ALGORITMI - Teoria da lezioni e slide

Insertion Sort con ordinamento in modo lineare pos j nelle posizioni da j-1 a 1.

6 5 3 1 8 7 2 4

caso medio: $\Theta(n^2)$

Web visualization:
https://www.hackerearth.com/practice/algorithms/sorting/insertion-sort/visualize/

ottimo se scambio solo elementi contigui

In place/in loco; 1 bit di spazio richiesto.

Nota. In place ossia algoritmo è in grado di trasformare una struttura dati utilizzando soltanto un piccolo e costante spazio di memoria extra.

versione iterativa:

```
InsertionSort(int [ ]A){
n = A.length;
for(int i=1; i<n; i++){
        x=A[i];
        for(int j=i-1;j>=0 && A[j]>x ;j--){//!!>=0
            A[j+1]= A[j]; //non devo fare swap doppio ma solo verso destra
        }
        A[j+1]=x; //essendo uscito dallo shift sono già sull'elemento più piccolo di x
nella posizione j quindi vado sul suo successivo che è già stato copiato dallo shift
e inserisco il mio valore
}
}
```

versione ricorsiva:

```
InsertionSortRic(int[ ] A,int n){
if(n>1) //salto avanti di n ricorsioni
        insertionSortRic(A,n-1);
else{ //ma poi procedo a ritroso sistemando dal primo elemento fino all'ultimo
essendo entrato in n ricorsioni precedentemente
        int k = A[n];
        i = n - 1;
```

```
        while( i >= 0 && A[i]>k){
        A[i+1]=A[i]
        }
        A[i+1]=k
}
}
```

Divide et Impera

secondo questa tecnica un programma è diviso in tre parti:

- *Divide*: in questa parte si procede alla suddivisione dei problemi in problemi di dimensione minore; con un costo di D(n) pari alla divisione nei sottoproblemi Π(n1),Π(n2),...Π(nk)
- *Impera*: nella seconda parte i problemi vengono risolti in modo ricorsivo. Quando i sottoproblemi arrivano ad avere una dimensione sufficientemente piccola, essi vengono risolti direttamente tramite il caso base; con costo di C(n) per combinare i risultati.
- *Combina*: l'ultima fase del paradigma prevede di riunire l'output ottenuto dalle precedenti chiamate ricorsive al fine di ottenere il risultato finale. T(ni) i k costi delle soluzioni dei sottoproblemi e c il costo del problema.

$$T(n) = \begin{cases} c & n \leq k \\ D(n) + C(n) + \sum_{i=1}^{k} T(n_i) & n > k \end{cases}$$

Merge Sort

caso medio: $T(n) = \Theta(n \log n)$

sia nel caso medio che nel caso pessimo. Infatti:

- la funzione merge qui presentata ha complessità temporale $\Theta(n)$
- mergesort richiama se stessa due volte, e ogni volta su (circa) metà della sequenza in input

6 5 3 1 8 7 2 4

Da questo segue che il tempo di esecuzione corrisponde alla ricorrenza: $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$

la cui soluzione in forma chiusa è $\Theta(n \log n)$, per il secondo caso del teorema principale.

Web visualization:
https://www.hackerearth.com/practice/algorithms/sorting/merge-sort/visualize/

versione iterativa introdotta col pensiero Divide et Impera:

```
Mergesort (int A[ ],int l,int r){//l=left r=right
if(l < r)//finché la mia r non coincide con l quindi sono su un singolo elemento
        c = (l+r)/2; //trovo la metà da cui dividere DIVIDE
        mergesort(A,l,c); //da sinistra alla metà IMPERA
        mergesort(A,c+1,r);//dal successivo alla metà alla fine IMPERA
        merge(A,l,c,r); //FONDI
}
```

funzione di supporto merge iterativa:

```
Merge(int A,int l,int c,int r){
n1 = c−l+1;//numero di elementi sinistra
n2 = r−c;//numero di elementi destra
for(i = 1 to n1)//copio in L quelli sinistra
        L[i] = A[l + i − 1];
for(j = 1 to n2)//in R quelli di destra
        R[j] = A[c + j];
L[n1 + 1] = R[n2 + 1] = infinito;
i=j=1; //riparto dall'inizio di L e R e inserisco in A k dal primo in l fino a r
for(k = l to r){
        if(L[i] <= R[j]){//inserisco in A solo il più piccolo di L e R in maniera ordinata
siccome i contatori dei due proseguono indipendentemente da k
                A[k]=L[i];
                i++;
        }else{
                A[k]=R[j];
                j++;
        }//else
}//for
}
```

Notazione asintotica

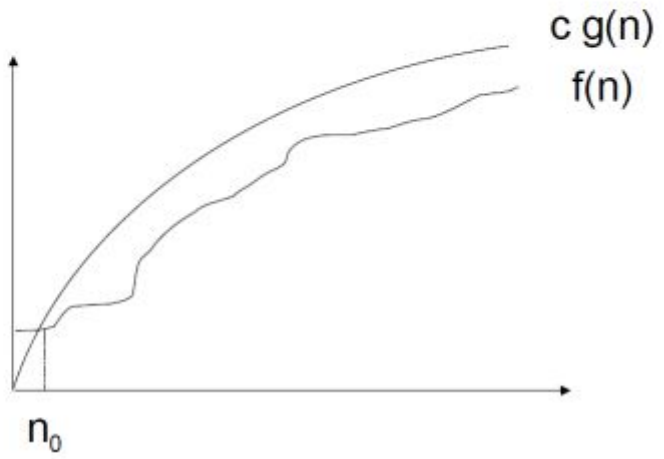
Limite superiore O

Date due funzioni $f(n)$, $g(n) \geq 0$ si dice che

$$f(n) \text{ è un } O(g(n))$$

se esistono due costanti c ed n_0 tali che

$$0 \leq f(n) \leq c\, g(n) \text{ per ogni } n \geq n_0$$



f(n) = O(g(n)) si legge: **f(n) è "o grande" di g(n)**

Se f(n) = O(g(n)) rappresenta il tempo calcolo richiesto da un algoritmo diciamo che **O(g(n)) è un limite superiore asintotico** per la **complessità tempo** di tale algoritmo.

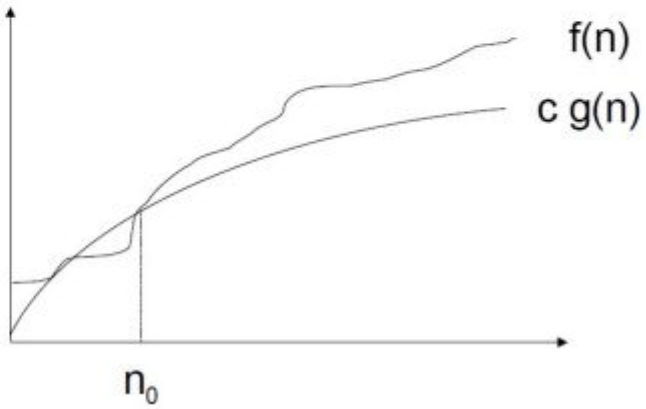
Limite inferiore Ω

Date due funzioni $f(n)$, $g(n) \geq 0$ si dice che

$$f(n) \text{ è un } \Omega(g(n))$$

se esistono due costanti c ed n_0 tali che

$$f(n) \geq c\, g(n) \text{ per ogni } n \geq n_0$$



f(n) = Ω(g(n)) si legge: **f(n) è "omega" di g(n)**

Se f(n) = Ω(g(n)) rappresenta il tempo calcolo richiesto da un algoritmo diciamo che Ω**(g(n)) è un limite inferiore asintotico** per la **complessità tempo** di tale algoritmo.

Unione del limite superiore e inferiore per la maggiore precisione possibile nel determinare la complessità di un algoritmo

Abbiamo visto come, in entrambe le notazioni esposte in precedenza, per ogni funzione f(n) sia possibile trovare più funzioni g(n).

In effetti O(g(n)) e Ω(g(n)) sono insiemi di funzioni, e dire "f(n) è un O(g(n)" oppure "f(n) = O(g(n)" ha il significato di "f(n) appartiene a O(g(n))".

Tuttavia, poiché i limiti asintotici ci servono per stimare con la maggior precisione possibile la complessità di un algoritmo, vorremmo trovare – fra tutte le possibili funzioni g(n) – quella che più si avvicina a f(n).

Per questo cerchiamo la più piccola funzione g(n) per determinare O e la più grande funzione g(n) per determinare Ω.

Limite asintotico stretto Θ

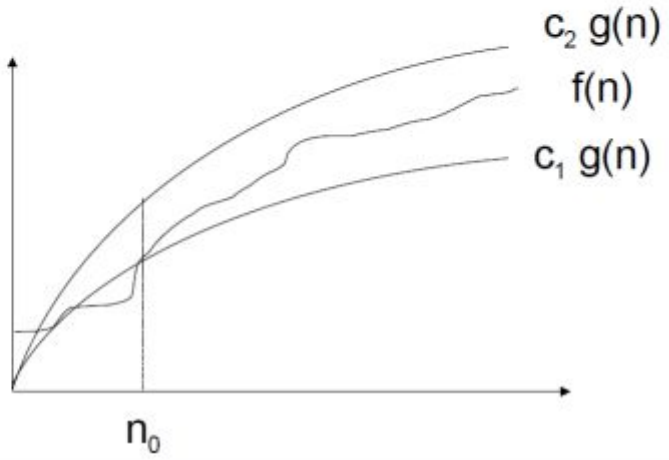
Date due funzioni $f(n)$, $g(n) \geq 0$ si dice che

$$f(n) \text{ è un } \Theta(g(n))$$

se esistono tre costanti c_1, c_2 ed n_0 tali che

$$c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ per ogni } n \geq n_0$$

In altre parole, $f(n)$ è $\Theta(g(n))$ se è contemporaneamente $O(g(n))$ e $\Omega(g(n))$.



f(n) = Θ(g(n)) si legge: **f(n) è "theta" di g(n)**

Se f(n) = Θ(g(n)) rappresenta il tempo calcolo richiesto da un algoritmo diciamo che **Θ(g(n)) è un limite asintotico stretto** per la **complessità tempo** di tale algoritmo.

Metodo del limite

è possibile determinare dei limiti asintotici calcolando il limite di un rapporto.

Se limite per infinito di n con f(n)/g(n) = k > 0 allora f(n) = Θ(g(n))

Equazioni di ricorrenza

è un'equazione o una diseguaglianza che descrive una funzione in termini del suo valore su valori inferiori.
E' relativamente semplice esprimere la complessità con una relazione di ricorrenza.
Può essere però piuttosto complicato risolvere la relazione di ricorrenza.
Esistono due metodi per la risoluzione di equazioni di ricorrenza:

- metodo iterativo(richiede una maggiore quantità di calcoli algebrici ma è più meccanico)
- metodo di sostituzione(serve soprattutto nelle dimostrazioni mentre si sconsiglia di utilizzarlo nella pratica)
- metodo dell'esperto o principale

Metodo di sostituzione

utile se si chiede di dimostrare un T(n)=Θ/O(valore_da_dim) prevede due passi:

1. stimare l'ordine di grandezza asintotico per T(n)
2. si dimostra per induzione su n la correttezza dell'ordine di grandezza stimato

Risoluzione:

1. porre T(n) <= c *(valore_da_dim) quindi determinare un valore di c e di n0 per ogni n>n0
2. dimostrare che i valori scelti nel caso base funzionano con n=n0
3. quindi passare al passo induttivo per n>n0
4. guardare per quale valore T(n) dimostrare quindi calcolare dalla ricorrenza il T(del sottoinsieme di n specificato) e sostituirlo nella equazione quindi procedere con semplificazioni
5. infine porre valore ottenuto <= di c *(valore_da_dim) da step1 e dire quindi per quali valore di c vale la dimostrazione e in caso si avesse determinato un valore di c già all'inizio quindi dire se l'ipotesi iniziale è dimostrabile oppure no

$$T(n) = 2T\left(\frac{n-1}{2}\right) + 1 \leq cn$$

$$= 2 \cdot c\left(\frac{n-1}{2}\right) + 1 \leq cn$$

$$= c(n-1) + 1 \leq cn$$

$$= cn - c + 1 \leq cn$$

$$c \geq 1$$

$$T\left(\frac{n-1}{2}\right) \leq c\,\frac{n-1}{2}$$

Metodo dell'Esperto/Teorema principale

data T(n) una funzione positiva non decrescente definita dalla relazione:

$$\mathbf{T(n) = aT(n/b) + f(n)}$$

dove a≥1,b>1 sono costanti intere e n/b indica indifferentemente n/b oppure n/b. E sia k = $\log_b a$. e **a** corrisponde al numero di sottoproblemi e **b** il tasso di divisione degli elementi nei sottoproblemi. La relazione ha soluzione:

1. se $f(n) \in O(n^{k-\varepsilon})$ per qualche $\varepsilon>0$, allora $\mathbf{T(n) \in \Theta(n^k)}$;
2. se $f(n) \in \Theta(n^k)$, allora $\mathbf{T(n) \in \Theta(n^k \log n)}$;
3. se $f(n) \in \Omega(n^{k+\varepsilon})$ per qualche $\varepsilon>0$, e se $af(n/b)<cf(n)$ per qualche $c<1$ e $n\to\infty$, allora $\mathbf{T(n) \in \Theta(f(n))}$.

Risoluzione:

1. elimino parentesi sopra sotto laterali da CAPIRE
2. sviluppo $\log_b a$
3. confronto $n^{\log_b a}$ con la f(n)
4. sviluppo un limite per $f(n)/n^{\log_b a}$ e se domina il numeratore abbiamo f(n) come teta ma quindi come ultima cosa devo controllare che esista un k>1 che rispetti la condizione af(n/b) < cn
5. confronto f(n) con valore n^k e se:
    a. n^k > f(n) CASO 1
    b. n^k = f(n) CASO 2
    c. n^k < f(n) CASO 3 dimostrando pure che a*f(n/b) <= c*f(n) per un qualche c < 1 ed n sufficientemente grande

Metodo Iterativo

Consiste nello srotolare la ricorsione fino ad ottenere una sommatoria dipendente da n potendo applicarlo a una qualsiasi ricorrenza come nel seguente esempio.

$$\begin{aligned} T(n) &= 1 + T(n/2) \\ &= 1 + 1 + T(n/4) = 2 + T(n/4) = 2 + T(n/2^2) \\ &= 2 + 1 + T(n/8) = 3 + T(n/8) = 3 + T(n/2^3) \\ &= 3 + 1 + T(n/16) = 4 + T(n/16) = 4 + T(n/2^4) \\ &= \cdots \\ &= k + T(n/2^k) \end{aligned}$$

Continuiamo a srotolare la ricorsione fin quando $n/2^k = 1$; ora $n/2^k = 1$ implica $2^k = n$ e quindi $k = \log_2 n$ (k è il logaritmo in base 2 di n). Allora:

$$T(n) = \log_2 n + 1 = O(\log_2 n)$$

Arrivo ad ottenere un certo valore costante con T(n pari a ....) lo pongo uguale a 1 e da questo ricavo k e lo vado a sostituire nella mia approssimazione ottenendo tempo per T(n) avendo quindi un limite superiore O(...).

Albero di Ricorsione

Alternativo al precedente, rappresentazione visiva delle chiamate ricorsive dove ad ogni nodo associamo il costo del lavoro esterno inoltre è una tecnica semplice per calcolare un limite superiore.

Per il calcolo dell'altezza dell'albero di ricorsione, ovvero del numero di passi necessario per terminare la ricorsione, si considera il cammino più lungo dalla radice ad una foglia.
Risoluzione:

1. troviamo cammino più lungo nella i-esima iterazione
2. si calcola n rispetto a valore x elevato per i
3. si fissa il $\log_x x^i$ e poi si rimette al posto di x^i il valore n avendo quindi trovato l'altezza
4. si moltiplica l'altezza h_alb per cn

Stima del costo totale: T(n) = Σ i=0 =>h_alb per: cn

Dopo aver ottenuto questo T(n) ne calcoliamo il limite asintotico stretto.

Algoritmo

serie di operazioni definite in sequenza per risolvere un problema

Heapsort

6 5 3 1 8 7 2 4

Tempo caso peggiore, migliore, medio: $O(n \log n)$, in place
con heap minimo e massimo => heap è un albero

web visualization:
https://people.ok.ubc.ca/ylucet/DS/HeapSort.html

Sia per MaxHeap che per MinHeap web visualization:
https://kanaka.github.io/rbt_cfs/trees.html

Heap(mucchio)

struttura a nodi assimilabile a un albero binario nel quale ogni nodo corrisponde a un solo elemento ed esso è **QUASI COMPLETO** (completo almeno fino al penultimo livello(ogni nodo non foglia ha due figli e tutti i cammini radice foglia hanno la stessa lunghezza)), implementato come un array A[1...n] dove 1 è la radice e l'i esimo nodo padre ha il suo figlio sinistro nella posizione 2*i e quello destro in 2*i+1.

Avremo quindi due dimensioni di A, A.length(capacità massima albero) e A.heapsize(numero di nodi effettivi nell'albero).

Ma la proprietà fondamentale degli Heap è il fatto che il **valore associato al nodo padre è sempre maggiore o uguale a quello associato ai nodi figli.**

Left(i){return i*2;} Lo spostamento verso Left o Right è del doppio di i non di i+1 che

Right(i){return i*2+1;} porterebbe invece al nodo fratello di i sullo stesso livelo

Parent(i){return i/2;}

MaxHeap

ogni nodo i ha chiave maggiore uguale a quella del sottoalbero radicato in i, Ogni chiave di i >= chiavi Left(i)&&Right(i) e ogni nodo ha chiave minore uguale agli antenati, Ogni i chiave <= a Parent(i).

```
MaxHeapify(int[] A, int i){//O(logn) figli l e r sono heap ma i padre forse no quindi fa in modo di rendere heap l'intero albero
	int max=i;
	if(Left(i) <= Right(i))&&(A[Left(i)]>A[max])
		max=Left(i);
	if(Right(i) <= A.heapsize)&&(A[Right(i)]>A[max])
		max=Right(i);
	if(max!=i){
		A[i]<-swap->A[max];//si scambia il più grande con i
		MaxHeapify(A,max);//l'elemento figlio max non tiene più il massimo per forza siccome è possibile sia stato scambiato e messo nel padre quindi si devono controllare i figli per essere sicuri che possa stare in quella posizione
	}
}//A è modificato così che i sia la radice heap
```

Se h è altezza di i, il costo associato è O(h)=O(logn) se n #nodi nel sottoalbero.

T(n)=c+T(⅔ n)

f(n)=c con a=1 e b=3/2 applicato th.esperto siamo nel primo caso quindi T(n)=Θ(logn)

n>= 2^ht

log2^ht <= logn diventa semplificando log primo membro

ht <= logn dove ht è altezza albero

n_h numero di sottoalberi di altezza h

n_h <= 2^ht-h = 2^ht/2^h <= n/2^h

T(n)=Sommatoria di nh * O(h) = O(Sommatoria di h=1 fino a logn di n*h/2^h)=O(n)

```
BuildMaxHeap(A){//O(n) da input array A non ordinato diventa un heap
	A.heapsize=A.length;
	for(i=A.length/2 downto 1)//fino a metà dell'array ossia al penultimo livello di padri siccome gli ultimi padri saranno senza figli
```

```
                MaxHeapify(A,i);
}//A modificato per rappresentare un heap
```

Si può usare l'**Heapsort** per ordinare un array A

- si mette gli elementi in un maxheap
- l'i esimo elemento è il massimo elemento nell'array
- si scambia il primo elemento A[1] con l'ultimo elemento dell'array
- si decrementa la size dello heap di uno
- si mantiene l'ordine dell'heap richiamando maxheapify

```
HeapSort(int []A){//O(n*logn)
        BuildMaxHeap(A);//tempo n
        for(int i=A.length; i > 0; i--){//tempo n*logn
                A[1]<-swap->A[i];//scambio elemento più grande alla fine dell'albero
attuale
                A.heapsize=A.heapsize-1;//l'ultimo elemento essendo il più grande
dell'albero attuale è l'iesimo elemento ordinato e quindi decremento heapsize per
lasciarlo inalterato in A
                MaxHeapify(A,1);
        }
}//A ordinato dal più piccolo al più grande
```

Avremo alla fine ottenuto un A con heapsize=0 ma riportando alla dimensione iniziale avremo un array ordinato dal più piccolo al più grande siccome ogni volta veniva messo nell'ultima posizione il più grande, nella penultima il secondo più grande, terz'ultimo terzo,...

Questo algoritmo non è però STABILE quindi se array ha elementi uguali già ordinati dopo l'ordinamento potrebbero essere stati spostati, il che non è consigliato come caratteristica legata a un argomento futuro, rispetto ai due precedenti.

Min-Heap Code con priorità

Ma heap utile per realizzare code con priorità nonostante i suoi difetti.

Massimo nodo del mio heap. Implementazione con max heap A[1...n] chiave A[i] accesso all'elemento A[i] uso i

```
Max(int []A){//O(1) ritorna il massimo nodo da A heap
        if(A.heapsize >= 1)//il massimo in un heap è sempre il nodo padre
                return A[1];
        else //ho meno di un oggetto ossia zero oggetti
```

```
                error("underflow non ci sono elementi/oggetti in coda");
}//
```

Estra il nodo massimo dal nostro albero binario.

```
ExtractMax(int []A){//O(logn) estrae da A heap il massimo
        if(A.heapsize >= 1){//massimo presente solo se ho almeno un elem
                max=A[1];
                A[1]<-swap->A[A.heapsize];//ragionamento simile heapsort solo che
a questo punto dopo aver diminuito heapsize devo pure riordinare l'albero per
avere il massimo elemento in padre ma l e r sono heap mentre solo 1 primo
elemento ora è forse non heap quindi richiamo MaxHeapify
                A.heapsize--;
                MaxHeapify(A,1);
    }else //ho meno di un oggetto ossia zero oggetti
                error("underflow");
        return max;
}
```

Premessa: Per realizzare Insert e Increase-Key ci serve una Max-Heapfy diversa che invece della proprietà: "A[i] è maggiore o uguale di ogni suo discendente" usa la proprietà simmetrica: **"A[i] è minore o uguale di ogni suo ascendente"!!!**

MaxHeapifyR oppure MaxHeapifyUp

```
MaxHeapifyUp(int []A, int i){//O(log i)
while(i>1 && A[i/2] < A[i])
        A[i/2]<-swap->A[i];//solo A[i/2] può non soddisfarla
        i=i/2;
}
```

Inserisce in A elemento con chiave/elemento k

```
Insert(int []A, int k){//O(log n) inserisco k in A e lo ordino verso l'alto
if(A.heapsize < A.lenght){//ho ancora spazio nel mio heap
        A.heapsize++;
        A[A.heapsize]=k;
        MaxHeapifyUp(A,A.heapsize);
}
```

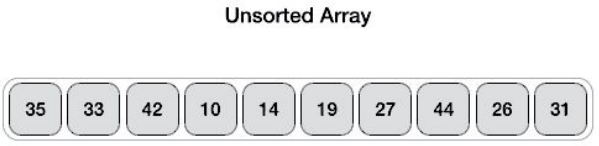
Metodo Incrementa

```
IncreaseKey(int []A, int i, int k){//O(log i)
        if(A[i]>k)
                error("la nuova priorità è minore");
        else{
                A[i]=k;
                MaxHeapifyUp(A,i);//non parte da 1 siccome heap possibilmente non ordinato solo rispetto ai figli che potrebbero avere priorità minore di k mentre ascendenti di A[i] sono per forza con priorità inferiori a k siccome la pre richiede che A sia heap prima della posizione di A[i] sia prima che dopo
        }
}
```

Metodo cambia chiave

```
ChangeKey(int []A, int i, int k){//O(log n)
        if(k >= A[i]){//come IncreaseKey
                A[i]=k;
                MaxHeapifyUp(A,i);
        }else{ //qui cambia la situazione e si passa col ragionamento MaxHeapify
                A[i]=k;
                MaxHeapify(A,i);
        }
}
```

Quicksort



caso peggiore n^2

caso medio = caso migliore nlogn

ideato nel 1961 Sir Tony Hoare, divide et impera con divisione di x in A e ordinamento parte sinistra e parte destra senza far nulla nella combinazione oltre a unione base siccome due parti saranno già ordinate.

Web visualization:
https://www.hackerearth.com/practice/algorithms/sorting/quick-sort/visualize/

```
QuickSort(int []A, int l, int r){//caso peggiore O(n^2) se array già ordinato con l=left r=right
```

```
        if(l<r){
        q=Partition(A,l,r);//q vale posizione di A[r] pivot nella posizione corretta con array ordinato quindi si divide array nelle due parti sinistra e destra
                QuickSort(A,l,q-1);
                QuickSort(A,q+1,r);
        }
}
```

```
Partition(int []A, int l, int r){//O(n)
//fino a i ordinato e da i a j in riordino mentre da j in poi ancora non visitato
        int piv=A[r];//pivot è ultimo elemento tra l e r
        int i=l-1;//i parte da sinistra
        for (int j = l; j > r-1; j++)//percorro array in avanti senza spostare pivot
        {
                if(A[ j ] < piv){//se j ossia i+1 minore di pivot
                        i++; //incremento posizione corretta di pivot
                        A[i]<-swap->A[ j ];//scambio a sinistra minore di pivot
                }
        }
    i++;
    A[i]<-swap->A[r];//scambio elemento più grande di pivot e lo scambio con la posizione corretta che avrà il pivot ossia posizione i
    return i;
}
```

TQS(n)=an+b+T(QS(1/100n)+TQS(99/100n)

- caso medio partizione proporzionale con 1/100n e 99/100n

TQSmedio(n)=an+b+1/n*(Sommatoria da q=1 a n di TQSmed(q-1)+TQSmed(n-q))

=an+b+2/n*Sommatoria da j=0 a n-1 di TQSmed(j)

- alternanza tra partizionamenti bilanciati e non

TQSmedio(n) circa nlogn

Quicksort randomizzato

Nota Bene!!! algoritmo probabilistico (randomizzato) è un algoritmo la cui computazione dipende da una o più scelte casuali poiché generalmente un algoritmo probabilistico ha una complessità migliore nel caso medio rispetto all'equivalente deterministico.

(per maggiori approfondimenti: random)

Randomized_Partition

```
Randomized_Partition(int []A, int l, int r){
        i=Random(l,r);//numero random tra l e r
        A[i]<-swap->A[r];//scambio posizione causale con ultimo elemento
        return Partition(A,l,r);//partiziono col nuoto pivot ottenuto da posizione i
}
```

Randomized_Quicksort

```
Randomized_Quicksort(int []A, int p, int r){//identico Quicksort ma l'unica
differenza è che richiama metodi random
        if(p < r){
                q=Randomized_Partition(A,p,r);
                RandoAlmized_Quicksort(A,p,q-1);
                Randomized_Quicksort(A,q+1,r);
        }
}
```

Limite inferiore con tempo di ordinamento

con input A[1...n], con output B[1...n] dove A[i] appartiene a [0,k] e gli elementi di B sono quelli di A ma ordinati; inoltre ci limitiamo a contare i confronti e in array ma solo SENZA ripetizioni.

Albero di decisione: albero binario

- nodi interni operazioni di confronto i:j -> A[i]<=A[j]
- foglie permutazione dell'input

numero di confronti n! >= altezza dell'albero con N foglie = log2(N) ma quindi nel caso pessimo l'algoritmo deve eseguire almeno log(n!) confronti.

Dimostrando per induzione che log2(n!) >= log2(n/2)^n/2 = n/2 log2(n/2)=1/2log2(n)-n/2= Θ(nlogn) ma quindi Tmax=**Ω(nlogn)** è limite inferiore di complessità del problema dell'ordinamento

Heapsort esegue ordinamento con complessità Tmax pari a **O(nlogn).**

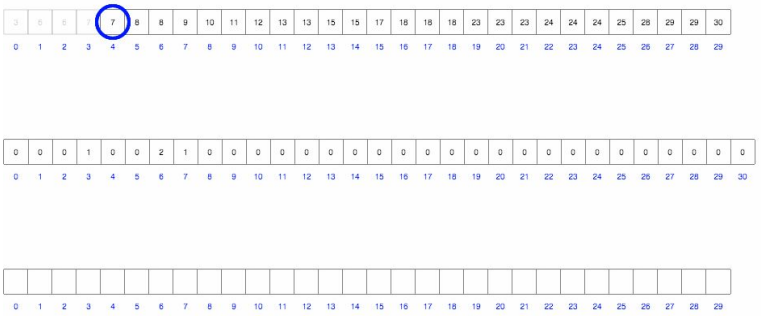
Ω(nlogn) e O(nlogn) => Θ(nlogn)

ATTENZIONE: Il limite inferiore Ω(n log n) da noi dimostrato vale solo per algoritmi di ordinamento generali, ossia algoritmi che non fanno alcuna ipotesi sul tipo degli elementi da ordinare: le uniche operazioni ammesse su tali elementi sono confronti e assegnazioni.

Ordinamento in tempo Lineare

facciamo opportune ipotesi restrittive sul tipo degli elementi possiamo trovare algoritmi più efficienti. Naturalmente il limite inferiore banale Ω(n) vale comunque per tutti gli algoritmi di ordinamento.

Counting sort



N.B. assumendo che gli elementi dell'array siano interi compresi tra 0 e k con k costante.
Θ(n+k) e se k=O(n) allora Θ(n)i **NON è IN PLACE!!**

Web visualization:
https://www.cs.usfca.edu/~galles/visualization/CountingSort.html

```
CountingSort(int []A,int []B, int k){//k=chiave da ordinare
        C[0,k]<-0;//reset di k posizioni in c con valore 0
        for(j=1 to A.length){
                C[A[j]]++;//segno la presenza di elemento A[j] nella relativa posizione
nell'array
        }
        for(i=1 to k){//somma i count con tutti i precedenti incrementale
                C[i]=C[i]+C[i-1];
        }
```

```
        //ora in C[k] abbiamo la posizione in cui inserire la variabile di valore k
        for(j=A.length downto 1){//voglio inserire elemento
                B[C[A[j]]]=A[j];//inserisco nella posizione corrispondente C[A[j]] dell'elemento A[j] nell'array di output B
                C[A[j]]--;//decremento la posizione del mio elemento A[j] utile solo nel caso in C[A[j]] avessi avuto un numero maggiore di 1
        }
}
```

Tabella di dimensione problema array	K	C
64 bit	2^64	512TB
32 bit	2^32	16TB
16 bit	2^16	256MB

Radix Sort

N.B. assumendo che i valori degli elementi dell'array siano interi rappresentabili con al più d cifre in una certa base b.
Origini nel 1877 Hollerith, censimento popolazione con ordinamento crescente delle schede perforate tramite macchinario di ordinamento.
Si ordina array usando d volte l'algoritmo Counting Sort, per esempio, per ciascuna delle d cifre **ma** partendo dalla cifra **meno significativa**.

	4	3	2	1	4	3	2	1	4	3	2	1	4	3	2	1	4	3	2	1
$A[1]$	1	4	2	7	9	8	9	0	8	2	2	3	0	0	3	9	0	0	3	9
$A[2]$	0	2	4	1	0	2	4	1	7	5	2	5	3	1	6	2	0	2	4	1
$A[3]$	7	5	2	5	3	1	6	2	1	4	2	7	8	2	2	3	1	2	3	9
$A[4]$	3	1	6	2	8	2	2	3	1	2	3	9	1	2	3	9	1	4	2	7
$A[5]$	9	8	9	0	7	5	2	5	0	0	3	9	0	2	4	1	3	1	6	2
$A[6]$	1	2	3	9	1	4	2	7	0	2	4	1	1	4	2	7	7	5	2	5
$A[7]$	8	2	2	3	1	2	3	9	3	1	6	2	7	5	2	5	8	2	2	3
$A[8]$	0	0	3	9	0	0	3	9	9	8	9	0	9	8	9	0	9	8	9	0

Web visualization: https://www.cs.usfca.edu/~galles/visualization/RadixSort.html

```
RadixSort(int []A,int d){
        for(j=1 to d){//A[j-1] è ordinato
                //ordina A rispetto alla cifra j ma con un algoritmo stabile ossia fino alla cifra j-esima
                //usiamo per ordinare il Counting Sort
                Acopy=[ ];
                CountingSort(A,Acopy,j);//usa un algoritmo stabile per ordinare A rispetto alla j-esima cifra
```

```
                A=Acopy;
        }//fine ciclata A[j] è ordinato
}
```

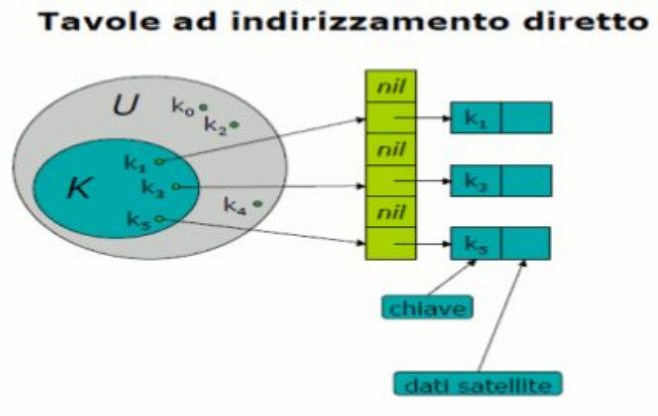
Il tutto con complessità: **Θ(d(n+b)** dove b è la base della numerazione e d è il numero di cifre dei numeri da ordinare nell'array A.

Insiemi Dinamici

x.key Insert(x), Search(key), Delete(x)

Indirizzamento diretto

(non funziona, ma si vede per quello)
universo delle chiavi U={0,1,...,m-1} dove array T non può essere troppo grande.
Chiavi stringhe 8 caratteri pari a (2^8)^8=2^64 per array.
Generalmente T[k] è un puntatore al record con chiave k.
Se la tavola non contiene un record con chiave k allora T[k]=nil(simile a NULL).
Ma con questo metodo avremo una memoria necessaria pari al numero di celle per tutte le chiavi possibili.



Idea: avere una memoria **proporzionale** al N. di elementi con media O(1) ma dimensione della tabella m << |U|
Soluzione: (Funzioni di Hash) usare tabelle hash con h: U =>{0,1,...m-1} con una funzione di hash dove ogni chiave k viene memorizzata nella cella h(k) avendo un totale di m celle.
x ⇒ T[h(x.key)]
Problem: elementi con chiavi Distinte che però tramite h(k1)==h(h2) provocando una **collisione** tra le due chiavi k1 e k2.
Soluzioni: Open addressing oppure liste sottostanti.

Chaining (liste di trabocco)

per il quale T[j]= lista di elementi x tali che h(x.key)=j

- Insert(T,x) con inserimento di x nella lista T[h(x.key)] (testa) **Θ(1)**
- Search(T,k) cerca in T[h(k)] **Θ(n)**
- Delete(T,k) elimina x alla lista T[h(x.key)] **Θ(1)** altrimenti se passiamo chiave **Θ(n)** richiedendo una ricerca con Search dopo il quale elimina effettivamente l'elemento

Ma nel caso in cui tutti gli n elementi siano nella stessa lista della chiave k cercata allora avremo un tempo di ricerca pari a O(n)(come un array lineare quindi)
m=numero di celle di T
n=numero di elementi inseriti
a=n/m fattore di carico della complessità media per Search

Hashing Uniforme Semplice (ideale)
ogni elemento di input inviato dalla funzione di hash in una qualunque delle celle ma TUTTI con la stessa probabilità di essere mandati in una delle m celle.

Ricerca di una chiave k presente: Θ(1+ a/2)= Θ(1+a) dove l'uno e legato alla funzione di hashing per accedere a T[k]

Lo stesso tempo è impiegato per una chiave non presente.

Metodo della divisione

Non funziona bene per tutti gli m.
k => [0,1….,m] da 0 a m-1
h(k)= k mod m

Randomizzazione Hash/ Hash Universale
Siccome un utente potrebbe forzare la funzione facendo inserire forzatamente tutti elementi nella stessa cella allora viene introdotta la randomizzazione che permette di rendere il comportamento della funzione hash indipendente dall'input siccome per ogni inserimento viene usata una funzione di hash scelta da un insieme "universale" di queste.

Così facendo probabilità collisione di due chiavi è pari a 1/m.
E[n per h(k)] quindi la lunghezza attesa della lista per h(k) è pari a a=n/m se k non è presente nella tavola oppure è minore di a+1 se k è presente quindi il Search richiede come tempo medio **Θ(1+a)** ma se n=O(m) è **Θ(1)**.

Metodo di moltiplicazione

scelgo A tra (0,1) ma non uguale a estremi
h(k)=m(k*A mod 1) con m non critico ma A diventa uguale a = $\frac{\sqrt{5}-1}{2}$ inverso rapporto aureo
m=2^p
w= numero di bit della parola di memoria
A=q/2^w tale che 0<q<2^w
Peccato che qualcuno conoscendo la nostra sequenza di hash può sovraccaricare il sistema

Indirizzamento aperto

tutti gli elementi sono memorizzati nella tavola e quindi funzione hash non individua la singola cella ma l'ordine con cui ispezionare le celle.

N.B.!! La variabile “i” indica il numero di elementi inseriti fino ad ora.
L’intestazione diventa quindi h(k,i) dove i varia tra 0 e m-1 detto anche tentativo.

```
Insert(T,k){
        i=0;
        repeat{
                j=h(x.key,i);
                if(T[j] == nil or T[j]== deleted){
                        T[j]=x;
                        return j;
                }
                i++;
        }until(i==m)
}
```

```
Search(T,k){
        i=0;
        repeat{
                j=h(k,i);
                if(T[j].key == k)
                        return j;
                i++;
        }until(i==m or T[j]==nil)
}
```

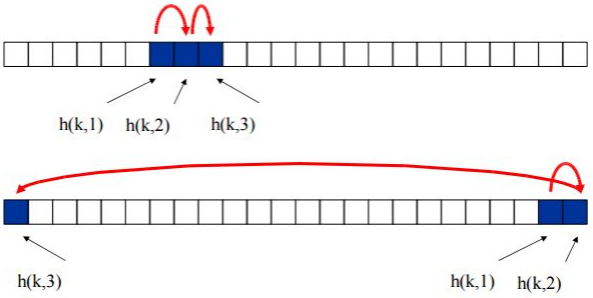
```
Delete(T,j){
        T[j]=deleted;
}
```

Ma con questo indirizzamento la funzione di hashing fornisce sequenza di ispezione quindi ogni chiave la stessa probabilità **1/m!** di generare una qualsiasi delle **m!** possibili sequenze.

Con tre tecniche decidiamo l’ordine ossia il valore ritornato da h(key):

1. **Ispezione lineare (m)**
2. **Ispezione quadratica (m)**
3. **Doppio hash (m^2)**

Ispezione lineare
funzione hash **h(k,i)=[h’(k)+i]mod n.**
Data la chiave k si genera la posizione h’(k) quindi la posizione h’(k)+1 e così via fino a m-1



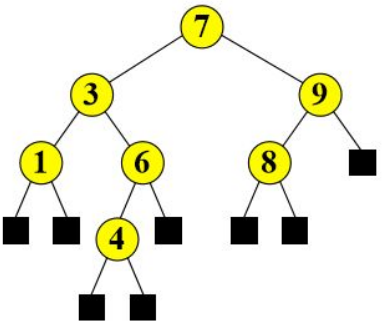
e poi si riparte dalla posizione 0 fino a ritornare h'(k).
Problema: i nuovi elementi inseriti nella tavola tendono ad addensarsi attorno agli elementi già presenti(clustering).

Ispezione quadratica
costruisce una **h(k,i)=(h'(k)+c1i+c2i^2) mod m** dove c1, c2 e m sono costanti vincolate per usare l'intera tabella inoltre evita agglomerazione primaria(clustering isp.lineare) ma produce una agglomerazione secondaria ma risulta essere meno dannosa.

Doppio hash
h(k,i)=(h1(k)+i*h2(k)) mod m permette di avere prima posizione esaminata pari a h1(k) ma le successive distanziate di h2(k) dalla prima, approssimando la proprietà di uniformità rispetto alle isp. precedenti.
Dove h2(k) e m sono primi tra loro e per farlo rendo m sempre numero primo di partenza e poi scelgo un h2 per cui varrà sempre la proprietà h2(k) < m.
Il valore di i riparte da zero per ogni elemento che inserisco e aumenta se ho collisione con calcolo hash e ripete calcolo aumentando i finché si ha soluzione.
Fattore di carico: 0 <= a = n/m <= 1

1. Ricerca di un elemento NON PRESENTE
    a. a=1 m
    b. a < 1 1/(1-a)
2. Ricerca di un elemento PRESENTE
    a. a=1 (m+1)/2
    b. a < 1 1/a*log(1/(1-a)

Alberi binari di ricerca



h altezza albero, operazioni con O(h) se bilanciato h=Θ(logn).
Un albero binario di ricerca è un albero binario in cui ogni nodo x è tale che ogni y nel sottoalbero sinistro ha y.key >= x.key e di quello destro ha y.key <= x.key;
un albero vuoto è rappresentato con un quadratino nero.
In parole povere A è ABR se radice ha tutti nodi sinistri più piccoli di se stesso e quelli di destra più grandi, lo stesso per i sotto rispettivi nodi figli stesso ragionamento.
Web visualization: https://www.cs.usfca.edu/~galles/visualization/BST.html

Stampa lista ordinata nodi

```
Stampa(x){//T(0)=c mentre con T(n)=T(k)+b+T(n-k-1)
        if(x!=nil){
                Stampa(x.left);//stampo parte sinistra minore di x
                print x;//poi quindi x
                Stampa(x.right);//infine la parte maggiore di x
        }
}
```

Ricerca

```
Search(x,k){//Complessità O(h) pari all'altezza dell'albero x
	if(x==nil)or(x.key==k)//se albero nullo ritorno nil altrimenti in tutti gli altri
casi ritorno x siccome sarà figlio esistente per forza
		return x;
	else
		if(k < x.key)//se k minore di nodo attuale vado a sinistra
			return Search(x.left);
		else//altrimenti a destra dove ci sono quelli più grandi di x
			return Search(x.right);
}

Search(x,k){//versione alternativa non ricorsiva ma iterativa
	while(x!=nil)and(x.key != k){
		if(k<x.key)
			x=x.left;
		else
			x=x.right;
	}
	return x;
}
```

Massimo

```
Max(x){//O(h)
	if(x.right == nil)
		return x;
	else//il massimo lo trovo nel nodo più a basso a destra dell'albero dovendo
scendere fino a quando il mio nodo x non possiede figlio destro ossia sono
all'elemento più grande dell'albero
		return Max(x.right);
}
```

Minimo

```
Min(x){//O(h)
	if(x.left == nil)
		return x;
	else//come Max ma discorso opposto ossia albero in basso a sinistra
		return Max(x.right);
}
```

Successore

```
Succ(x){//Successore con x!=nil O(h)
	if(x.right != nil)//se esiste albero a destra di questo voglio il minimo dopo x ossia per forza maggiore di x ma minore di tutti gli altri nodi
		return Min(x.right);
	else{//se albero destro di x è nullo allora il successore è uno degli antenati di x ripercorrendo verso l'alto l'albero
		y=x.p; //y = a genitore di x
		while(y!=nil)and(x==y.right){//ripercorro genitori finché trovo un nodo che risulta essere il figlio sinistro del genitore!!! Questo genitore sarà il successore di x
			x=y;
			y=y.p;
		}
		return y;
	}
}
```

Inserimento di un nuovo elemento

```
Insert(T,z){//
	x=T.root,y=nil;//padre di x
	while(x!=nil){//finché non sono alla fine dell'albero
		y=x;//y variabile locale per ricerca
		if(z.key < y.key)//se z minore di y ossia x vado a sinistra in basso
			x = y.left; //cerco ramo sinistro
		else
			x = y.right; //cerco nel ramo destro
	//modifico x essendo var su cui ricerco
	}
   z.p=y;//metto come genitore del nodo che inserisco y essendo il predecessore a -> x
	if(y==nil)//se y è NULL allora il mio albero è vuoto quindi radice uguale a z
		T.root=z;
	else if(z.key < y.key)//stesso ragionamento di prima solo che devo impostare correttamente il figlio di y se a destra o sinistra mentre z è già con genitore corretto
		y.left=z;
	else
		y.right=z;
}
```

Eliminazione di un elemento

```
Delete(T,z){//z!=nil con complessità O(h)
        if(z.left == nil)or(z.right == nil){//tolgo z
                y=z;//che ha al più un solo figlio
        }
        else{//tolgo il successore di z che non ha sottoalbero sinistro
                y=Succ(z),z.key=y.key;//Succ(z) è elemento che metterò al posto di z
        }
        //cerco l'eventuale unico figlio x di y
        if(y.left == nil)//se figlio di y sinistro
                x=y.right;
        else
                x=y.left;
        //metto x al posto di y
        if(x!=nil)
                x.p=y.p;

        if(y.p == nil)//allora x va messo nella radice
                T.root=x;
        else if(y==y.p.left)//guardo dove mettere elemento se a sinistra genitore
                y.p.left=x;
        else //o a destra del genitore
                y.p.right=x;
}
```

E ora la nuova versione con “trapianto” albero

```
Transplant(T,u,v){
        if(u.p == nil) //se u non ha genitore u è radice
                T.root = v;
        else
                if(u==u.p.left) //se u è un figlio sinistro del genitore
                        u.p.left=v; //v rimpiazza u come figlio sinistro genitore di u
                else
                        u.p.right=v;
        if(v != nil)//se v ha rimpiazzato u e non è nil
            v.p=u.p;//allora v deve avere lo stesso genitore di u
}
```

```
Delete(T,z){
        //caso 1 z non ha figlio sinistro
```

```
        if(z.left==nil)
                Transplant(T,z,z.right);
        else if(z.right == nil){//caso 2 z ha un figlio sinistro ma non uno destro
                        Transplant(T,z,z.left);
                else{
                        y=Min(z.right);
                        if(y.p!=z){//caso 3 y non è figlio di z e ha dei passi aggiuntivi
                                y.right=z.right;
                                z.right.p=y;
                        }
                        //caso 4 y è il figlio destro di z
                        y.left=z.left;
                        z.left.p=y;
                        Transplant(T,Z,y);
                }
        }
}
```

Provando a ordinare array A ottenendo un albero binario di ricerca T

```
ABR_Sort(A){
        new T;
        for( i=1 to A.length){ //Θ(n2)
                Insert(T,A[i]);
}
        InOrder(T); //Θ(n)
}
```

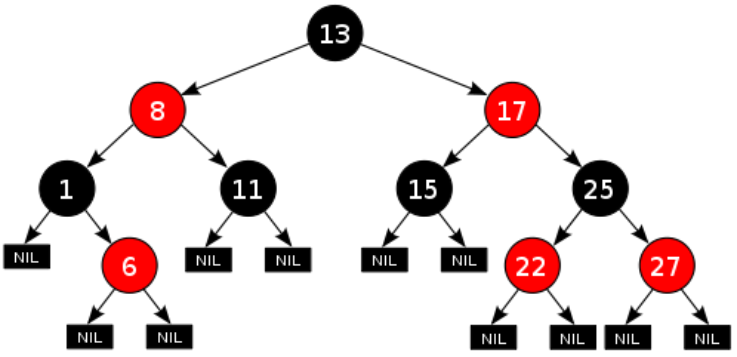
A=[1 2 5 7 8], Produrrà un albero che si sviluppa come un albero solo verso destra se array ordinato in modo crescente. Costo peggiore= $\frac{n(n-1)}{2}$ = $\Theta(n^2)$

Non esiste procedura In Order per maxheap che sia $\Theta(n)$

Alberi Rosso-Neri

(ARN) ossia RedBlack trees: ossia RB-tree sono ABR con nodi
aggiunta di un bit per il colore quindi
x.color{nero o rosso}
con le seguenti proprietà:

1. ogni nodo ha colore **rosso** o **nero**
2. radice è **nera**
3. foglie(T.nil) sono **nere**



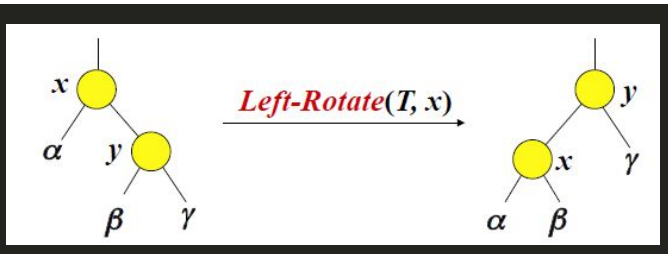
4. nodo **rosso** ha entrambi i figli **black**
5. per ogni nodo n, tutti i percorsi da n a una qualsiasi delle sue foglie discendenti contiene lo stesso numero di nodi **neri**.

Web visualization: https://people.ok.ubc.ca/ylucet/DS/RedBlack.html

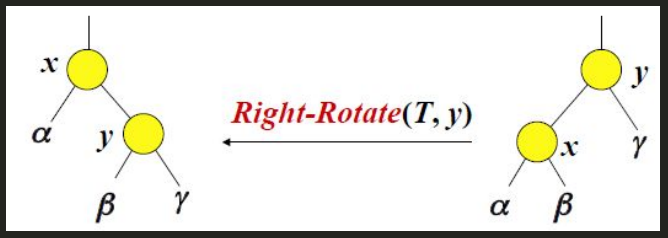
Questa **caratteristica rende** possibile avere l'albero approssivamente **bilanciato**. Presentano quindi una Insert e Delete opportunamente modificate per garantire tale proprietà. Osservo che: T è RB-Tree con n nodi interni(non foglia) e h è altezza tale che **h <= 2 log(n+1) = Θ(log n)**

Da questa proprietà capiamo che ARN con n nodi interni le operazioni di Search,Max,Min,Succ,Predec richiedono tutte tempo O(log n)
Mentre le Insert e Delete richiedono sempre tempo O(log n) ma siccome possono introdurre delle violazioni delle proprietà richiedono del tempo extra per ripristinarle tramite delle rotazioni.

```
LeftRotate(T,x){//Θ(1)
        y=Right(T,x);//y=x.right
        x.right=y.left; y.left.p=x;

        Transplant(T,x,y);

        y.left=x;
        x.p=y;
}
```




```
RightRotate(T,y){//Θ(1)
      x=y.left;//x non deve essere la sentinella T.nil
      y.left=x.right,x.right.p=y;

       //Inizio metodo Transplant
      //x.right non può essere T.nil
     x.p=y.p;
      if(y.p == T.nil)
            T.root=x;
      elseif(y==y.p.left)
            y.p.left=x;
      else
            y.p.right=x;
       //Fine metodo Transplant

      y.p=x;     x.right=y;
```



```
}
```

Inserimento di un nuovo elemento

```
RBInsert(T,z){//z.left=z.right=T.nil
        Insert(T,z);
        z.color=RED;
        RBInsertFixUp(T,z);
}
```

Sistemazione delle proprietà violate inserendo elemento in albero rosso nero

```
RBInsertFixUp(T,z){//O(log n)
        while(z.p.color=RED){//per ogni ciclo il puntatore z risale di due posizioni
quindi ripetuto al massimo h volte
                if(z.p = z.p.p.left){
                        y=z.p.p.right;
                        if(y.color=RED){
                                z.p.color=y.color=BLACK;
                                z.p.p.color=RED;
                                z=z.p.p;
                        }else if(z==z.p.right){
                                z=z.p;
                                LeftRotate(T,z);
                                //z figlio sinistro
                                z.p.color=BLACK;
                                z.p.p.color=RED;
                                RightRotate(T,z.p.p);
                        }
                }else{//MACRO CASO B
                        z.p=z.p.p.right;

                        //simmetrico con right e left scambiati, CODICE
SOTTOSTANTE
                        //da ricontrollare
                        if(y.color=RED){
                                z.p.color=y.color=BLACK;
                                z.p.p.color=RED;
                                z=z.p.p;
                        }else if(z==z.p.left){
                                z=z.p;
```

```
                        RightRotate(T,z);
                        //z figlio destro
                        z.p.color=BLACK;
                        z.p.p.color=RED;
                        LeftRotate(T,z.p.p);
                    }

            }
        }//end while
        T.root.color=BLACK;
}
```

Cancellazione di un elemento

```
RBDelete(T,z){
      if(z.left==T.nil or z.right == T.nil)
            y=z;
      else
            y=Succ(z),z.key=y.key;
      //elimino y che ha almeno un sottoalbero
      if(y.left == T.nil)
            x=y.right;
      else
            x=y.left;
            //x sottoalbero di y, l'altro è sicuramente vuoto
            //metto x al posto del padre y
            x.p=y.p;
      if(y.p == T.nil)
            T.root=x;
      else if(y == y.p.left)
            y.p.left=x;
      else
            y.p.right=x;
      //Se y è rosso non ci sono violazioni
      if(y.color==BLACK) //se y era nero violazione è che i cammini
che passano per x contengono un nodo nero in meno quindi lo
sistemo
            RBDeleteFixup(T,x);
}
```

```
RBDelete_Fixup(T,x){//O(log n)
```

```
      while(x!=T.root and x.color==BLACK){//al massimo ripetuto h
volte
      if(x==x.p.left)
            w=x.p.right;
            if(w.color == RED){//Caso 1
                  w.color=BLACK;
                  x.p.color=RED;
                  LeftRotate(T,x.p);
                  w=x.p.right;
            }
            if(w.left.color==BLACK && w.right.color=BLACK){//Caso
2con w che sarà per forza nero per condizione sopra non rispettata
                  w.color=RED;
                  x=x.p;
            }else if(w.right.color == BLACK){//Caso 3
                  w.left.color=BLACK;
                  w.color=RED;
                  RightRotate(T,w);
                  w=x.p.right;
                  w.color=x.p.color;
                  x.p.color=w.right.color=BLACK;
                  LeftRotate(T,x.p);
                  x=T.root;
            }else{//Caso 4
                  w.left.color=BLACK;
                  w.color=RED;
                  LeftRotate(T,w);
                  w=x.p.right;
                  w.color=x.p.color;
                  x.p.color=w.right.color=BLACK;
                  RightRotate(T,x.p);
                  x=T.root;
            }
      }
      x.color=BLACK; //Caso 0
}
```

Arricchimento di strutture dati

Per esigenze di alcuni problemi algoritmici è possibile progettare una struttura dati appropriata aumentando una struttura già nota.

Estendiamo ALBERI_ROSSO_NERI con Select(k) che ritorna il nodo con la chiave k-esima. Aggiungeremo a ogni nodo il campo intero **x.size** in cui memorizzare il numero di nodi interni del sottoalbero di radice x.

tutte le operazioni di ABR + select(i)=elemento di posizione i nella sequenza ordinata
rank(k)= posizione di x dell'ordine

```
Select(x,i){
      //nodo posizione i per il sottoalbero Tx con 1 <= i <= x.size
      r=x.left.size+1;//posizione di x
      if(i == r)
            return x;
      else if(i<r)
            return Select(x.left,i);
      else
            return Select(x.right,i);
}
```

Trova posizione k della chiave di x nell'albero T

```
Rank(T,x){//devo risalire fino a quando trovo la radice albero
      r=x.left+1;
      y=x;
      while(y!=T.root){
            if(y = y.p.right)
                  r=r+y.p,left+size+1;
            y=y.p;
      }
      return r;
}
```

```
Insert(T,z){
      z.size=1; //istruzione aggiunta
      X=T.root;
      y=T.nil;
      while(x!=T.nil){
            x.size=x.size+1; //istruzione aggiunta
            y=x;
            if(z.key < x.key)
                  x=x.left;
            else
                  x=x.right;
      }
```

```
            ...
            come originale
            ...
}
```

```
LeftRotate(T,z){
      ...//uguale come precedente funzione senza size
      y.size=x.size;   //istruzioni aggiunte
      x.size=x.left.size + x.right.size + 1;
}
```

```
Delete(T,z){
      ...//uguale come precedente funzione senza size
      w=x.p;
      while(w!=T.nil){
            w.size=w.size-1;
            w=w.p;
      }
      if(y.color==BLACK)
      RBDelete_Fixup(T,x)
}
```

Teorema dell'arricchimento ABR:
sia x.field tale che: x.field si calcola in tempo O(1) usando x, x.left e x.right allora posso ridefinire Insert, Delete in modo da mantenere field con complessità O(logn). Osservo infatti x.field dipende solo dai discendenti Tx, non sale verso genitori ma questo vuol dire che è necessaria la modifica di field degli antenati di x non dei suoi discendenti.

```
RB_Insert(T,z){
      Insert(T,z);
      z.color=RED;
RB_Insert_Fixup(T,z);
}
```

Teorema dell'arricchimento generale:

1. scelta della struttura dati di base
2. scelta delle ulteriori informazioni da memorizzare nella struttura
3. verifica che esse si possano mantenere durante le operazioni della struttura di base senza aumentarne la complessità asintotica.
4. sviluppo delle nuove operazioni

Insert(T,z) e LeftRotate(T,z) e RightRotate(T,z) avranno questa sezione in più alla fine del corpo che viene eseguito:

```
w=x;
while(w!=T.nil){
      Ricalcola_Field(w);
      w=w.p;
}
```

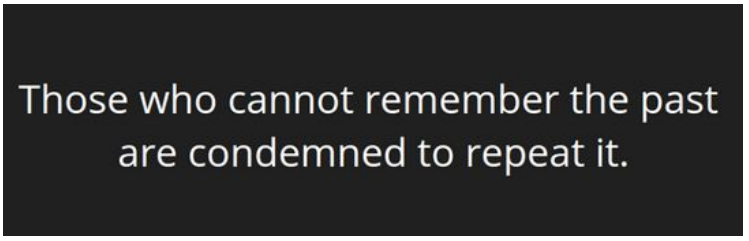
ABR aumentati con operazioni su intervalli i=[low, high]
nodi x avranno → x.i & [x.int.low, x.int.high]

```
IntervalSearch(x,i){//O(logn)
      if(x==T.nil or x.int.intersezione con i!=0)
            return x;
      else if(x.left != T.nil and x.left.max >= i.low)//esiste in x.left
nodo con intervallo i'
            return IntervalSearch(x.left,i);
      else
            return IntervalSearch(x.right,i);//x.left.max < i.low
}
```

i'.low <= i.high --> i' è ok
i'.low > i.high --> i' intersezione i è uguale a vuoto

Programmazione Dinamica

Il fondamento è pure basato su sottoproblemi che memorizziamo e risolveremo solo una volta se si presenteranno casi identici che verranno riutilizzati basandosi sul primo caso base calcolato (vedi foto sopra).



Sottoproblemi non indipendenti. **Sviluppo con seguenti passi:**

1. caratterizzazione strutturale di una soluzione ottima
2. definizione ricorsiva di un metodo per il calcolo di una soluzione ottima
3. algoritmo calcolo valore soluzione ottima con schema bottom-up
4. algoritmo soluzione ottima dalle informazioni raccolte nella fase precedente

Si può pensare alla programmazione dinamica come un algoritmo di riempimento di tabelle: si conoscono tutti i calcoli e devo scegliere l'ordine migliore e ignorare quelli che non bisogna usare per il risultato massimo.

RodCutting problem:
Web visualization: http://rosulek.github.io/vamonos/demos/rod_cutting.html
input: lunghezza totale asta n
P1,...Pn → Pi = prezzo asta di lunghezza i
output: ricavo massimo da asta di lunghezza n (e un taglio)

L	1	2	3	4	5	6	7
Prezzo	1	5	8	9	10	17	17

n=7 il ricavo massimo avrò:

- 6+1 → 18
- 2+2+3 → 18

Problema abbiamo 2^{n-1} possibilità da contare per tutti i possibili tagli

Espressione ricorsiva costo massimo (ottimo)

r0=0

rn=max tra 1 <= i < =m {Pi+r_{n-1}} con n>0

```
CutRod(p,n){//T(n^2)
      if(n=0)
            return 0;
      else{
            q=-1;
            for(i=1 to n)
                  q=max{q,CutRod(p,n-1)+p[i]};
      }
      return q;
}
```

Due modi per risolvere in modo differente:

1. TOP DOWN con memorisation (tabella degli ottimi dei sottoalberi)

```
MemorisedCutRod(p,n){
      for(i=0;i<n;i++){
            r[i]=-1;
      }
      return MemorisedCutRodAux(p,n,r);
}
```

```
MemorisedCutRodAux(p,j,r){
      if(r[j]<0)
            if(j==0)
                  r[j]=0;
            else{
                  q=-1;
                  for(i=1 to n)
                        q=max{q,p[i]+MemorisedCutRodAux(p,j-1,r)};//se ho
già risolto il problema non faccio nulla
                  r[j]=q;
                  }
      return r[j];
}
```

2. BOTTOM UP risolvo i problemi dal più piccolo al più grande

```
BottomUpCutRod(p,n){//migliore non essendo ricorsiva a parità di
complessità ma più facile da scrivere
      //alloca r[0...n]
      r[0]=0;//il problema più semplice
      for (j=0; j < n; j++){
            q=-1;
            for(i=1; i < j; i++)
                  q=max(q,r[j-i]+p[i]);
            r[j]=q;//assegno il massimo che ho trovato
      }
      return r[n];
}
```

Calcolo della soluzione ottima:
r[0...n] r[i]=valore ottimo per lunghezza i
s[1...n] s[i]=posizione del primo taglio che realizza l'ottimo

```
BottomUpCutRodExt(p,q){//O(n^2)
      for(i=1 to n){
            r[i]=-1;
      }
      r[0]=0;
      for (j=0; j < n; j++){
            q=-1;
            for(i=1; i < j; i++)
                  if(q < p[i]+r[j-i]){
                        s[j]=i;//memorizzo il taglio ottimo
                        q=p[i]+r[j-i];
                  }
      return r,s;
}
```

```
PrintCutRodSolution(p,n){//O(n^2)
      r,s=BottomUpCutRodExt(p,n);//ottengo i due array
      j=n;
      while(j>0){
            print s[j];
            j=j-s[j];//devo togliere all'asta il pezzo che ho rimosso
      }
}
```

Moltiplicazione di matrici:
A(pXq) x B(qXr) = C(pXr)

C[i,j]=sommatoria A[i,k]*B[k,j]
costo = numero di moltiplicazioni = p*q*r
A1: 10x100 A2: 100x5 A3: 5x50

http://computerscience.unicam.it/merelli/algoritmi06/%5b09%5dmatrixChain.pdf

Parentetizzazione di Matrici: A1 A2 … An
P(n)= numero di differenti parent = $\Omega(2^n)$ con P(n)>= c 2^n quindi provare tutte comb possibili non risulta possibile.

1. Sottostruttura ottima sarebbe una frazione da Ai … Aj 1<=i<=j<=n
2. Soluzione ricorsiva dei sottoproblemi dei prodotti parziali $A_{i..j}$
3. Calcolo del costo ottimo ossia minimo ricorsivo

```
MatrixCost(p,i,j){
      if(i=j)
            return 0;
      else
            cmin=infinito;
      for(k=1 to j-1){
            q=MatrixCost(p,i,k)+MatrixCost(p,k+1,j)+ p[i-1] * p[k]
* p[j];
            if(q < cmin)
                  cmin=q;
      }
      return cmin;
}
```

4. procedimento top-down e bottom-up

```
MatrixCost(p,n){
      for(int i=1; i<n; i++){
            for(int j=i to n){
                  m[i,j]=infinito;
            }
      }
      return MatrixCostRec(p,1,n,m);
}
```

```
MatrixCostRec(p,j,n,m){
      if(m[i,j]==infinito){
            if(u==j)
                  m[i,j]=infinito;
            else{
                  for(k=i to j-1){
```

```
q=MatrixCostRec(p,i,k,m)+MatrixCostRec(p,k+1,j,m)+p[i-1] * p[k] *
p[j];
                        if(q < m[i,j])
                              m[i,j]=q;
                    }
              }
        }
        return m[i,j];
}
```

```
BottomUpMatrixCost(p,n){//
        for(i=1 to n)//O(n)
              m[i,j]=0;
        for(l=2 to n)//O(n^3)
              for(i=1 to n-l+1){
                    j=i+l-1;
                    for(k=i to j-1){
                          q=m[i,k]+m[k+1,j]+p[i-1]*p[k]*p[j];
                          if(q<m[i,j])
                                m[i,j]=q;
                                s[i,j]=k;
                          }
              }
        return m[1,n];
}
```

```
PrintOptimum(s,i,j){
        if(i=j)
              cout << "Ai";
        else{
              cout << "C";
              PrintOptimum(s,i,s[i,j]);
              cout << ")(";
              PrintOptimum(s,s[i,j],j);
              cout << ")";

        }
}
```

Problemi risolvibili con Programmazione Dinamica:
Ma ora abbiamo risolto due problemi con questa programmazione dinamica **ma quali altri problemi possiamo risolvere con questa tecnica?**
Tutti i **problemi di Ottimizzazione**, in cui da un insieme generalmente molto grande di soluzioni possibili vogliamo estrarre una soluzione ottima rispetto ad una determinata misura.
Le condizioni vantaggiose sono:

1. **sottostruttura ottima:** soluzione ottima che si possa costruire partendo da soluzioni ottime dei sottoproblemi
2. **ripetizione di sottoproblemi:** numero di sottoproblemi distinti sia molto minore del numero di soluzioni possibili tra cui cercare soluzione ottima ossia uno stesso problema deve comparire **molte** volte come sottoproblema di altri sottoproblemi

Con queste due condizioni verificate dobbiamo scegliere l'ordine con cui calcolare soluzioni sottoproblemi:

1. bottom-up
2. top-down ricorsivo con memorizzazione soluzioni trovate in modo che non vengano ricalcolate più volte

Attenzione se per il calcolo della soluzione globale finale servono tutte le soluzioni di tutti i sottoproblemi utilizzare bottom-up conviene rispetto a top-down siccome quest'ultimo è ricorsivo ed effettua un controllo in più.
Ma top-down non calcola tutte le soluzioni ma solo quelle che servono effettivamente per il costo ottimo e quindi conviene in casi come il seguente:
Cammini su grafi: da una sorgente a una destinazione, sottostruttura ottima?
m(s,t) = lunghezza del cammino minimo
m(s,t)=0 se s=t
m(s,t)=1+min di tutti gli archi uscenti

LCS(LongestCommonSequence) Massima sottosequenza comune:
Web visualization: http://www.cs.usfca.edu/~galles/visualization/DPLCS.html
Date due sottosequenze X=x1x2 … xn e Y=y1y2 … yn, Focus: trovare la più lunga sequenza Z=z1z2 … zn che è **sottosequenza sia** di X che di Y.
1) Sottostruttura ottima, idea: riduzione ai prefissi:

1. se Xm=Ym allora Wk=Xm e W(k-1) è in LCS(X(m-1),Y(m-1))
2. altrimenti se sono diversi
    a. se Wk!=Xm allora W è in LCS(X(m-1),Y)
    b. se Wk!=Ym allora W è in LCS(X,Y(m-1))

2) Calcolo ricorsivo della lunghezza LCS
X=ABACA
Y=ACDAB
C=matrice con valori X(varia la i) e Y(varia la j) spostandosi dall'ultima casella in basso a destra dei questa cercando da risalire fino a raggiungere un bordo della tabella sinistra o bordo superiore che avendo epsilon vale zero.
3) Soluzione Bottom-Up:
C[i,j] b[i,j] e spostandomi in diagonale se xi=yj superiore sinistra altrimenti se xi!=yk e C[i,j]=C[i-1,j] vado a sinistra altrimenti vado verso l'alto.

Triangolazione ottima

Greedy

non è un algoritmo ma una tecnica dove viene sempre fatta la scelta che sembra la migliore al momento, ottima localmente e si spera solamente sia ottima globalmente.
Un aspetto negativo è che essendo facile avere idee greedy è difficile provarne la correttezza che faremo quasi sempre come segue:

- prova per contraddizione assumendo che quello che vogliamo dimostrare sia falso in questo caso assumendo che l'algoritmo greedy non produce una soluzione ottima e che esiste un'ulteriore soluzione che è migliore di quella proposta.

1. Sottoproblema con Sottostruttura Ottima
2. scelta(greedy), ossia esiste **sempre** una soluzione del problema **ottima** che include la scelta greedy e ciò che resta della soluzione senza la scelta greedy; oppure: soluzione ottima del sottoproblema e aggiungo la scelta greedy ottengo soluzione ottima del problema originale se A1 è soluzione ottima di S1 allora A1∪{a1} è ottima per S.

(spiegare che scelta giusta non impedisce di ottenere soluzione ottima)
esempio pratico con selezione di attività
S: a1 a2 a3 ... an
a1 ha si inizio e fi fine [si,fi]
ai e aj compatibili se [si,fi] intersecato con [sj,fj]
AIM:trovare A contenuto S compatibili, massimo.
Sottostruttura ottima:
|--ai--|fi(sij)sj|--aj--|
sij={am | fi =< sm intersecato fm <= sj}

```
ActivitySelector(s,f){//s e f ordinati in senso crescente
  n=s.length;
  A.push_back(a(1));
  k=1;
  for(m=2;m<n;m++){
    if(s[m]>=f[k]){
      A.push_back(a(m));
      k=m;
    }
  }
  return A;
}
```

Knapsack ossia problema dello zaino

```
Knapsack(q,c,Q){
  Spazio=Q;
```

```
  n=length(q||c)
  for(i=1;i<n;n++){
    if(Spazio >= q[i]){
      Z[i]=q[i];
      Spazio=Spazio-Z[i];
    }else{
      Z[i]=Spazio;
      Spazio=0;
    }
  }//for i
  return Z;
}
```

Versione aggiornata del prof:
C[i,Q]= 0 se i=0 oppure Q'=0
C[i-1,Q'] se qi > Q'
max{ciqi + C[i-1,Q'-qi],C[i-1,Q']

```
Knapsack(q,c,Q){
      n=q.length;
      for(Q1=0 to Q){
            C[0,Q1]=0;
            for(i=1 to n)
                  for(Q1=1 to Q){
                        if(q[i]>Q1)
                              C[i,Q1]=C[i-1,Q1];
                        else{
                              vyes=C[i]*q[i]+C[i-1,Q1-q[i]];//metto
oggetto nello zaino
                              vno=C[i-1,Q1];//non lo metto
                              if(vyes > vno)//guardo il massimo e tengo
oggetto C[i,Q1]
                                    C[i,Q1]=vyes;
                              else
                                    C[i,Q1]=vno;
                        }
                  }
            }
            return C[n,q];
}
```

Algoritmo goloso di Huffman
https://people.ok.ubc.ca/ylucet/DS/Huffman.html
Codice pref come Alberi PRECEDENZA A FOGLIE RISPETTO NODI GENERATI
Albero binario:

- foglie = simboli
- archi etichettati 0/1
- codice=circa cammino la radice al simbolo
- Leggo in discesa il valore degli archi da numero più significativo al primo valore

```
Nodo(f,c){...} //costruttore dei nodi foglia
Nodo(x,y){...} //costruttore dei nodi interni
Huffman(c,f,n){//O(n logn) siccome Insert e ExtractMin richiedono tempo
O(nlogn)
  //c array caratteri f array frequenza n numero totale
  Q=0;//coda con priorità
  for(i=1;i<n;i++){
    Push(Q,Nodo(f(i),c(i)));
  }
  for(j=n;n>2;j--){
    x=ExtractMin(Q);
    y=ExtractMin(Q);
    Push(Q,Nodo(x,y));
  }
  return ExtractMin(Q);
}
```

Differenza tra tecnica Greedy e programmazione Dinamica:

La differenza principale è il fatto che invece che trovare prima tutte le soluzioni ottime dei sottoproblemi e poi fare una scelta ponderata, la tecnica greedy fa per prima cosa una scelta immediata che sembra essere la migliore al momento attuale e poi risolve i sottoproblemi senza preoccuparsi di risolvere tutti i sottoproblemi relativi.

Analisi ammortizzata

Se nel caso pessimo del tempo richiesto per l'esecuzione di una sequenza di operazioni, in queste le operazioni costose sono meno frequenti allora il costo richiesto per eseguirle è ammortizzato con l'esecuzione di quelle meno costose.
Complessità di n operazioni caso pessimo = O(f(n)) mentre il suo costo ammortizzato è ottenuto dividendo per n tale complessità = O(f(n)/n).

Costo medio di un'operazione: considero op1,...,opn

1. Aggregazione, complessità peggiore O(f(n)) concludo che in media opi vale O(f(n)/n)
   Esempio: Stack P, inserimento rimozione, top, empty. Multipop(P,k)

Multipop(P,k){//O(n) nel caso peggiore sarebbe O(n^2) ma siccome rimuovendo elemento
i=k;
while(not Empty(P)and(i>=0))

```
        pop(P);
        i=i-1;
}
```

2. Accantonamenti, si caricano le operazioni meno costose di un costo aggiuntivo assegnato come credito prepagato a certi oggetti che saranno usati per pagare le operazioni più costose su tali oggetti.

3. Potenziale associa alla struttura dati D un potenziale f(D) tale che un'operazione meno costosa comporti aumento del potenziale mentre quelle più costose lo fanno diminuire, il costo sarà somma algebrica del costo effettivo e della variazione di potenziale.

Applicazioni dell'analisi ammortizzata sono:

- espansione contrazione delle tabelle hash
- heap di Fibonacci
- compressione dei cammini su insiemi disgiunti

B-ALBERI:

sono struttura dati che permettono la rapida localizzazione dei file records o keys, derivano dagli alberi di ricerca in quanto ogni chiave appartenente al sottoalbero sinistro di un nodo è di valore inferiore rispetto ogni chiave che appartiene ai sottoalberi alla sua destra,
La loro peculiarità è il fatto che ogni nodo è bilanciato ossia le altezze del sottoalbero sinistro o destro differiscono al più di una unità, permettendo operazioni di inserimento, cancellazione e ricerca in tempi ammortizzati.

Struttura di un B-tree