Algoritmi e strutture dati

Umberto Domenico Ciccia

November 20, 2024

Il documento è stato scritto con L'Intenzione di essere un ripasso su tutti gli argomenti studiati durante il corso. Non consiglio l'utilizzo come fonte di studio ma solo come fonte di ripasso. Ciò non toglie che gli argomenti sono approfonditi e ben spiegati.

Contents

Introduzione

Definizioni basilari

Un **problema** nella teoria degli algoritmi è una **funzione** che ad **associa** ad un **input** un **output**; Alla definizione di problema possiamo associare la definizione di **algoritmo risolutore**. Un algoritmo risolutore dato un problema P, è un algoritmo che **non termina** e che **dato** un **input**, **restituisce** per quel dato input sempre lo **stesso output**. Un **problema** si dice **risolvibile** se **esiste** un **algoritmo risolutore** per quel determinato problema.

Analisi della complessità per casi

Non esiste sempre una funzione che associa alla dimensione dell'input la sua complessità. Per questo analizziamo la complessità degli algoritmi in 3 diversi casi:

- Caso Migliore: E' il caso in cui l'algoritmo effettuerà il minor numero di operazioni, $T_A(x) = min \;\; T_A(x) | x \in I \land \;\; |x| = n$
- Caso Peggiore: E' il caso in cui l'algoritmo effettuerà il massimo numero di operazione,$T_A(x) = max \;\; T_A(x) | x \in I \land \;\; |x| = n$
- Caso Medio: Bisogna considerare la probabilità che le istanze considerate vengano date in input

Calcolo della complessità algoritmi iterativi

Se volessimo caratterizzate la complessità temporale degli algoritmi allora andremmo a considerare la quantità di tempo per eseguire un determinato algo-

ritmo, mentre caratterizzando la complessità spaziale confideremo lo spazio occupato dall'algoritmo per essere eseguito. Nel modello a costo uniforme consideriamo che ogni istruzione abbia costo unitario, Il costo temporale di esecuzione dell'algoritmo sarà dato dalla somma dei costi. Per facilitare l'analisi degli algoritmi iterativi possiamo utilizzare la definizione di istruzione dominante. Dato un algoritmo A con costo $T_a(n)$ un istruzione dominante è un istruzione che viene eseguita esattamente $\theta(n)$ volte.

Calcolo complessità algoritmi ricorsivi

Per calcolare la complessità temporale degli algoritmi ricorsivi dobbiamo prima avere chiaro che ad ogni algoritmo ricorsivo possiamo associare un'equazione di ricorrenza, ovvero un'equazione che ne rappresenta matematica le operazioni. Per il calcolo della complessità temporale possiamo analizzare la complessità dell'algoritmo o attraverso il metodo dello srotolamento, o attraverso il metodo di sostituzione.Per quanto riguarda la complessità spaziale è rappresentata dal numero di chiamate ricorsive contemporaneamente attive sullo stack.

Complessità intrinseca del problema

Lower bound upper bound

Partiamo col dare la definizione di upper bound e lower bound.

- Upper Bound: Con upper bound di un problema p indichiamo il limite superiore sulla complessità degli algoritmo per quel determinato problema. Per essere più rigorosi un problema P ha Upper Bound $O(f(N))$ se esiste un algoritmo risolutore che abbia complessità $O(f(N))$ nel caso peggiore.

- Lower Bound: Con lower bound di un problema p indichiamo il limite inferiore sulla complessità degli algoritmi per quel determinato problema. Per essere più rigorosi, un problema P ha come upper bound $\Omega(f(n))$se tutti gli algoritmi risolutori hanno complessità $\Omega(f(n))$ nel caso peggiore

Complessita intrinseca

La complessità intrinseca corrisponde al lower bound del problema. Per determinare la complessità intrinseca per gli algoritmi decisionali possiamo utilizzare il metodo dell'albero di decisione, altrimenti utilizzeremo la tecnica dell'avversario.

Algoritmo ottimale

Collegandoci a quanto appena detto un algoritmo si dice ottimale per quel problema se P ha complessità intrinseca $\Omega(f(n))$ e l'algoritmo ha complessità $O(f(n))$ nel caso peggiore.

Definizioni asintotiche

- O $f(n) \in O(g(n))$ $\forall n > n_0, \exists c > 0 | f(n) < cg(n)$
- θ entrambi
- Ω $f(n) \in \Omega(g(n)) \forall n > n_0 \exists c > 0 | f(n) > cg(n)$

Tecniche logaritmiche

Dividi et impera

La tecnica di dividi et impera è una tecnica algoritmica che consiste nel suddividere il problema originario in un certo numero di sotto istanze della stessa dimensione, risolvere le sotto istanze con lo stesso algoritmo, combinare i risultati per ottenere la soluzione finale del problema. Esistono due tipi di algoritmi divide et impera:

$$f(x) = \left\{ \begin{array}{cc} aT(\frac{n}{c}) + bn^d & n \geq 1 \\ b & n \leq 1 \end{array} \right\} \tag{1}$$

$$f(x) = \left\{ \begin{array}{cc} aT(n-k) + bn^d & n \geq 1 \\ b & n \leq 1 \end{array} \right\} \tag{2}$$

Greedy

La tecnica goloso è una tecnica algoritmica che si presta bene alla risoluzione di problemi di ottimizzazione, ovvero di problemi in cui vogliamo ottenere la migliore soluzione possibile. Un generico algoritmo che utilizza la tecnica golosa, utilizza queste strutture dati.

- Insieme di tutte i canditati a soluzione
- Funzione selezione: seleziona la migliore scelta
- Funzione ammissibile: verifica se la soluzione è ammissibile
- Funzione ottimo: verifica che la funzione ammissibile sia ottima
- Funzione obbiettivo: restituisce il valore della soluzione trovata

Algorithm 1: Generico algoritmo goloso

```
S=∅ ;
C=insiemeCanditati ;
while not Ottimo(S) and (s ≠ ∅) do
    x=seleziona(C);
    C=C-x;
    if ammissibile(S ∪ x) then
        S = S ∪ x ;
    end
end
if ottimo(S) then
    return S ;
end
```

Dinamic

La tecnica dinamica è una tecnica simile alla divide et impera. La divide et impera infatti lavora top to down la dinamica down to top. Attraverso questa tecnica utilizziamo una struttura dati aggiuntiva come una tabella o un'array nel quale memorizziamo i risultati intermedi delle operazioni. Utilizziamo i risultati intermedi come base per arrivare alla soluzione finale

Master theorem

Per algoritmi divide et impera di tipo 1 dove a rappresenta il numero di sotto istanze c la loro dimensione e d il costo della fase combina

- $\frac{a}{c^d} > 1$ implica $T(n) = \theta(n^{log_c a})$
- $\frac{a}{c^d} < 1$ implica $T(n) = \theta(n^d)$
- $\frac{a}{c^d} = 1$ implica $T(n) = \theta(n^d log_c n)$

Proof. Applicando lo sdrotolamento all'equazione di ricorrenza divide et impera tipo 1

$$T(n/c) = aT(n/c^2) + b(n/c)^d$$
$$T(n/c^2) = aT(n/c^3) + b(n/c^2)^d$$

$$T(n) = a^3T(\frac{n}{c^3}) + a^2b(\frac{n}{c^2})^d + ab(\frac{n}{c})^d + a^0b(\frac{n}{c^0})^d$$

Applicando lo srotolamento con $i < n$ (suppongo che il lettore sappia arrivare da questa equazione al caso base finale)

$$bn^d \sum_{i=0}^{k} (\frac{a}{c^d})^i$$

- $\frac{a}{c^d} = 1$ avremo $bn^d \sum_{i=0}^{k} (\frac{a}{c^d})^i = bn^d k = bn^d log_c(n)$ che è $\theta(n^d log_c(n))$
- $\frac{a}{c^d} < 1$ la serie $bn^d \sum_{i=0}^{k} (\frac{a}{c^d})^i$ converge a c per cui l'equazione di ricorrenza $bn^d c$ che è $\theta(n^d)$
- Vedi appunti ipad

□

Algoritmi di ordinamento

MergeSort

Il merge sort è un algoritmo di ordinamento temporalmente molto efficiente(ma non spazialmente). Si basa sulla tecnica divide et impera, l'idea è infatti quella di dividere ricorsivamente in due sotto istanze di dimensione n/2 il nostro problema, ordinare sotto problemi piu semplici, combinare i risultati tramite la merge per ottenere alla fine un array ordinato.

```
1  def mergeSort(v,inf,sup)->void:
2      if inf > sup:
3          return
4      med=(inf+sup)/2
5      mergeSort(v,inf,med)
6      mergeSort(v,med+1,sup)
7      merge(v,inf,med,sup)
```

```
8  def merge(v,inf,med,sup)->void:
9      x={}
10     i1=inf i2=med+1
11     aux=0
12     while i1<=med and i2<sup:
13         if v[i1]<=v[i2]:
14             x[aux]=v[i1]
15             aux+=1
16         else
17             x[aux]=v[i2]
18             aux+=1
19     for i1<=med:
20         x[aux]=v[i1]
21         aux+=1
```

```
22      for i2<=sup
23          x[aux]=v[i2]
24          aux+=1
25      copia x in v
```

La complessità della funzione merge è $\theta(n)$ per cui questo algoritmo divide et impera combina i risultati a questo costo. Per quanto riguarda il costo dell'algoritmo possiamo sfruttare il teorema master. L'algoritmo divide il problema in due sotto istanze di dimensione la meta. $\frac{a}{c^d} = \frac{2}{2} = 1$ quindi la complessità è $\theta(nlogn)$ mentre la spaziale $\theta(n)$

QuickSort

Il quicksort si basa su un idea simile a quella del merge sort. L'idea di questo algoritmo è quella di dividere il vettore logicamente in due parti, una prima di un valore perno e una dopo questo valore scelto perno. Dopodiché tramite una funzione partiziona spostiamo tutti gli elementi piu piccoli del perno a sinistra di esso e tutti i piu grandi a destra di esso, dopodiché effettuiamo ricorsivamente questa operazione sul sotto vettore a sinistra e destra del perno.

```
26  def quickSort(v,i,f):
27      if inf>=f:
28          return
29      pindxe=random(f-i)+i
30      pivot=v[pindex]
31      swap(v,pindex,f)
32      int p=partiziona(v,i,f,pivot)
33      quickSort(v,i,p-1)
34      quickSort(v,p+1,f)
35  def partiziona(v,i,f,pivot):
36      low=i high=f-1
37      while low<high:
38          while v[low]<=pivot and low<high
39              low++
40          while v[high]>pivot and low<high
41              high--
42          swap(v,low,high)
43      if v[low]>v[f]
```

```
44          swap(v,low,f)
45      else
46          low=f
47      return low
```

Nel caso peggiore (ovvero quando il perno è sempre il massimo o il minimo del vettore) l'algoritmo ha complessità $O(n^2)$. Nel caso medio avremmo la stessa complessità del MergeSort.

Dimostrazione della complessità

Il numero di confronti è dato dalla seguente equazione $C(n) = n-1+C(a)+C(b)$ dove a è il vettore con gli elementi minori del pero e b maggiori.

- Nel caso peggiore il perno sarà il minimo del vettore per cui a=0 $C(n) = n-1+C(n-1)$. Notiamo facilmente che questo è riconducibile alla sommatoria di gauss e quindi $C(n) = O(n^2)$
- Nel caso medio scegliamo il perno in maniera randomica in modo tale che ogni elemento abbia la stessa probabilità di essere scelto. $T(n) = \sum_{p=0}^{n-1} \frac{1}{2} + T(p) + T(n-p-1) = n-1 \sum_{p=0}^{n-1} \frac{2}{n} T(p)$ attraverso il metodo di sostituzioni e integrali che non ho idea come si risolvano arriviamo a concludere che $T(n) \leq 2nlogn$ quindi $T(n) = O(nlogn)$

HeapSort

L'heap sort sfrutta le proprietà dell'heap per ordinare un vettore. Non in loco:

```
48  def heapSort(v):
49      heap h
50      for x in v:
51          h.add(v)
52      for i in range(len(v)):
53          v[i]=heap.remove()
```

Complessità spaziale $\theta(N)$, temporale $\theta(NlogN)$.

CountingSort

Il CountingSort è un algoritmo di ordinamento non basato su confronti. Per ordinare il vettore ne istanzia uno di dimensione $max(v) - min(v) + 1$. Ogni cella conterrà il valore 1 se l'elemento è presente 0 altrimenti. Per convertire l'indice dell'array al corrispondente valore basta sommare alla posizione il minimo del vettore principale $elemento = i + min(v)$. Una volta inizializzato l'array di appoggio basterà scandirlo e sostituire gli elementi (che saranno ordinati) con valore 1 nel principale.

```
54 def countingSort(v):
55     l=nuovo vettore[(max(v)-minv(v))+1]
56     for i in v:
57         index=v[i]-min(v)
58         l[index]=1
59     pos=0
60     for j in l:
61         while a[j]>=0:
62             v[pos]=j+min(v)
63             pos++;
```

RadixSort

Il Radix sort è un algoritmo di ordinamento basato sull'utilizzo di un array di code. L'idea sta nell'inserire l'ultima cifra di ogni numero in un array di code contenente una coda per ogni cifra. Una volta effettuato questo procedimento scandiamo l'array di code e togliamo ogni volta l'elemento che ha maggiore priorità sostituendolo nel vettore principale. Otterremo cosi l'ordinamento per quella cifra. Effettuiamo il procedimento per n cifre e otteniamo cosi l'array ordinato.

Alberi

Alberi binari

Definizioni

Un albero binario è una struttura dati, gerarchica in cui esiste una relazione gerarchica fra gli elementi, ogni nodo ha n figli. Un albero è composto da un certo numero di nodi e archi. Essendo binario avra grado di ogno nodo due.

- Radice: Nodo senza padre
- Grado di un nodo: Numero di figli di un nodo
- Foglia: Nodo senza figli
- Livello di un nodo: Numero archi attraversati per raggiungere un nodo
- Altezza: livello foglia più bassa

Rappresentazione alberi binari

- Puntatore figli: Ogni oggetto è rappresentato come un nodo contenente il suo valore e un puntatore per ogni figlio
- Lista figli: Ogni oggetto è rappresentato come un nodo contenente il suo valore e un puntatore ad una lista dei suoi figli
- Lista figli-fratello: Ogni oggetto è rappresentato come un nodo contenente il suo valore e un puntatore al primo figlio e al suo fratello

Algoritmi visita

Visita in profondità: PREORDER

```
64 def visitaPreorder(a):
65     if(!a is None)
66         print(a.val)
67         visita(a.sx)
68         visita(a.dx)
```

Visita in profondità: INORDER

```
69 def visitaInorder(a):
70     if(!a is None)
71         visita(a.sx)
72         print(a.val)
73         visita(a.dx)
```

Visita in profondità: POSTORDER

```
74 def visitaPostorder(a):
75     if(!a is None)
76         visita(a.sx)
77         visita(a.dx)
78         print(a.val)
```

Complessità spazio temporale visita

Il costo di ogni algoritmo visita appena citato è $T(N) = O(N)$ mentre per quanto riguarda la complessità spaziale $S(H) = O(H)$

Algoritmo visita: AMPIEZZA

```
79 def visitaAmpiezza(a):
80     Coda c
81     c.push(a)
82     while c.size!=0:
83         n=a;
84         if n!=null:
85             print(n)
86             c.push(a.sx)
87             c.push(a.dx)
```

Calcolo complessità spazio temporale visita

Il costo di ogni algoritmo visita appena citato è $T(N) = O(N)$ mentre per quanto riguarda la complessità spaziale $S(H) = O(H)$.

Alberi binari di ricerca

Un albero binario si dice di ricerca se per ogni nodo vale la seguente proprietà: Il figlio sinistro del nodo ha valore più piccolo della radice mentre il figlio destro un valore più grande o viceversa.

```
88 def alberoRicerca(a):
89     max=sys.maxInt()
90     min=sys.minInt()
91     return verifica(a,min,max)
92 def verifica(a,min,max)
93     if a is none:
94         return true
95     if not (min<a.val<max):
96         return false
97     return verifica(a.fs,min-1,a.val) and verifica(a.fd,a.
          val+1,max)
```

Operazioni sugli alberi binari di ricerca

Ricerca elemento

```
98  def search(a,x):
99      if(a is none)
100         return false
101     if(a.val==x)
102         return true
103     if(a.val>x)
104         return search(a.fs,x)
105     return search(a.fd,x)
```

Complessità spaziale $S(H) = O(H)$ complessità temporale $T(H) = O(H)$

Inserimento su un albero binario di ricerca

Per inserire un nodo in un albero binario di ricerca bisogna trovare prima di tutto la posizione perché bisogna mantenere la proprietà di ordinamento. Una volta trovata la posizione si inserisce il nodo come foglia.

```
106 def insert(x):
107     a=insert(x,a)
108 def insert(x,a)
109     if a is none
110         return new nodo(x)
111     if a.val>x
112         a.sx=insert(x,a.sx)
113     else
114         a.dx=insert(x,a.dx)
115     return a
```

Rimozione su un albero binario di ricerca

Per rimuovere il nodo la prima cosa da fare è trovarlo, una volta trovato è possibile rimuoverlo solo se è una foglia. altrimenti se è un nodo con un figlio basta cambiare il puntatore del padre, altrimenti bisogna eliminare il nodo predecessore e sostituire il suo valore nel nodo precedentemente trovato

```
116 def remove(x):
117     a=remove(x,a)
118 def remove(x,a)
119     #caso 1
120     if a is none
121         return null
122     if a.val>x
123         a.sx=remove(x,a.sx)
124     else if a.val<x
125         a.dx=remove(x,a.dx)
126       #in questo momento abbia trovato il nodo
127       #caso 2
128       if a.sx is none:
129         return a.dx
130       if a.dx is none:
131         return a.sx
132       #caso 3
133       successore=successor(a.dx)
134       a.val=successore.val
135       a.dx=remove(a.dx,successore.val)
136     return a
137 def successor(a):
138     current = a
139     while current.left:
140         current = current.left
141     return current
```

Albero AVL

Un albero AVL è un albero che si mantiene sempre bilanciato, infatti per ogni nodo può essere definito un valore detto coefficiente di bilanciamento dato dalla differenza tra l'altezza del sotto albero sinistro e il destro in valore

assoluto. Se questo valore è minore di uno per ogni nodo l'albero è bilanciato. Essendo l'albero bilanciato l'altezza sarà sempre proporzionale al logaritmo del numero di nodi.

Bilanciamento

Il bilanciamento viene garantito attraverso le rotazione a sx e a dx di base che hanno costo costante essendo un banale cambio di puntatore. Il bilanciamento viene effettuato sui nodi sbilanciati. Possiamo avere diverse posizioni dell'albero che sbilancia il nodo

- SS: T sotto albero sinistro del figlio sinistro, il bilanciamento viene effettuato ruotando a destra il nodo v
- DD: T sotto albero destro del figlio destro, il bilanciamento viene effettuato ruotando a sinistra il nodo v
- SD: T sotto albero destro del figlio sinistro, il bilanciamento viene effettuato ruotando a sinistra il figlio sinistro e a destra v
- DS: T sotto albero sinistro del figlio destro, il bilanciamento viene effettuato ruotando a destra il figlio destro e a sinistra v

Operazioni

inserito o rimosso il nodo, ricalcolati i fattori bilanciamento, bilanciamento nodi se necessario

Tabelle Hash

Supponiamo di avere un insieme di chiavi ed ad ogni chiave associamo un valore. Possiamo rappresentare questo tipo di elementi attraverso le tabelle hash. Grazie alle tabelle hash effettueremo le operazioni in modo molto efficiente sfruttando le proprietà di accesso costante alle celle degli array.

Tabella ad accesso diretto

Supponiamo di avere come chiavi un insieme di interi distinti. Possiamo associare ad ogni indice dell'array una chiave. A tale chiave sarà associato un valore se il contenuto della cella per quella determinata chiave sia diverso da null. E' chiaro che le operazioni di inserimento rimozione e ricerca saranno supportate in tempo costante.

Algorithm 2: Tabella Hash

Dati: Array m chiavi, l'elemento chiave i è presente se v[i]!=null
S(m)=O(m) ;
Operzioni: ;
insert (chiave k,elemento e);
v[k]=e T(N)=O(1);
remove (chiave k);
v[k]=null T(N)=O(1);
search(chiave k);
return v[k] T(N)=O(1);

Fattore Carico

Per misurare il riempimento delle tabelle hash utilizzeremo un parametro chiamato fattore di carico dato dal rapporto tra il numero di elementi contenuti nella tabella e il numero di chiavi che può contenere $\alpha = \frac{n}{m}$

Tabelle Hash

Se volessimo rappresentare anche delle chiavi non intere all'interno della nostra tabella potremmo utilizzare questo approccio. Attraverso una funzione hash associamo l'insieme delle chiavi ad un itero che rappresenterà la posizione della chiave nell'array.

Algorithm 3: Tabella Accesso Diretto

Dati: Array m chiavi, l'elemento chiave k è presente se v[h(k)]!=null
S(m)=O(m) ;
Operzioni: ;
insert (chiave k,elemento e);
v[h(k)]=e T(N)=O(1);
remove (chiave k);
v[h(k)]=null T(N)=O(1);
search(chiave k);
return v[h(k)] T(N)=O(1);

Definizione della funzione hash

Per essere una funzione hash buona, deve valere della proprietà dell' uniformità semplice. Una funzione hash gode di questa proprietà se tutte le celle hanno la stessa probabilità di essere associate ad una determinata chiave

Funzione Hash perfetta

Una funzione hash perfetta è una funzione iniettiva ovvero una funzione tale per cui ogni chiave riceverà un indice diverso.

Collisioni

E' inevitabile che due stesse chiavi possano ottenere lo stesso indice. Questo fenomeno si chiama collisione. Per gestire le collisioni possiamo utilizzare due apporci, lista di collisione o funzione di indirizzamento.

Liste di collisione

Utilizzando questa strategia risolutiva, gli oggetti sono rappresentati in una lista esterna puntata dalla posizione. Se due chiavi avranno lo stesso hash code, saranno contenuti nella stessa lista di collisione. La dimensione di una lista di collisione dipenderà dal fattore di carico della funzione hash. Il costo delle operazione in questa struttura dati sarà diverso.

Algorithm 4: Tabella Hash Lista collisione

Dati: Array m chiavi, l'elemento chiave k è presente se v[h(k)] è presente nella lista di collisione S(m,n)=O(m+n) ;
Operzioni: ;
insert (chiave k,elemento e);
aggiungi e alla lista v[h(k)] T(N)=O(1);
remove (chiave k);
rimuovi k dalla lista v[h(k)] T(N)=O(1+n/m);
search(chiave k);
cerca e restituisci l'elemento corrispondente alla chiave h(k) T(N)=O(1+n/m);

Indirizzamento aperto

A differenza della strategia a lista di collisione, la strategia a indirizzamento aperto mantiene tutte le chiavi nella stesso array. Se la posizione naturale è già occupata attraverso una funzione di indirizzamento occuperemo un'altra posizione. La ricerca della nuova posizione avviene tramite una scansione

- Scansione lineare: $c(k,i) = (h(k) + i) mod(m)$
- Scansione quadratica: $c(k,i) = [h(k) + c_1 \cdot i + c_2 \cdot i^2] mod(m)$
- Hashing doppio: $c(k,i) = [(h_1(k) + i \cdot h_2(k)] mod(m)$

La strategia a scansione lineare soffre del problema dell'agglomerazione primaria. Chiavi uguali finiranno in agglomerati di posizioni vicine, la scansione quadratica invece del problema dell'agglomerazione secondaria, chiavi uguali finiranno nella stessa posizione. La complessità delle operazione dipenderà

Algorithm 5: Tabella Hash Indirizzamento

```
Dati: Array m chiavi, ogni elemento rappresenta una coppia chiave
  valore S(m)=O(m) ;
Operzioni: ;
insert (chiave k,elemento e);
for(i to m) ;
if(v[c(k,i)]==null ;
v[c(k,i)]=e ;
remove (chiave k);
for(i to m) ;
if(v[c(k,i)]==k ;
v[c(k,i)]=null ;
search(chiave k);
for(i to m) ;
if(v[c(k,i)]==k ;
return v[c(k,i)] ;
```

dal tipo di scansionamento, (nel caso peggiore tutte O(n)), nel caso medio invece

- Scansione lineare: $O(\frac{m}{(m-n)^2})$
- Hashing doppio/quadratico: $O(\frac{m}{(m-n)})$

Code Priorità

Heap

L'heap è una particolare struttura dati che mantiene sempre il massimo o il minimo elemento alla radice. Questo perché l'heap è rappresentabile logicamente come un albero dove ogni nodo ha un valore di priorità maggiore/minore dei suoi figli. L'heap è un albero sempre completo tranne fino all'ultimo livello.

Theorem 1 (Altezza heap)**.** *l'altezza dell'heap è sempre proporzionale al* $log(n)$

Proof. Sappiamo che l'heap fino all'ultimo livello è completo. il numero di nodi al livello h-1 sarà $n = 2^{h-1}$ mentre il numero di nodi al livello h $n <= 2^h$, quindi $2^{h-1} <= n <= 2h$, passando al logaritmo naturale dimostriamo facilmente la proprietà □

L'heap può essere rappresentato tramite un array nel quale tralasciamo la prima posizione. A questo punto per individuare la posizione di un nodo, il figlio sinistro si troverà in posizione $2 \cdot i$ mentre il figlio destro in posizione $2 \cdot i + 1$, il padre invece $floor(\frac{i}{2})$

Operazione Inserimento

L'inserimento di un elemento in un heap avvera sempre come ultima foglia, si andrà poi a ripristinare l'ordinamento dell'heap salendo verso l'alto, scambiando i nodi se necessario. Nel caso peggiore la complessità sarà T(n)=O($log_2(n)$) nel caso migliore costante

Operazione rimozione

La rimozione invece avvera sempre dalla radice perché vogliamo restituito l'elemento a priorità maggiore/minore. La rimozione logica pero avviene in maniera differente. Si salva in una variabile il valore da restituire, eliminiamo la foglia più a destra sostituendone il valore alla radice, scendiamo ripristinando la proprietà di priorità dell'heap.Nel caso peggiore la complessità sarà T(n)=O($log_2(n)$) nel caso migliore costante

Array code

In questa particolare struttura dati abbiamo come insieme di priorità un insieme finito di priorità intere. Possiamo quindi rappresentarne gli elementi con quella priorità attraverso un array in cui ogni cella punta ad una coda. Se gli elementi hanno la stessa priorità verranno estratti in ordine fifo.

Insiemi

Gli insiemi sono una particolare struttura dati in cui gli elementi sono tutti disgiunti e le operazioni supportate sono quelle insiemistiche.

	Array	Array Ordinato	Lista	Lista Ordinata
Unione	$\theta(nm)$	$\theta(n+m)$	$\theta(nm)$	$\theta(n+m)$
Intersezione	$\theta(nm)$	$\theta(n+m)$	$\theta(nm)$	$\theta(n+m)$
Differenza	$\theta(nm)$	$\theta(n+m)$	$\theta(nm)$	$\theta(n+m)$

Nelle tabelle hash invece sarà sempre $\theta(n+m)$

Union-Find

Il problema dell'union-find consiste nel mantenere una collezioni di insiemi disgiunti, le operazioni consentite su questa collezione sono l'operazione union makeset e find. Ogni insieme sarà caratterizzato da un nome e dai suoi elementi.

1. Makeset(x): crea un nuovo insieme con di nome x con elemento x
2. Union(A,B): unisce A,B in un unico insieme di nome nome di A
3. Find(x): restituisce il nome dell'insieme che contiene x

Le due implementazione più comuni sono la rappresentazione quick-find che velocizza le operazioni di ricerca e quick-union che velocizza le operazioni di unione

Quick-Find

Nell'implementazione quick-find rappresentiamo gli insiemi come alberi di altezza massimo due. La radice rappresenta il nome degli insiemi mentre i figli gli elementi. L'operazione makeset(x) crea un albero di altezza due con radice x e figlio x. L'operazione union(A,B) sostituisce a tutti gli elementi di B il puntatore al nome dell'insieme, il nuovo nome sarà il nome di A.

- makeset(x): T(n)=O(1)
- union(A,B): T(n)=O(n) spostare n puntatori
- find(x): T(n)=O(1)

Quick-Union

Nell'implementazione quick-union rappresentiamo gli insiemi come alberi di altezza anche maggiore di due. In questo caso la radice non conterrà solo il nome dell'insieme ma anche il suo valore. L'operazione quick-union può essere implementata in modo costante facendo solamente puntare un nuovo figlio alla radice del secondo albero. Diverse saranno le operazioni di ricerca dovendo scorrere in altezza nel caso peggiore tutto l'albero.

- makeset(x): T(n)=O(1)
- union(A,B): T(n)=O(1)
- find(x): T(n)=O(n)

Bilanciamento operazioni tramite union

Quick-Union

Per migliorare la complessità dell'operazione find dobbiamo evitare che l'altezza dell'albero cresca troppo. Per fare ciò è logico pensare che conviene effettuare l'operazione union in modo tale per cui, l'albero di dimensione minore diventi figlio dell'albero di dimensione maggiore. Dimostriamo ora che l'altezza dell'albero è limitata superiormente da $log_2(n)$ dove n è il numero di makeset effettuabili. Per fare ciò ci aiutiamo dimostrando prima un lemma intermedio.

Lemma 2 (Operazione find $O(log_2(n))$)**.** *Il numero di nodi dell'albero è sempre* $n \geq 2^{rank(a)}$

Proof. Come detto in precedenza l'operazione union(A,B) attraverso questa variante viene bilanciata per non far crescere troppo l'albero.

1. $size(A) > size(b)$: la size di $|A \cup B| = |A| + |B|$ e il rank(A $\cup$ B) = rank(A). Per cui $|A \cup B| = |A| + |B| \geq 2^{rank(a)} + 2^{rank(b)} \geq 2^{rank(a)} = 2^{rank(a \cup b)}$

2. $size(A) < size(b)$: la size di $|A \cup B| = |A| + |B|$ e il rank(A $\cup$ B) = rank(B). Per cui $|A \cup B| = |A| + |B| \geq 2^{rank(a)} + 2^{rank(b)} \geq 2^{rank(b)} = 2^{rank(a \cup b)}$

3. $size(A) = size(b)$: la size di $|A \cup B| = |A| + |B|$ e il rank(A ∪ B) = rank(a). Per cui $|A \cup B| = |A| + |B| \geq 2^{rank(a)} + 2^{rank(b)} \geq 2 \cdot 2^{rank(a)} = 2^{rank(a \cup b)}$

□

Theorem 3. *Operazione find $O(log_2(n))$ L'altezza dell'albero è limitata superiormente da $log_2(n)$ dove n è il numero di makeset effettuabili*

Proof. Dal lemma sappiamo che il numero di nodi e quindi la size è sempre $size(a) \geq 2^{rank(a)}$ ma la dimensione è limitata dal numero di makeset $n \leq size(a) \leq 2^{rank(a)}$ □

Capiamo quindi che essendo l'altezza limitata superiormente dal logaritmo del numero di makeset la complessità temporale nel caso peggiore dell'operazione quick-union in questo caso avrà costo T(n)=O($log_2(n)$)

Quick-Find

Per migliorare la complessità temporale dell'operazione union dobbiamo logicamente unire l'albero di dimensione minore all'albero di dimensione maggiore per evitare di cambiare piu puntatori. Possiamo dimostrare che tramite questa piccola variazione riusciremo ad effettuare operazione union in tempo $T_{amm}(n) = O(log_2(n))$

Lemma 4. *A seguito di un operazione union su un albero quick-find la foglia dell'albero si troveranno sempre in un insieme di cardinalità almeno doppia*

Proof. Supponiamo che $size(A) \geq size(B)$, prima dell'operazione union $p_b \in B||B| = size(B)$. La nuova size post union Sara $size(A)+size(A) \geq size(A)$ con la foglia $p_b \in A||A| = size(B) + size(A)$. □

Theorem 5. $T_{amm} = \frac{T_{tot}}{noperazioni} = \frac{nlogn}{n} = O(logn)$

Proof. Dimostriamo che il tempo totale per eseguire n union è $O(logn)$. Assegniamo ad una operazione makeset ($logn$) crediti e togliamone k ogni spostamento di foglia nell'altro insieme. Dopo k spostamenti sappiamo grazie al lemma che le foglie si troveranno in un insieme di cardinalità 2^k e avremmo a disposizione ($logn - k$) crediti. Il numero massimo di union è limitato dal massimo numero di makeset n. La cardinalità dell'insieme finale in cui si troverà la foglia sarà $2^k < n$. Passando al logaritmo dimostriamo che il costo di k spostamenti quindi è $k \leq logn$ □

Stringhe

Edit Distance

Il problema dell'edit distance consiste nel trovare il minimo costo per trasformare una stringa X in una stringa Y attraverso 3 operazioni di base, col minimo costo:

1. inserimento(a): Inserisci il carattere stringa
2. rimozione(a): Rimuovi carattere dalla stringa
3. sostituzione(a,b): Sostituisci il carattere a con il carattere b

L'algoritmo risolutivo si basa sulla tecnica dinamica. Utilizziamo infatti una tabella di dimensione N per M dove n è la dimensione della prima stringa ed m della seconda. La posizione P(i,j) rappresenta la minima distanza per trasformare la stringa $x_i$in y_j. La tabella sarà inizializzata inizialmente nella prima riga e nella prima colonna con valori che vanno da 0 a 1 perché rappresentano il costo per ottenere se stessi.

$$P(i,j) = \left\{ \begin{array}{l} 1 + min[P(i-1,j), P(i,j-1), P(i-1,j-1)], \quad P[i]! = P[j] \\ P[i-1,j-1] \end{array} \right\} \tag{3}$$

Il costo dell'algoritmo temporale e spaziale sarà $\theta(nxm)$ Ovviamente scegliamo $1+min[P(i-1,j), P(i,j-1), P(i-1,j-1)]$se i caratteri sono diversi perché vogliamo l'operazione che ci porti al costo minore.

Massima sotto sequenza comune

Il problema della massima sotto comune consiste nel trovare la stringa X di lunghezza maggiore ottenibile rimuovendo alcuni caratteri da una stringa

Y. Anche questo problema è risolvibile attraverso la tecnica dinamica considerando pero questa volta la tabella verrà esaminata dalla fine verso l'inizio ottenendo il risultato in posizione (0,0). L'ultima riga e colonna verranno inizializzati con valore 0. Per capire l'equazione di avanzamento lungo la tabella ci viene in aiuto il teorema sulla massima sotto sequenza comune.

Theorem 6 (Massima sotto sequenza comune)**.** *Date due stringhe X, Y e W la loro massima sotto sequenza piu lunga(le tre stringhe hanno cardinalità rispettivamente n,m,k).*

1. $X[i] == Y[i]$ *allora* $W[i] = X[i] = Y[i]$ *e* $W[i+1...k]$ *è anche essa massima sotto sequenza a* $X[i+1...m]Y[i+1...n]$
2. $X[i] \neq Y[i]$ *se* $W[i] == X[i]$ *allora W è la massima sotto sequenza comune a* $X[i+1...n]$ *e* $Y[i...m]$
3. $X[i] \neq Y[i]$ *se* $W[i] == Y[i]$ *allora W è la massima sotto sequenza comune a* $X[i...n]$ *e* $Y[i+1...m]$

$$D(i,j) = \left\{ \begin{array}{l} max[P(i+1,j), P(i,j+1)], \quad D[i]! = D[j] \\ 1 + D[i+1, j+1] \end{array} \right\} \tag{4}$$

Il costo dell'algoritmo temporale e spaziale sarà $\theta(nxm)$

String Matching

Il problema dello String Matching consiste nel verifica l'occorrenza di un certo pattern P all'interno di un testo T.

```
142 def patternMatching(P,T):
143     m=len(T)
144     n=len(P)
145     for(i to m-n+1):
146         j=0
147         while p[j]==T[i+j] and j<m:
148             j++;
149         if j==m
150             occorrenza trovata pos i
```

Complessità $O(mxn)$

Grafi

Un grafo è una struttura dati che rappresenta relazioni gerarchiche tra i suoi elementi. Un grafo può essere definito attraverso i suoi due insiemi principali: Insieme dei nodi(vertici) e l'insieme degli archi. I grafi possono essere orientati o non orientati.

Definizioni

- Adiacenza: Due nodi x,y sono adiacenti se esiste un arco (x,y)
- Incidenza: Un arco (x,y) incide sul nodo x sul nodo y
- Grado vertice: Numero di archi entranti e uscenti dal nodo
  - Entrata: Numero di archi entrano nel nodo
  - Uscita: Numero di archi escono dal nodo

  Importante proprietà dei gradi dei nodi è la seguente:

$$\sum_{x \in v} \delta(x) = 2m$$

$$\sum_{x \in v} \delta_{in}(x) = \sum_{x \in v} \delta_{out}(x) = m$$

- Cammino: Sequenza di nodi tale per cui per ogni coppia di nodi esiste un arco che li collega
- Ciclo: Cammino parte e arriva stesso nodo
- Grafo connesso. Esiste un cammino per ogni coppia di nodi

- Grafo fortemente connesso: Esiste un cammino orientato per ogni coppia di nodi

Rappresentazione

Liste

- Lista archi: Un grafo viene rappresentato attraverso una lista che contiene tutti i suoi archi.
- Lista adiacenza: Un grafo viene rappresentato attraverso una lista di nodi, ogni nodo punta ad una lista contenente i nodi a loro adiacenti.
- Lista incidenza: Un grafo viene rappresentato attraverso una lista di nodi, ogni nodo punta ad una lista contenente un indice che rappresenta la posizione dell'arco che incide su questo nodo. Questo arco è rappresentato in un'altra lista.

Operazione	Lista archi	Lista adiacenza	Lista incidenza
Grado(v)	O(m)	O($\delta(v)$	O($\delta(v)$
ArchiIncidenti(v)	O(m)	O($\delta(v)$	O($\delta(v)$
sonoAdiacenti(v,u)	O(m)	O(min$\{\delta(v), \delta(u)\}$)	O(min$\{\delta(v), \delta(u)\}$)
add(v)add(v,u)	O(1)	O(1)	O(1)
remove(v)	O(m)	O(m)	O(m)
remove((v,u))	o(m)	O($\delta(v) + \delta(u)\}$)	O($\delta(v) + \delta(u)\}$)

Matrici

- Matrice incidenza: Attraverso questa rappresentazione manteniamo una tabella n x m in qui poniamo 1 il valore se l'arco incide su quel determinato nodo.
- Matrice adiacenza: Attraverso questa rappresentazione dei grafi manteniamo una tabella nodi per nodi in cui poniamo il valore ad 1 se i due nodi sono adiacenti.

Operazione	Matrice incidenza	Matrice adiacenza
Grado(v)	O(m)	O(n)
ArchiIncidenti(v)	O(m)	O(n)
sonoAdiacenti(v,u)	O(m)	O(1)
add(v)add(v,u)	O(nm)	O(n^2) O(1)
remove(v)	O(nm)	O(n^2)
remove((v,u))	o(n)	O(1)

Visite

```
151 def visitaAmpiezza(Grafo g, Nodo p):
152     visitati=false per ogni nodo in g
153     Coda c
154     c.enque(p)
155     while c is not empty:
156         u=c.deque()
157         visita(u)
158         visitati[u]=true
159         for(arco (u,v) in G)
160             if not visitati[v]
161                 c.enque(v)
```

La complessità spaziale dipende dal numero di nodi contenuti nella coda. Per quanto riguarda la complessità temporale dipenderà da quali strutture dati abbiamo utilizzato per rappresentare il grafo.

- Lista di archi: Con questa rappresentazione il ciclo for a riga 145 impiegherà per ogni nodo visiterà i suoi adiacenti. Questo costa come sappiamo O(m) per cui T(n,m)=O(nm)
- Lista adiacenza/incidenza: In questo caso sappiamo che la visita degli incidenti / adiacenti è proporzionale al grado del nodo considerato, nel caso peggiore quindi scorreremo tutta la lista che avra dimensione del grado. Sappiamo che la somma di tutti i gradi vale $\sum_{x \in v} \delta(x) = 2m$ per cui $T(n,m) = (n+m) = O(n + n\sum_{x \in v} \delta(x)) = O(n+m)$
- Matrici incidenza/adiacenza: $O(n+n^2) = O(n^2)$

Minimo albero ricoprente

Definizione minimo albero ricoprente

Un albero ricoprente T è un sottoinsieme del grafo g tale che: T è un albero, tutti i nodi di G sono presenti in T. Il costo dell'albero ricoprente è definito dalla sommatoria dei pesi di tutti gli archi che lo costituiscono. Il minimo albero ricoprente è l'albero ricoprente di costo minimo. Se gli archi hanno tutti costo diverso è unico altrimenti possono esserci piu minimi alberi ricoprenti.

Proprietà minimi alberi ricoprenti

Gli algoritmi per il calcolo del minimo albero ricoprente si basano sulla tecnica golosa. Prima di enunciare gli algoritmi dobbiamo dare prima delle definizioni preliminari.

Definition 1 (Taglio)**.** *Un taglio è una suddivisione dell'insieme dei nodi in due. Un arco (u,v) è un arco del taglio se u appartiene al primo insieme e v al secondo*

Diamo ora la definizione di arco blu e arco rosso

Definition 2 (Arco blu)**.** *Un arco blu è un arco incluso nella soluzione del metodo goloso*

Definition 3 (Arco rosso)**.** *Un arco rosso è un arco escluso dalla soluzione del metodo goloso*

Il metodo goloso si occuperà di colorare gli archi o di blu o di rosso usando o la regola del taglio la regola del ciclo. In questo modo costruiremo attraverso il metodo goloso il minimo albero ricoprente.

Definition 4 (Regola del taglio)**.** *Scegliamo un taglio del grafo che non contiene archi blu e coloriamo di blu l'arco di costo minimo*

Definition 5 (Regola del ciclo)**.** *Scegliamo un ciclo del grafo che non contiene archi rossi e coloriamo di rosso l'arco di costo massimo*

Correttezza metodo goloso

Abbiamo intuito che per calcolare il minimo albero ricoprente utilizzeremo la tecnica golosa che selezionerà gli archi in base alla tecnica precedentemente descritta. Dobbiamo pero dimostra che la tecnica golosa effettivamente calcola il minimo albero ricoprente. Per fare ciò dimostreremo che la tecnica golosa mantiene sempre la seguente invariante.

Definition 6 (Proprietà invariante)**.** *Il metodo goloso mantiene sempre ad ogni passo un albero che contiene solo archi blu; termina quando tutti gli archi sono stati colorati.*

Proof. Dimostreremo per induzione

- Il passo base è verificato, l'albero vuoto infatti contiene nessun arco rosso
- Passo Generico Regola taglio: Consideriamo un taglio (X,X') ed un arco e=(u,v) che è stato colorato di blu. Se l'arco e appartiene al taglio allora l'invariante è verifica logicamente perché l'albero blu deve contenere solo gli archi blu. Se l'arco e non appartiene al taglio allora esisterà un altro arco e' utilizzato per colorare e. Questo arco e' logicamente ha costo maggiore di e altrimenti la regola del avrebbe colorato questo arco. Considerando l'albero $T^{'} = (T - e^{'}) \cup e$ verificherà l'invariante perché conterrà solo archi blu
- Passo Generico Regola ciclo: Consideriamo un ciclo del grafo ed un arco e=(u,v) colorato di rosso. Se l'arco non appartiene a T allora l'invariante è verificata perché l'albero non conterrà archi rossi ma solo blu. Se l'arco invece appartiene possiamo notare che eliminando l'arco e dal grafo divideremo il grafo in due parti, esisterà quindi un arco e' che apparteneva al ciclo non colorato (altrimenti sarebbe stato colorato lui) utilizzato per colorare e. Considerando l'albero $T^{'} = (T - e) \cup e^{'}$ verificherà l'invariante perché non conterrà archi rossi ma solo blu

- Il metodo goloso termina dopo aver colorato tutti i nodi. Per dimostrare ciò possiamo supporre per assurdo che il metodo goloso è terminato ma alcuni ancora non sono colorati. Se non sono colorati pero possiamo applicare la regola del ciclo se l'arco incide nella stessa foresta di alberi blu, la regola del taglio se l'arco incide in due foreste di alberi blu diversi. Quindi essendo applicabile il metodo goloso l'ipotesi è un assurdo.

□

Prim

L'algoritmo di prim si basa sull'applicazione della regola del taglio fin quando è possibile. E' dunque chiaro che l'algoritmo restituirà correttamente il risultato avendone prima dimostrato la correttezza.

```
162 def primGenerico(Grafo g)->albero ricoprente:
163     T=0
164     while T ha meno di g nodi:
165         trova arco di costo minimo incide in T #regola
              taglio
166         aggiungi arco a t
```

La complessità del generico algoritmo di Prim sarà T(n,m)=O(nm), il metodo infatti andrà avanti fin quando T non conterrà tutti i nodi, inoltre genericamente la ricerca degli archi incidenti implicherà la visita di tutti gli archi.

Ottimizzazione Prim

Attraverso una coda di priorità possiamo rappresentare il taglio. Ogni elemento della coda sarà una coppia<nodo,costo arco collega all'albero>

```
167 def prim(Grafo g,Nodo p)->albero ricoprente:
168     T=0
169     for nodo n in g
170         d(n)=+inf
171     CodaPriorita c
172     c.insert(p,0)
173     d(p)=0
174     while c is not empty:
175         u=c.remove()
176         for arco u,v in g:
177             if d(v)=+inf:
178                 u.insert(v,w(u,v))
179                 d(v)=w(u,v)
180                 u padre di v in T
181             elif w(u,v)<d(v):
182                 s.decreasekey(v,d(v)-w(u,v))
183                 d(v)=w(u,v)
184                 u nuovo padre di v in T
185     return T
```

La complessità dell'algoritmo sarà $\theta(mlogn)$

Kruskal

Anche l'algoritmo di kruskal è corretto, questo perché si basa sull'applicazione della regola del taglio o della regola del ciclo. La regola del ciclo verrà applica se l'arco incide nello stesso albero blu, altrimenti applicheremo la regola del taglio.

```
186 def kruskalGenerico(Grafo g)->albero ricoprente:
187     ordina archi di G in base al peso
188     T=0
189     for arco (u,v) in archi ordinati:
190         if !(u,v) connessi in T:
191             T.add((u,v))
192     return T
```

L'analisi della complessità è la stessa.

Ottimizzazione Kruskal

Attraverso la struttura dati union find possiamo rappresentare le foreste di alberi come insiemi disgiunti.

```
193 def kruskalottimizzato(Grafo G)->albero ricoprente:
194     UnionFind uf
195     T=0
196     ordina archi di G
197     for nodo in G:
198         uf.makeSet(nodo)
199     for arco (u,v) in archiordinati:
200         if uf.find(u)!=uf.find(v):
201             uf.union(u,v)
202             t.add(u,v)
203     return T
```

La complessità dell'algoritmo sarà $\theta(mlogn)$

Cammini Minimi

Definizioni base

Definition 7 (Cammino minimo)**.** *Il costo di un cammino è definito tramite la somma dei pesi di gli archi che costituiscono il cammino. Il cammino minimo è il cammino di costo minore.*

Theorem 7 (Sotto cammini)**.** *La proprietà principale dei cammini è che ogni loro sotto cammino è un cammino minimo*

Proof. Supponiamo per assurdo un cammino sia minimo ma un suo sotto cammino no. Allora esisterà un altro sotto cammino che abbiamo costo minore; Osserviamo pero che formato questo nuovo cammino il suo costo è maggiore del cammino minimo iniziale. Arriviamo dunque ad un assurdo □

Nel caso in cui nel grafo sia presente un ciclo negativo sarà impossibile determinare il cammino minimo.

Distanza tra due nodi: Condizione di Bellman

Theorem 8 (Condizione Bellman)**.** *Dato un grafo g orientato e pesato, per ogni arco (u,v) e per ogni nodo S vale la seguente proprietà.*

$$d_{su} + w(u, v) \geq d_{sv}$$

Theorem 9 (Appartenenza ai cammini minimi)**.** *Dato un grafo g orientato e pesato, per ogni arco (u,v) e per ogni nodo S, l'arco (u,v) appartiene al cammino minimo da s a v se:*

$$d_{su} + w(u, v) = d_{sv}$$

Djkastra

L'algoritmo di Djkastra calcola l'albero dei cammini minimi attraverso la tecnica golosa scegliendo ad ogni passo l'arco che minimizza la distanza dal nodo appartenente all'albero dei cammini minimi e un nodo non appartenente a questo albero.
Supponiamo di avere un grafo G con tutti archi positivi

Proof. Dimostriamo la correttezza del metodo goloso: Supponiamo che un arco (u,v),con u che appartiene all'albero dei cammini minimi e v che non appartiene, minimizzi la distanza dal nodo di partenza s a v ma per assurdo non appartenga al cammino minimo da s a v

Esisterà allora un altro arco (x,y) tale che $x \in T$ e $y \notin T$ che appartiene al cammino minimo da s a v. Appartenendo al cammino minimo allora $d_{sv} = d_{sx} + w(x,y) + w(\pi^*_{yv})$ dall'ipotesi pero era l'arco (u,v) a minimizzare d_{sv} quindi $d_{sv} = d_{sx} + w(x,y) + w(\pi^*_{yv}) \geq d_{su} + w(u,v)$ Abbiamo cosi trovato l'assurdo □

Algoritmo

```
204 def dijkastraGenerico(Grafo g,Nodo s)->albero cammini
    minimi:
205     T=0
206     inizializza distanza da s a v a +inf per ogni nodo
207     while T ha meno di g nodi:
208         trova arco incidente in T minimizza distanza dsv
209         dsv=dsu+w(u,v)
210         rendi u padre di v in t
```

```
211 def dijkastra(Grafo g,Nodo s)->albero cammini minimi:
212     T=0
213     for nodo v in G:
214         dsv=+inf
215     CodaPriorita c
216     c.offer(s,0)
217     while c.size()!=0:
218         u=c.poll()
219         for arco (u,v) in G:
```

```
220             if dsv=+inf:
221                 c.offer(v,dsu+w(u,v))
222                 dsv=dsu+w(u,v)
223                 u padre v in T
224             elif Dsv>dsu+w(u,v)
225                 c.decreasekey(v,dsv-(dsu+w(u,v))
226                 dsv=(dsu+w(u,v))
227                 u nuovo padre di v in T
228     return T
```

Floyd Warshall

L'algoritmo di Floyd Warshall calcola la distanza minima fra ogni coppia di nodi anche in presenza di costi negativi(basta non avere cicli negativi). In particolare dobbiamo definire due elementi che ci permetteranno attraverso la tecnica dinamica di calcolare la distanza minima tra ogni coppia di nodi.

- Distanza k vincolata: La distanza d_{xy}^k si dice k vincolata se è la distanza piu corta che possiamo percorrere per arrivare da x a y senza passare dai nodi intermedi $v_{k+1}...v_n$
- Cammino minimo k vincolato: Un cammino minimo k vincolato se ha costo minimo tra tutti i cammini che non usano vertici intermedi da $v_{k+1}...v_n$.

Lemma 10. *Esistenza cammini minimi k-vincolati: Dato un grado $G < V, E, \lambda >$ se non ci sono cicli negativi esiste sempre un cammino k-vincolato* ***semplice*** *tra ogni coppia di nodi.*

Lemma 11. *Costruzione cammini minimi k-vincolati:* $d_{xy}^k = min\ \ d_{xy}^{k-1}, d_{x,v_k}^{k-1} + d_{v_k,y}^{k-1}$

Proof.
- v_k non appartiene al cammino minimo k-vincolato. Logicamente allora anche il k-1 esimo sarà un cammino minimo k-vincolato non contenendo nodi intermedi $v_{k+1}...v_n$
- v_k appartiene al cammino minino k vincolato. Dalle ipotesi sappiamo che il cammino minimo k-vincolato deve essere semplice, allora $v_{xv_k}^{k-1} + vv_k y^{k-1} = v_{xy}^k$

□

```
229 def algoritmoFloydWarshall(Grafo g)
230     D[n][n][n]
231     for i to n
232         for j to n
233             if i==j
234                 D[0][i][j]==0
235             else if w(i,j)!=+inf
236                 D[0][i][j]=w(i,j)
237             else
238                 D[0][i][j]=+inf
239     for k=1 to n
240         for i to n
241             for j t n
242                 D[k][i][j]=min{D[k-1][i][j],D[k-1][i][k]+D[
                       k-1][k][j]}
243     return D
```

L'algoritmo avrà ovviamente complessità spaziale e temporale cubica

```
244 def algoritmoFloydWarshallQuadratica(Grafo g)
245     D[n][n]
246     for i to n
247         for j to n
248             if i==j
249                 D[i][j]==0
250             else if w(i,j)!=+inf
251                 D[i][j]=w(i,j)
252             else
253                 D[i][j]=+inf
254     for k=0 to n
255         for i to n
256             for j t n
257                 D[i][j]=min{D[i][j],D[i][k]+D[k][j]}
258     return D
```

Possiamo ridurre la complessità spaziale perché per ogni coppia (x, y) la cella D^k e D^{k-1} relativa allo stessa coppia conterranno lo stesso valore dato che $D^{k-1}_{v_k v_k} = 0$

Hoofman

Il problema risolto dal codice Hoofman consiste nel codificare una stringa in una sequenza binaria di 0 ed 1. La codifica effettuata dall'algoritmo di Hoofman è la piu efficiente.

Prefix Code

Prefix Code è una funzione che associa ad ogni carattere della stringa una sequenza di bit 0 1 tale per cui per ogni coppia di caratteri appartenenti alla stringa x,y, il risultato della funzione prefisso su x non è un prefisso di y.

Il numero medio di bit per carattere data una sequenza sarà dato da $\sum_{x \in S} f_x C(x)$ dove f_x è la frequenza del carattere x nella stringa è C(x) è il numero di bit utilizzati per rappresentare x.

La sequenza sarà codificata in maniera ottimale se l'albero nel quale sarà rappresentata è completo ovvero tutti i nodi non foglia hanno due figli.

Algoritmo Hoofman

L'algoritmo di hoofman rispetta 3 proprieta basilari

- Caratteri frequenza minore in basso all'albero
- Albero completo
- Ordine arrivo caratteri è ininfluente

L'algoritmo applica la tecnica golosa:

1. calcola la frequenza di ogni carattere
2. crea un albero dal basso all'alto
3. crea ricorsivamente il padre e prosegue fino in cima

```
259 def Hoofman(s):
260     #frequenza rappresentata come coppia character,int
261     frequenze f=calcolaFrequenze(s)
262     heap h
263     for (c,f) in f:
264         nodo = new Nodo(c,f)
265         h.add(nodo)
266     while h.size()!=1:
267         nodo1=h.poll()
268         nodo2=h.poll()
269         padre=new Nodo(null,nodo1.f+nodo2.f,nodo1,nodo2)
270         h.offer(padre)
271     return h.poll()
```

La complessità dell'algoritmo sarà dato da $T(n) = O(nlogn)$ poiché le operazioni sull'heap andranno effettuate per ogni nodo.

Knapsack

Il problema del knapsack consiste nell'inserire un totale di oggetti in un contenitore di peso massimo W non sforando mai la capacita del contenitore. Ne esistono due varianti. Knapsack intero nel quale gli oggetti o sono inseriti tutti o solo una parte e knapsack frazionario nel quale possiamo inserire anche una percentuale dell'elemento.

Knapsack Frazionario

Il knapsack frazionario viene risolto attraverso la tecnica golosa ordinando tutti gli oggetti in ordine crescente in base al loro valore dato dal rapporto $\frac{valore}{peso}$. Inseriremo quindi via via gli elementi al contenitore partendo da quelli a valore maggiore.

```
272 def knapsackFrazionaro(pesi:list,valori:list,capacita:int)
273     valoreTotale=0;
274     vOggetti
275     for i in pesi:
276         vOggetti[i]=valori[i]/pesi[i]
277     sort(vOggetti) #ordine decrescente
278     #ovviamente gli oggetti andavano inseriti nell'array
           come oggetti di una classe che contiene peso valore
           e il loro rapporto
279     for i in vOggetti:
280         if capacita-peso[i] >0
281             capacita-=capacita[i]
282             valoreTotale+=valore[i]
283         else
284             frazione = capacita/peso[i]
285             valoreTotale+=valore[i]*frazione
```

```
286             capacita-=peso[i] * frazione
287             break
288     return valoreTotale
```

Knapsack Intero

Per risolvere il problema del Knapsack intero possiamo utilizzare la tecnica dinamica. Identifichiamo con W il peso totale dello zaino. Con $B(k, w)$ il beneficio utilizzando i primi k elementi assumendo peso massimo w.

$$B(i,j) = \left\{ \begin{array}{ll} B(k-1,w), & \text{peso wk maggiore w} \\ maxB(k-1,w), B(k-1, w-w_k+v_k) & \end{array} \right\} \tag{5}$$

Nota che se peso wk maggiore w non puoi inserire l'elemento o sforeresti il vincolo sul peso.
Nel secondo caso invece inseriremo l'elemento se effettivamente ci porta un beneficio