# Esercizio 3

Florian Sabani 881665

L'algoritmo QuickSelect viene presentato con la funzione di riordinare parzialmente un array, in particolare :

Dopo l’esecuzione di QuickSelect(A, p, q, r) valgono le seguenti condizioni su A:

A[r] contiene l’elemento che si troverebbe in posizione r in A se il sottovettore A[p..q] fosse ordinato;

A[p..q] è partizionato rispetto al pivot A[r], precisamente:

A[i] < A[r] per ogni i tale che $p \le i < r$

A[r] < A[j] per ogni j tale che $r < j \le q$.

Per risolvere il primo punto basterebbe trovare in modo efficente r-1 numeri in A, tale che siano tutti minori dell'elemento in posizione r, in questo modo non sappiamo in che ordine vadano messi gli elementi negl'intervallo :

[p...r-1] e [r+1...q]

Ma sappiamo di per certo, che se quei due sotto-array fossero ordinati, l'array A sarebbe ordinato :

A[i] < A[r] per ogni i tale che $p \le i < r$

A[r] < A[j] per ogni j tale che $r < j \le q$.

in quanto tutti i numeri presenti nell'intervallo [p...r-1] sono minori di A[r], e tutti gli elementi presenti nell'intervallo [r+1...q] sono presenti tutti i numeri non inferiori ad A[r].

Bisogna quindi trovare una soluzione al problema : trova il r-esimo numero piu' piccolo, e nel mentre, partizioni l'array come precedentemente definito.

Una possibile soluzione consiste nell'utilizzo della funzione partition (utilizzata dal quicksort).

```c
#define swap(a,b) tmp=a;a=b;b=tmp;

int partition(int* arr, int l, int r){
    int x=arr[r];
    int i=l;
    int tmp; // usato dalla swap
    for(int j=l;j<=r-1;j++)
        if(arr[j]<=x){
            swap(arr[i],arr[j]);
            i++;
        }

    swap(arr[i], arr[r]);
    return i;
}
```

linguaggio : C

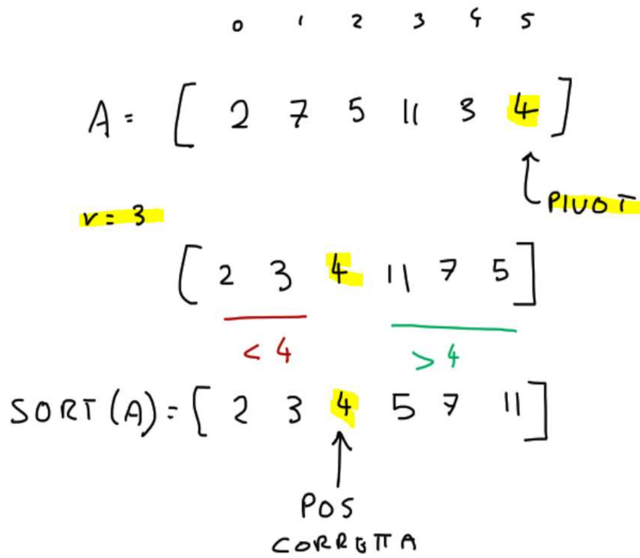
Dato un array, ed un intervallo l...r, usando come pivot l'elemento piu' a destra dell'intervallo (quindi in pos r), partition e' una funzione che si presta per swappare tutti gli elementi minori di A[r] alla "sinistra" dell'elemento presente in posizione A[r]. In conclusione (riga 13) imposta la posizione dell'elemento piu' a destra con i, che, coincide con il numero di elementi minori di A[r].

Ora ipotizziamo di avere il seguenti valori :

A = [ 2, 7, 5, 11, 3, 4 ]

r = 3 , p=0, q=5

In questo particolare e specifico caso, possiamo notare che il comportamento della funzione Partition e QuickSelect sarebbe il medesimo :




Notare che r e' 1-based, mentre gli array sono 0-based.

Quindi, nel fortunato caso in cui, l'elemento alla fine, sia esattamente il r-esimo numero piu' piccolo dell'array, eseguire partition, risolverebbe il problema del QuickSelect.
Nel caso in cui cio' non avvenga? Basta ripetere l'operazione, spostandosi nel sottoarray di destra o sinistra, in base all'indice restituito dal primo partition.

```c
int QuickSelect(int* arr,int p, int q,int r){
    if(r<0||r>q+1)return -1;  // se siamo fuori il range del vettore
    int index = partition(arr,p,q); // classico partition
    if(index==r-1) return index; // trovato il r-esimo numero piu piccolo
    if (index>r-1) return QuickSelect(arr,p,index-1,r); // sottoarray di sinistra
    return QuickSelect(arr, index+1,q,r); // sottoarray di destra
}
```

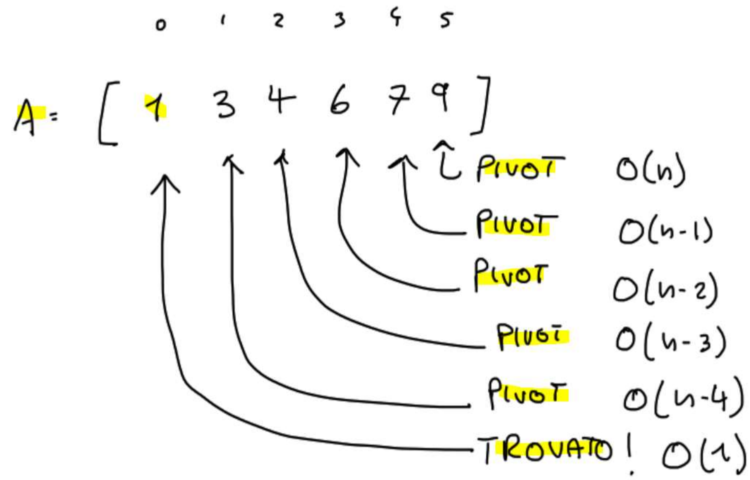
Linguaggio : C

Complessita' Temporale :

L.B. ci dice che QuickSelect(A, p, q, r) ha complessità O(log m), dove m = $q - p + 1$.

Ma l'ingegnere L.B. si sbaglia, infatti nel caso migliore, la funzione QuickSelect esegue partition una sola volta, senza ricorrere alla ricorsione, in quel caso la complessita' dipende dall'implementazione del partition, ossia O(n) lineare.

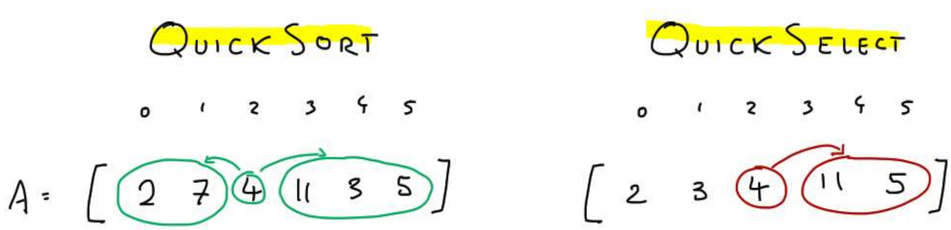
Il peggiore dei casi consiste invece nella presenza di un array gia' ordinato e r=1, quindi ci interessa che il numero piu' piccolo di un array gia’ ordinato sia posizionato all'inizio :




In questo caso partition viene eseguito N volte, fino a quando non arriviamo al primo elemento. Di conseguenza la complessita' del QuickSelect nel caso peggiore e' O(n*n)

Nel caso medio invece?

Nel caso medio, il range p-q si dimezza (circa) ogni volta, proprio come nell'algoritmo QuickSort, la differenza consiste nel fatto che ad ogni ricorsione, dopo aver trovato il pivot, non ricorriamo in entrambi i lati, ma solo nella parte che contiene il r-esimo numero piu' piccolo.




Questa differenza, ci permette di ridurre la complessita' media da O(n log n) a O(n).

## QUICK SORT

NB: * NEL QuickSort RICORRO SIA A SX CHE A DX

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

$a = 2$ $\quad g(x) = n$

$b = 2$

$d = \log_2 2 = 1$

CASO 2) $\quad n = \Theta(n^1)$

$f(n) \cong g(n)$

$$T(n) = \Theta(n^1 \cdot \log n)$$

## QUICK SELECT

NB: * NEL QUICK SELECT RICORRO SOLO IN UN LATO

$$T(n) = 1T\left(\frac{n}{2}\right) + O(n)$$

$a = 1$ $\quad g(x) = n^0 = 1$

$b = 2$

$d = \log_2 1 = 0$ $\quad$ 1) $n = \Omega(n^{-\varepsilon})$

CASO 3) $\quad$ 2) $\exists\, c < 1 : a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n)$

$f(n) > g(n)$

$$T(n) = \Theta(f(n)) = \Theta(n)$$

Ricapitolando quindi la complessita' temporale del QuickSelect implementato come abbiamo visto e' :
Caso Peggiore O(n*n)
Caso Medio THETA(n)
Caso Migliore O(n)

Quindi QuickSelect non puo' avere complessita' O(log(m)) con dove m = $q - p + 1$.

Un algoritmo efficente per ordinare un Vettore A puo' essere una versione del QuickSort, che come funzione di partizionamento usa QuickSelect. Se la funzione di partizionamento e' ottimale (quindi con complessita' lineare nel caso medio), allora anche l'algoritmo di ordinamento costruito con quella funzione e' efficente, quindi con complessita' O(n log n).

```
void QuickSelectSort(int* a,int l,int r){
    if(l>=r)return;
    int i = QuickSelect(a,l,r,r+1);
    QuickSelectSort(a,l,i-1);
    QuickSelectSort(a,i+1,r);
}
```

Quindi asintoticamente parlando, questo algoritmo di ordinamento che chiameremo QuickSelectSort, e il noto QuickSort, hanno complessita’ temporali medesime.

Ma in termini di tempo, i due algoritmi come si comportano per grandi N ?

| N | QuickSort | QuickSelectSort |
|---|---|---|
| 10 | 0 ms | 0 ms |
| 15 | 0 ms | 0 ms |
| 22 | 0 ms | 0 ms |
| 33 | 0 ms | 0 ms |
| 49 | 0 ms | 0 ms |
| 73 | 0 ms | 0 ms |
| 109 | 0 ms | 0 ms |
| 163 | 0 ms | 0 ms |
| 244 | 0 ms | 1 ms |
| 366 | 0 ms | 0 ms |
| 549 | 1 ms | 0 ms |
| 823 | 1 ms | 2 ms |
| 1234 | 2 ms | 2 ms |
| 1851 | 5 ms | 6 ms |
| 2776 | 15 ms | 14 ms |
| 4164 | 46 ms | 33 ms |
| 6246 | 106 ms | 73 ms |
| 9369 | 158 ms | 159 ms |
| 14053 | 399 ms | 579 ms |
| 21079 | 827 ms | 858 ms |
| 31618 | 2060 ms | 1904 ms |

Con A gia’ ordinato.

| N | QuickSort | QuickSelectSort |
|---|---|---|
| 10 | 0 ms | 0 ms |
| 15 | 0 ms | 0 ms |
| 22 | 0 ms | 0 ms |
| 33 | 0 ms | 0 ms |
| 49 | 0 ms | 0 ms |
| 73 | 0 ms | 0 ms |
| 109 | 0 ms | 0 ms |
| 163 | 0 ms | 0 ms |
| 244 | 0 ms | 0 ms |
| 366 | 0 ms | 0 ms |
| 549 | 0 ms | 0 ms |
| 823 | 0 ms | 2 ms |
| 1234 | 0 ms | 2 ms |
| 1851 | 0 ms | 5 ms |
| 2776 | 0 ms | 13 ms |
| 4164 | 0 ms | 31 ms |
| 6246 | 1 ms | 83 ms |
| 9369 | 1 ms | 170 ms |
| 14053 | 1 ms | 392 ms |
| 21079 | 2 ms | 805 ms |
| 31618 | 3 ms | 2062 ms |

Con A disordinato (randomicamente)

Come possiamo vedere i due algoritmi si comportano quasi ugualmente nel caso in cui A sia gia’ ordinato (quindi nel worst case di cui abbiamo parlato precedentemente).

Mentre abbiamo una netta differenza nel caso medio, differenza dovuta dal fatto che mentre QuickSort utilizza partition come metodo di partizione dei sotto-array, QuickSelectSort utilizza QuickSelect.

QuickSelect e partition hanno complessita’ temporali diverse, come abbiamo gia’ visto. E’ molto raro che QuickSelect esegua lo stesso numero di iterazioni di partition, in particolare la probabilita’ e’ 1/N con N il numero di elementi in A.

E’ chiaro quindi perche’ con l’aumentare di N, la differenza temporale dell’esecuzione dei due algoritmi aumenta.

Come possiamo quindi evitare questa mancanza di efficenza?

Impostando un r tale che il QuickSelect degradi in un partition.

```c
void QuickSelectSort2(int* a,int l,int r){
    if(l>=r)return;

    int i,rPos= l + 1; // e' 1-based

    for(i=l;i<r;i++)
        rPos+=a[i]<a[r]; // quanti num sono minori

    // O(n) cerco in che pos sarebbe l ultimo elem
    i = QuickSelect(a,l,r,rPos);
    QuickSelectSort2(a,l,i-1);
    QuickSelectSort2(a,i+1,r);
}
```

Linguaggio : C

A questo punto QuickSelect si occupera’ di eseguire partition una sola volta, scoprendo che la posizione dell’elemento piu’ a destra, se l’array fosse ordinato, coinciderebbe con l’indice “rPos” che e’ stato passato come parametro, quindi si limiterebbe a non eseguirsi in ricorsione, ma semplicemente restituire il risultato di partition.

| N | QuickSort | QuickSelectSort2 |
|---|---|---|
| 10 | 0 ms | 0 ms |
| 100 | 0 ms | 0 ms |
| 1000 | 0 ms | 1 ms |
| 10000 | 1 ms | 1 ms |
| 100000 | 10 ms | 14 ms |
| 1000000 | 171 ms | 238 ms |
| 10000000 | 7797 ms | 9698 ms |

In questo modo cio’ che differisce i due algoritmi e’ la presenza di un ciclo lineare in piu’ sul QuickSelectSort2 : scannerizzazione che non influisce la complessita’ ma influenza sicuramente il tempo di esecuzione.