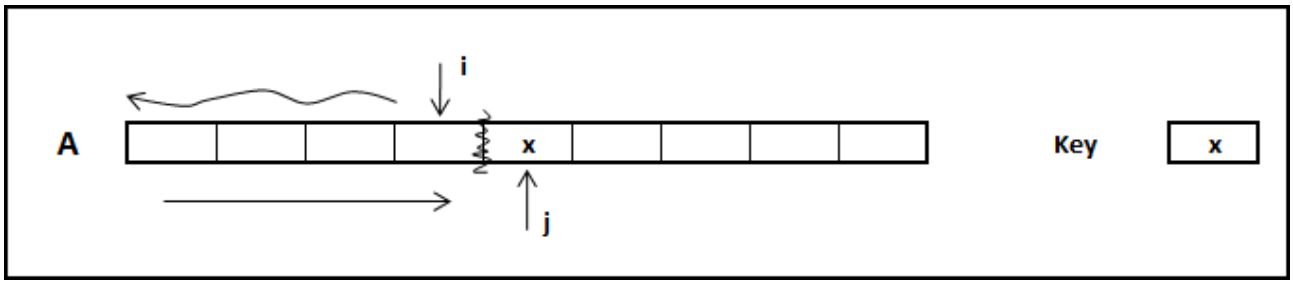
ALGORITMI DI ORDINAMENTO

INSERTION SORT

Algoritmo **STABILE** e **IN-PLACE**.

Scandisco il vettore da sx a dx ed ogni volta che trovo un elemento più piccolo dei precedenti lo metto in una variabile temporanea, spostando gli elementi più grandi (già scanditi) di una posizione a dx, inserendo l'elemento prima di questi.

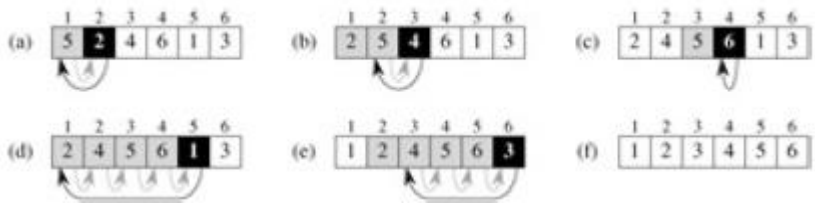
Ho un vettore A già ordinato in parte.




```
InsertionSort(A){
      for(j←2 to lenght[A]){
            key ← A[j]
            i←j-1

            while(i>0 && A[i]>key){
                  A[i+1] ← A[i]
                  i ← i-1
            }
            A[i+1] ← key
      }
}
```

Esempio:



COMPLESSITA':

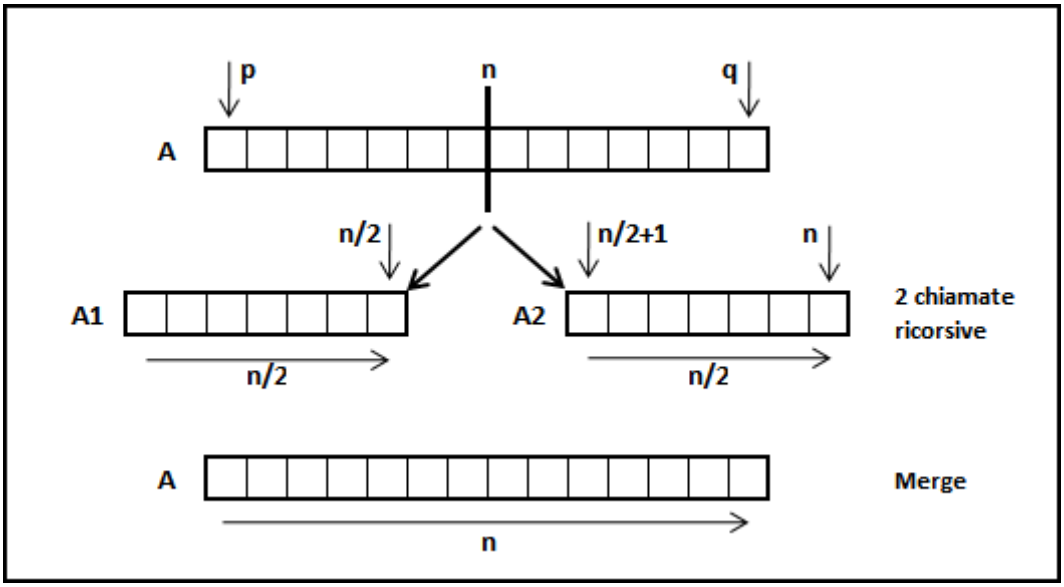
Caso Migliore: vettore già ordinato **$T(n) = \theta(n)$**

Caso Peggiore: vettore ordinato al contrario **$T(n) = \theta(n^2)$**

MERGE SORT

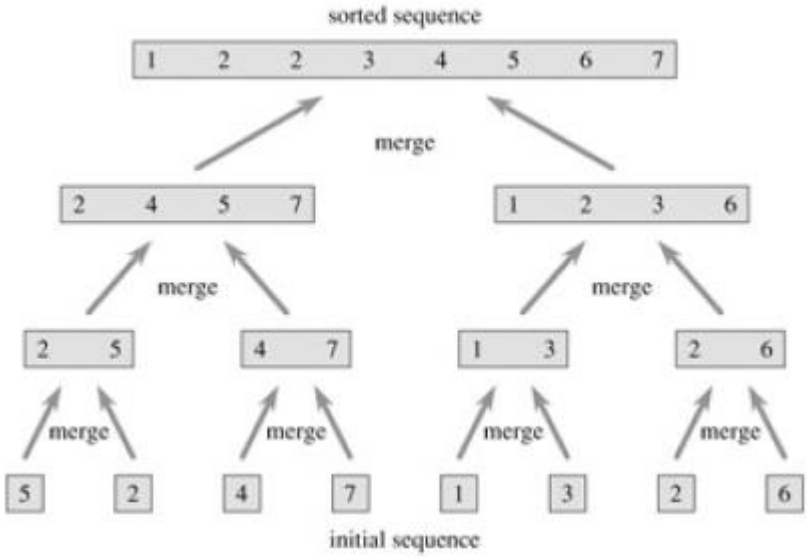
Algoritmo **STABILE** e **NON IN PLACE**. E' stabile perché in Merge c'è `if(A[i] ≤ A[j])`

`Divide il vettore in 2 metà. Utilizzando la ricorsione arrivo ad avere tutti array di un solo elemento. Poi richiamo Merge che fonde gli array e li ordina.`




```
MergeSort(A,p,q){
      if(p<q){
            r ← (p+q)/2
            MergeSort(A,p,r)
            MergeSort(A,r+1,q)
            Merge(A,p,q,r)
      }
}
```

Esempio:



COMPLESSITA':

T(n) = θ(n log n)

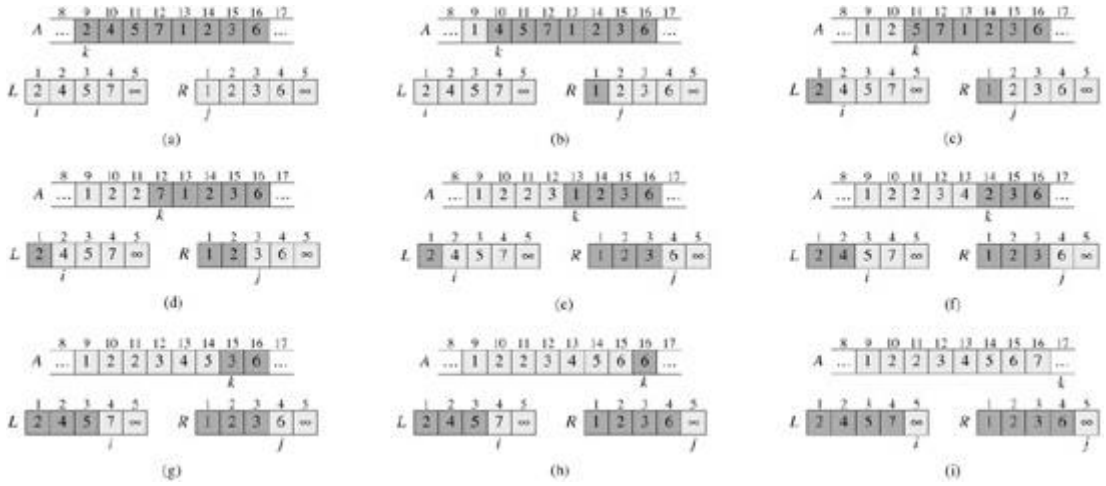
MERGE

Merge rimette insieme i pezzi di che erano stati divisi da Merge Sort, ordinandoli.

```
Merge(A,p,q,r){
      B ← new_Array (q-p+1)                    %n=q-p+1
      i ← p
      j ← r+1
      for(k←1 to length[B]){
            if(i≤r && j≤q){
                  if(A[i]≤A[j]){
                        B[k] ← A[i]
                        i ← i+1
                  }else{
                        B[k] ← A[j]
                        j ← j+1
                  }
            }else if(i≤r){
                  B[k] ← A[i]
                  i ← i+1
            }else{
                  B[k] ← A[j]
                  j ← j+1
            }
      }
      for(k←1 to length[B]){
            A[p+k-1] ← B[k]
      }
}

Libero la memoria di B.
```

Esempio:



COMPLESSITA':

T(n) = θ(n)

HEAPSORT

Algoritmo **NON STABILE** e **IN PLACE**.

Dato un vettore, creo una heap (con BuildHeap). Scambio il max con l'ultimo elemento, non considerandolo più. Richiamo Heapify che riordina la Heap (senza la radice di prima). E ripeto.

Al termine della procedura ho un vettore ordinato in ordine crescente, la trasformazione in heap è solo un passaggio momentaneo.

```
HeapSort(A){
      BuildHeap(A)
      for(i←length[A] down to 2){
            Scambia(A,1,i)
            heapsize[A]←heapsize[A]-1
            Heapify(A,1)
      }
}
```

COMPLESSITA':

T(n) = O(n log n)

BUILDHEAP

Dato un vettore lo trasforma in heap.

```
BuildHeap(A){
      heapsize[A]←length[A]
      for(i← ⌊length[A]/2⌋ down to 1){
            Heapify(A,i)
      }
```

```
}
```

HEAPIFY

Riordino una heap partendo dalla posizione i e confrontandola con i suoi figli. Se uno dei 2 è maggiore dell'elemento in posizione i, scambio il figlio corrispondente con l'elemento in posizione i.

```
Heapify(H,i){
      if(2i≤heapsize[H] && H[2i]>H[i]){
            m←2i
      }else{
            m←i
      }
      if(2i+1≤heapsize[H] && H[2i+1]>H[m]){
            m←2i+1
      }
      if(m≠i){
            Scambia(H,m,i)
            Heapify(H,m)
      }
}
```

HEAPINSERT

Inserisco un nuovo elemento nella heap.

```
Heapinsert(H,k){
      if(heapsize[H]<length[H]){
            heapsize[H]←heapsize[H]+1
            H[heapsize[H]]←k
            i←heapsize[H]
            while(i>1 &&H[i]>H[⌈i/2⌉]){
                  Scambia(H,i,⌊i/2⌋)
                  i←⌊i/2⌋
            }
      }else{
            Print "non c'è posto"
      }
}
```

HEAP-EXTRACT-MAX

Estraggo il max da una heap. Riordinando quello che mi rimane.

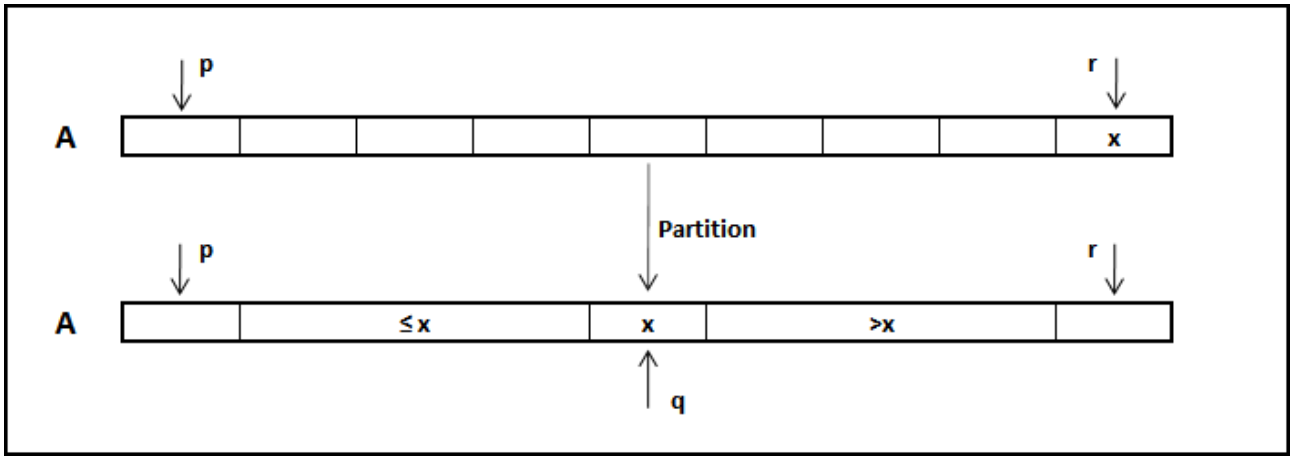
```
Heap-extract-max(H){
```

```
        if(heapsize[H]=0){
                return "errore"
        }else{
                max←H[1]
                Scambia(H,1,heapsize[H])
                heapsize[H]←heapsize[H]-1
                Heapify(H,1)
                Return max
        }
}
```

QUICK SORT

Algoritmo **NON STABILE**(perché c'è Partition) e **NON IN-PLACE**(ci sono le chiamate ricorsive).

Dato un vettore, scelgo un elemento e lo uso come perno. Richiamo Partition che mi restituisce la posizione in cui finirebbe il perno se il vettore fosse ordinato. Chiamate ricorsive per ordinare ogni pezzo del vettore.




```
QuickSort(A,p,r){
        if(p<r){
                q ← Partition(A,p,r)
                QuickSort(A,p,q-1)
                QuickSort(A,q+1,r)
        }
}
```

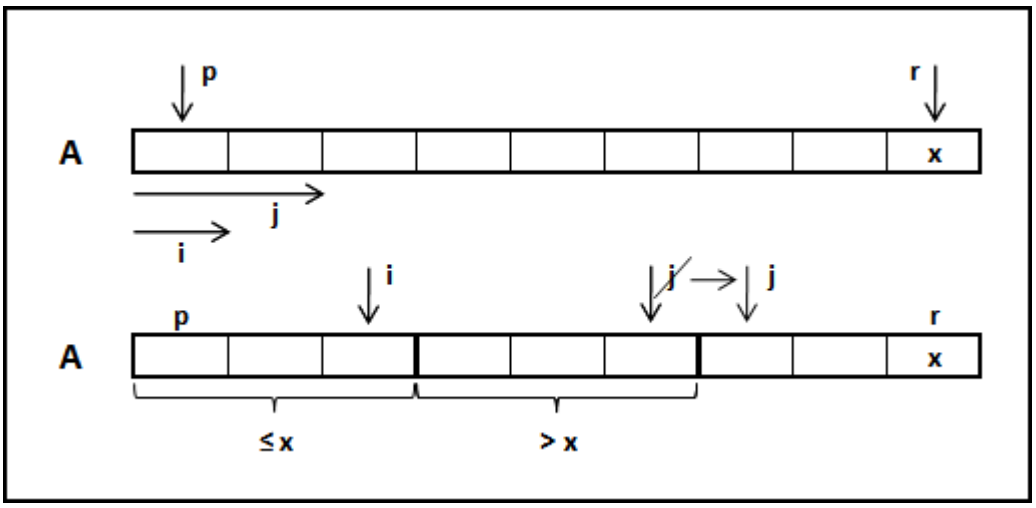
COMPLESSITA':

Caso Migliore: quando le parti dx e sx sono equilibrate **To(n) = θ(n log n)**

Caso Peggiore: il vettore è già ordinato, o è ordinato al contrario, cioè il perno finisce sempre in prima o in ultima posizione **Tp(n) = θ(n^2)**

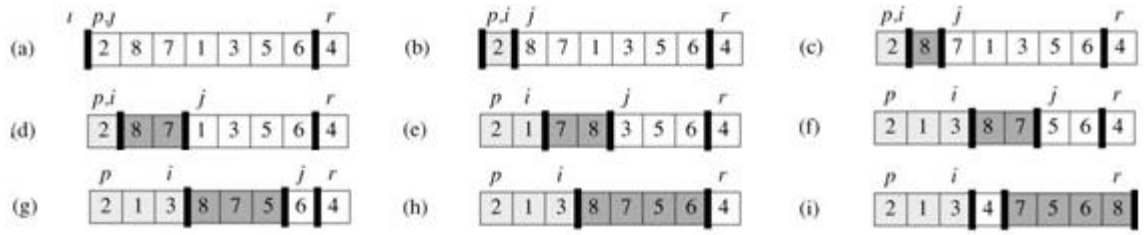
PARTITION

Partition mi restituisce la posizione q in cui è finito il perno x, mettendo alla sua sinistra gli elementi minori o uguali del perno ed alla sua destra quelli maggiori.




```
Partition(A,p,r){
      x ← A[r]
      i ← p-1
      for(j←p to r-1){
            if(a[j]≤x){
                  i ← i+1
                  Scambia(A,i,j)
            }
      }
      i ← i+1
      Scambia(A,i,r)
      return i;
}
```

Esempio:



COMPLESSITA':

T(n) = θ(n) , con n = r-p+1

SELECT

Algoritmo **NON STABILE**(perché c'è Partition) e **NON IN-PLACE.**

(cerco l'elemento che sta in posizione i se il vettore fosse ordinato).

Dato un vettore lo spezza in blocchi di 5 elementi, trova il mediano in ogni blocco copiandolo in un nuovo vettore. Richiamo Select sul nuovo vettore per trovare il mediano dei mediani. Richiamo Partition sul vettore iniziale usando l'elemento trovato (mediano) come perno. Confronto la posizione che cercavo con la posizione del perno di Partition. Se è uguale restituisco il valore, altrimenti applico la ricorsione sulla metà dell'array che contiene la posizione che mi interessa trovare.

COMPLESSITA':

$T(n) = \theta(n)$

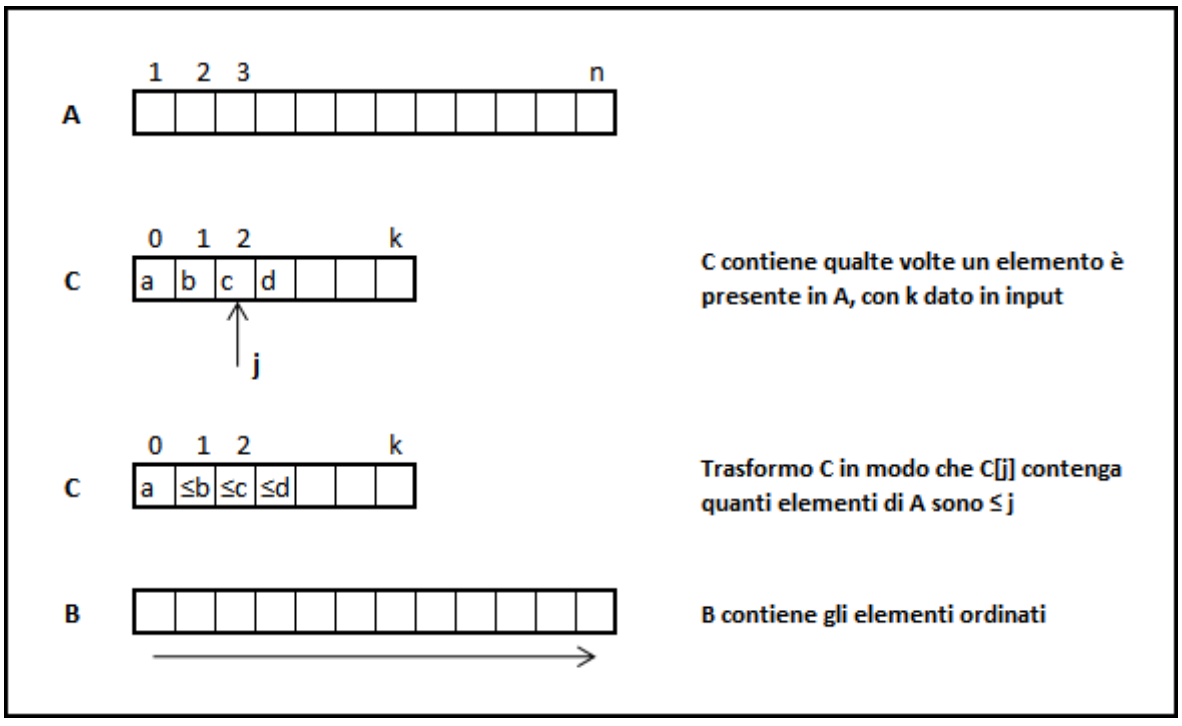
COUNTING SORT

Algoritmo **STABILE** ma **NON IN-PLACE**. (con ipotesi sull'input)

Dato un vettore A, conta le occorrenze di ogni elemento di A e le scrive in un vettore C. Sovrascrivo il vettore C cumulando le occorrenze. Scorro A da dx verso sx, posiziono gli elementi in un vettore B scrivendoli nella posizione datami dall'occorrenza cumulata scritta nel vettore C in posizione "elemento di A"

A vettore contenente interi positivi tali che per ogni i:

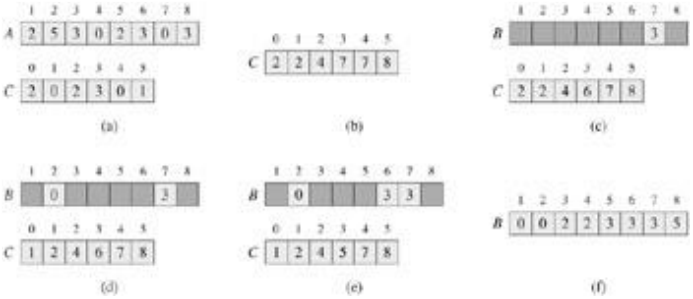
- $1 \leq i \leq n$, con n=length[A]
- $A[i] \leq k$, con k funzione costante e $k \in O(n)$
- $0 \leq A[i] \leq k$




```
CountingSort(A,B,k){
```

```
        C←new_vector(k+1)
        for(i←0 to k){
              C[i] ← 0
        }
        for(j←1 to length[A]){
              C[A[j]] ← C[A[j]]+1
        }
        for(i←1 to k){
              C[i] ← C[i]+C[i-1]
        }
        for(j←length[A] down to 1){
              B[C[A[j]]] ← A[j]
              C[A[j]] ← C[A[j]]-1
        }
}
```

Esempio:



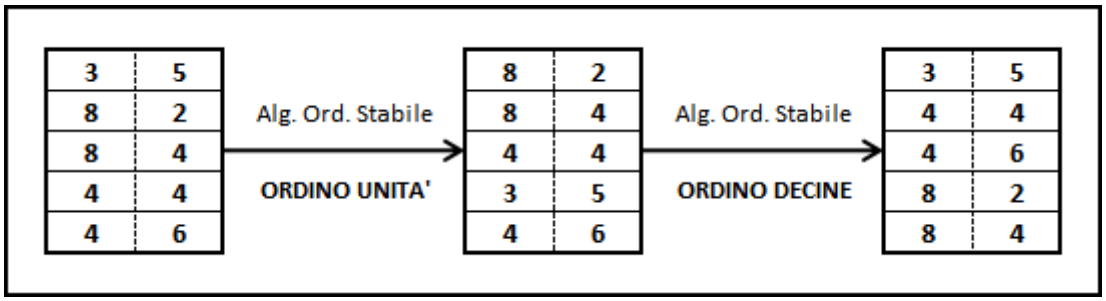
COMPLESSITA':

T(n) = θ (k) + **θ(n) = θ(n)** (per l'ipotesi)

RADIX SORT

Algoritmo **STABILE**. L'essere IN-PLACE o NON IN-PLACE dipende da che algoritmo uso al suo interno.

Radix Sort ordina n numeri di d digit:
1-Guarda un numero per volta
2-Parte dal digit meno significativo(unità)
3-Ad ogni passo applica un algoritmo di ordinamento stabile



BUCKET SORT

Algoritmo **STABILE** ma **NON IN-PLACE**.

Bucket Sort funziona bene solo se:

1- A vettore $\in R[0,1)$
2- Elementi del vettore distribuiti in modo uniforme in [0,1)

Dato un vettore A con elementi distribuiti in modo uniforme, suddivido l'intervallo [0,1) in *n* sottointervalli, determinando gli elementi di A contenuti in ognuno di questi. B è il vettore di liste degli elementi di A. ordino ogni lista ed infine concateno le liste ordinate.

```
BucketSort(A,B){
      for(i←1 to length[A]){
            Insert_in_list(B[⌊A[i] · length[A]⌋],A[i])
      }
      for(j←0 to length[B]){
            Sort(B[j])
      }
      k←1
      for(j←0 to length[B]){
            k←Copy(A,B[j],k)
      }
}
```

STRUTTURE DATI

Vettori

struttura dati statica, accesso diretto (vantaggio: poco tempo x accedere agli elementi, svantaggio: molto spazio da allocare)

Liste

struttura dati dinamica, accesso sequenziale (svantaggio: molto tempo xke devo scorrere la lista, vantaggio: poco spazio da allocare, in base agli elementi in memoria)

- 2 campi: 1 campo key (elemento), 1 campo next (puntatore)
- Stabile solo se inserisco un elemento in fondo alla lista
- Il costo dell'inserimento dipende dal numero di puntatori che passo. (inserimento in testa = θ(1))
- Per scandire una lista si deve usare il while perché non ne conosco la lunghezza, mi fermo quando il next del prossimo elemento è nill.

Tabelle di Hash

Utilizzano poco spazio e poco tempo nel caso medio, per mantenere in memoria un insieme di elementi che varia dinamicamente.

1. **Hash con chaining**: vettore di liste (se 2 elementi vogliono andare nella stessa cella della tabella, creo una lista all'interno della cella.
   Costi: inserimento θ (1) ricerca/cancellazione O(|K|)
   a. *ipotesi di hashing uniforme semplice*: estraendo casualmente un elemento dall'universo, esso ha la stessa probabilità di finire in una qualsiasi cella della tabella.

Se è soddisfatta questa condizione, le operazioni di ricerca/cancellazione hanno costo in media θ (1+α). α = lunghezza media delle liste T[0], T[1], … T[m-1].

α (fattore di carico) = n/m *n=|K|=n° elementi da inserire*

m=|T|=n° celle tabella

2. **Hash con open addressing:** memorizza i dati in una tabella senza liste. In caso di collisione scandisco la tabella in cerca di una cella libera.
   a. *ipotesi di hashing uniforme*: per ogni elemento dell'universo, tutte le sequenze di scansione hanno uguale probabilità. se una funzione di hash soddisfa "hashing uniforme" allora *inserimento, ricerca e cancellazione* hanno costo medio θ(1).
   b. **funzioni di hash con sequenza**:
      - *scansione lineare*: $h(k,i)=(h'(k)+l\cdot i) \bmod m$ (primary clustering)
        posso avere elementi x,y t.c. $h'(x)\neq h'(y)$
      - *scansione quadratica*: $h(k,i)=(h'(k)+c_1 i+c_2 i^2) \bmod m$ (secondary clustering)
        se $h'(x)\neq h'(y)$ sequenze diverse
        se $h'(x)=h'(y)$ sequenze uguali
      - *hash doppio*: $h(k,i)=(h_1(k)+ h_2(k)\cdot i) \bmod m$ (evita i clustering, migliore)

Devo sempre controllare che:

- ***a)*** Vk $0\leq h(k)\leq m-1$
- ***b)*** V $0\leq i\leq m-1$ Ex ∈ U $h(key(x))=i$
- ***c)*** non esistano alcuni indici di T in cui finirebbero moltissimi elementi di U ed altri in cui ne finirebbero pochissimi
- ***d)*** h utilizzi tutti i bit di k

Inserimento: uso la sequenza di scansione di x per inserire x nella prima cella libera che trovo (NIL o DEL).

Caso peggiore : O(m)

```
Insert(T,x,h){
      m←length[T]
      i←0
      j←h(key[x],0)
      while(T[j]≠NIL && T[j]≠DEL && i≤m-1){
            i←i+1
            j←h(key[x],i)
      }
      if(i>m-1){
            return "T piena"
      } else {
            T[j]←x
            return j
      }
}
```

Cancellazione: uso la sequenza di scansione di x per cercare x. se prima di trovare x trovo un NIL, x non è in tabella. Se trovo x lo cancello sostituendolo con DEL. Caso peggiore : O(m)

```
Delete(T,x,h){
      j←Search(T,x,h)
      if(j≠-1){
```

```
            T[j]←DEL
      }
}
```

Ricerca: uso la sequenza di scansione di x per cercare x. Se trovo DEL proseguo. Se trovo NIL mi fermo (x non è in T). Se trovo x restituisco la posizione in cui è x. Caso peggiore : O(m)

```
Search(T,x,h){
      m←length[T]
      i←0
      j←h(key[x],i)
      while(T[j]≠x && T[j]≠NIL && i≤m-1){
            i←i+1
            j←h(key[x],i)
      }
      if(T[j]=NIL || i>m-1){
            return -1
      } else {
            T[j]←x
            return j
      }
}
```

Alberi binari di ricerca (BST)

Albero binario

Struttura dati dinamica.

Nodi: key, left, right, parent.

BST

T è un BST:

- Vx€T key[x] è un numero
- Vx€T se y sta nel sottoalbero radicato in left[x] allora key[y]<key[x]
- Vx€T se y sta nel sottoalbero radicato in right[x] allora key[y]>key[x]

Visite di alberi binari

- **Pre-order**: visito il nodo x, poi left[x], poi right[x]
- **In-order**: visito left[x], poi il nodo x, poi right[x]
- **Post-order**: visito left[x], poi right[x], poi il nodo x

Pre-order

Pre-Order(x){

```
        If(x≠nil){
                print(key[x])
                Pre-Order(left[x])
                Pre-Order(right[x])
        }
}
```

In-order

```
In-Order(x){
        If(x≠nil){
                In-Order(left[x])
                print(key[x])
                In-Order(right[x])
        }
}
```

Post-order

```
Post-Order(x){
        If(x≠nil){
                Post-Order(left[x])
                Post-Order(right[x])
                print(key[x])
        }
}
```

Costo: Tutte le visite hanno costo θ (n)

Complessità: Si può dimostrare solo per induzione su n e non con l'albero delle chiamate ricorsive