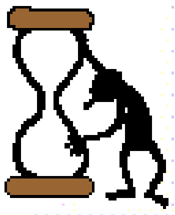
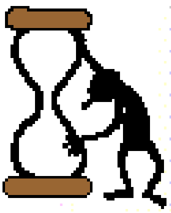


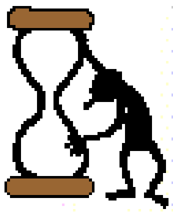
Algoritmoen analisia

Algoritmoen konplexutasun neurketa



Aurkibidea

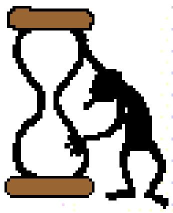
- ***Zergatik da beharrezkoa ?***
- ***Nola neurtu exekuzio denbora ?***
  - Zeren menpe dago
  - Nola kalkulatu
    - Esperimentalki
    - Matematikoki neurtuz
- ***Algoritmo analisia***
  - Oinarrizko eragiketak
  - Notazio asintotikoa
- ***Analisi asintotikoaren murriztapenak***



Konplexutasuna

Zergatik da beharrezkoa bere analisia?

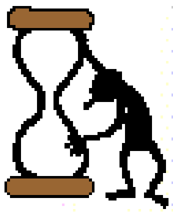
- Algoritmo bat garatu eta zuzena den frogatu ondoren.
- Bere **konplexutasun konputazionala** zehaztu behar da (exekuziorako behar dituen baliabideak)
- ***Nola neurtu?***
  - Exekuzio denbora
  - Tokia
    - Memorian
    - Diskoan



Konplexutasuna

Nola neurtu exekuzio denbora ?

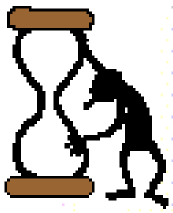
- Zer neurtzen da? Exekuzio denbora
- Zeren menpe dago?
  - Sarrera tamainaren arabera
  - Beste faktore (Hw & Sw)
    - Ordenadorearen abiadura
    - Konpiladoreen kalitatea
    - Programaren kalitatea
- Nola neurtu?
  - Esperimentalki
  - Matematikoki estimatuz



Konplexutasuna

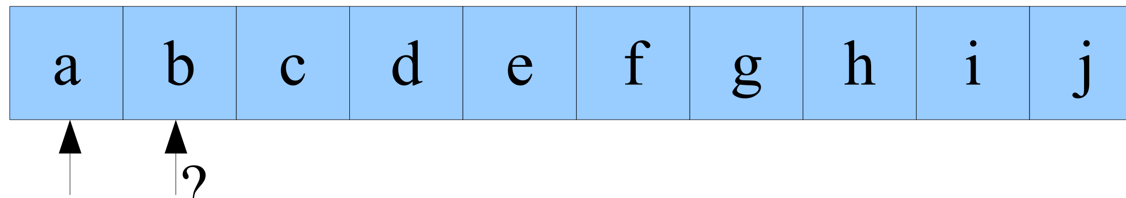
Nola neurtu? Esperimentalki

- Exekuzio denbora neurtuz sarrerako tamainaren arabera
  - Ezin dezakegu sarrera guztiak frogatu
  - Algoritmoa inplementatzea beharrezkoa da
  - Software eta Hardware-n menpe dago
- Beste neurri bat behar dugu
  - Abstraktuagoa
  - Errazagoa lortzeko



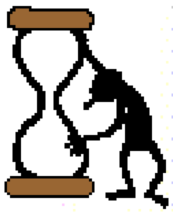
Algoritmoen konplexutasuna

Demagun array baten lehenengo eta bigarren elementuak berdinak diren kalkulatzeko algoritmo bat dugula.



* Zenbat konparaketa behar ditugu arrayak 10 elementu baditu?

* Zenbat konparaketa behar ditugu arrayak 100 elementu baditu?

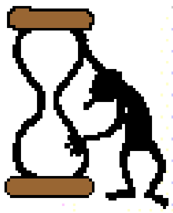


Algoritmoen konplexutasuna

Algoritmoa, array neurriarekiko independentea da

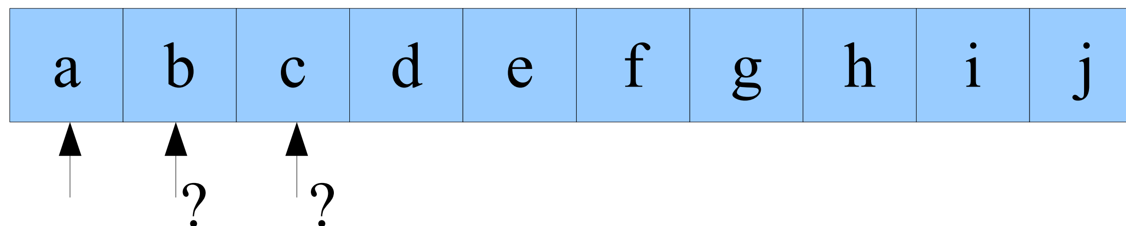
Algoritmoaren konplexutasuna O(1)ean dago
(algoritmoaren hazkunde abiadura konstantea da)

Egin behar den konparaketa kopurua konstantea da
(ez dugu konparaketa gehiago egin behar array-an elementu gehiago izateagatik)

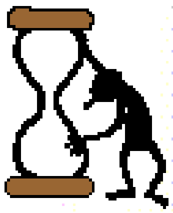


Algoritmoen konplexutasuna

Array baten lehenengo elementua bigarren edo hirugarrenaren berdina den kalkulatzeko algoritmoa: O(1)

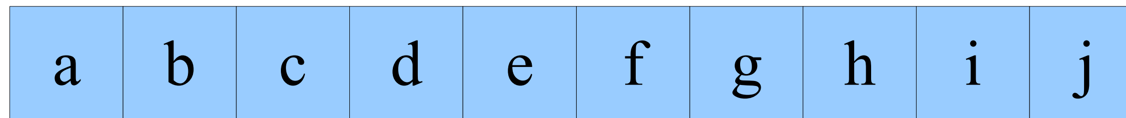


Egin behar den konparaketa kopurua konstantea da
(ez dugu konparaketa gehiago egin behar array-an elementu gehiago izateagatik)

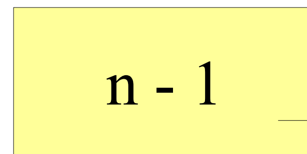


Algoritmoen konplexutasuna

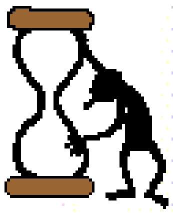
Array baten lehenengo elementua arrayan errepikatuta agertzen den edo ez kalkulatzeko algoritmo bat dugu.



* Kasu txarrenean zenbat konparaketa egin beharko ditugu?



zein da termino nagusia?
(n oso handia denean...)



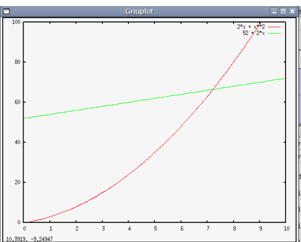
Konplexutasuna

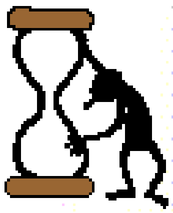
Nola neurtu?. Estimazio matematikoa

- Algoritmo bakoitzentzat, bere **sarreraren menpe dagoen funtzio bat** bilatzen da: f(n)

- Exekuzio denbora bezala **kasu okerrena** hartuko da.

- Gure helburua, sarrera neurriaren arabera, **algoritmoen hazkunde abiadura** lortzea da

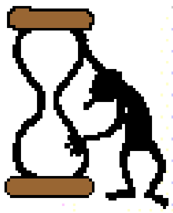




Konplexutasuna

Nola neurtu?. Estimazio matematikoa

- Denbora konstante
  - asignazio, metodo deialdi, erag. aritmetikoa, array index, idatzi/irakurri balio bat .....
- Iterazio bakoitzeko denbora
  - Zikloak (gehi baldintzaren ebaluazioa pauso bakoitzean)
- Denbora handiena
  - Baldintzazko eragiketatan
- Denbora osagarri
  - azpiprogramen deialdiak



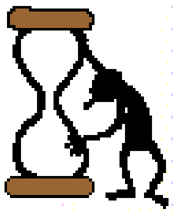
Konplexutasuna
Adibidea

- Suposatu hurrengo metodoa garatu nahi dela:

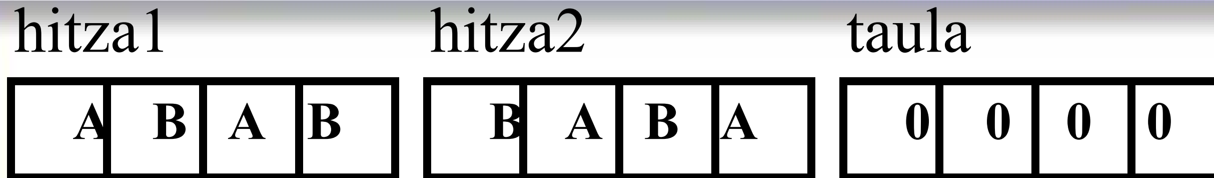
```
public boolean isAnagrama(Hitza h1, Hitza h2)
```

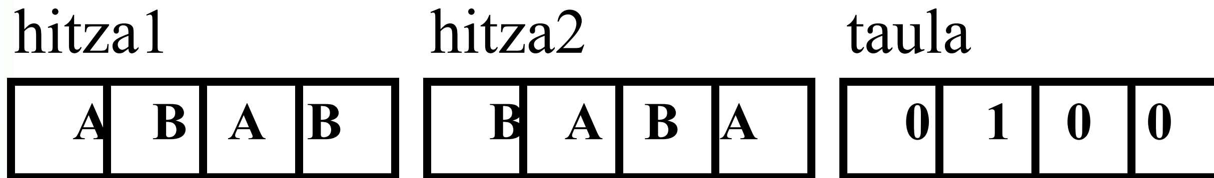
```
//aurre: h1 eta h2 hitz bat daukate letra minuskuletan
//post: true, h1eta h2  letra berdinak badauzkate.
```

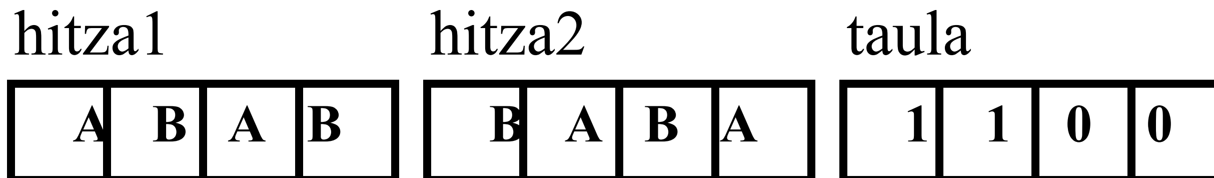
Adibide: h1="donostia" eta h2="itsaondo" anagramak dira

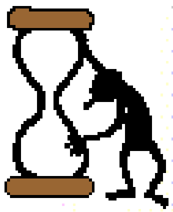


Adibidea : Anagrama? (I)

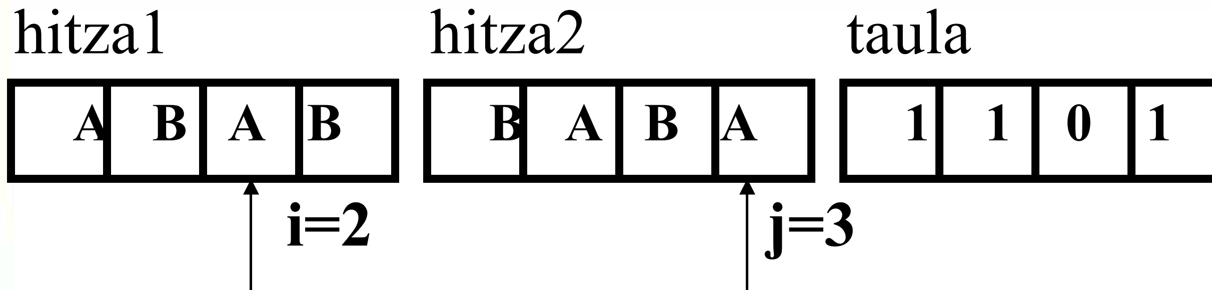


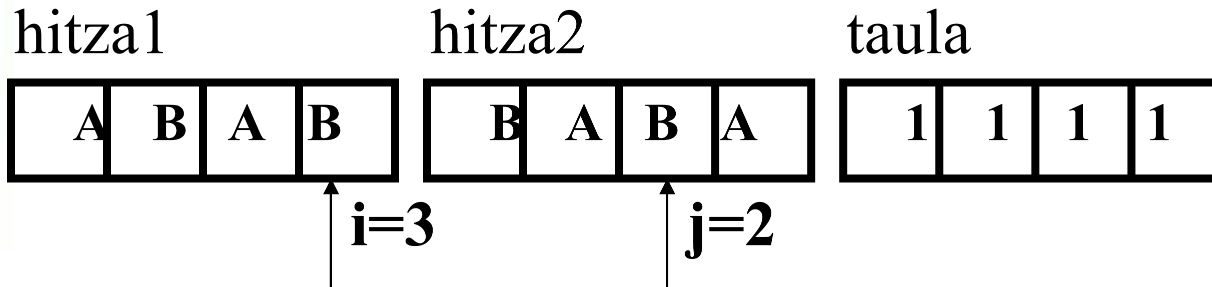






Adibidea: Anagrama? (I)





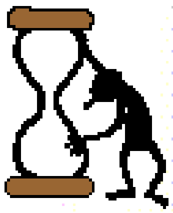
Balioa:

4 hasierazko marka

hitza1-en letra guztiak hitza2-an bilatu

4*4 konparaketa okerren kasuan

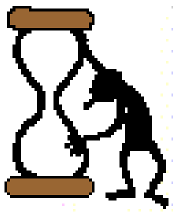
Ikusi 4 letrak markatuak izan diren (kasu okerrenean)



Anagrama(I)

Java programa

```
public static boolean isAnagrama(Hitza h1, Hitza h2) {
  char[] hitza1=h1.getWord();
  char[] hitza2=h2.getWord();
  int[] taula=new int[4]; int p,j;
  //Initialize visited positions
  for (int i=0;i<4;i++)
    taula[i]=0;
 for (int i=0;i<4;i++) {
   j=0;
   while ( (j<4) && !(hitza1[i]==hitza2[j] && taula[j]==0) )
    j++;
   if ( (j<4) && hitza1[i]==hitza2[j] && taula[j]==0 )
     taula[j]=1;
  }
  p=0;
  while ((p<4) && (taula[p]==1))
  p++;
  if (p==4) return true;
    else return false;
•       }
```



Adibidea: Anagrama? (II)

hitza1 hitza2

A	H	A	H

H	A	H	A

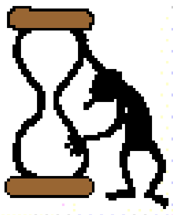
A	B	C	D	E	F	G	H	....	W	X	Y	Z
0	0	0	0	0	0	0	0	....	0	0	0	0

A	B	C	D	E	F	G	H	....	W	X	Y	Z
2	0	0	0	0	0	0	2	....	0	0	0	0

A	B	C	D	E	F	G	H	....	W	X	Y	Z
0	0	0	0	0	0	0	0	....	0	0	0	0

Balioa:

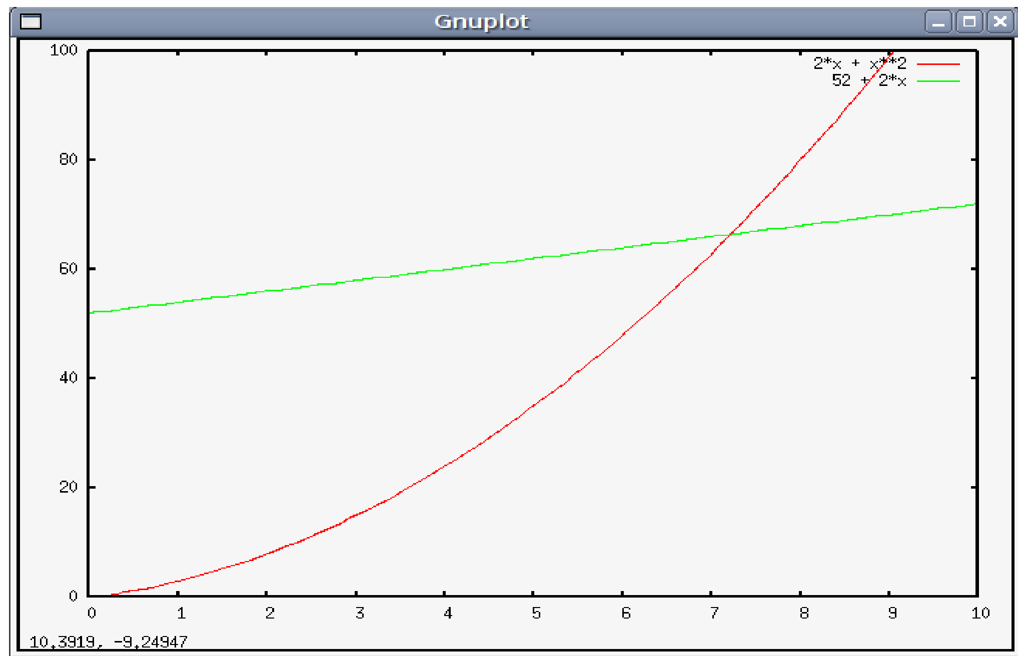
26 zero ipini
4 batuketa egin (hitza1)
4 kenketa egin (hitza2)
26 konprobaketa egin (okerren kasuan)

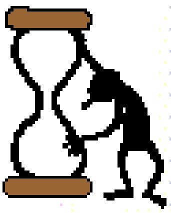


Algoritmoen konparaketa

Elementuak	Algoritmo 1	Algoritmo 2
4	4+4*4+4=24	26+4+4+26=60
n	$n+n*n+n=2n+n^2$	26+n+n+26=52+2n

N handia denean bigarren algoritmoa lehenengoa baina hobeagoa da.

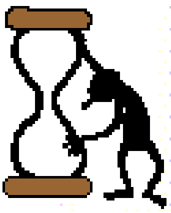




O Notazioa

Nola adierazten da algoritmo baten exekuzio-denboraren balioa

- Exekuzio-denbora nola igotzen den tamainaren araberako funtzioen sailkapena eginez.
  - $t(n) = 2n+n^2$
  - $t(n) = 52+2n$
  - $t(n) = 2n^2/5+6n+3\pi \log_2 n+2$



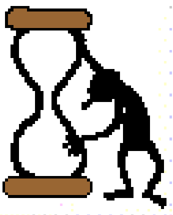
O Notazioa

Nola adierazten da algoritmo baten exekuzio-denboraren balioa

– f(n) funtzioaren konplexutasun ordena: O(f)

- f funtzio bat emanda $f:\mathbb{N}\rightarrow\mathbb{R}^+$, f-ren ordena, funtzio multzo bat da, non n zenbaki batetik aurrera, funtzio multzo hori akotatuta dago f funtzioaren multiplo positibo batengatik.

$$O(f)=\{g:\mathbb{N}\rightarrow\mathbb{R}+/\exists c\in\mathbb{R}+,\exists n0\in\mathbb{N},\forall n\geq n0:g(n)\leq c\cdot f(n)\}$$

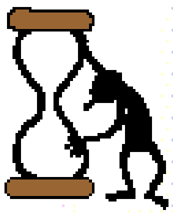


Konplexutasuna

O Notazioa

g(n) ϵO(f(n)) bsb edozein n>=n_0 g(n)<=c.f(n)

$$O(f)=\{g:\mathbb{N}\rightarrow\mathbb{R}+/\exists c\in\mathbb{R}+,\exists n0\in\mathbb{N},\forall n\geq n0:g(n)\leq c\cdot f(n)\}$$



O Notazioa

Gnuplot

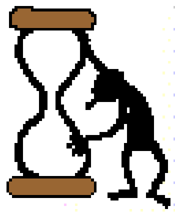
Gnuplot

log(x)
x
x*log(x)
x**2
x**3
2**x

2500
2000
1500
1000
500
0

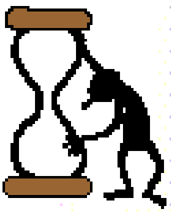
0 1 2 3 4 5 6 7 8 9 10 11

11.3600, 260.967



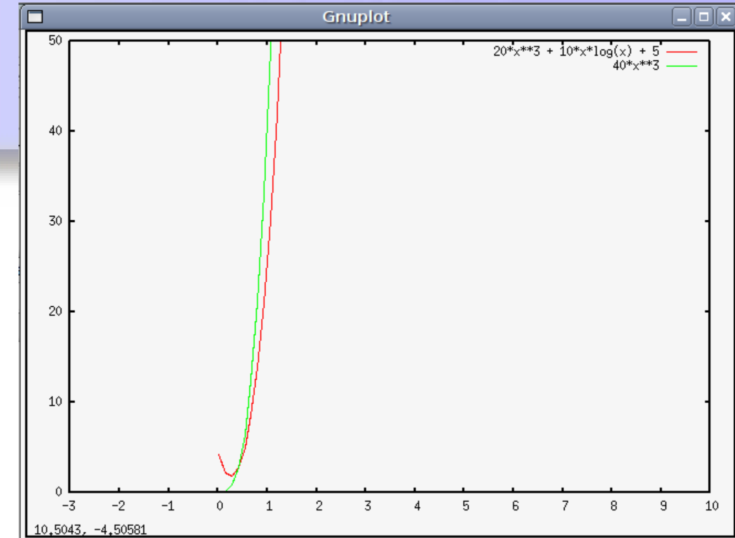
Funtzioen haziera

log n	√n	n	n log n	n^2	n^3	2^n
1	1,4	2	2	4	8	4
2	2,0	4	8	16	64	16
3	2,8	8	24	64	512	256
4	4,0	16	64	256	4.096	65.536
5	5,7	32	160	1.024	32.768	4.294.967.296
6	8,0	64	384	4.096	262.144	$1{,}8 * 10^{19}$
7	11,0	128	896	16.536	2.097.152	$3{,}4 * 10^{38}$

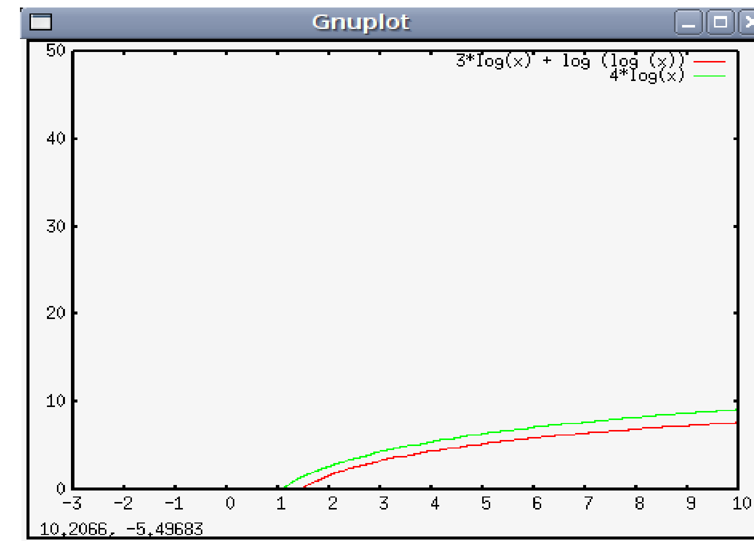


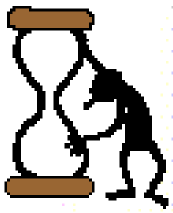
Konplexutasuna

O Notazioa



- 7n-3 ∈ O(n)
  - c=7, n0=1
  - 7n-3 <= 7n
- 20n3+10nlogn+5 ∈ O(n3)
- 3 logn+loglogn ∈ O(log n)
- 2^{100} ∈ O(1)
- 5/n ∈ O(1/n)

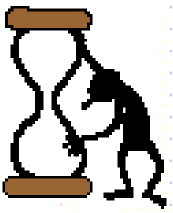




Definizioak

– $20n3+10nlogn+5 \in O(n3)$

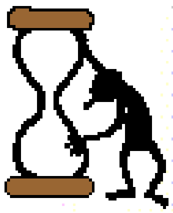
- Funtzio baten **termino nagusia**, haziketa abiadura handien duen terminoa da
- Funtzio baten **haziketa abiadura**, bere termino nagusien menpe dago, beste elementu guztiak deuseztatuz.
- Funtzio baten **ordena** bere haziketa abiadurarekin erlazionatuta dago, beraz termino nagusia aztertu beharko da soilik.



Adibidea I

- Array baten elementu handiena lortu
- Sarrera: A Array bat n elementu (osoak)
- Irteera: A arrayaren elementu handiena
- Algoritmo:arrayMax(A,n)

```
current=A[0];
for(i=1;i<n;i++)
    if (A[i]>current) current=A[i]
return current
```

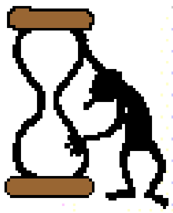


Adibidea I

Ebazpena

```
current=A[0];
for(i=1;i<n;i++)
    if (A[i]>current) current=A[i]
return current
```

Hoberen kasuan=2+4*(n-1)+1=4n-1
Okerren kasuan=2+6*(n-1)+1=6n-3

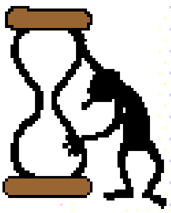


Adibidea II

- A array bat, n elementu osoak
- Kalkulatu beste array B, non:

$$B[i] = \sum_{j=0}^{i} A[j]$$

A[i] sekuentzia hartuta, beste sekuentzia bat lortzen da non bigarren taularen elementu j bakoitza lehenengo taularen 0 elementutik j-elementura lortzen den batura balioa gordetzen da.



Adibidea II

Ebazpena I

```
for (i=0;i<n;i++) {
        b=0;
        for(j=0;j<=i;j++)
                b=b+A[j]
        B[i]=b;
}
return B
```

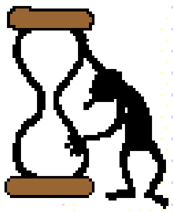
Pausoak:

Hasieratu eta taula itzuli :O(n)

i begizta: n aldiz exekutatzen da: O(n)

j begizta: 1+2+3+...+n aldiz exekutatzen da=(1+n)n/2 : $O(n^2)$

Totala: $O(n)+O(n)+O(n^2) \in O(n^2)$

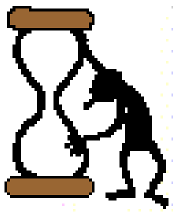


Adibidea II

Ebazpena II

- B[i]=A[0]+.......+A[i-1]+A[i]
- B[i]=B[i-1]+A[i]

Aurreko terminoan egin ditugun eragiketak erabiltzen ditugu algoritmoaren konplexutasuna murrizteko



Adibidea II
Ebazpena II

```
b=0;
B[0]=A[0]
for(i=1;i<n;i++)
     B[i]=B[i-1]+A[i]
return B
```

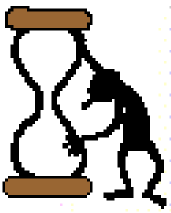
Zatiak:

Hasieratu eta taula itzuli:O(n)

Hasierako eragiketak:O(1)

i begizta: n aldiz exekutatzen da

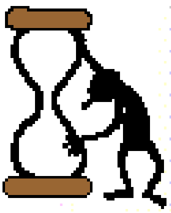
Totala: $O(n)+O(1)+O(n) \in O(n)$ (Orden lineala)



Adibidea III-A

- Elementu bat array batean bilatzen du
- Sarrera: A Array bat, n elementurekin eta e osoko bat
- Irteera: true bsb e ∈ A, false beste kasuan
- Algoritmo:dago(A,n,e)

```
for(i=0;i<n;i++)
    if (A[i]==e) return true;
return false
```



Adibidea III-B

- Elementu bat array batean bilatzen du
- Sarrera: A Array ordenatu bat, n elementurekin eta e osoko bat
- Irteera: true bsb $e \in A$, false beste kasuan
- Algoritmo:dago(A,n,e)

```
int l=0;
int a=n-1;
while (l<=a) {
        int i=(l+a)/2;
        if (A[i]==e) return true;
    else { if (A[i] > e) a=i-1;
            else l=i+1;}}
return false;
```

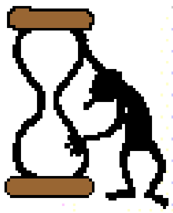
Zenbat aldiz exekutatzen da while begizta?

$n, n/2, n/4, \ldots, n/2^i$

Kasu okerrenean: $2^i = n$

Orduan: $i = \log n$

$O(\log N)$



Konplexutasuna

Analisi asintotikoaren murriztapenak

- **Elementu gutxi daudenean** ez da egokia
- analisi asintotikoa **goi-estimazio bat da**
- **Exekuzio denbora kasu erdian, okerrenen kasuan baino askoz txikiago izan daiteke**
- Batzuetan kasu okerrena ez da esanguratsua