1) {Z=Z/5 }                 log5(Z)
            K=1
            J=1
            L=X-Y = n  —> n/K
            Mientras (J>=1 {
n               K=K+1  => hasta K=n
laps            J=L/K  ——> not a recursive division
   K=1            n/K  => 1
   ↓            Para I=1 hasta J {C=G(I,J)}
   K=L=n        }
2t(n/2)     I=1                                  n/2 = (X-Y)/2
 2 laps     Repite {H=H+Suerte(X/2,Y/2,V)*I
                    I=I+1} Hasta que I²>6
            Devolver (H*I)
    }
```

Z · n —> $Z^2 n^2$ } n!

Z(X-Y)

$\log_5 n! \Rightarrow n \cdot \log_5 n$

$\log_5(n \cdot (n-1)(n-2) \cdots (n-n)) = n \log_5 n$

inequality sign

constant · $n \le t(n) \le n^3 \sum_{K=1}^{n} \frac{1}{K^3}$

$\sum_{K=1}^{n} \left( "3" + \underbrace{\sum_{I=1}^{n/K} I^2}_{} + "2" \right)$ —> $2 \cdot n/K$

$\frac{I^3}{3} \simeq \frac{n^3}{K^3}$

$\frac{(n/K)^3}{3} = \cancel{\frac{1}{3}} \cdot \frac{n^3}{K^3}$

$$\boxed{n^3 \sum_{K=1}^{n} \frac{1}{K^3}}$$

—> constant · $K^3$

log(n)

size of the problem $X - Y = n$

$$t(n) = \begin{cases} \text{"}4\text{"} & n < 5 \\ \text{"}7\text{"} + 2t(n/2) + \text{"}5\text{"}n + \text{"}3\text{"}n\lg(n) + kn^3 & \end{cases}$$

HOMEWORK

# Case 9: $G(x,y) \in O(x\cdot y)$

```
Function Septiembre (A:integer):integer;
Si (A<=5) Entonces Devolver (1)
En Caso Contrario
    B=A;
    C=0;
    D=0;
    Mientras (B>=1)
        C=C+1;
        B=A/C;   => B será 1 cuando A=C y C va de 1 en 1
A       D=D+G(C,B);
        Si (C<3) Entonces
  2      C=1       D=D+Septiembre(A/4)*5;
         C=2       D=D*D;
        Fin Si
    Fin Mientras
Fin Si
Devolver (A*A)
```

# Case 10

*a) Enero(n) ∈ O(1) b) Enero(n) ∈ O(n^2)*

```
entero Funcion Diciembre(entero X)
    A=1
    Si (X<1)
        Para I=1 hasta 10.000
            A=A*I
        Fin Para
    Sino
        A=Diciembre(X-1)*A
        Para I=1 hasta X
            A=I*X
            Para J=1 hasta A
                Z=Z+Enero(J)
            Fin Para
        Fin Para
        return (A*Diciembre(X-2))
    Fin Si
```

# Case 11

```
function  f(x: integer) : integer;
var z,i : integer;
begin
z:=0;
if x>2 then
    begin
    for i:=1 to 5  do
          z:=z+i* f(x-1);
          z:=z+2* f(x-2);
    end;
f:=z;
end;
```

# Case 12

```
procedure uno(matriz[1..n],n){
    para i=1 hasta n {
        y=matriz[i]
        tres(i)
        para j=n hasta i {
            matriz[i]=matriz[j]
            matriz[j]=dos(n/2, tres(n/2))
        }
    }
    si n>1 uno(matriz[1..n/3], n/3)
    retornar
}
```

dos(m,k) $\in O(m \log m)$

tres(m) $\in O(\log m)$

Fri 15/2/19

# [SOLVE RECURSIVE CALLS]

$$t(n) = 2t(n-1) + 4 \in \Omega(1)$$

$$\underbrace{t(n) - 2t(n-1)}_{\text{homogenic ecuation} \Rightarrow \text{roots}} = \boxed{4}$$

→ non - homogeneous part

→ contribution to the ecuation => $4 = b^n p(n)$ → $1^n = 1$ → $r = 1 \; ; \; \alpha = 1$

$$t(n) - 2t(n-1) = 0$$

$\div r^{min}$

$r^n - 2r^{n-1} = 0$ (characteristic ecuation)

$r - 2 = 0$ => minimum exponent on r

$\boxed{r = 2 \Rightarrow \alpha = 1}$ algebraic degree of the root

↳ Example

$(r-3)^2 = r^2 - 6r + 9$

$r = 3 \Rightarrow \alpha = 2$

$t(n) = K 2^n$

$$\begin{cases} t(n-1) = 2t(n-2) + 4 \\ t(n) = 2(2t(n-2)+4)+4 \\ t(n) = 2(2(2t(n-3)+4)+4)+4 \end{cases}$$

↳ $t(n) = \underline{2^K \cdot t(n-K)} + 4 \cdot K$ ↑↑

$*\; (r-a)^2 (r-b)^3 (r-c) = 0$

Every root will contribute w/ a term multiplying by a general polynomial w/ degree less than (?)

$$t(n) = \underline{a^n (A + Bn)} + b^n (C + Dn + En^2) + F \cdot c^n$$

↳ $Aa^n + Bna^n$

## Non - homogeneous part

$2^n \cdot 7n = b^n p(n) \Rightarrow b = 2 \; ; \; p(n) = 7n$

Algebraic degree 1+ degree of polynomial

$$\begin{cases} r = 2 & \alpha = 2 \\ r = b & \alpha = 1 + \partial(p) \end{cases}$$ → degree of polynomial

## General solution of the expression

$$t(n) = A \cdot 2^n + B 1^n = A 2^n + B$$

Fri 15/12/19

# Complexity of recursive algorithms I

- $T(n)=3T(n-1)+4T(n-2)$ when $n>1$
  - $T(0)=0$; $T(1)=1$
- $T(n)=2T(n-1)+(n+5)3n$ when $n>0$
  - $T(0)=0$
- $T(n) = a + c + T(n-1)$, when $n > 1$
  - $T(1) = a + b$ a,b,c from $R^+$
- $T(n) = 5T(n-1)-8T(n-2)+4T(n-3)$, when $n > 2$
  - $T(2) = 2$; $T(1) = 1$; $T(0) = 0$
- $T(n) = T(n-1)+2T(n-2)-2T(n-3)$, when $n > 2$
  - $T(n) = 9n^2-15n+106$ when $n = 0,1,2$

$T(n) = 3\,T(n-1) + 4T(n-2)$ when $n > 1$ $T(0) = 0$ ; $T(1) = 1$

$$\underline{T(n) - 3T(n-1) - 4T(n-2)} = 0$$ → homogeneous part

$$r^n - 3r^{n-1} - 4r^{n-2} = 0$$

$\div r^{n-2}$

$$r^{+2} - 3r - 4 = 0 \quad \begin{cases} r = 4 \Rightarrow \alpha = 1 \\ r = -1 \Rightarrow \alpha = 1 \end{cases}$$

→ algebraic degree $\left( \frac{\text{degree}}{\#\text{ roots}} \Rightarrow \alpha = \frac{2}{2} = 1 \right)$

$$\boxed{t(n) = A \cdot 4^n + B(-1)^n}$$

bigger term

$\to \in O(4^n)$

$\to \in \cancel{\Theta(1)}$ (?) ↳ $\Theta(4^n)$

$$\begin{cases} A = B = 0 \\ A = 0 \cancel{, B} \neq 0 \Rightarrow 6 \cdot 4^n < t(n) < 8 \cdot 4^n \quad \Theta(4^n) \\ A \neq 0 , B \neq 0 \\ A \neq 0 \neq B \end{cases}$$

minimum cost (if) + 3 $\Rightarrow t(n) = A4^n + 3(-1)^n$

- A cannot vanish because the computational cost would be sometimes possitive and others negative, so that cannot happen.

$T(n) = 2T(n-1) + (n+5)3n$ when $n>0$ => $T(0)=0$

$\underline{T(n) - 2T(n-1)} = \underline{3n(n+5)}$

$r^n - 2r^{n-1}$ $\qquad b^n P(n) = 3n(n+5) = 1^n \cdot 3n(n+5)$

$1^n \cdot (3n^2 + 15n)$ → degree 2

$r - 2 = 0$ $\qquad$ (constant variation) $\qquad$ → $r=1$ ; $\alpha = 3$

$(r = 2 \; ; \alpha = 1)$

$P(n) = A2^n + B1^n + Cn + Dn^2$

$P(n) = A2^n + \boxed{B + Cn + Dn^2} \in \begin{cases} O(2^n) \\ \Omega(1) \end{cases}$

most important constant $\qquad$ $2^n$ most important

Fri 22/02/19

$\underbrace{t(n) - 2t(n-1)}_{r=2 \quad \alpha=1} = \underbrace{(n+5)3n}_{r=1 \quad \alpha=3}$

$t(n) = A + Bn + Cn^2 + D2^n \in \theta(2^n)$

$t(0) = 0$

$t(1) = 18$

$t(2) = 36 + 42 = 78$

$t(3) = 156 + 72 = 228$

$A = -D$

$18 = B + C + D \to B = 18 - C - D$

$78 = 2B + 4C + 3D \qquad 78 = 36 + 2C + D \to C = \frac{42-D}{2}$

$228 = 3B + 9C + 7D \qquad 228 = 54 + 6C + 4D$

$228 = 54 + 126 - 3D + 4D$

$\boxed{D = 48} \neq 0$

CASE 1 (code)

$$T_{31}(n) = \begin{cases} 4 & n < 20 \\ \leq \text{"}7\text{"} + \text{"}5n\text{"} + \text{"}3\log_3(n)\text{"} + C\log_3(n)\,n^2 + 3T_{31}(n-2) \end{cases} \in \begin{cases} O(\sqrt{3}^n) \\ \Omega(n) \end{cases}$$

$\Omega(1) \geq \Omega(\log(n)) \geq \Omega(n)$

$T(n) = 3T(n-2) + \text{"something"}$

$T(n) - 3T(n-2) = 0$

$r^n - 3r^{n-2} = 0$

$r^2 - 3 = 0$

$r = \pm\sqrt{3}$

$\begin{cases} r = +\sqrt{3} & \alpha = 1 \\ r = -\sqrt{3} & \alpha = 1 \end{cases}$

$T_{31}(n) = \underbrace{A\sqrt{3}^n}_{\text{biggest term}} + \underbrace{B(-\sqrt{3})^n}_{\text{if case statement}} + \text{"something"}$

# Complexity of recursive algorithms II

- $T(n) = 2T(n-1)+(n + 5)3^n$, when $n > 0$
  - $T(0) = 0$
- $T(n) = 2T(n-1)+n$ when $n > 0$
  - $T(0) = 0$
- $T(n) = 4T(n-1)-2^n$, when $n > 0$
  - $T(0) = 1$
- $T(n) = 2T(n-1)+n+2^n$, when $n > 0$
  - $T(0) = 0$
- $T(n) = 7+T(n-1)+T(n-2)$, when $n > 1$
  - $T(1) = T(0) = 2$

Fri 22/02/19

$$T(n) = 2T(n-1) + n$$

$$\underbrace{T(n) - 2T(n-1)}_{r=2 \quad \alpha=1} = \underbrace{n}_{r=1 \quad \alpha=2} \in \Omega(n) \to \Omega(2^n)$$

$$T(n) = A + Bn + C2^n \in O(2^n) \to \Theta(2^n) //$$

$$T(0) = 0 \longrightarrow 0 = A + C$$

$$T(1) = 1 \longrightarrow 1 = A + B + 2C = B = 1 - C$$

$$T(2) = 4 \longrightarrow 4 = A + 2B + 4C \Rightarrow 4 = C + 2 \Rightarrow \boxed{C = 2 \neq 0}$$

# Variable changes

- $T(n) = 4T(n/3) – 3T(n/9) + \boxed{\lg_3(n)} \rightarrow \Omega(\log(n))$
  - $T(n) = n^2$, when $n<4$

- $T(n) = 2T(\sqrt[3]{n}) + \lg_2(n)$
  - $T(n) = 5n$, when $n<4$

- $T(n) = nT^2(n/2)$
  - $T(1) = 1/3$

$$T(n) = 4\,T(n/3) - 3\,T(n/9) + \log_3(n)$$

$$T(n) = n^2 \text{ , when } n<4$$

$n = 3^i$ → tells us which values to choose for the initial conditions values substitution

$$T(3^i) - 4\,T(3^{i-1}) + 3\,T(3^{i-2}) = \log_3(3^i)$$

$$T(3^i) - 4\,T(3^{i-1}) + 3\,T(3^{i-2}) = i \longrightarrow r=1 \quad \alpha = 2$$

$$r^i - 4r^{i-1} + 3r^{i-2} = 0$$

$$r^2 - 4r + 3 = 0 \quad \left\{ \begin{array}{ll} r=3 & \alpha=1 \\ r=1 & \alpha=1 \end{array} \right.$$

$$\left\{ \begin{array}{ll} r=1 & \alpha=3 \\ r=3 & \alpha=1 \end{array} \right.$$

$$t(3^i) = \overbrace{A + Bi + Ci^2}^{r=1 \quad \alpha=3} + \overbrace{D\,3^i}^{r=3 \quad \alpha=1}$$

$$t(n) = A + B\log_3(n) + \underbrace{C(\log_3(n))^2}_{\text{(cannot make this simpler) !!!}} + Dn \in \left\{ \begin{array}{l} O(n) \text{ (worst case)} \\ \Omega(\log(n)) \text{ (best case)} \end{array} \right.$$

↗ from the original equation

↓

$$\log_3(n^2) = 2\log_3(n)$$

$$\left\{ \begin{array}{l} D \neq 0 \Rightarrow \Omega(n) \Rightarrow \Theta(n) \\ D = 0 \;\&\; C \neq 0 \Rightarrow \Theta((\log(n))^2) \\ D = 0 = C \Rightarrow \Theta(\log(n)) \end{array} \right.$$

$$T(n) = n^2 \;;\; n<4$$

$$T(3) = 9$$

$$T(1) = 1$$

los valores los coges del exponente del cambio de variable $n = 3^i$ //

$$T(9) = 36 - 3 + 2 = 35$$

$$T(27) = 140 - 27 + 3 = 116$$

$$\left\{ \begin{array}{l} 1 = A + D \longrightarrow 1 - A = D \\ 9 = A + B + C + 3D \\ 35 = A + 2B + 4C + 9D \\ 116 = A + 3B + 9C + 27D \end{array} \right.$$

$x_1 = -3'625$

$x_2 = -1$

$x_3 = -0'25$

$$\boxed{x_4 = 4'625} \neq 0$$

Mon 25/02/19

## Example

- $T(n) = 3T(n/2) + \dots$ $\quad \left\{ \begin{array}{l} n \rightsquigarrow n/2 = n \cdot 2^{-1} \\ n = 2^i \rightsquigarrow 2^i/2 = 2^{i-1} \end{array} \right.$

- $T(n) = 2T(\sqrt[3]{n}) + \log_2(n)$ $\quad n \rightsquigarrow \sqrt[3]{n} = n^{1/3} = n^{3^{-1}}$

$\hookrightarrow \in \{ \Omega(\log_2(n))$ $\quad (a^b)^c = a^{b \cdot c}$

variable change $\Rightarrow n = \square^{(3^K)} = 2^{(3^K)}$

$\neq (\square^3)^K$

$$\left[ \sqrt[3]{(\square^{3^K})} = (\square^{3^K})^{3^{-1}} = \square^{3^K \cdot 3^{-1}} = \square^{3^{K-1}} \right]$$

$$T(\square^{(3^K)}) = T(\square^{(3^{K-1})}) + lg_2(n)$$

$\hookrightarrow$ strictly increasing / decreasing functions

$\hookrightarrow$ base $> 1$

$$T(2^{(3^K)}) - 2T(2^{(3^{K-1})}) = \overset{b^i p(i)}{3^K} \quad \left\{ \begin{array}{l} r = 3 \\ \alpha = 1 \end{array} \right.$$

$$\left. \begin{array}{l} r^K - 2r^{K-1} = \emptyset \\ r - 2 = \emptyset \end{array} \right\} \begin{array}{l} r = 2 \\ \alpha = 1 \end{array}$$

$2^K = (\underbrace{3^{\log_3(2)}}_{2})^K = (3^K)^{\log_3(2)}$

$$T(2^{(3^K)}) = A2^K + B3^K \longrightarrow 3^K = \log_2(\overset{3^{2^i}}{n})$$

Example

$\left\{ \begin{array}{l} n = 2^i \\ 3^i = ?? = 2^{\log_2(3)} \end{array} \right.$

$$T(n) = A(\log_2(n))^{\log_3(2)} + B\log_2(n) \in \left\{ \begin{array}{l} O(\log_2(n)) \\ \Omega(\log_2(n)) \end{array} \right.$$ -> from the original ecuation

$$\Downarrow$$

$$\Theta(\log_2(n))$$

## Example

$T(n) = 3T(n/2) + n \quad \in \Omega(n)$

$T(1) = 1$

$T(n) - 3T(n/2) = n$

- $n = 2^i \to (i)$ Determines the values we can use for the recursive call info.

$T(2^i) - 3T(2^{i-1}) = 2^i \Rightarrow b^i p(i)$

$r^i - 3r^{i-1} = 0 \qquad r = 2 \quad \alpha = 2$

$r - 3 = \emptyset$

$r = 3 \quad \alpha = 1$

$$\begin{cases} r = 3 & \alpha = 1 \\ r = 2 & \alpha = 1 \end{cases}$$

$p(i) = 0 \Rightarrow \alpha = 0 + 1$

$t(2^i) = A3^i + B2^i$

$t(n) = An^{\log_2(3)} + Bn$

$\in O(n^{\log_2(3)})$

$\in \Theta(n^{\log_2(3)})$

$$\begin{cases} 3^i \longrightarrow f(n) \\ (2^{\log_2(3)})^i = (2^i)^{\log_2(3)} = n^{\log_2(3)} \end{cases}$$

$T(1) = 1 \to 1 = A + B \Rightarrow B = 1 - A$

$T(2) = 5 \to 5 = 2^{\log_2(3)}A + 2B$

$5 = 3A + 2B \Rightarrow 5 = 3A + 2(1 - A)$

$5 = 3A + 2 - 2A$

$\boxed{3 = A}$

$T(n) = 4T(n-1) - 2^n$

$T(0) = 1$

$p(-1) \rightarrow \alpha = 0$

$T(n) - 4T(n-1) = \boxed{-}\boxed{2^n} \rightarrow b^i \; p(i)$

$\rightarrow r = 2 \quad \alpha = 2$

$r^n - 4r^{n-1} = 0$

$r - 4 = 0$

$r = 4 \quad \alpha = 1$

$$\begin{cases} r = 4 & \alpha = 1 \\ r = 2 & \alpha = 1 \end{cases}$$

$T(n) = A4^n + B2^n$

$$\left.\begin{array}{l} T(0) = 1 \\ T(1) = 2 \end{array}\right\} \quad \begin{array}{l} 1 = A + B \Rightarrow 1 - B = A \\ \boxed{A = 0} \\ 2 = 4A + 2B \\ 1 = 2A + B \\ 1 = 2(1-B) + B \\ 1 = 2 - 2B + B \\ 1 - 2 = -B \\ -1 = -B \\ \boxed{B = 1} \end{array}$$

$A = 0 \Rightarrow B \neq 0$

$\hookrightarrow \underline{\underline{\theta(2^n)}}$

$T(n) = n\,(T(n/2))^2$

Fri 1/3/19

$n = 2^i$

$T(2^i) = 2^i \cdot (T(2^{i-1}))^2$

$\lg(T(2^i)) = \lg(2^i \cdot (T(2^{i-1}))^2)$

$\lg_2(T(2^i)) = \lg_2(2^i) + 2\lg(T(2^{i-1}))$

$\lg_2(T(2^i)) - 2\lg_2(T(2^{i-1})) = i$

$\underbrace{r^i - 2r^{i-1}} = \underbrace{i}$

$r = 2 \quad \alpha = 1 \qquad r = 1 \quad \alpha = 2$

$\boxed{\lg_2(T(2^i))} = \underline{A + Bi + C \cdot 2^i}$

[DOMAIN CHANGE]

$2^{\lg_2(T(2^i))} = 2^{(A + Bi + C \cdot 2^i)}$

$T(2^i) = 2^A \cdot 2^{Bi} \cdot 2^{C \cdot 2^i}$

$A \qquad (2^i)^B \quad (2^C)^{2^i} \simeq C^n$

$\boxed{T(n) = A \cdot n^B \cdot C^n}$

$T(1) = 1/3 = 3^{-1}$

$T(2) = 2 \cdot 3^{-2} \Rightarrow T(2) = 2 \cdot T^2(1) = 2 \cdot 3^{-1} \cdot 3^{-1} = 2 \cdot 3^{-2}$

$T(4) = 2^2 \cdot 2^2 \cdot 3^{-4} = 2^4 \cdot 3^{-4}$

$3^{-1} = A \cdot C \Rightarrow A = 3^{-1} \cdot C^{-1}$

$2 \cdot 3^{-2} = 3^{-1}\, 2^B \cdot C \Rightarrow C = 3^{-1} \cdot 2^{1-B}$

$2^4\, 3^{-4} = 3^{-1}\, 2^{2B}\, C^3 \Rightarrow 2^4 \cdot \cancel{3^{-4}} = \cancel{3^{-1}}\, 2^{2B}\, \cancel{3^{-3}}\, 2^{6-3B}$

$2^4 = 2^{2B} \cdot 2^{6-3B}$

$4 = 2B + 3 - 3B$

$\boxed{B = -1 \qquad C = 4/3 \qquad A = 1/4}$

$$T(n) = \frac{1/n \, (1'\tilde{3})^n}{4} \Rightarrow T(n) \in \Theta\left(\frac{(1'\hat{3})^n}{n}\right)$$

Fri 8/2/19

# Basic laws for computing complexities

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}=k>0\Rightarrow\theta(f)=\theta(g)\Rightarrow O(f)=O(g)\,\&\,\Omega(f)=\Omega(g)$$

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}=0\Rightarrow\begin{cases}O(f)\subset O(g) & \& \quad g\notin O(f)\\ \Omega(g)\subset\Omega(f) & \& \quad f\notin\Omega(g)\end{cases}$$

# Exercises I

- **True or False?**

1.- $n^2 \in O(n^3)$ T

2.- $n^3 \in O(n^2)$ F

3.- $2^{n+1} \in O(2^n)$ T

$(n+1)! = (n+1)n!$
$n!$

4.- $(n+1)! \in O(n!)$ F

F 5.- $f(n) \in O(n) \Rightarrow 2^{f(n)} \in O(2^n)$

6.- $3^n \in O(2^n)$

7.- $\log n \in O(n^{1/2})$

8.- $n^{1/2} \in O(\log n)$

9.- $n^2 \in \Omega(n^3)$ F

10.- $n^3 \in \Omega(n^2)$ T

11.- $2^{n+1} \in \Omega(2^n)$ T $\Rightarrow \lim \frac{2^{n+1}}{2^n} = 2$

12.- $(n+1)! \in \Omega(n!)$ T

13.- $f(n) \in \Omega(n) \Rightarrow 2^{f(n)} \in \Omega(2^n)$ F

14.- $3^n \in \Omega(2^n)$

15.- $\log n \in \Omega(n^{1/2})$

16.- $n^{1/2} \in \Omega(\log n)$

3) $2 \cdot 2^n \leq 3 \cdot 2^n$

5) $f(n) \in O(n) \Rightarrow 2^{f(n)} \in O(2^n)$

. $f(n) = 2n \in O(n)$ $\quad k = 3 \Rightarrow \quad 2n \le 3n$

. $2^{2n} \in O(2^n)$

$$(2^2)^n \approx (2^n) \cdot (\cancel{2^n}) \le K \cdot \cancel{2^n} \Rightarrow 2^n \le K$$

13) $f(n) = \frac{n}{2}$ $\qquad 2^{1/2\,n} > K\,2^n$

$$(\cancel{\sqrt{2}})^n > K(\sqrt{2})^n (\cancel{\sqrt{2}})^n$$

$$\frac{1}{K} > (\sqrt{2})^n$$

# Exercises II

- Let “a” belong from R, 0<a<1. Order according to $\subset$ and = the Complexity Order of the following functions:

$n \cdot \log n$, $n^2 \log n$, $n^8$, $n^{1+a}$, $(1+a)^n$, $(n^2+8n+\log^3 n)^4$, $n^2/\log n$, $2^n$.

Mon 4/2/19

# Unit 1

## ALGORITHMIC COMPLEXITY

## 1.1: Definition. Time Cost/Complexity

- ALGORITHM: A structured sequence of instructions to do a certain job
- A precise statement to solve a problem on a computer
- Our goal to be studied: Time Complexity of a given algorithm.
  Informally: how long does it take to be run in a computer?
  - It depends on the input data:
    - How many values?
    - How big are them?

# Analysis of algorithms

- We are interested in measuring the efficiency of a given algorithm
- We assume efficiency as a measure of the time (how long?) and / or space (amount of bytes?) required by the algorithm when running to find a solution to a given problem.
  We are concerned with the former.
- Such efficiency can order the set of algorithms that solve a given problem. We want the most efficient, i.e., least cost

# Time efficiency

- Quantitative value (in time units usually, no matter whether they are real) that computes the number of comparisons, basic operations, function calls,… of the time requiremnets of an algorithm.
- We are concerned with “a priori” efficiency (clocks measure the true one)
- I.e: An estimation of Time Complexity

# COMPLEXITY ORDER ~ ~ Behaviour of an Algorithm

- An algorithm’s (time) behaviour is defined by its time complexity / efficiency; just efficiency from now on. (regardless constants).
- Invariance Principle for algorithms: “Given an algorithm and a couple of implementations of it, I1, I2 which take T1(n), T2(n) seconds to be run, then, there exists a constant c>0 and a Natural n’ such that for all n>n’, T1(n)<cT2(n) holds”

# Size of a problem

- Variable, parameter or function, over which the time complexity of an algorithm has to be computed.
- It usually is related with the number or size of INPUT data
  - Number of elements to be ordered
  - Number of rows or columns or total elements in matrices
  - The biggest value to be computed
  - Always has to be the same for the algorithms we want to analyse and therefore compare.
  - …

# Complexity Function I

- It is a mathematical function.
- A size of the problem dependant function which measures the time complexity of the algorithm.
- We are concerned with the behaviour in the limit, i.e. when such size of the problem grows.
- This is called Asymptotic efficiency

– behaviour when "n" goes to ∞

# Complexity Function II

- It can depend on the INPUT data state.
  - "Finding a a value among an ordered or non-ordered set"
- There are three functions:
  - Best case $f_b(n)$ (fastest)
  - Worst case $f_w(n)$ (slowest). Default one.
  - Average case $f_a(n)$: Probabilistic issues have to be taken into account to properly compute it. Too expensive to compute.

    $\sim f_w(n)$

# Complexity Function => BEHAVIOUR

- Just interested in studying Behaviour of an algorithm.
- Some usual complexity functions includes (Complexity Orders) :
  - 1, log n, n, n·log n, $n^2$, $n^3$, $2^n$, $3^n$, ...
- We then refer it as Behaviours, that are so ordered:

  constant

  quite close to constant

  - Ord(1) < Ord(log n) < Ord(n) < Ord(n·log n) < < Ord($n^2$) < Ord($2^n$) < Ord(n·$2^n$) < Ord($3^n$)

# Tractable Problems

- **Edmonds law**:
  - Tractable problem : Polynomial Complexity
  - Non-tractable problem: Exponential Complexity
- Algorithms which Complexity are greater than $(n \cdot \log n)$ are almost useless => Better find a “cheaper” one.
- Exponentials are only useful as a matter of theoretical examples.

# 1.2: Asymptotic Complexity Orders

## Behaviours: O, Ω, Θ

- These asymptotic orders, summarize the behaviour of a given algorithm.

  They range over the time taken by the algorithm when the size of the problem grows unbounded.
- This is the key fact to be taken into account when we want to compare the efficiency of two algorithms solving the same problem.
- They are valid even for midsized instances

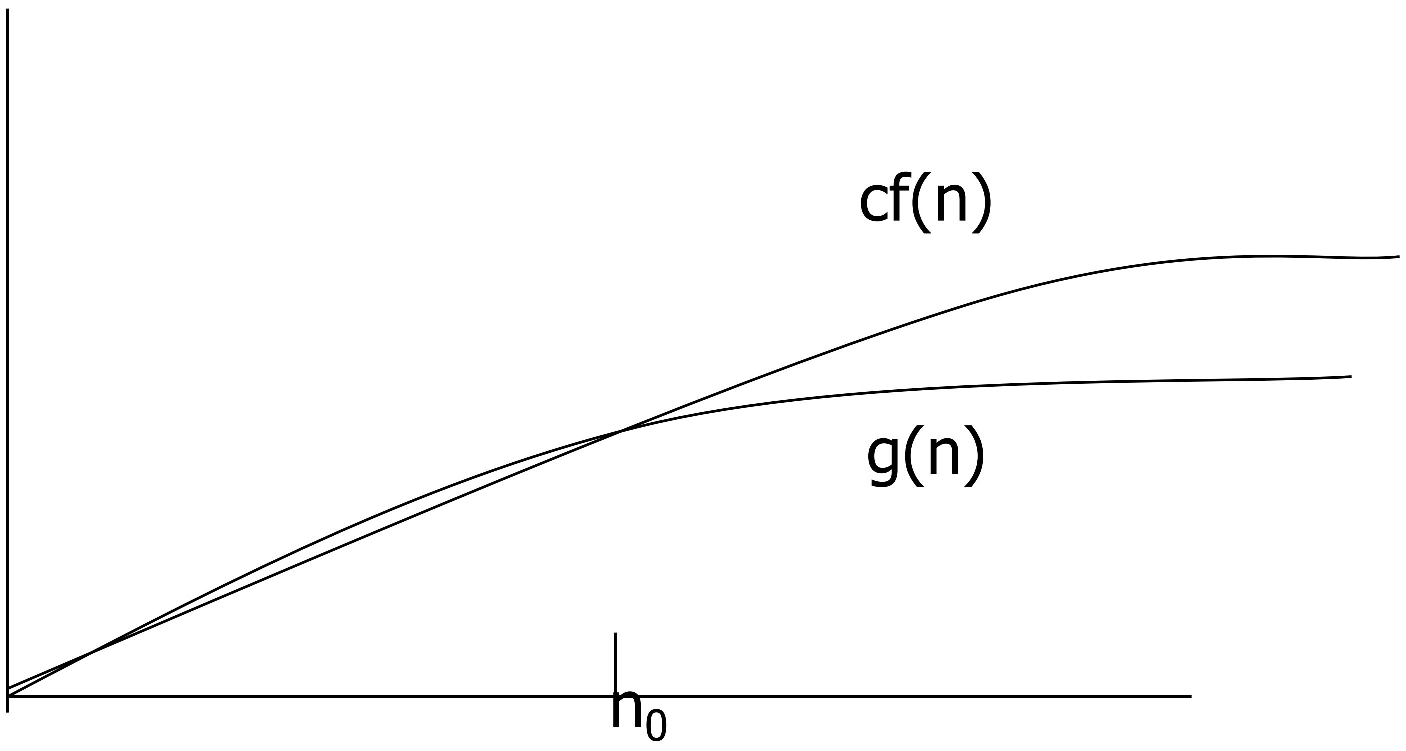
## Asymptotic Orders

# Big O notation

- Given a function f, we want to refer those functions that at most grows as f does. The set of these functions is the upper bound of f, and it is written as O(f).
- Once this value is known, we can guarantee that always the algorithm can be run within the time upper bounded by this value. Formally:
- $g(n) \in O(f(n))$ if $\exists\ c>0, n_0 \ /\ g(n) \leq c \cdot f(n), \forall n \geq n_0$

Graphically

# Big O notation



Example

# Big O notation

- If the map of g is upper bounded by f then g's efficiency is better or equal than that of f.
- Constants must be discarded regarding behaviour issues.
- O(f(n)): Set of all the functions which are upper bounded by f (g belongs to this set).
- **Example**:
  - $P(n) = a_m \cdot n^m + \ldots + a_1 \cdot n + a_0$ (polynomial on "n")
  - $P(n) \in O(n^m)$ ; c must be any value bigger than am

Asymptotic Orders

# Properties of Big O notation

1. $g(n) \in O(g(n))$
2. $O(c \cdot g(n)) = O(g(n))$ (c is constant)
   - Ex : $O(2 \cdot n^2) = O(n^2)$
3. $O(g(n)+h(n)) = \max\{O(g(n)),O(h(n))\}$
   - Ex: $O(n+n^2) = O(n^2)$
4. $O(g(n)-h(n)) = \max\{O(g(n)),O(h(n))\}$
   - Ex: $O(n-n^2) = O(n^2)$

# Big O notation

- The big O notation is used when searching for upper bounds (should be interested in the smallest) for the behaviour of a complexity function in the worst case.
- Notice:

  smallest upper bound you're able to provide

  - If $f(n) \in O(n) \Rightarrow f(n) \in O(n^2)$ ; $f(n) \in O(n^3)$…
  - Immediate, but the least of them is always preferable against the rest.

Asymptotic Orders

# Lower Bound: Ω Notation

# Exact Bound: θ Notation

# Unit 1

ANALYSIS OF ALGORITHMS

# 1.3: Basic computations EOs

- Basic/Atomic/Elementary Operations (EO), t<k, take constant time (referred to INPUT size “n”):
    - Basic arithmetic operations
    - Asignments of basic types (loads, stores)
    - Calls (to Functions, to Procedures) “return”
    - Basic comparison / boolean operations
    - Arrays, and other indexed types, accesing
- Each time of an EO is taken as “1”

# How to compute it in practice?

- 1º.-Time for basic instructions is constant.
- 2º.- These times are properly “included” as they appear in the higher level control instructions structures:
  - Sequences
  - Binary or n-ary conditionals (if , case)
  - Iterative structures (for, while, repeat)
  - Procedures or functions (sometimes recursive) calls
- When an iterative structure includes another one (or even more) inside, it should be computed inside - out.
  Mind the indices relations!

# 1.4: Recursive calls appears

- Characteristic equation
- Non homogeneus
- Variable change
- Domain change
- Initial conditions

# 1.4: Recursion in practice

- Characteristic Equation
  - $T(n) \to r^n$
  - Homogeneus part (only has terms on T(n)) is divided by $r^{min}$
    - if a proper polynomial appears… GREAT!
    - Otherwise, a variable change for the former equation would be required
- The homogeneus part becomes a polynomial which roots has to be computed
- Non homogeneus part has to be fitted with terms $b^n\, p(n)$. Each of which adds the root “b” with algebraic degree 1+degree(p)
- In the worst case a Domain change would be also required to get a *proper polynomial structure* (*both coeficients and exponents are required to be constant*)

$$T\underbrace{\left(\frac{7n+3i}{\lg_i(x^2)}\right)}_{2^i} = T\underbrace{\left(\frac{7n+3i}{\lg_{i+1}(x^2)}\right)}_{2^{i-1}}$$

# 1.4: Initial Conditions -> $\theta$ computation

- Once previous process has been developed, a general expression for the computational complexity has been obtained and from it, the big O order is set out.
- From the recursive equation a lower asymthotic order, $\Omega$, can be computed.
- If both coincide the problem is done: $\theta$.
- Otherwise, the constants from that general expression has to be determined using the initial conditions, until any of $\Omega$ and O is refined to reach the other, and so $\theta$.

# Unit 2

GREEDY / EAGER ALGORITHMS

# Requirements

- Only in those cases where the solution can be reached after the binary decision of including each element belonging from a set of “candidates” into the solution.
- These binary decisions should be taken in a greedy way, i.e., guaranteed that each element of the candidates set could be checked once at most. Which usually implies to do the checking process according to some order

# Greedy algorithms

- Set of algorithms that make decisions taking into account the current state of the problem.
  - Such decisions must guarantee to reach the objective.
- Usually applied to solve optimization problems
  - Easy to design, to implement and very computationally efficient.

# Rough description

- Candidates set is somehow ordered, sometimes in a static way. Exceptionally can be reordered after each decision.
  - The problem objective defines the order of candidates set.
  - Sometimes an extra ordination could be required and this conform the selection function. (every candidate checked at most once)
- The candidate has to be checked, **FEASIBILITY**, to be included in the solution:
  - When the answer is NO, the candidate is definitively discarded
  - When the answer is YES, the candidate is almost always included in the solution definitively, o.w. is half-greedy
- **SOLUTION** function checks the END

# Non-directed graphs: Minimum spanning tree

- G=<N, A> Connected non directed graph.
- Every pair of nodes are connected by some path.
- Each edge has a weight (or length) ≥ 0 (avoiding the existence of negative-length cycles).
- PROBLEM: Find T, a subset of A, that guarantees the connectivity on the set of nodes N, such that, minimizes the sum of the lengths of the edges in T.
- An n-nodes connected graph has at least n-1 edges.
    - An n-nodes connected graph having more than n-1 edges, has got at least one cycle.

# Greedy algorithms to compute the minimum spanning tree

- **Kruskal Algorithm**: T is an empty set and every node conform a component, to begin.

  From the length-increasingly ordered set of edges, an edge is accepted if it connects current unconnected components, and so became into the same component. Until N-1 edges are accepted or only a component remains.

- **Prim Algorithm**: Starting on a given node, the set of edges is assumed length-increasingly ordered.

  Every time is selected the first edge that increases the number of connected nodes.

  Until N-1 accepted edges or all the nodes are connected.

# Kruskal Algorithm

- List of candidates formed by the length-increasingly ordered set of edges
- Initially:
  - T=empty set
  - N current unconnected components (= N nodes)
- Feasibility: An edge is accepted if, and only if, it connects 2 unconnected components so far (which means that reduces the number of unconnected components)
  - If so, the edge is included into the T set and the two unconnected components become one.
  - Otherwise the edge is discarded
- Solution: T has got N-1 edges = There is only ONE connected component
- Finally only a connected component remains in T, therefore T is a minimum spanning tree for all the nodes in G

# Prim Algorithm

- List of candidates formed by the edges increasingly-length ordered
- First step:
  - T=empty set
  - A node is taken as root of the tree A to be constructed
- Feasibility: An edge is accepted in T if, and only if, it connects A with a node currently not included in A
  - If so, the edge is included into the T set and A is enriched with the new node provided by this edge
  - Otherwise the edge is discarded
- Selection: The first edge in the list of candidates, having any node in A is chosen
- Solution: T has got N-1 edges = A has got N edges (as G has)

# Directed graphs: Shortest paths from a given node

- G=<N, A> Directed graph with non-negative length arcs
- From a given node all the rest must be accessible, i.e., must be connected through some path
- PROBLEM: Finding the shortest path (the path that the sum of all its constituent edges is minimized) from a given node to all the rest.
- Dijkstra algorithm: Candidates are arcs (increasingly length ordered) the first having the origin in A is selected
  - If the target node is not in the current graph, it is included in it (together with its cost to be reached from the root node) and the graph is refreshed
  - If the target node is already in the current graph, the arc is included only when this new cost of the target node is better that the already existing reachability cost, the previous arc providing reachability is discarded. Otherwise the arc under consideration is discarded
  - The algorithm ends when all the arcs have been checked
- It is a half-greedy algorithm

# Unit 2

## GREEDY ALGORITHMS: SOME PROBLEMS

# Supermarket cashier I

Implement an algorithm that solves the problem of change by means of the maximum number of coins:

- Non-limited # ( number ) of coins available
- Only some coins available
- Random values for currency system

- From an already defined set of candidates, choose which ones should be in the solution.

- Checking only once each value would solve the problem?

- Question for 1st candidate, repeated for each of the following ones.
  ↳ Fisibility function

Quick Short -> $n\log_2(n)$ -> shortest time to sorting a one dimensional array.

```
order candidates... ; O(nlog(n))
change = 37
for(i=0 ; i<14 ; i++){
    while(change!=0){
        printf(%d, change/Value[i]);
        change %= Value[i]; }
```

O(n)

# Supermarket cashier II

Implement an algorithm that solves the problem of change by means of the minimum number of coins:

- Non-limited coins available within the european currency system.

# Memory storage optimization

Given a set of n files of sizes $s_1$, $s_2$, ..., $s_n$, and an external storage device with capacity d ($d <= s_1 + ... + s_n$). Provide an algorithm that maximizes the number of files to be moved to the storage device, until the device comes full (files cannot be divided).

- The order of the algorithim should be increasing: from smallest size to the biggest.
- Know the ending condition, when fisibility answer is NO => algorithm must finish.

Fri 15/3/19

# Knapsack / Rucksack problem

- Given a set of items, each with a mass and a value, determine the number of each item to be included in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible
- These items can be proportionally divided as required.

Example

8 litres $h^2O$ —> 20u benefit

6 Kg bread —> 15 u benefit

5kg butter —> 10u benefit

} we have a bound -> capacity of the knapsack

- Decreasing order (with proportions)
  - by means of an aux. variable we control the current state of our knapsack.
  - ENDING CONDITION: (boolean function)

[SOLUTION]

CANDIDATES : Items that can be carried

ORDER : decreasing according to $\frac{v(i)}{w(i)}$ → value → weight

- definition of variables

W = capacity ; V = 0 ; i = 0

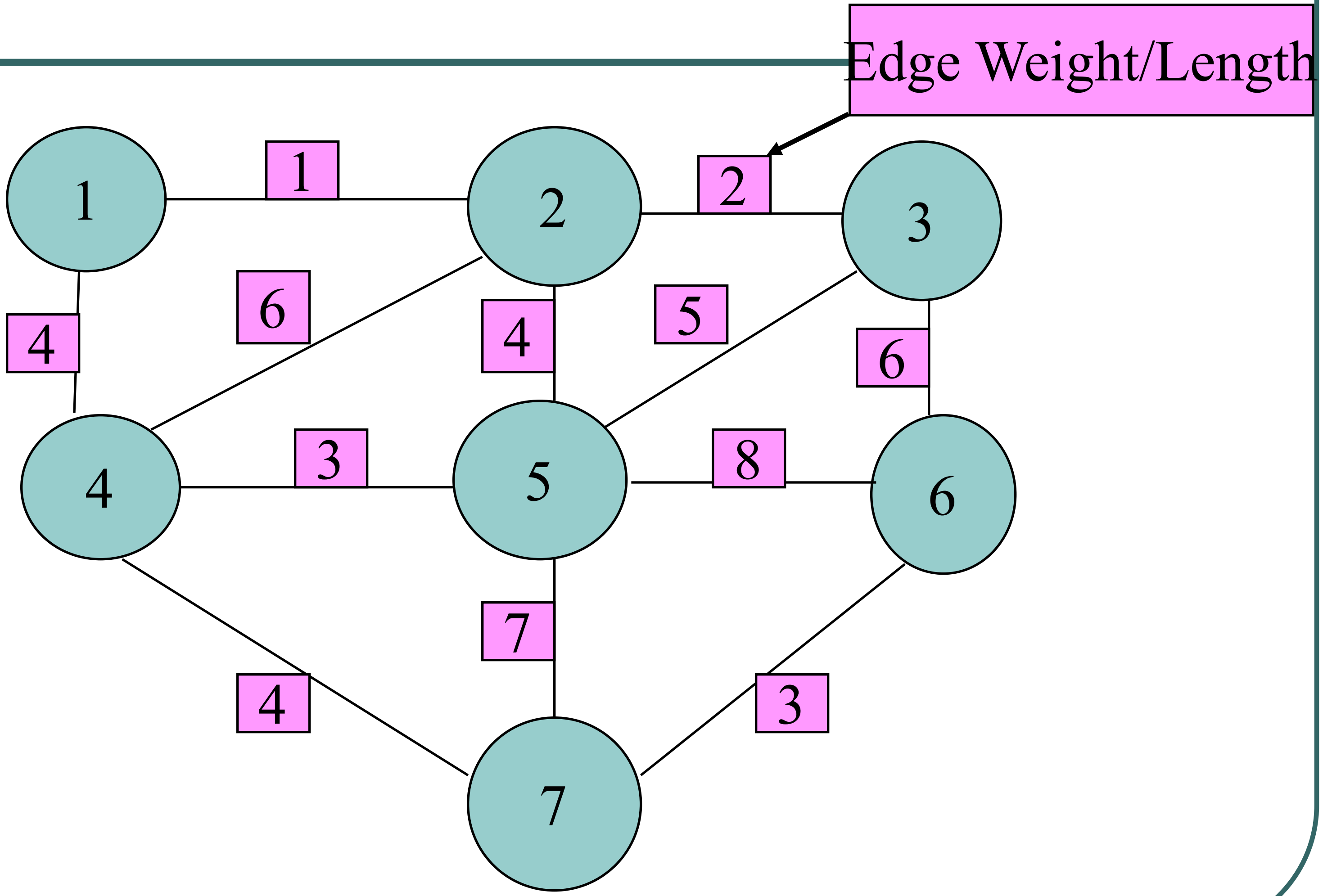
of knapsack / current value of the knapsack / controls iterative structure

FEASIBILITY :
```
if (W > w(i)) {
      V += v(i);
      W -= W(i);      > including the item
      i++;
} else {  (include the proportion that exactly fits)
      V += $\frac{W}{w(i)} \cdot v(i)$ ;
      W = 0;   ⟶ max. proportion of the object that can be included.
      END;
}
```

last step

ENDING CONDITION : else of feasibility

# Minimum Spanning Tree



Edge Weight/Length

CANDIDATES : edges

ORDER : increasing according to the weight of the edges

$U = \{ 1, 2, 3, 4, 5, 6, 7 \}$ ; $N = \emptyset$

↳ nodes included in the solution

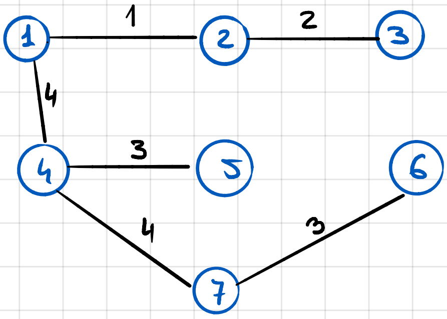
↳ non-connected nodes

FEASIBILITY :
```
if ( |U| decreases ) {
        Accept it ;
        Update (U) ;  N++ ;
} else {
        Discard it ;
}
```

ENDING CONDITION : if $|U| == 1$

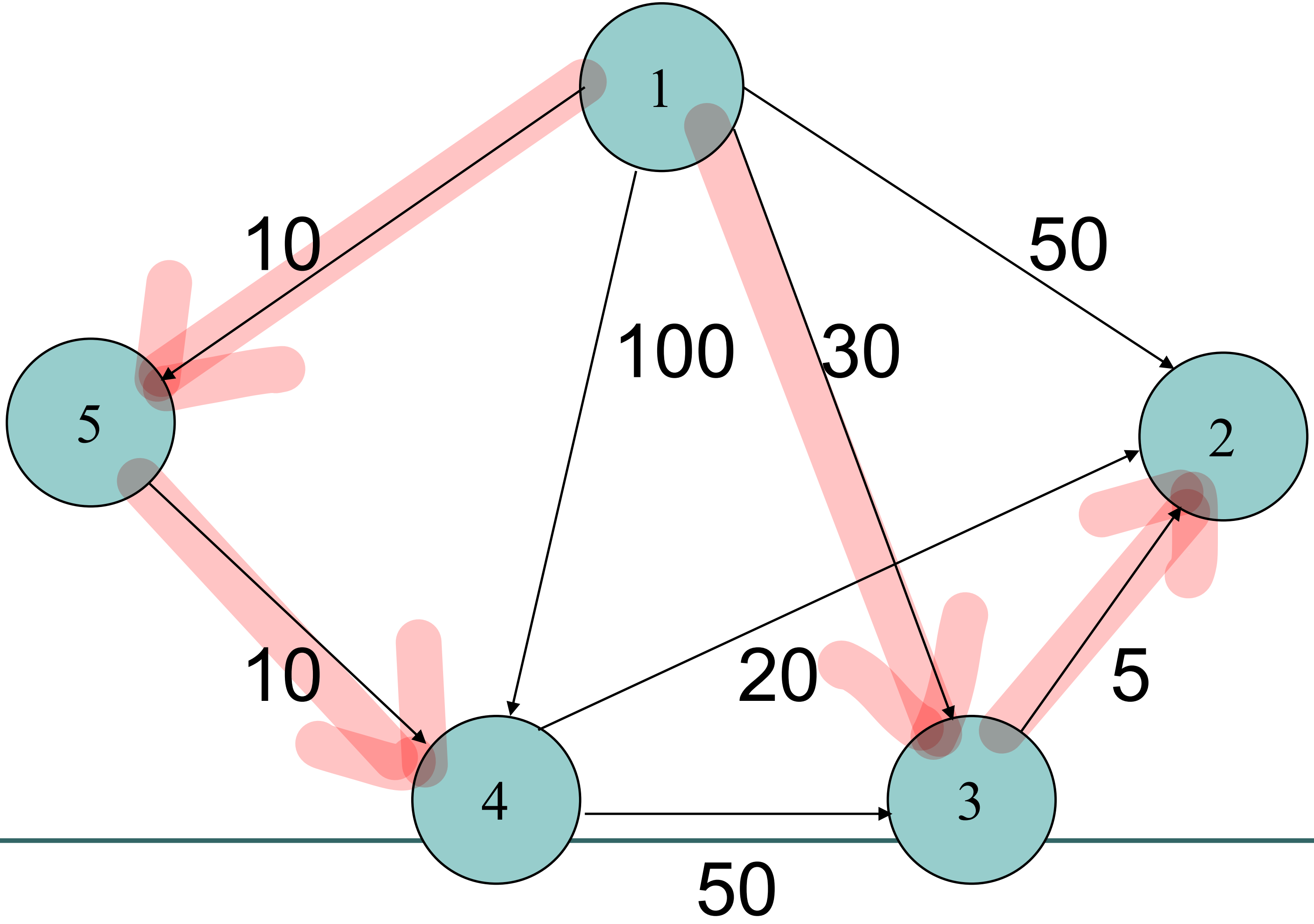
i have accept 1 less than #nodes

N = = ? - 1 ;



[ HALF GREEDY ALORITHM ]

# Shortest paths from 1: DIJKSTRA



CANDIDATES : Arcs

ORDER : increasing according to weight.

SELECTION : arcs having the origin in the already built tree

FEASIBILITY : if (destination is out of the current tree) Accept!
↳ it can be discarded later

else { Discard it; }

ENDING CONDITION : all the arcs have been checked

# Unit 2

## GREEDY ALGORITHMS: SOME PROBLEMS

# Array I

- Design an algorithm to choose the maximum number of pairs such that their sum is possitive, from an increasingly ordered array of whole numbers.
- Each element from the array can contribute at most to one of the pairs.

[-2, -2, -1, 2, 3, 4]

CANDIDATES : pairs of indexes of the array

ORDER : increasing order   i = 0 ; j = last ; N = 0 ;

always

FEASIBILITY : if (A[i] + A[j] > 0) { N++; j--; } i++;

SELECTION : Not required.

ENDING CONDITION : (i >= j) || (A[j] <= 0) ; return N;

# Array II

- Design an algorithm to choose the maximum number of pairs such that their product is possitive, from an increasingly ordered array of whole numbers.
- Each element from the array can contribute at most to one of the pairs.

CANDIDATES : pairs of indexes in the array

CANDIDATES : pairs of indexes in the array

ORDER : increasing

ORDER : increasing

FEASIBILITY : if ( A[i] * A[i+1] > 0 ) { N++ ; i += 2 } else { i++ } ;

FEASIBILITY : if ( A[i] * A[i+1] > 0 ) { N++ ; i += 2 } else { i++ } ;

SELECTION : Not required

SELECTION : Not required

ENDING CONDITION : i = last ; return N ;

ENDING CONDITION : i = last ; return N ;

Other solution :

Other solution

add possitives -> whale division by 2 } add

add possitives -> whale division by 2 } add

add negatives ->

add negatives ->

# Array III

- Design an algorithm to choose the maximum number of pairs such that their product is negative, from an increasingly ordered array of whole numbers.
- Each element from the array can contribute at most to one of the pairs.

CANDIDATES : pairs of indexes in the array

ORDER : increasing  i = 0 ; j = last ; N = 0

FEASIBILITY : if (A[i] * A[j] < 0) { N++ ; j-- ; } else { i++ };

SELECTION : Not required

ENDING CONDITION : i = last ; return N ; A[i] * A[j] >= 0

# Array IV

- Design an algorithm to choose the maximum number of pairs such that their product is zero, from an increasingly ordered array of whole numbers.
- Each element from the array can contribute at most to one of the pairs.

CANDIDATES :

ORDER :

FEASIBILITY : # 0 in the array

min ( # 0 , half of the array )

# Closest average subsets

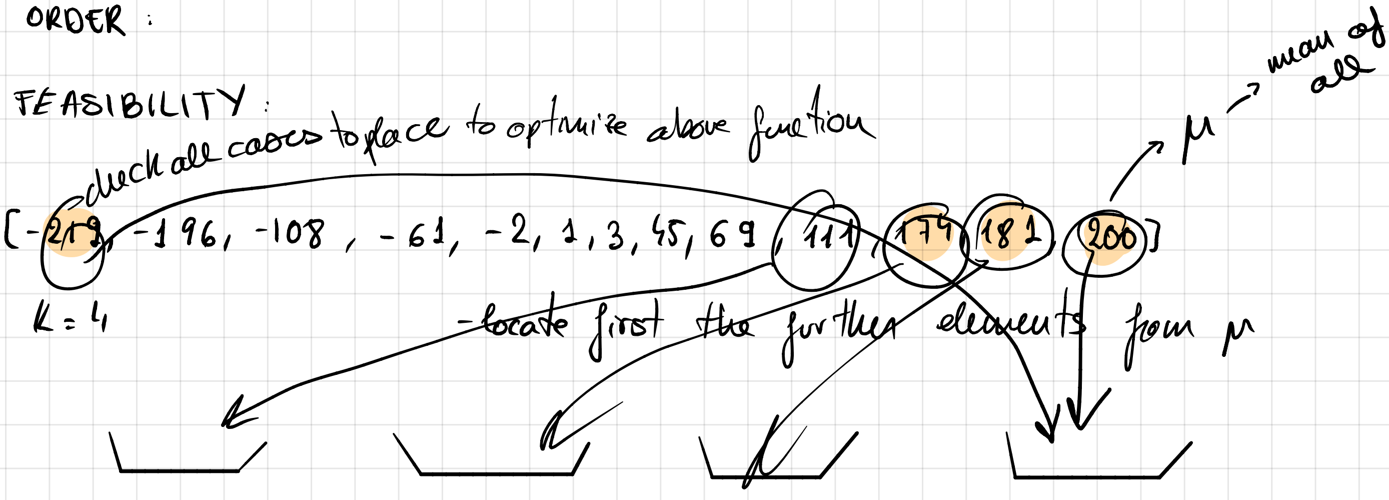
- Given a set D of N numbers and K sets (N > K), design an algorithm to distribute the elements of D into the K sets, such that, the average value of these K sets are as close as possible. The seeking sets $S_i$, $1 \leq i \leq k$ must be non-empty, i.e., they are required to provide a disjoint cover of D.
  - Minimize the value:

$$Z(s) = \sum_{i=1}^{k} \left| \overset{\text{set K}}{\mu_i} - \overset{\text{all elements in set D}}{\mu} \right|$$

CANDIDATES :

ORDER :

FEASIBILITY :



- check all cases to place to optimize above function
μ → mean of all
[-219, -196, -108, -61, -2, 1, 3, 45, 69, 111, 174, 181, 200]
k = 4
- locate first the further elements from μ

Mon 25/3/19

# Courses registration

- A company offers a range of courses for its employees.
- Each course only can be completed within a given set of days that are not required to be sequential.
- Every employee has at his disposal an interval of days in order to complete his formation.
- Each employee has already chosen a set of courses, but not always is possible for each employee to complete the courses has already registered.
- Design an algorithm to check this last scenario.

Example Info:

Course 1: 2 - 11 ; 4 days;<br>
Course 4: 3 - 5 ; 3 days;<br>
Course 5: 2 - 7 ; 3 days;

} not required to be consecutive

student's availability: 1 - ~~10~~ 12 days ~~NO~~ YES

CANDIDATES: days of disposal of the student

ORDER: decreasing priorities of courses depending on days available and days left.

FEASIBILITY: for each day taking into account the number of days left for each course, choose the one w/ higher priority.

SELECTION: ———

ENDING CONDITION: check all days

# Unit 3

## DYNAMIC PROGRAMMING

# USE NEEDS

- Some cases where the solution can be reached just deciding the inclusion of elements from the set of candidates, can’t be done in a serial form without looking forward to the next possible scenarios as concecuences of this one.
- This is mainly because the feasibility function grows when trying to take into account all the possible future evolutions, so getting into a complexity bigger than constant

# Brief description

- The set of candidates is sorted in a range of ordered subsets (conforming stages) such that each of them is strictly included in the next, and strictly includes the previous, as follows:
    - The first is the empty set.
    - Each differs from the previous and from the next in just a single element (candidate).
    - The last conforms the whole set of candidates.

# Principle of Optimality I

- Dynamic Programming algorithms decide the inclusion of candidates, in the order determined by the stages, for any input value (column).
- Such sequence of including decisions follows the Bellman´s Optimality Principle:
  - Each row/stage assumes the previous one as correct/optimal
  - The inclusion of the new candidate of the present subset (row or stage) is taken according to the function to be optimized.

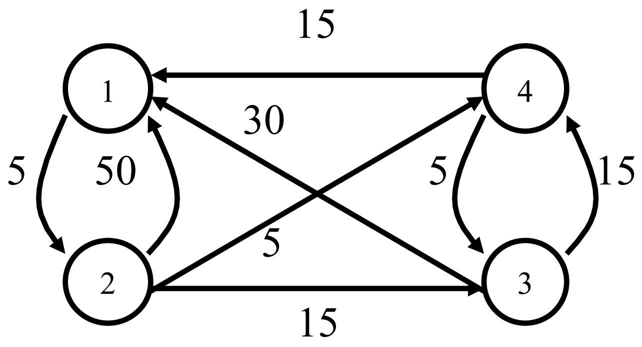
# Principle of Optimality II

- A problem is said to satisfy the Bellman's Principle of Optimality if the subsolutions of an optimal solution are themselves optimal solutions for their subproblems.
- A problem is considered as fully solved when the set of accepted candidates (solution) can be identified from the value of the solution provided by the algorithm (usually in a cell of the last row).

# How does it work?

1. Define a mathematical notation that can express the solution and subsolutions of the problem, usually the value to be optimized.
2. Develop a recurrence relation that relates a solution to its subsolutions, according to the mathematical notation previously defined.
3. Find out the value of the function to be optimized in the immediate cases that fits with such recurrence
4. Explain how from the values of the last stage, the solution set can be found

# Floyd’s Algorithm: All-Pairs Shortest Path Problem



- It computes the shortest paths between every ordered pair of nodes in a negative-length cycles free, weighted directed graph.
- The set of candidates is formed of tentative intermediate nodes to be included in the optimum paths.
- Optimality: “Node **i** is included in the optimum path **j->k** if and only if **(j->i + i->k)** is less than **j->k** before had considered node **i** as possible node to pass through”

| paths<br>interm. points | 1 -> 2 | 1 -> 3 | 1 -> 4 | 2 -> 1 | 2 -> 3 | 2 -> 4 | 3 -> 1 | 3 -> 2 | 3 -> 4 | 4 -> 1 | 4 -> 2 | 4 -> 3 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ∅ | 5 | no way<br>∞ | ∞ | 50 | 15 | 5 | 30 | ∞ | 15 | 15 | ∞ | 5 |
| {1} | 5 | ∞ | ∞ | 50 | ~~50+15~~<br>15 | ~~50+5~~<br>5 | 30 | 30+5<br>35 | 15 | 15 | 20 | 5 |
| {1, 2} | 5 | 5+15<br>20 | 10 | 50 | 15 | 5<br>same | 30 | 35 | 15 | 15<br>same | 20 | 5 |
| {1, 2, 3} | 5 | 20 | 10 | 45 | 15 | 5 | 30 | 35 | 15 | 15 | 20 | 5 |
| {1, 2, 3, 4} | 5 | 15 | 10 | 5+15<br>20 | 5+5<br>10 | 5 | 30 | 35 | 15 | 15 | 20 | 5 |

$M(i, \overset{j \to k}{j, k})$ = minimum distance from "j" to "k" having as posible intermediate nodes set "i"

$$M(i, j, k) = \begin{cases} \text{graph} & (i == 0) \\ \min\{ M(i-1, j, k), M(i-1, j, i^*) + M(i-1, i^*, k) \} & (\text{o.w}) \end{cases}$$

| Capacity | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ∅ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $i_1$ $i=1$ | 0 | 0 | 0 | | | | | | | | | | | | | |
| $i_1, i_2$ $i=2$ | | | | | | | | | | | | | | | | |
| $i_1, i_2, i_3$ | | | | | | | | | | | | | | | | |
| ⋮ | | | | | | | | | | | | | | | | |
| {set} | | | | | | | | | | | | | | | | |

M(i, j) = maximum benefit carried in a knapsack of capacity "j" with items from set "i"

$$M(i,j) = \begin{cases} 0 & i == 0 \\ M(i-1,j) & j < m(i^{*}) \text{ new element of this row with respect to the previous} \\ \max\left(M(i-1,j),\ b(i^{*}) + M(i-1,\ j - \underbrace{m(i^{*})}_{\text{mass}})\right) \end{cases}$$

↳ Bellman's optimallity principle

| items | mass | benefit |
|---|---|---|
| $i_1$ | 4 | 13 |
| $i_2$ | 2 | 6 |
| $i_3$ | 3 | 8 |
| $i_4$ | 3 | 7 |
| $i_5$ | 6 | 15 |

| capacity | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ∅ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $i_1$ $i=1$ | 0 | 0 | 0 | 0 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 |
| $i_1, i_2$ $i=2$ | 0 | 0 | 6 | 6 | 13 | 13 | 19 | 19 | 19 | 19 | 19 | 19 | 19 | 19 | 19 | 19 |
| $i_1, i_2, i_3$ | 0 | 0 | 6 | 8 | 13 | 14 | 19 | 21 | 21 | 27 | 27 | 27 | 27 | 27 | 27 | 27 |
| $i_1, i_2, i_3, i_4$ | 0 | 0 | 6 | 8 | 13 | 14 | 19 | 21 | 21 | 27 | 28 | 28 | 34 | 34 | 34 | 34 |
| {set} | 0 | 0 | 6 | 8 | 13 | 14 | 19 | 21 | 21 | 27 | 28 | 29 | 34 | 36 | 36 | 42 |

4

2

mass 3

mass 6

$i_5$

# Memory storage optimization b)

| Capacity | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ∅ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| $i_1$ $i=1$ | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| $i_1, i_2$ $i=2$ | | | | | | | | | | | | | | | | |
| $i_1, i_2, i_3$ | | | | | | | | | | | | | | | | |
| $i$ $i_1, i_2, i_3, i_4$ | | | | | | | | | | | | | | | | |
| $1, i_2, i_3, i_4, i_5$ | | | | | | | | | | | | | | | | |

| items | Petabytes |
|---|---|
| $i_1$ | 4 |
| $i_2$ | 2 |
| $i_3$ | 3 |
| $i_4$ | 3 |
| $i_5$ | 6 |

$$M(i,j) = \begin{cases} 0 & i == j \\ M(i-1,j) & j < size(i^*) \\ Min\{M(i-1,j),\ M(i-1,\ j-size(i^*))) & \hookrightarrow o.w. \end{cases}$$

$M(i,j)$ = minimum non-used space in a USB of capacity "j" with files from "i"

# Unit 3

## DYNAMIC PROGRAMMING ALGORITHMS: SOME PROBLEMS

# Supermarket cashier

Implement an algorithm that solves the problem of change by means of the minimum number of coins:

- Random values for currency system

M(i, j) = minimum # coins from set "i" to get a return of "j"

$$M(i,j) = \begin{cases} j & (i == 0) \\ M(i-1, j) & (\text{value}(i^*) > j) \\ \text{Min}\{ M(i-1, j), \underset{\text{include it}}{1} + M(\overset{\text{currificación}}{i}, j - \text{value}(i^*)) & \text{o.w.} \end{cases}$$

* As many coins of value X as you require.

| capacity | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| {1} | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| {1,2} $i=1$ | 0 | 1 | 1 | 2 | 2 | 3 | 3 | 4 | 4 | 5 |
| {1,2,4} $i=2$ | 0 | 1 | 1 | 2 | 1 | 2 | 2 | 3 | 2 | 3 |
| {1,2,4,5} | 0 | 1 | 1 | 2 | 1 | 1 | 2 | 2 | 2 | 2 |

not the same

same

→ same line

no coin of value 5

1, 2, 4, 5

# Memory storage optimization

- Given a set of n files of sizes $s_1$, $s_2$, ..., $s_n$, and an external storage device with capacity d (d <= $s_1$ + ... + $s_n$). Provide two algorithms that minimize:
  - The number of files to be moved to the storage device
  - The amount of non-used space in the storage device

… until the device comes full (files cannot be divided).

# Memory storage optimization a)

$M(i, j)$ : minimize # files from set "$i$" to get full a "$j$"-capacity disk.

$$M(i,j) = \begin{cases} j & (i == 0) \\ \underset{\text{[discard]}}{M(i-1, j)} & (\text{size}(i^*) > j) \\ \text{Min}\{\underset{\text{[discard]}}{M(i-1, j)},\ \underset{\hookrightarrow \text{we have include it}}{1 + M(i-1, \overbrace{j - \text{size}(i^*)}^{\text{new capacity}})}\} & (\text{o. w.}) \end{cases}$$

| capacity | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| $\emptyset$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| $i_1$ $i=1$ | 0 | 1 | 2 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| $i_1, i_2$ $i=2$ | 0 | 1 | 2 | 1 | 1 | 2 | 3 | 2 | 3 | 4 |
| $i_1, i_2, i_3$ | 0 | 1 | 2 | 1 | 1 | 2 | 3 | 2 | 2 | 3 |

$s(i^*) = \text{size of (file)} == 1$

$f_1 = 3$

$f_2 = 4$

$f_3 = 4$

# Minimizing translating costs

- In a company that offers translating services, every pair of languages belonging from a set of 10 can be translated into the other.
- This company rents dictionaries at different  prices for every ordered pair of languages.
- Implement an algorithm that computes the minimum renting dictionaries cost to develop any possible translation.

# Knapsack / Rucksack problem 0/1

- Given a set of items, each with a mass and a value, determine the number of each item to be included in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible
- These items can’t be divided.

# Pairs of files to be stored

- Given an even number of files, **n**.
- They are named holding that files **f1** & **f2** weigh **k1** Megas each; Files **f3** & **f4** weigh **k2** Megas each,…).
- Provide an algorithm that minimize the amount of non-used space in the storage device when it comes full (files cannot be divided).

# Unit 3

## DYNAMIC PROGRAMMING ALGORITHMS: A PROBLEM WITH NON-BINARY INCLUSION

# Optimal Investment

- Given a quantity of money that can be invested in a range of banks in order to maximize the benefits.
- It is asked an algorithm to develop such investments. As an example, below can be found a table relating the amount invested in each bank, to the benefit provided.

invertir 1M$ y obtener 2M$

| Bank / Quantity | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| BSCH | 0 | 2 | 5 | 6 | 7 |
| BBVA | 0 | 1 | 3 | 6 | 7 |
| CCM | 0 | 1 | 4 | 5 | 8 |

- Problem can be solved deciding which candidate should be included.

- Candidates: banks (not binary)

- Maximize the benefit

M(i, j) = maximum benefit investing "j" M$ in banks belonging to set "i".

$$M(i,j) = \begin{cases} 0 & i == 0 \\ \text{Max}\left\{ \underbrace{M(i-1,j)}_{\text{benefit not included}}, \underbrace{\overset{\text{table}}{BC}(i^*,\overset{1M\$}{1})}_{\text{benefit included}} + M(i^*,j-1),\ B(i^*,2)+M(i-1,j-2),\ B(i^*,3)+M(i-1,j-3),\ B(i^*,4)+M(i-1,j-4) \right\} \end{cases}$$

previous case
( included
or
not included )

MORE THAN
2 CASES

Max

$$\forall K \in \{0, \dots, \min\{4, \overset{\text{upper bound}}{j}\}$$

$$\{ \overset{\text{tabla}}{BC(i^*, K)} + \overset{\text{fila anterior}}{M(i-1, j-K)} \}$$

3M to BSCH

3M to BBVA

12+8 (4M to 3* (CCM))

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| ~~∅~~ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| {BSCH} | 0 | 2 | 5 | 6 | 7 | 7 | 7 | 7 | 7 | 7 | 7 |
| {BSCH, BBVA} | 0 | 2 | 5 | 6 | 8 | 11 | 12 | 13 | 14 | 14 | 14 |
| {ALL} | 0 | 2 | 5 | 6 | 9 | 11 | 13 | 15 | 16 | 19 | 20 |

(knapsack)
if same
not included

jump back previous
column

( in this
case we have
5 previous options )

$$M(i, j) = \begin{cases} j & (i == 0) \quad ✓ \\ M(i-1, j) & (\text{weight}(i^*) > j) \\ \text{Max}\left\{ M(i-1, j), 1 + M(i-1, j - \text{weight}(i^*)) \right\} & \text{O.W.} \end{cases}$$

$$M(i, j) = \begin{cases} j & i == 0 \\ M(i-1, j) & \text{weight}(i^*) > j \\ \text{MIN}\left\{ M(i-1, j), M(i-1, j - \text{weight}(i^*)) \right\} & \text{o.w.} \end{cases}$$

# Unit 4

Backtracking Algorithms

# Why should it be used?

- Motivation:
  - The last option
  - Although it is not the only option, other algorithm designs are extremely complex to be described or extremely expensive from the computational point of view.

# Brief description

- All the possible options has to be checked until the solution is found.
- It is equivalent to build a searching tree to be explored according to a depth-first policy. Actually, a one-dimensional tuple will be used to seek for the solution.
- When the branch is identified as fruitless, it is pruned → DEAD NODE

# Key Elements

- Solution is encoded as a one-dimensional tuple/array
- Potentially all the tuple values must be checked, i, e, it is an exhaustive searching process.
- It is required to identify every type of node:

1. Dead Node
2. Live Node
3. Solution Node

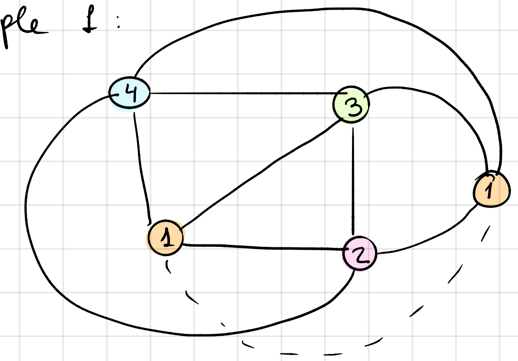
# Unit 4

## BACKTRACKING ALGORITHMS: SOME PROBLEMS

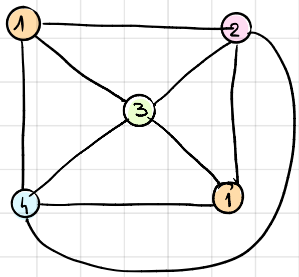
# Colored Graphs

- Given a non-directed graph G, we look for an algorithm to associate a color to each node such that every pair of adjacent nodes have different colors.
- Every political map corresponds with a plain* non-directed graph (country~vertex, border~edge). It can be coloured using at most four colors.
- *A plain graph can be drawn en 2D without crossing edges.

Example 1:



Example 2:



- Bellman does not hold.
- No ordination of candidates.

=> Backtracking: checking all possible solutions.

- one dimensional array

{ C: C[i] == j means
i-node takes color j

---

DATA: 1 dimensional array C: C[i] == j means i-node takes color j.

EXHAUSTIVITY: 0 --> 1 --> 2 --> 3 --> 4 --> 5

state in which we have not decided so far the colour

look for another path / posible solution (backtraking)

(1, [illegible], [illegible], [illegible], [illegible])

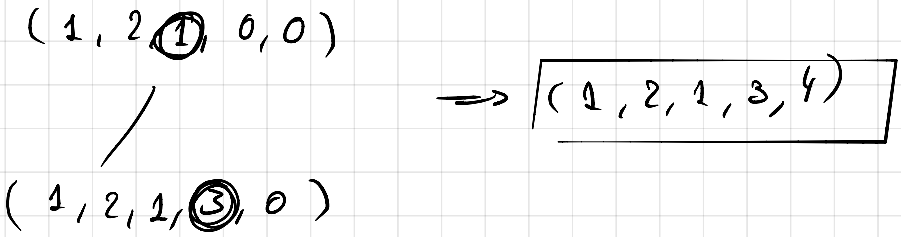
check 1 again
EXHAUSTIVITY

(0, 0, 0, 0, 0) live node

(1, 1, 0, 0, 0) dead node
-no possibility of getting a solution.

- Increase the value of a possition when you arrive and then check whether the restriction is fulfilled.

(1, 2, 0, 0, 0) (1, 3, 0, 0, 0) (1, 4, 0, 0, 0)



NODES: alive    else

dead    for (k = 0; i-1; k++) ((C[k] == C[i] &&

b.t.    A[i, k] == 1)

solution    (i == n) & LIVE-NODE

# A partition in two subsets of equal sum

- Given a set of **n** integers, we wonder if it can be found a partition of it into 2 subsets which sum the same.

# All the subsets of sum S

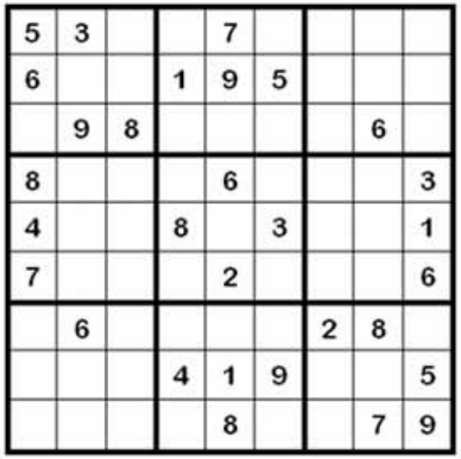
- Given a set of **n** integer numbers, it is asked to design an algorithm that identifies all its subsets which sum equals **S**.

# Chessboard

- How to locate **n** queens which don't threaten each other
- How to fully traverse it by a horse in **nxn** jumps
- Laberinth*

# Sudoku

- The objective is to fill a 9×9 grid with digits so that each column, each row, and each of the nine 3×3 sub-grids that compose the grid (also called "boxes", "blocks", "regions", or "sub-squares") contains all of the digits from 1 to 9. The puzzle setter provides a partially completed grid, which for a well-posed puzzle has a unique solution.
- Design an algorithm to solve them.



| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 5 | 3 | | | 7 | | | | |
| 6 | | | 1 | 9 | 5 | | | |
| | 9 | 8 | | | | | 6 | |
| 8 | | | | 6 | | | | 3 |
| 4 | | | 8 | | 3 | | | 1 |
| 7 | | | | 2 | | | | 6 |
| | 6 | | | | | 2 | 8 | |
| | | | 4 | 1 | 9 | | | 5 |
| | | | | 8 | | | 7 | 9 |

# Sequential Registered Courses

- A company offers a range of courses for its employees.
- Each course only can be completed within a given set of sequential days
- Every employee has at his disposal an interval of days in order to complete his formation.
- Each employee has already chosen a set of courses, but not always is possible for each employee to complete the courses has already registered.
- Design an algorithm to check this last scenario.

# Backtracking

General algorithm for finding all (or some) solutions to some computational problems, notably constraint satisfaction problems, that incrementally builds candidates to the solutions, and abandons a candidate ("backtracks") as soon as it determines that the candidate cannot possibly be completed to a valid solution.

It enumerates a set of partial candidates that, in principle, could be completed in various ways to give all the possible solutions to the given problem. The completion is done incrementally, by a sequence of candidate extension steps.

The partial candidates are represented as the nodes of a tree structure, the potential search tree.
The backtracking algorithm traverses this search tree recursively, from root down, in depth-first order; and checks whether each node can be competed to a valid solution. If it cannot, the whole sub-tree is rotated at that node which is skipped (pruned).

otherwise, the algorithm checks if the node itself is a valid solution, reports the user and recursively enumerates all sub-trees of that node.

The total cost of the algorithm is the number of nodes of the actual tree times the cost of obtaining and processing each node.