Per ottenere dati più robusti e calcolare varianze di essi

$n = 10 \ldots\ldots \quad 100000$

se generate l'input prima
bla bla bla
Il Puppis dice di essere
(l'aggiunte un po' si nota)
≈ DavideCapra

Per generare gli input, è molto meglio generare 1000 input già pronti all'uso e nel momento in cui misuro il tempo medio, uso tali input già generati

Per quanto riguarda il c, bisogna cercare di allocare tutto subito e deallocare alla fine in quanto anche l'allocazione prende tempo

Per il for che fa in modo di far variare n nel tempo, non ha senso variare n nell'ordine dell'unità, bensì si cerca conversione di tipo geometrico (progressione geometrica)

Come fare? $N[j]$ con j da 1 a 1000 che segue una distribuzione esponenziale $\rightarrow A \cdot B^j$ e fare in modo tale che $n(1) = n_{min}$ e

$n(1000) = n_{max}$

$A \cdot B^1 = 100 = A = \frac{100}{B}$ $\quad A \cdot B^1 = n_{min}$ e $A \cdot B^{1000} = n_{max}$ e in questo

maniera ottengo i risultati desiderati

$B^{1000-1} = N_{max} / N_{min}$ quindi $\log B = \frac{\log N_{max} - \log N_{min}}{999}$

e $B = \exp \frac{(\log N_{max} - \log N_{min})}{999}$ e $A = \frac{N_{min}}{B}$

Quindi ottengo $n =$,

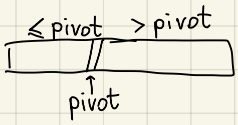
Lezione del 17/12

Selezione → var quickSort. Input array $A[0, \ldots, n-1]$;

posizione se ≥ 1 $\quad k \in [1, \ldots, n]$

$\frac{\text{naive}(A, n, k)}{\text{sort}(A, n)}$

return $A[k-1]$

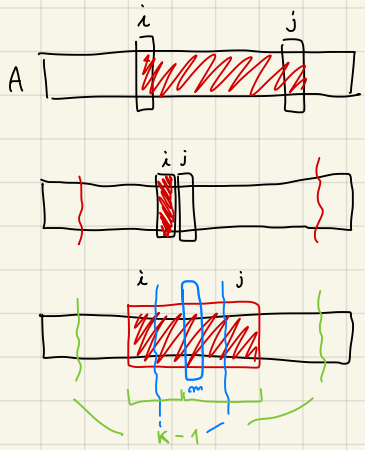


quicksort (A, i, j) base ≥ j - 1

m = partition

quicksort (A, i, m - 1)

quicksort (A, m+1, j)



quickselect (A, i, j)

if i ≥ j

return INT_MIN // limits.h

if i == j - 1

if K - 1 == i

return A[K-1]

else

return INT_MIN

if i < j

m = position (A, i, j)

quickselect (A, $^{0}_{i}$, $^{m}_{j}$)

copre anche il caso i ≥ j

if (K - 1 < i) OR (K - 1 ≥ j)

return INT_MIN

if (i == j - 1) AND (K - 1 == i)

return A[K-1]

// i < j AND

// i ≤ K AND K < j

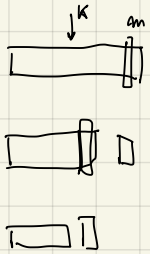
if K - 1 < m

return quickselect (A, i, m, K)

else

return quickselect (A, m, j, K)

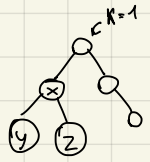
caso peggiore (complessità quadratica)



caso ottimo, divido sempre a metà

lineare

heapselect MIN - HEAP → BuildHeap → Θ(n)



x ≤ y

x ≤ z

heapselect (A, n, K)

buildheap (A, n)

for i = 1 ... K

r = extract min (A, h ...)

return r

se K < n

Θ(K log n + n)

log K

H_1 = buildheap (A)

r = get min (H)

H_2 = extract heap

for

e il prof non si ricorda l'algoritmo! 😉

↓

l'ha cambiato tipo 5 volte 😉

```
H1 = buildheap(A)
H2 = buildheap({A[0]})                    inizio H2 contiene 1 elem; passo dopo 2...
for i = 1...k
    r = extract(H2)
    if (modo sx di r in H1 esiste)
        insert(H2, modo sx di r)          O(log i)
    if (modo dx di r in H1 esiste)
        insert(H2, modo dx di r)
return r
```

```
heap select(A, n, k)
    build heap(A, n)
    int B[n]
    B[0] = &(A[0])
    m = 1
    for i = 1 .k
        int* r = extract(B, m)
        int* l = &(A[2(r-A)+1])
        if l-A < n
            insert2(B, m, l)
            m++
        if r-A < n
            insert2(B, m, r)
            m++
    return *r
```

```
heapify
    if x < y
    min
heapify2
    if *x < *y
```

"vite bonus" ad es implementando median of medians, senza allocare nuovi array (In Place)