Algoritmi e Strutture Dati 2023-24
(M. Benerecetti)

Contents

1 Lezione 02 - 15/09/2023

1.1 Algoritmo di Conteggio

Descrivere un algoritmo che accetta come input un intero $N \geq 1$ e produce in output il numero di coppie ordinate $i, j \in \mathbb{N} \quad (i, j) : 1 \leq i \leq j \leq \mathbb{N}$
Esempio:

- Input:N=4
- Output:10 {(1,1),(1,2),(1,3),(1,4),(2,2),(2,3),(2,4),(3,3),(3,4),(4,4)}

```
1 Conta(N):
2   ris = 0; //Assegnamento costante (1 operazione elementare)
3   for i=1 to N do //Assegnamento/Incremento + confronto (2 op. elementari)
4     for j=1 to N do //idem for di sopra
5       if i<=j then //2 letture + confronto (3 op. elementari)
6         ris = ris+1 //lettura+scrittura+assegnamento (3 op. elementari)
7   return ris //1 op. elementare
```

Ogni riga ha un costo che corrisponde alle operazioni elementari effettuate, mediamente ogni op. elementari ha un costo di 1 unità di tempo, prendiamo come esempio il for al primo giro: Assegnamento + Confronto (2 op. elementari), invece i successivi giri: Incremento+confronto (2 op. elementari).

Ognuna di queste operazioni (righe) vengono eseguite più di una volta, quindi il costo sarà maggiore, andiamo ad esprimerlo:

- 2) Costo = 1 (fuori dal ciclo)
- 3) La testa viene eseguita $n+1$ volte poiché abbiamo anche l'ultima operazione per uscire dal ciclo, quindi Costo $= 2 * (n+1) = 2 * \sum_{i=1}^{N+1} 1$
- 4) Questo for verrà ripetuto N volte poiché il corpo del for viene eseguito N volte, quindi il suo costo sarà:

$$\underbrace{2}_{\text{costo dell'operazione}} * \underbrace{\sum_{i=1}^{N}}_{\text{for esterno}} \underbrace{\sum_{j=1}^{N+1}}_{\text{for interno}} 1$$

- 5) L'if stando in entrambi i for avrà un costo di: $3 * \sum_{i=1}^{N} \sum_{j=1}^{N+1} 1$
- 6) Questa operazione non ha un numero fisso di volte di esecuzione. Pertanto e necessario stabilirne un algoritmo per decretarne il numero. Pensandoci il numero di volte che questa operazione esegue dipende da N e dall' i fissate in precedenza. Calcolando, anche banalmente a mano, quante operazioni vengono eseguite ci troveremo con:

$$N - i + 1 \text{volte che l'operazione viene eseguita.}$$

- 7) Costo = 1 (fuori dal ciclo)

Dopo che viene effettuata l'analisi, possiamo andare a sommare tutti i risultati che abbiamo ottenuto in termini di unita (correggere accento) di tempo.
La funzione $T(n)$ e(correggere) la funzione che ci tiene traccia della complessita dell'algoritmo.

Andiamo semplicemente a sommare i nostri risultati di ogni riga:

$$T(n) = 1 + 2 * (N) + 2 * (N^2 + N) + 3 * \frac{N(N+1)}{2}$$

Questo risultato e ottenuto semplificando le nostre sommatorie:

- 3) $2 * \sum_{i=1}^{N+1} 1 = 2 * (N + 1)$
- 4) $2 * \sum_{i=1}^{N} \sum_{i=1}^{N+1} 1 = 2 * \sum_{i=1}^{N} N + 1 = 2 * (N^2 + N)$
- 5) $3 * \sum_{i=1}^{N} \sum_{i=1}^{N} 1 = 3 * \sum_{i=1}^{N} N = 3N^2$
- 6) $\sum_{i=1}^{N}(N-i+1) = N-(k-1)$ cioe ad ogni ciclo il numero delle volte che viene eseguita questa operazione diminuisce costantemente di 1 (cioe dipendente dal salire di i).

$$T(n) = \frac{13}{2}N^2 = \frac{9}{2}N + 4$$

Come vediamo questa funzione e quadratica, quindi cresce esponenzialmente nel tempo, molto pesante e lenta come funzione.

1.1.1 Algoritmo di conteggio v2

Dopo aver ottenuto i risultati dell'analisi sopra, possiamo dire che e sicuramente possibile semplificare il nostro codice in modo tale da far eseguire meno operazioni al nostro processore e quindi utilizzare meno tempo.

```
1 Conta(N):
2   ris = 0;
3   for i=1 to N do
4     ris = ris + (N-i+1)
5   return ris
```

Cosi facendo abbiamo semplicemente detto al nostro codice che deve sommare soltanto gli elementi che nel momento in cui i e fissato, sono $\leq$ di se stesso.
Cosi facendo si dovrebbero eliminare molte operazion inutili, analizziamo.

- 2)sempre 1 operazione
- 3) $2 * \sum_{i=1}^{N+1} 1$
- 4) $\sum_{i=1}^{N} 7 = 7 \cdot N$

Come possiamo osservare abbiamo eliminato il secondo for, dunque abbiamo eliminato la quadraticita, ora l'operazionea riga 4 viene eseguita solamente N volte, e il numero di operazioni semplici che esegue e fissato.

$$T(n) = 1 + 2(N + 1) + 7N + 1 = 9N + 4$$

1.1.2 Algoritmo di conteggio v3

Da come possiamo notare e possibile di nuovo semplificare l'ultima sommatoria della riga 4 dello scorso algoritmo.

Da

$$\sum_{i=1}^{N} N - i + 1 \rightarrow \sum_{i=1}^{N} i$$

Il risultato della sommatoria e lo stesso, se andiamo a semplificarlo.

$$\sum_{i=1}^{N} N - i + 1 = \sum_{i=1}^{N} i = \frac{N(N + 1)}{2}$$

```
Conta(N):
  ris = 0
  ris = ris+(N-i+1)
  return ris
```

In questo modo abbiamo eliminato qualsiasi ciclo e quindi il risultato sara un numero fisso di operazioni.

- 1) 1 operazione
- 2) 5 operazioni elementari

$$T(n) = 5 + 1 = 6$$

In questo caso la funzione tempo per eseguire queste operazioni e costante, non dipendente da nessun N, dunque e la migliore soluzione possibile per questo algoritmo.

1.2 Algoritmo della massima sottosequenza contigua

Preso un array di n elementi, vorremmo provare a trovare la sottosequenza la cui somma di tutti i valori e massima.

Come fare? La soluzione piu naive possibile e effettuare 3 cicli for innestati:

```
int Max_seq_sum_1(int N, array a[])
  maxsum = 0
    for i=1 to N
      for j=i to N
        sum = 0
        for k=i to j
```

```
          sum = sum + a[k]
        maxsum = max(maxsum,sum)
  return maxsum
```

Come possiamo notare l'avere 3 for innestati ci fa avere una complessita computazionale di N^3, comunemente rappresentato dalla formula della notazione asintotica $O(N^3)$.

1.2.1 Algoritmo v2

Come e facile notare, nel terzo for innestato c'e una ripetizione abbastanza inutile di operazioni che effettuiamo per andarci a sommare i valori delle sottosequenze. In particolare andiamo a ripetere scorrere piu volte lo stesso numero di celle, solamente per trovare la sottosequenza poco piu grande (a volte anche di una cella).

Gli stessi valori delle sottosequenze possono essere semplicemente trovati scorrendo avanti l'indice e sommando alla sequenza precedente il valore successivo. Analiticamente ci troveremmo che:

$$\sum_{k=i}^{j+1} a_k = a_{j+1} + \sum_{k=i}^{j} a_k$$

Il valore sum rimarra inalterato, ma verra solamente aggiornato del valore successivo. Il codice si presentera in questo modo:

```
int Max_seq_sum_1(int N, array a[])
  maxsum = 0
    for i=1 to N
      sum = 0
      for j=i to N
        sum = sum + a[j]
        maxsum = max(maxsum,sum)
  return maxsum
```

2 Lezione 03 - 21-09-2023

2.1 Algoritmo v3

L'algoritmo può essere anche migliorato, riuscendo ad arrivare ad una complessità **lineare**, nel seguente modo:

```
int Max_seq_sum_3(int N, array a[])
  maxsum = 0
  sum = 0
  for j=1 to N
    if (sum + a[j] > 0) then
      sum = sum + a[j]
    else
      sum = 0
    maxsum = max(maxsum,sum)
  return maxsum
```

Il ragionamento è il seguente: Se prendiamo un insieme di numeri da sommare, (da i ad a), possiamo controllare se esso è positivo o negativo. Nel caso in cui $\sum_{e=i}^{a} A[e]$ risultasse positiva, andiamo a espandere il nostro range fintantochè il risultato della sommatoria riamanga positivo. Nel caso in cui invece il risultato fosse negativo, non ci conviene tenere traccia dei numeri più piccoli di quel range, dato che se quella sommatoria è minore del numero successivo alla sommatoria, non ha senso tenerne conto. E quindi invece ha senso tenere traccia del numero successivo. Da quel numero poi sommare i numeri successivi continuando il processo sopracitato.

2.2 Strutture Dati - Insieme Dinamico

Vediamo come rappresentare un insieme di dati dinamico S (con insieme dinamico si intende una collezione di elementi variabile nel tempo, quindi è possibile aggiungere o rimuovere elementi);

$$S = \{a_1, a_2, ..., a_n\} \quad n \geq 0$$

Andiamo a definire alcune operazioni:

- Insert$(S, a) \rightarrow S'$ $(S' = S \cup \{x\})$
- Deletes$(S, a) \rightarrow S'$ $(S' = S \backslash \{x\})$
- Search$(S, a) \rightarrow \{True, False\}$
- Massimo$(S) \rightarrow a$
- Minimo$(S) \rightarrow a$
- Successore$(S, a) \rightarrow a'$
- Predecessore$(S, a) \rightarrow a'$

3 Lezione 04 - 23/09/2023

3.1 Array non ordinato

Abbiamo un insieme S che vogliamo rappresentare in A.

$$|S| = |A|$$

La funzione $A.free$ ci restituirà la posizione della prima cella libera.

3.1.1 Ricerca

Dato un array e un elemento da cercare, restituisce la posizione in cui è presente il valore

```
Search(A,e) //A: array in input, e: elemento da cercare
  pos = A.free-1
  while A[pos] != e and pos >= 0 do
    pos = pos-1
  return pos
```

Ci posizioniamo all'ultima cella piena e mano a mano tornando all'indietro andiamo a cercare il valore che abbiamo in input, se non è presente nell'array andiamo a restistuire il valore -1. La complessità computazione sarà **lineare** nello specifico:

$$T_s(n) = c \cdot n$$

Dove c è il costo fisso delle operazione e n è quante volte si ripete il ciclo.

3.1.2 Inserimento

Dato un array e un elemento da inserire, andiamo a verificare tramite la funzione **ricerca** se l'elemento non sia già presente poiché non vogliamo elementi duplicati.

```
Inserimento(A,e) //A: array in input, e: elemento da cercare
  pos = search(A,e) //-1: non trovato, >=0: indice del valore
  if pos = -1 then
    if(A.free < length(A)) then
      A=resize(A)
    A[A.free]=e
    A.free=A.free+1
```

La complessità computazione sarà **lineare** nello specifico:

$$T_i(n) = 2c \cdot n + c'$$

3.1.3 Cancellazione

Dato un array e un elemento da cancellare, andiamo a verificare tramite la funzione **ricerca** se l'elemento sia presente in modo da portelo eliminare.

```
Delete(A,e) //A: array in input, e: elemento da cercare
  pos = search(A,e) //-1: non trovato, >=0: indice del valore
  if pos >= 0 then //Elemento trovato
    A[pos]=A[A.free-1]
    A.free=A.free-1
```

3.2 Array Ordinato

Uno dei grosso problemi dell'array visto in precendenza era la ricerca, poiché al più avevo costo **lineare** cioè la lunghezza di tutto l'array, con l'array ordinario andiamo a sopperire a questo problema ma ovviamente aggiungendone degli altri.

3.2.1 Ricerca Binaria

La ricerca binaria è un algoritmo applicabile solo ad array ordinato che permette una ricerca molto rapida.
L'algoritmo è simile al metodo usato per poter trovare una parola sul dizionario: sapendo che il vocabolario è ordinato alfabeticamente, l'idea è quella di iniziare la ricerca non dal primo elemento, ma da quello centrale, cioè a metà del dizionario. Si confronta questo elemento con quello cercato:

- se corrisponde, la ricerca termina indicando che l'elemento è stato trovato;
- se è superiore, la ricerca viene ripetuta sugli elementi precedenti (ovvero sulla prima metà del dizionario), scartando quelli successivi;
- se invece è inferiore, la ricerca viene ripetuta sugli elementi successivi (ovvero sulla seconda metà del dizionario), scartando quelli precedenti.

Andiamo a definirlo per ricorsione:

```
BinSearch(A, e, i, j) //i: punto inizio, j: punto fine
  if i <= j then
    q=(i+j)/2 //punto centrale
    if A[q] > e then
      i=BinSearch(A, e, i, q-1) //ricerca a "destra"
    else if A[q] < e then
      i=BinSearch(A, e, q+1, j) //ricerca a "sinistra"
    else return q //trovato
  else return -1 //non trovato
```

Per studiare la complessità di questo algoritmo (ricorsivo), andiamo a rappresentare le varie chiamate tramite un "albero degenere" in cui ogni nodo ha un solo figlio e così via, il costo di ogni nodo sarà c, ad ogni chiamatà andiamo a dimezzare la lunghezza dell'array $n ... n/2 ... n/4 ....$ il massimo delle chiamate possono essere $h+1$ dove h è l'altezza dell'albero andiamo a generallizare con:

$$\frac{n}{2^h} \quad h : \text{è il numero dei richiami}$$

Dobbiamo trovare il valore di h per il quale l'intervallo si riduce a 1 elemento, in quanto l'elemento desiderato sarà stato trovato. Quindi, dobbiamo risolvere l'equazione:

$$\frac{n}{2^h} = 1$$

Per risolvere questa equazione per h, possiamo moltiplicare entrambi i lati per 2^h:

$$n = 2^h$$

Ora, per isolare h, possiamo applicare il logaritmo in base 2 ad entrambi i lati:

$$h = log_2(n)$$

Quindi, il numero di chiamate ricorsive necessarie per trovare l'elemento desiderato è logaritmico rispetto alla dimensione dell'array n. Pertanto, la complessità computazionale dell'algoritmo di ricerca binaria è $O(logn)$ nel caso peggiore.
N.B Una complessita computazionale logaritimica è più efficente di una lineare.

3.2.2 Inserimento/Cancellazione nella ricerca binaria

Abbiamo visto come in un array ordinato la ricerca (binario) è molto efficemente ma abbiamo come contro che l'inserimento/cancellazione hanno costo peggiore poiché dobbiamo mantenere l'ordinamento, quindi se vogliamo inserire un elemento dobbiamo andare a spostare i valpri per creare un posto disponibile.

4 Lezione 05 - 26-09-2023

4.1 Ricerca Binaria Iterativa

Proviamo a riscrivere la ricerca binaria su un array ordinato in maniera iterativa:

```
BinSearchIterative(A,e) //A: array in input, e: elemento da cercare
  ret=-1
  i=0
  j=length(A)-1
  while i <=j AND ret=-1 do
    q=(i+j)/2
    if A[q] < k then
      i=q-1
    else if A[q] > k then
      j=q+1
    else
      ret = q
  return ret
```

Possiamo notare che a meno di piccoli cambiamenti il funzionamento di questo algoritmo rispetto alla sua versione ricorsiva è pressoche identico.

4.2 Somma

Andiamo a creare un algoritmo per sommare tutti i valori presenti in un array tramite ricorsione.
Come tutti gli algortmi ricorsivi dobbiamo andare a trovare una base di induzione (caso base) e poi un passo di induzione, rappresentiamolo tramite sommatorie:

$$\sum_{i=i}^{n} i = \begin{cases} 0 & \text{se } n = 0 \quad \text{base induzione} \\ n + \sum_{i=1}^{n-1} i & \text{se } n \geq 0 \quad \text{passo induttivo} \end{cases}$$

Ora che abbiamo capito il ragionamento andiamo a scrivere l'algoritmo

```
Sum(A,i,n) // i=inizio, n=fine
  if n=0 then
    ret = 0
  else
    x=sum(A,1,n-1)
    ret=n+x
  return ret
```

4.3 Lista Linkata

La lista rappresenta una struttura dati dinamica dove le operazioni di inserimento e cancellazione sono meno dispendiose, a differenza di quanto accade negli array (che sono implementati come struttura statica, il che rende problematiche le suddette operazioni).
La lista condivide con l'array la proprietà di linearità (o sequenzialità) ma è una struttura più flessibile poiché non richiede la contiguità in memoria come l'array. La lista permette di avere gli elementi in una qualsiasi area di memoria rendendo le operazioni di inserimento e cancellazione eseguibili in tempo costante; infatti, la sua struttura è composta da nodi, i quali contengono sia un certo dato (key), sia un'informazione su dove si trovi il nodo successivo (next).

4.3.1 Ricerca (interativo)

Andiamo a definire un algoritmo di ricerca su lista linkata tramite iterazione

```
Search(L,k) // L=lista, k=elemento da cercare
  ris=-1
  temp=L // "puntatore" al primo elemento della lista
  while temp != NIL and ris=-1 do
    if temp->key = k then
      ris=k
    else
      temp = temp->next
  return ris
```

La variabile "temp" ci serve per evitare di andare a perdere "l'inizio" della lista in ingresso dato che la lista in questo caso può solo andare avanti, quindi se non ci salviamo il punto di inizio non possiamo più tornare all'inizio

4.3.2 Ricerca (ricorsiva)

Andiamo a rifare lo stesso algoritmo ma tramite la ricorsione

```
Search(L,k) // L=lista, k=elemento da cercare
  ris=NIL
  if L != NIL then
    if L->Key = k then
      ris=L
    else
      search(L->Next, k)
  return ris
```

In questo caso ad ogni prossima chiamata di "search" andiamo a passare una "sottolista" cioè la stessa ma partendo da un nodo più avanti fino ad arrivare alla sua fine.
Andiamo a calcoalre la complessità computazione, dato che ad ogni chiamata andiamo a passare la lista ma ridotta di un nodo avremo $n, n-1, n-2, n-i$ dove i sarà il numero di nodi, quindi abbiamo la sequenza dei primi i numeri naturali (formula di Gauss) quindi l'algoritmo ha complessità **lineare**.

4.3.3 Ragionamento Ricerca Binaria su Lista Ordinata

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

5 Lezione 06 - 29/09/2023

Continuiamo i nostri algoritmi sulle liste linkate

5.1 Inserimento

Andiamo a scrivere un algoritmo per l'inserimento in testa, cioè andiamo ad "attaccare" il nostro nuovo nodo all'inizio della lista, è il tipo di inserimento più semplice perché non prevede particolari modifiche alla struttura.

Funzione Ausiliaria Useremo una funzione ausiliare per aiutarci in tutte le situazione di creazioni di un nuovo nodo:

```
newNode(K,L) //K: Elemento, L: Lista
  temp = allocanodo() //tipo una malloc
  temp->key = k //assegnamento valore
  temp->next = L //attacchiamo il nodo al primo elemento della lista in
      input
  return temp
```

Ora che abbiamo definito la nostra funzione ausiliaria andiamo a scrivere l'algoritmo di inserimento

```
Insert(L,K)
  ret = search(L,K) //cerchiamo il valore
  if ret=NIL then //se non e' presente lo inseriamo
    L=newNode(K,L)
  return L
```

5.2 Inserimento Ordinato (iterativo)

Per strutturare al meglio le liste possiamo prevedere un inserimento ordinato, fare questo nelle liste è molto più efficiente rispetto a un array poiché non dobbiamo spostare tutti i valori per fare spazio al valore da inserire ma possiamo banalmente staccare i puntatori e "riattaccarli" nel modo corretto:

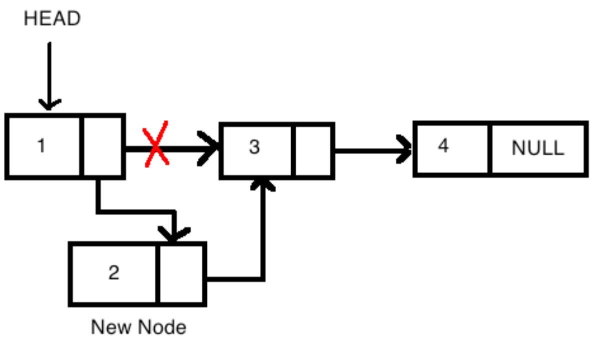

Fig. Insert after a given Node

Andiamo a definire l'algoritmo in maniera iterativa:

```
InsertO(L,K)
  temp=L //salviamo il puntatore al primo elemento
  P=NIL //creiamo una variabile di appoggio
  while [temp != NIL and temp->key < k] do
    P=temp //mettiamo P sul nodo "precedente" a dove inseriamo
    temp=temp->next //mettiamo temp sul nodo "successivo"

  if [temp = NIL or temp->key > k] then
    new=newNode(k,temp) //creiamo il nodo attaccandolo al "successivo"
    if P != NIL then //se "esiste" il precedente allora
      P->next = new //attacchiamo il "precedente" al nuovo
    else //se non esiste
      L=new //spostiamo la testa al nuovo valore

  return L
```

5.3 Inserimento Ordinato (ricorsivo)

Come sempre per scrivere un buon algoritmo ricorsivo bisogna ragionare per casi, andiamo ad esaminare i vari possibili:

- Inserimento in testa (valore minimo)
- Inserimento "centrale" (valore compreso tra due numeri)
- Inserimento in coda (valore massimo)

Sulla base di ciò andiamo a scrivere il nostro algoritmo:

```
InsertOR(L,K)
  if L=NIL then //Inserimento in coda
    L=newNode(K,L)
  else if [L->key > K] then //Inserimento in testa
    temp=L
    L=newNode(K, temp)
  else //Inserimento "centrale"
    L->next = InsertOR(L->next, K) //da esaminare
  return L
```

5.4 Cancellazione (iterativa)

Andiamo a scrivere un algoritmo di cancellazione simile a quello visto per l'inserimento, cioè andiamo a cercare il valore e ci posizioniamo sia "prima" che "dopo" il valore da cancellare.

```
Delete(L,K)
  temp=L //salviamo il puntatore al primo elemento
  P=NIL //creiamo una variabile di appoggio
  while [temp != NIL and temp->key != k] do
    P=temp //mettiamo P sul nodo "precedente" a quello da cancellare
    temp=temp->next //mettiamo temp sul nodo da cancellare

  if temp != NIL then
    if P != NIL then
      P->next=temp->next
    else
      temp=L
      L=L->next
    dealloca(temp)
  return L
```

5.5 Cancellazione (ricorsiva)

Ragioniamo per casi anche se in questo caso sono solo i due banali, cioè il valore è presente oppure no, nello specifico noi andiamo a considerare sempre la testa della lista che piano a piano a decrementarsi fino ad arrivare ad essere vuota

```
Delete(L,K)
  if L!=NIL then //La lista ha almeno un valore
    if L->key=k then //elemento in testa
      temp=L
      L=L->next
      dealloca(temp)
    else //elemento non in testa, "spostiamo" la testa in avanti
      L->next = delete(L->next, k)
  return L
```

6 Lezione 07 - 29/09/2023

6.1 Albero Binario

L'albero binario è una particolare struttura dati composto da nodi, ogni nodo ha tre campi:

- Valore
- Figlio Sinistro (puntatore)
- Figlio Destro (puntatore)

Ogni nodo può avere al più due figli, il primo nodo è chiamato **radice** e i nodi finali (senza figli) sono chiamati **foglie**.
Immagine
Andiamo a dare una definizione più corretta:

$$T \begin{cases} 1) T = NIL \\ 2) T = \text{ nodo } + \text{ sottoalbero} \end{cases}$$

6.2 Altezza di un Albero

Definiamo l'altezza di un albero come la **quantità** di nodi che scorreremo per raggiungere il nodo desiderato. Nota bene: Ad ogni livello (di altezza) il numero di nodi dell'albero aumenta esponenzialmente con un ritmo di

$$2^i$$

. Questo vale solo per un albero bilanciato. Osserveremo la sua complessità tra poco.

6.3 Complessità Computazione

Un albero binario può essere degenere, cioè un albero i cui nodi hanno solo un figlio. Questo tipo di albero, come si vede in foto, creano in un certo senso una lista dinamica. Infatti questo tipo di albero condividerà la stessa complessità computazionale di una lista ordinata.
Analizziamo l'equazione che definisce quanti nodi ha ogni livello di un albero.
$n = \sum_{i=0}^{n} 2^i$ Questa sommatoria è risolvibile con una serie.

$$n = \sum_{i=0}^{k} x^i = \frac{x^{k+1} - 1}{x - 1}$$

Diventerà nel nostro caso:

$$n = \sum_{i=0}^{n} 2^i = \frac{2^{n+1} - 1}{2 - 1} = 2^{n+1} - 1$$

Il risultato che abbiamo ottenuto è il valore della tabellina del 2 2^{n+1}, a meno di un numero per il -1.

Per andare dunque a calcolare la complessità computazione per la creazione di ogni nodo per ogni grado d'altezza dell'albero è dunque utile andare a prendere il risultato di questa funzione qui sopra scritta e risolverlo.

$$n = 2^{n+1} - 1 \rightarrow \log_2 n = \log_2 2^{n+1} - 1 \rightarrow\rightarrow$$

Sapendo che $n = 2^{n+1} - 1$ allora potremmo sostituirlo all'interno della seconda parte dell'equazione, facendo in modo che l'equazione diventi

$$\log_2 n = \log_2 n$$

Andiamo a ragionare su quando possa valere $2^{n+1} - 1$ andiamo a considerare in quale spazio esso può essere compreso: $2^h \leq 2^{n+1} - 1 \leq 2^{n+1}$

6.4 Visite

Esistono vari tipi di visita ognuna con i suoi pregi e difetti

6.4.1 PreOrder

L'algoritmo di visita pre-order è un particolare algoritmo usato per l'esplorazione in profondità dei nodi di un albero. L'esplorazione dell'albero parte dalla radice per poi scendere alle foglie, prima il sotto albero sinistro poi quello destro, che sono gli ultimi nodi ad essere visitati.

```
PreOrder(T):
  T != NIL then
    Visita(T->key) //funzione qualsiasi
    PreOrder(T->sx)
    Preorder(t->dx)
```

6.4.2 InOrder

L'algoritmo di visita in-order è un particolare algoritmo usato per l'esplorazione in profondità dei nodi di un albero binario. In questo tipo di visita, per ogni nodo, si esplora prima il sottoalbero sinistro poi si visita il nodo corrente ed infine si passa al sottoalbero destro. Più precisamente, l'algoritmo esplora i rami di ogni sottoalbero fino ad arrivare alla foglia più a sinistra dell'intera struttura, solo a questo punto si accede al nodo. Terminata la visita del nodo corrente si procede poi con l'esplorazione del sottoalbero a destra, visitando sempre i nodi a cavallo dell'esplorazione del sottoalbero sinistro e quello destro.

```
InOrder(T):
  T != NIL then
    InOrder(T->sx)
    Visita(T->key) //funzione qualsiasi
    InOrder(t->dx)
```

6.4.3 PostOrder

L'algoritmo esplora i rami dell'albero fino ad arrivare alle foglie prima di accedere ai singoli nodi, ad esempio si supponga di essere alla ricerca di alcuni oggetti molto pesanti e che per trovarli si debba esplorare un sentiero che comporta diverse diramazioni (albero).

Essendo gli oggetti pesanti non conviene raccoglierli subito e portarli con sé lungo il cammino ma conviene prima esplorare tutto il territorio quindi prelevarli quando si torna indietro.

L'algoritmo post-order visita un albero nello stesso modo, arrivando prima più in fondo possibile ad ogni diramazione ed accedendo agli elementi solamente al ritorno, il che corrisponde ad un'implementazione ricorsiva al fronte di risalita della ricorsione.

```
PostOrder(T):
 T != NIL then
   PostOrder(T->sx)
   PostOrder(t->dx)
   Visita(T->key) //funzione qualsiasi
```

6.5 Ricerca Albero Binario

Andiamo a definire la ricerca su l'albero binario, la miglior visita per eseguire una ricerca è la **preorder** poiché è la visita che subito controlla i nodi senza aspettare la visita dei sottoalberi.

```
Search(T,k):
 ret=T
 if T != NIL then
   if T->key != k then //se il valore non c'e' in testa
     ret=Search(T->sx, k) //allora cerchiamo nel sottoalbero sinistro
     if ret=NIL then //se non c'e' a sinistra
       ret=Search(T->dx,k) //cerchiamo a destra
 return ret
```