Problema dell'ordinamento

DEF:

Dato un insieme S di n oggetti presi da un dominio totalmente ordinato, ordinare S. E' possibile effettuare ricerche in array ordinati in tempo $O(log(n))$ attraverso la **ricerca binaria**.

Algoritmi che ordinano in tempo $O(n^2)$

Sia n la dimensione dell'input, allora questi algoritmi ordinano in tempo $O(n^2)$

1. Selection Sort:

Sceglie sempre il minimo degli n-k elementi non ancora ordinati, mettendolo in posizione k.

```
SelectionSort(arr)
    for k = 1 to n-2 do
        min = k
        for j = k+1 to n-1 do
            if(arr[j] < arr[m]) then m = j
        scambia arr[m] con arr[k]
```

DEF (Invarianti):

Gli invarianti sono uno strumento utile per dimostrare la correttezza di un algoritmo, perché permette di isolare proprietà dell'algoritmo, spiegare il funzionamento, capire a fondo l'idea su cui si basa.

- i primi k elementi sono ordinati
- sono i k elementi più piccoli dell'array

Complessità, T(n) = $\theta(n^2)$

2. Insertion Sort:

Posiziona l'elemento k+1 - esimo nella pozione corretta rispetto ai primo k elementi

```
InsertionSort(arr)
    for k = 2 to n do
        while(k > 1)
            if(arr[k] < arr[k-1]) then
                scambia arr[k] con arr[k-1]
                decrementa k
```

Complessità Temporale: $T(n) = \theta(n^2)$, infatti in questo caso, in ogni iterazione il primo elemento della sottosequenze non ordinata viene confrontato solo con l'ultimo della sottosequenze ordinata. Il caso pessimo p invece quello in cui la sequenza di partenza sia ordinata al contrario. In questo caso, ogni iterazione dovrà scorrere e spostare ogni elemento della sottosequenza ordinata prima di poter inserire il primo elemento della sottosequenza non ordinata.

3. Bubble Sort:

Esegue n-1 scansioni. Ad ogni scansione guarda coppie di elementi adiacenti e li scambia se non sono nell'ordine corretto.

```
BubbleSort(arr)
    for k = 1 to n-2 do
        for j = 0 to n-1-k do
            if(arr[j] > arr[j+1]) then
                scambia arr[j] con arr[j+1]
```

Complessità Temporale: Il BubbleSort effettua all'incirca $\frac{n^2}{2}$ confronti ed $\frac{n^2}{2}$ scambi sia in media che nel caso peggiore. Il tempo di esecuzione è $\theta(n^2)$.

Algoritmi che ordinano in tempo meno che quadratico

1. Merge Sort

Usa la tecnica del divide et impera. (*Divide*) Dividi l'array a metà, risolvi i due sottoproblemi ricorsivamente e (*Impera*) fondi le due sottosequenze ordinate.

```
MergeSort(arr, i, f)
    if(i < f) then
        mid = (i + f) / 2
        MergeSort(arr, i, mid)
        MergeSort(arr, mid, f)
        Merge(arr, i, m, f)
```

La procedura Merge(arr, i, m, f), fonde arr[i:m] e arr[m+1:f], producendo arr[i;f] come output.

Come funziona la fusione di due array A e B:

- estrai rapidamente il minimo di A e B e copialo nell'array di output, finché A oppure B non diventa vuoto.
- copia gli elementi dell'array non vuoto alla fine dell'array di output

```
Merge(arr, sx, cx, dx)
    Sia AUX un array ausiliario di lunghezza (dx - sx)
    i = 1, k = sx, j = dx
    while (k <= cx AND j <= rx) do
        if (arr[k] <= arr[j]) then
            AUX[i] = arr[k]
            incrementa k
        else
            AUX[i] = arr[j]
            incrementa k
        incrementa i

    if(k <= cx) then copia arr[k:cx] alla fine di AUX
    else copia arr[j:rx] alla fine di AUX

    copia AUX in arr[sx:dx]
```

Lemma: La procedura Merge fonde due sequenze ordinate di lunghezza n e m in tempo $\theta(n+m)$

Dim: Ogni confronto "consuma" un elemente di una delle due sequenze. Ogni posizione di AUX è riempita in tempo costante. Il numero totale di elementi è $n + m$. Anche la copia del vettore AUX in arr costa $\theta(n + m)$.

Complessità Temporale: $T(n) = 2T(\frac{n}{2}) + O(n)$, usando il Teorema Master si ottiene: $T(n) = O(n \cdot log(n))$

Complessità Spaziale: Il MergeSort non ordina *in loco*, infatti occupa memoria ausiliaria pari a $\theta(n)$. Olte alla memoria ausiliaria, il MergeSort effetta $O(log(n))$ chiamate ricorsive, ciascuna usa memoriacostante, esclusa quella della chiamata Merge.

2. QuickSort

Anche questo algoritmo usa la tecnica del *divide et impera*:

- Divide: sceglie un elemento x della sequenza (perno) e partiziona la sequenza in elementi <= x ed elementi > x.
- Risolve i due sottoproblemi ricorsivamente.
- Impera: restituisci la concatenazione delle due sottosequenze ordinate.

Come funziona?
Si sceglie un perno. Si scorre l'array in "parallelo" da sinistra verso destra e da destra verso sinistra.

- da sinistra verso destra, ci si ferma su un elemento maggiore del perno.
- da destro verso sinistra, ci si ferma su un elementi minore del perno. Scambia gli elementi e riprendi la scansione.

```
Partition(arr, i, f)
    perno = arr[i]
    inf = i
    sup = f
    while(true) do
        do (incrementa inf) while (inf <= f AND arr[inf] <= perno)
        do (decrementa sup) while (arr[sup] > perno)
        if(inf < sup) then scambia arr[inf] e arr[sup]
    else break;

    scambia arr[i] e arr[sup] // mette il perno "al centro"
    return sup // restituisce l'indice del perno
```

Proprietà invariante: In ogni istante, gli elementi `arr[i],...,arr[inf-1]` sono <= del perno, mentre gli elementi `arr[sup+1],...arr[f]` sono > del perno.

```
QuickSort(arr, i, f)
    if(i < f) then
        m = Partition(arr, i, f)
        QuickSort(arr, i, m-1)
        QuickSort(arr, m+1, f)
```

Complessità Temporale:

- Ogni invocazione di Partition posiziona almeno un elemento in modo corretto (il perno).
- Quindi dopo n invocazioni di Partitio, ognuna di costo $O(n)$ ho il vettore ordinato. Il costo complessivo è quindi $O(n^2)$.
- Il caso peggiore si verifica quando il perno scelto ad ogni passo è il minimo o il massimo degli elementi nell'array.
- Costo caso peggiore: $T(n) = O(n^2)$.

- Costo caso migliore: $T(n) = O(n \cdot log(n))$.

Per ottenere il caso migliore, possiamo randomizzare la scelta del perno, fra gli elementi da ordinare.

Teorema
L'algoritmo QuickSort randomizzato ordina in loco un array di lunghezza n in tempo $O(n^2)$ nel caso peggiore e $O(n \cdot log(n))$ con altra probabilità, ovvero con probabilità almeno $1 - \frac{1}{n}$.

3. HeapSort

Idea: Selezionare gli elementi dal più grande al più piccolo, mediante una struttura dati efficiente, in modo tale da estrarre in tempo $O(log(n))$.

DEF (Tipo di dato e Struttura dati)

- Tipo di dato: specidica una collezione di oggetti e delle operazioni di interesse su tale collezione. (es. Dizionario: mantiene un insieme di elementi con chiavi soggetto a operazioni di inserimento, cancellazione, ricerca).
- Struttura dati: Organizzazione dei dati che permette di memorizzare la collezione e supportare le operazioni di un tipo di dato usando meno risorse di calcolo possibile.

Importante: progettare una struttura dati H su cui eseguire efficientemente le operazioni:

- dato un array A, generare velocemente H
- trovare il più grande oggetto in H
- cancellare il più grande oggetto da H

DEF (Heap)

La struttura dati heap associa ad un insieme S un albero binario radicato con le seguenti proprietà:

- completo fino al penultimo livello (struttura rafforzata: foglie sull'ultimo livello tutte compattate a sinistra).
- gli elementi di S sono memorizzati nei nodi dell'albero (ogni nodo v memorizza uno e un solo elemento, denotato con chiave(v).
- chiave(padre(v)) >= chiave(v) per ogni nodo v diverso dalla radice.

Proprietà:

- Il massimo è contenuto nella radice.
- L'albero con n nodi ha altezza $O(log(n))$.
- Gli heap con struttura rafforzata possono essere rapressentati in un array di dimensione pari a n.

Possiamo rappresentare l'Heap con un vettore posizionale di dimensione n. In generale la dimensione del vettore sarà diversa dal numero di elementi, perciò sarà necessario tenere conto del numero di elementi (HeapSize[arr]). Con il vettore posizionale abbiamo le seguenti proprietà: Dato un nodo i, possiamo calcolare il tempo costante

- sin(i) = 2i, *posizione del figlio sinistro.*
- des(i) = 2i + 1, *posizione del figlio destro.*
- padre(i) = $\frac{i}{2}$, *posizione del padre.*

FixHeap

Sia v la radice di H. Assumere che i sottoalberi radicati nel figlio sinistro e destro di v sono heap, ma la proprietà di ordinamento delle chiavi non vale per v. Posso ripristinarla così:

```
FixHeap(pos, arr)
    sx = sin(pos)
    dx = des(pos)
    max = pos
    if(sx < HeapSize[arr] AND arr[sx] > arr[max]) then max = sx
    if(dx < HeapSize[arr] AND arr[dx] > arr[max]) then max = dx
    if(max != pos) then
        scambia arr[pos] con arr[max]
        FixHeap(max, arr)
```

Complessità: $T(n) = O(log(n))$.

Come avviene l'estrazione del massimo?

- Copia nella radice la chiave contenuta nella foglia più a destra dell'ultimo livello. arr[HeapSize]
- Rimuovi la foglia, ovvero decrementare HeapSize.
- Ripristina la proprietà di ordinamento a heap richiamando FixHeap sulla radice

Heapify

```
Heapify_recursive(heap H, pos)
    sx = sin(pos)
    dx = des(pos)
    if(pos < HeapSize[H.arr] AND HeapSize[H.arr] > 1) then
        if(sin(sx) <= HeapSize[H.arr]) then Heapify_recursive(H, sx)
        if(des(dx) <= HeapSize[H.arr]) then Heapify_recursive(H, dx)
        FixHeap(pos, H)
```

Complessità Temporale: Sia h l'altezza di un heap con n elementi. Sia $n' >= n$ l'intero tale che un heap con n' elementi ha

- altezza h
- è completo fino all'ultimo livello

Vale: $T(n) <= T(n')$ e $n' <= 2n$

Tempo di esecuzione: $T(n') = 2T(\frac{n'-1}{2}) + O(log(n')) <= 2T(\frac{n'}{2}) + O(log(n'))$

Dal Teorema Master => $T(n') = O(n')$, quindi $T(n) <= T(n') = O(n') = O(2n) = O(n)$.

Alternativa

```
Heapify(heap H)
    i = H->heapSize / 2 - 1
    while(i >= 0)
        FixHeap(i, H)
        decrementa i
```

HeapSort

- Costruisce un heap tramite heapify.
- Estrae ripetutamente il massimo per $n-1$ volte, ad ogni estrazione memorizza il massimo nella posizione dell'array che si è appena liberata.

```
HeapSort(A)
    Heapify(A)
    HeapSize[A] = n
    for i=n down to 2 do
        scambia A[1] e A[i]
        HeapSize[A] = HeapSize[A] - 1
        FixHeap(1, A)
```

Teorema: L'algoritmo HeapSort ordina in loco un array di lunghezza n in tempo $O(n \cdot log(n))$ nel caso peggiore.

Algoritmi che ordinano senza fare confronti

1. IntegerSort

E' usato per ordinare n **interi** con valori [1, k], dove k è il massimo. Mantiene un array Y di k contatori tale che Y[x] = numero di volte che il valore x compare nell'array in input X. Dopo aver contao le occorenze di ciascun numero, scorre Y da sinistra verso destra e, se Y[x] = i, scrive in X il valore x per i volte.

```
IntegerSort(X, k)
    Sia Y un array di dimensione k
    for i = 1 to k to Y[i] = 0 // inizializzazione dei contatori a 0
    for i = 1 to n do incrementa Y[X[i]] // conta le occorenze dei elementi di X
    j = 1
    for i = 1 to k do
        while(Y[i] > 0)
            array[j] = i;
            incrementa j;
            decrementa Y[i];
```

Complessità Temporale: T(n) = $O(n+k)$, se k = O(n), allora l'algoritmo ordina in tempo lineare.

2. BucketSort

E' usato per ordinare n record con chiavi intere in [1, k], dove k è il massimo. Input: - n record mantenuti in un array - ogni elemento dell'array è un record con: campo chiave (rispetto al quale ordinare), altri campi associati alla chiave (varie informazioni)

Basta mantenere un array di liste, anziché di contatori, ed operare come per IntegerSort. La lista Y[i] conterrà gli elementi con chiave uguale a i. Infine, bisogna poi concatenare le liste. Tempo $O(n+k)$ come per IntegerSort.

```
BucketSort(X, k)
    Sia Y un array di dimensione k
    for i = 1 to k do Y[i] = lista vuota
    for i = 1 to n do
        appendi il record X[i] alla lista Y[chiave(X[i])]
    for i = 1 to k do
        copia ordinatamente in X gli elementi della lista Y[i]
```

DEF (Stabilità)

Un algoritmo è stabile se preserva l'ordine inziale tra elementi con la stessa chiave.

Il BucketSort è stabile se si appendono gli elementi di X in cosa alla opportuna lista Y[i].

3. RadixSort

- E' usato per ordinare n interi con valori in [1,k], dove k è il massimo. A differenza di BucketSort e IntegerSort, è usato per ordinare interi molto grandi.
- Rappresentiamo gli elemeti in base b, ed eseguiamo una serie di BucketSort.
- Partiamo dalla cifra meno significativa verso quella più significativa. Ordiniamo per l'i-esima cifra con una passata di BucketSort.

Correttezza:

- Se x e y hanno una diversa t-esima cifra, la t-esima passata di BucketSort li ordina
- Se x e y hanno la stessa t-esima cifra, la proprietà di stabilità del BucketSort li mantiene ordinati correttamente. Dopo la t-esima passata di BucketSort, i numeri sono correttamente ordinati rispetto alle t cifre meno significative.

Costo Temporale:

- Vengono eseguite $O(log(k))$ passate di BucketSort. Ciascuna passata costa $O(n + b) => O((n + b) \cdot log(k))$

Se b = $\theta(n)$, si ha un costo lineare se k = $O(n^c)$, c costante.