Appunti (Algoritmica)

Mario Di Modica

2020/2021

Indice

1 Introduzione agli algoritmi **2**

2 Algoritmi di sorting **4**

3 Master Theorem **8**

4 Limiti inferiori **10**

5 Array dinamici e Heap **11**

6 Condizioni logiche su Array e Matrici **16**

7 Strutture dati lineari **17**

8 Dizionari **25**

9 Tabelle hash **27**

10 Alberi binari **35**

11 Alberi binari di ricerca **38**

12 Alberi 2-3 **40**

13 Grafi **43**

14 Programmazione Dinamica **59**

15 Teoria della calcolabilitá **67**

16 Macchine di Turing **72**

17 Problemi intrattabili **75**

18 Teoria della complessitá **78**

1 Introduzione agli algoritmi

Un *algoritmo* è una strategia che serve per risolvere un problema ed è costituito da una sequenza finita di operazioni (dette anche istruzioni).
Uno tra i primi algoritmi creati si basa sull'antichissimo metodo della *moltiplicazione egizia* ed è il seguente:

Moltiplicazione egizia

```
MOLEG
  P=0;
  while (A!=0) {
    if (A%2!=0) {
      P=P+B;
      A=A/2;
      B=B*2;
    }
  }
  return P;
```

Costo in tempo
$T(n) = \Theta(n^2)$.

Un altro esempio fondamentale per apprendere al meglio a cosa serve un algoritmo è sicuramente quello del "*Case study*" dove ci viene proposto di trovare il numero minore di pesate per trovare una moneta falsa in un gruppo di dodici monete.
Ovviamente non è possibile farlo con solo due pesate ma una volta creato il relativo albero di decisione, con tutti i casi e tutte le soluzioni possibili, siamo in grado di concludere che questo "problema" può essere risolto con sole tre pesate.
Quindi abbiamo trovato un limite inferiore (lower bound), ovvero il numero minimo di operazioni da svolgere in tutti i casi possibili, ed il nostro algoritmo è *ottimo*.
Introduciamo adesso il concetto di *Input size*, in italiano "dimensione dei dati" rappresenta appunto il numero di elementi in ingresso o il numero di passi necessari da svolgere per completare un operazione.
Andando sempre più avanti ci troveremo a risolvere problemi sempre più complessi e di conseguenza anche l'algoritmo che andremo a scrivere per risolverlo sarà sempre più complicato sia da scrivere, che da comprendere.
È proprio in questo momento che abbiamo bisogno di approfondire un argomento fondamentale riguardo la risoluzione corretta di un problema. Non basterà più scrivere un qualsiasi algoritmo senza tenere conto di quanto sia efficiente, ma da ora in poi dovremo fare i conti con qualcosa di più importante.

Complessità di un algoritmo
La complessità di un algoritmo può essere vista secondo due aspetti:

- Tempo: è il costo delle operazioni di un algoritmo rispetto alla dimensione in input.
- Spazio: è aggiuntivo rispetto ai dati in input che dobbiamo aggiungere per risolvere un algoritmo (in sintesi: quanto spazio occupa un algoritmo).

Principalmente viene utilizzata la lettera n per indicare la input size o un numero qualunque di passi da compiere e come vedremo dopo, indicheremo con O, Θ e Ω i tre tipi di *ordini di complessità*.
Ma, tornando a noi, vediamo quali sono i problemi che affronteremo durante questo corso.
Risolveremo per esempio problemi di ricerca, di ordinamento e lavoreremo spesso con delle *strutture dati* che ci permettono di organizzare le informazioni per rendere più efficiente il nostro algoritmo.
Alla base di questo ci sono sicuramente i cosiddetti "problemi decidibili" che a loro volta si dividono in:

- Classe P: Per esempio problemi di ricerca o ordinamento, dove l'algoritmo risolve il problema in un tempo "raggiungibile".
- Classe NP: Nessuno conosce un algoritmo efficiente (brute-force) per risolverli.

Come vedremo nei capitoli successivi, esistono molti algoritmi di ricerca. Il problema principale che accomuna questi algoritmi è la ricerca di una nota chiave k all'interno di un array A.
Un esempio di ricerca è la ricerca sequenziale.

Sequential search

```
SEQ_SEARCH (k,A)
  i=1;
  trovato=false;
  while (i<=n && !trovato) {
    if (A[i]==k) trovato=true;
    else i++;
  }
  return i;
```

Costo in tempo
$T(n) = \Theta(1)$ (caso ottimo).
$T(n) \simeq \frac{n}{2}$ (caso medio).
$T(n) = O(n)$ (caso pessimo).

Definizione (Word model)

Tutti gli a_i devono avere dimensione tale da essere contenuti in una cella di memoria.
Un altro esempio fondamentale riguardo gli algoritmi di ricerca è la ricerca binaria.

Ricerca binaria

```
RICERCABINARIA (a,k)
  pos=-1;
  sin=1;
  des=n;
  while (sin<=des && pos==-1) {
  cen=(sin+des)/2;
    if (k==a[cen]) pos=cen;
    else if (k<a[cen]) des=cen-1;
    else sin=cen+1;
  }
  return pos;
```

Costo in tempo (caso pessimo)
$T(n) = O(logn)$.

Minimo

```
MIN (a)
  indicemin=1;
  min=a[1];
  for (i=2; i<=n; i++) {
    if (a[i]<min) {
      min=a[i];
      indicemin=i;
    }
  }
  return <min,indicemin>;
```

Costo in tempo (caso pessimo)
$T(n) = \Theta(n)$.

2 Algoritmi di sorting

Quando scriviamo un algoritmo è importante saperlo analizzare correttamente, in modo tale da verificarne la complessità in tempo nel caso ottimo, medio e pessimo.
Dato A un qualsiasi algoritmo e I come istanza di input, avremo che $|I| = n$ sarà la dimensione dell'input.
Complessità al caso ottimo: $minT_A(n)$.

Complessità al caso pessimo: $maxT_A(n)$.
Il problema principale quando parliamo di algoritmi di sorting è il seguente:
Input: array A di n numeri interi.
Output: array A ordinato.

Insertion sort

```
INSERTIONSORT (A)
  for j=2 to n {
    k=A[j];
    i=j-1;
    while (i>0 && A[i]>k) do {
      A[i+1]=A[i];
      i--;
    }
    A[i+1]=k;
  }
```

Invariante di ciclo: All'inizio di ogni iterazione j del for, il "sottoarray" A[1...j-1] è ordinato e contiene gli elementi che vi erano nell'array iniziale.

Costo in tempo
Caso ottimo: $T(n) = \Theta(n)$.
Caso pessimo: $T(n) = \Theta(n^2)$.

Selection sort

```
SELECTIONSORT (A)
  for i=1 to n-1 {
    min=i;
    for j=i+1 to n {
      if (A[j]<A[min]) then min=j;
    }
    swap (A[i],A[min]);
  }
```

Invariante di ciclo: All'inizio di ogni iterazione del for esterno, il "sottoarray" A[1...j-1] è ordinato e contiene gli i-1 elementi più piccoli dell'array iniziale.

Costo in tempo
$T(n) = \Theta(n^2)$.

Divide et Impera

Il paradigma Divide et Impera ci permette di "semplificare" un problema di grande dimensione, dividendolo (**Divide**) in tanti piccoli sotto-problemi (aventi a loro volta altrettante soluzioni (**Conquer**) per poi ricombinarli (**Combine**) tutti insieme ed ottenere un' unica soluzione.

Costo in tempo

$T(n) = D(n) + T(n_1) + ... + T(n_a) + C(n)$ se $n > 0$
$T(n) = \Theta(1)$ se $n <= 0$

Merge sort

```
MERGESORT (A,p,r)
  if (p<r) then {
    q=(p+r)/2;
    MERGESORT (A,p,q);
    MERGESORT (A,q+1,r);
    MERGE (A,p,q,r);
  }
```

Costo in tempo
$T(n) = \Theta(1)$ se $n <= 1$
$T(n) = 2T(\frac{n}{2}) + \Theta(n)$ se $n >= 2$

Dato $n = |A| = r - p + 1$.

```
MERGE (A,p,q,r)
  n1=q-p+1;
  n2=r-q;
  creo array L[1...n1+1] e R[1...n2+1];
  for i=1 to n1 do L[i]=A[p+i-1];
  for j=1 to n2 do R[j]=A[q+j];
  L[n1+1]=+Infinity;
  R[n2+1]=+Infinity;
  i,j=1;
  for k=p to r do {
    if (L[i]<=R[j]) then {
      A[k]=L[i];
      i++;
    }
    else {
      A[k]=R[j];
      j++;
    }
  }
```

Invariante di ciclo: All'inizio della k-esima iterazione del for: A[p,k-1] contiene i k-p elementi più piccoli di L e R, L[i] e R[j] sono gli elementi più piccoli dei rimanenti elementi di L ed R.

Costo in tempo
$T(n) = \Theta(nlogn)$.

Quick sort

```
QUICKSORT (A,p,r)
  if (p<r) {
    q=PARTITION (A,p,r);
    QUICKSORT (A,p,q-1);
    QUICKSORT (A,q+1,r);
  }
```

Costo in tempo (caso pessimo)
$T(n) = \Theta(n^2)$.

Osservazione
Come possiamo notare, all' interno della funzione QUICKSORT viene chiamata una funzione esterna che prende il nome di PARTITION che crea, appunto, una partizione dell'array A e assegna a k (*pivot*) il valore dell'array in posizione r. A questo punto il *pivot* viene spostato in una posizione tale da avere solo numeri minori di esso alla sua sinistra, e solo numeri maggiori alla sua destra. Una volta "diviso" l'array andremo a richiamare la funzione QUICKSORT sul sottoarray contenente solo elementi minori di k e sul sottoarray contenente solo elementi maggiori di k.

```
PARTITION (A,p,r)
  k=A[r];
  i=p-1;
  for j=p to r-1 {
    if (A[j]<=k) {
      i++;
      swap (A[i],A[j]);
    }
  }
  swap (A[i+1], A[r]);
  return i+1;
```

Costo in tempo
$T(n) = \Theta(n)$.

Il vero motivo per il quale dovremmo preferire QUICKSORT a SELECTION-SORT è che il primo occupa lo spazio in *loco*, il secondo no.

Esiste inoltre una variante del QUICKSORT e prende il nome di "RANDOM QUICKSORT", vediamo prima cos'è un algoritmo randomizzato.

Algoritmo randomizzato: è un algoritmo il cui comportamento è determinato, oltre che dall'input, anche da valori dati da un generatore di numeri casuali. Per esempio $random(a, b)$ genera un numero casuale compreso tra a e b.

Ci sono due cose di cui bisogna tenere conto:

1. L'array A viene "mischiato" in modo casuale prima dell'esecuzione di QUICKSORT.
2. L'array A è quello in input ma quando si esegue PARTITION, per la scelta del *pivot* si usa un procedimento casuale.

```
RANDOM-QS (A,p,r)
  if (p<r) then {
    q=RANDOM-PARTITION (A,p,r);
    RANDOM-QS (A,p,q-1);
    RANDOM-QS (A,q+1,r);
  }
  RANDOM-PARTITION (A,p,r) {
    i=random (p,r);
    swap (A[i], A[r]);
    return PARTITION (A,p,r);
  }
```

Costo in tempo
$T(n) = \Theta(nlogn)$ (caso ottimo).
$T(n) = O(n + x) = E(x) \rightarrow$ *Valore atteso di x* (caso medio).
$T(n) = \Theta(n^2)$ (caso pessimo).

x è il numero totale di confronti eseguiti in tutte le chiamate di RANDOM-PARTITION durante l'intera esecuzione di RANDOM-QS.

```
PARTITION (A,p,r)
  k=A[r];
  i=p-1;
  for j=p to r-1 {
    if (A[j]<=k) {
      i++;
      swap (A[i],A[j]);
    }
  }
  swap (A[i+1], A[r]);
  return i+1;
```

Costo in tempo
$T(n) = \Theta(1) + \Theta$(numero di confronti nel ciclo for).

3 Master Theorem

Una *relazione di ricorrenza* per una funzione $T(n)$ è un'equazione o una disequazione che esprime $T(n)$ rispetto a valori di T su variabili più piccole, completata dal valore di T nel caso base (o nei casi base).

Ci sono due principali tipi di relazioni di ricorrenza:

Bilanciate

- Metodo iterativo: sviluppo la ricorrenza fino ai casi base.
- Albero di ricorsione: ausilio grafico che rappresenta i costi ad ogni livello della ricorsione.
- Metodo di sostituzione: si ipotizza una soluzione e la si dimostra.
- Master theorem (Teorema dell'esperto)

Di ordine k

Non sono molto diverse dalle prime, otteniamo risultati diversi al variare di k.

Teorema (Master theorem)

Siano $a \geq 1$, $b > 1$ e $n_1 \geq 0$ delle costanti e $f(n) \geq 0$.

$$T(n) = \begin{cases} \Theta(1) & n \leq n_1 \\ aT(\frac{n}{b}) + f(n) & n > n_1 \end{cases}$$

Asintoticamente T(n) è limitata come segue:

1. $f(n) = O(n^{log_b a - \varepsilon})$ con $\varepsilon > 0 \rightarrow T(n) = \Theta(n^{log_b a})$
2. $f(n) = \Theta(n^{log_b a}) \rightarrow T(n) = \Theta(n^{log_b a} \cdot logn)$
3. $f(n) = \Omega(n^{log_b a + \varepsilon})$ con $\varepsilon > 0 \rightarrow T(n) = \Theta(f(n))$

Creando l'albero di ricorsione relativo a questo teorema, noteremo che dovremo suddividere T(n) $n^{log_b a}$ volte fino ad ottenere, come costo in tempo, la seguente formula:

$$\Theta(n^{log_b a}) + \sum_{j=0}^{log_b n - 1} a^j \cdot f(\frac{n}{b^j})$$

Passiamo adesso a confrontare $f(n)$ con $n^{log_b a}$.

1. $f(n)$ è polinomialmente più piccola di $n^{log_b a}$.
2. $f(n) \in \Theta(n^{log_b a})$.
3. $f(n)$ è polinomialmente più grande di $n^{log_b a}$.

Gli unici casi in cui non siamo in grado di utilizzare questo teorema sono quelli in cui, quando vado a confrontare $f(n)$ e $n^{log_b a}$ scopro che non asintoticamente equivalenti (quindi escludo il caso (2)) e che una è limitata superiormente dall'altra ma non c'è il fattore polinomiale di differenza. Al suo posto possiamo trovare, per esempio, un fattore logaritmico.

$$f(n) = \Theta(\frac{n^{log_b a}}{logn})$$

Dimostrazione del **Master theorem** (caso delle potenze esatte).

$$T(n) = \begin{cases} \Theta(1) & n \leq n_1 \\ aT(\frac{n}{b}) + f(n) & n > n_1 \end{cases}$$

$T(n) = aT(\frac{n}{b}) + f(n) = a(aT(\frac{n}{b^2}) + f(\frac{n}{b})) + f(n) = a^2(aT(\frac{n}{b^3}) + f(\frac{n}{b^3})) + af(\frac{n}{b}) + f(n) = a^3T(\frac{n}{b^3}) + a^2f(\frac{n}{b^2}) + af(\frac{n}{b}) + f(n) =$ ...dopo i-passaggi $= a^iT(\frac{n}{b^i}) + \sum_{j=0}^{i-1} a^j f(\frac{n}{b^j}) =$ ...quando $i = log_b n = a^{log_b n} \cdot T(\frac{n}{b^{log_b n}}) + \sum_{j=0}^{log_b n - 1} a^j f(\frac{n}{b^j}) =$

$n^{log_b a} \cdot \Theta(1) + \sum_{j=0}^{log_b n - 1} a^j f(\frac{n}{b^j})$.

Notiamo che $n^{log_b a} \cdot \Theta(1)$ è il *costo della risoluzione diretta di tutti i casi base*, mentre $\sum_{j=0}^{log_b n - 1} a^j f(\frac{n}{b^j})$ è il *costo della DIVIDE e COMBINE a tutti i livelli della ricorsione.*

4 Limiti inferiori

Dato π un qualunque problema da risolvere, la *complessità di* π rappresenta la complessità al caso pessimo del miglior algoritmo che risolve π.
Un algoritmo A, che risolve π, fornisce un limite superiore alla complessità di π (ovvero non posso impiegare più tempo di quanto ne impieghi A).

Indichiamo con $T_A(n)$ il limite superiore.
Se, per π, posso dimostrare un limite inferiore $L(n)$[1] (dove n è la input size di π), ho mostrato che:

$\forall$ algoritmo A che risolve π, ho che:

$T_A(n) \in \Omega(L(n))$ al caso pessimo.

Concludiamo dicendo che se ho $A : T_A(n) \in \Theta(L(n))$ allora A è un algoritmo risolutivo *ottimo*, ovvero il suo costo è uguale limite inferiore (asintoticamente).

[1] $L(n)$ rappresenta il numero minimo di operazioni che devono essere necessariamente svolte da un algoritmo per risolvere π.

Metodi per calcolare i limiti inferiori.

1. Dimensione dei dati.
2. Eventi ripetuti o contabili.
3. Albero di decisione.

Dimensione dei dati

Se la soluzione di un problema richiede l'esame di tutti i dati in input, allora $\Omega(n)$ è un limite inferiore.

Eventi ripetuti o contabili

Se la ripetizione di un certo evento è necessaria a risolvere π, allora il numero di volte che si deve ripetere, moltiplicato per il suo costo, è il limite inferiore.

Albero di decisione

Si applica a problemi risolubili attraverso sequenze di decisioni (confronti, pesate, partite...) che riducono via via lo spazio delle possibili soluzioni.
Indicheremo con AdD un albero di decisione (tipicamente binario).
In un qualsiasi AdD abbiamo tre tappe principali:

- Radice (situazione iniziale)
- Nodo interno (decisione)
- Foglie (soluzioni)

Il percorso radice $\rightarrow$ foglia è una qualsiasi esecuzione dell'algoritmo.

Un limite inferiore per π è l'altezza dell'albero, ovvero il numero di passi necessari al caso pessimo (profondità massima dell'albero).
L'altezza minima di un AdD si ottiene con un AdD *bilanciato*, ovvero se i nodi sono distribuiti equamente tra i sottoalberi.

$$h = log_2 n \quad e \quad h >= log_2(numero\ delle\ foglie)$$

Teorema

$AdD : limite\ inferiore\ di\ \pi = L(n) = \Omega(log_2 s(n))$.

Dove s(n) rappresenta il numero delle possibili soluzioni.

5 Array dinamici e Heap

Un *Heap* è una struttura dati dinamica (ovvero cambia nel corso del tempo) in grado di svolgere varie operazioni come l'*inserzione* o la *cancellazione*.
Possiamo utilizzare un Heap implementandolo su strutture a noi note come gli *array*, usufruendo di tutte le sue proprietà.

Gestione di un array dinamico
a: array.
d: dimensione dell'array.
n $\leq$ d (numero di elementi memorizzati).

Idea: Quando viene superata una certa soglia (valore) l'array viene raddoppiato (o dimezzato).
Questo tipo di operazioni purtroppo hanno un enorme costo, sia in spazio, che in tempo ($\Theta(n)$) infatti vengono utilizzate una volta tanto e la complessità $\Theta(n)$ viene diminuita nell'insieme di tutte le altre operazioni che invece hanno costo costante ($\Theta(1)$). Vogliamo quindi "verificare" che il numero di operazioni che hanno costo costante è abbastanza grande da "assorbire" il tempo necessario per fare un raddoppio (o un dimezzamento) ottenendo quella che viene chiamata *complessità ammortizzata* che ha come costo in tempo $\Theta(1)$ per operazione.

Vediamo adesso gli algoritmi relativi al raddoppio e al dimezzamento di un array.

Raddoppio

```
RADDOPPIO ()
  if (n==d) {
    b=nuovoarray(2*d);
    for (i=1;i<=n;i++) {
      b[i]=a[i];
      a=b;
      d=d*2;
    }
  }
```

Dimezzamento

```
DIMEZZAMENTO ()
  if (d>1 && n==d/4) {
    b=nuovoarray(d/2);
    for (i=1;i<=n;i++) {
      b[i]=a[i];
      a=b;
      d=d/2;
    }
  }
```

Costo in tempo
$T(n) = O(n)$.

Teorema
m operazioni di inserzione o cancellazione, in un array dinamico, richiedono un costo in tempo pari a $O(m)$.
Dimostrazione
Dopo un raddoppio, nel nostro array, avremo m=d+1 elementi e l'array avrà dimensione 2d, per avere un altro raddoppio ci vogliono m+1 inserimenti.

Dopo un dimezzamento, avremo m=$\frac{d}{4}$ elementi e per avere un altro dimezzamento occorrono m+1 inserimenti e $\frac{m}{2}$ cancellazioni.
Il costo di $O(m)$ operazioni richiesto da un raddoppio o da un dimezzamento viene assorbito da $\Omega(m)$ operazioni che l'hanno causato, possiamo concludere che il costo ammortizzato per operazione è $\Theta(1)$.

Per quanto riguarda la struttura di un Heap, possiamo considerarlo come un albero binario "quasi" completo ($n \neq 2^k - 1$).
Per essere considerato Heap un albero binario quasi completo deve seguire determinate proprietà, come ad esempio che, ogni nodo deve avere un Max-Heap ovvero il valore di ogni nodo deve essere maggiore del valore di tutti i suoi figli (se esistono).
Altezza di un nodo
Si indica con h_n ed è la massima distanza tra un nodo ed una foglia misurata in numero di archi.
Altezza di un albero
Si indica con h_a ed è la massima distanza tra la radice ed una foglia misurata in numero di archi.
Teorema
Un albero binario completo di altezza h ha:

$N_h = 2^{h+1} - 1$ nodi.
$F_h = 2^h$ foglie.
$I_h = 2^h - 1$ nodi interni.

Ma quanto vale h?

Per prima cosa studiamo la relazione tra n e h in un albero binario completo di altezza h.
$n = 2^{h+1} - 1$ (porto il -1 a primo membro).
$n + 1 = 2^h + 1$ (faccio il log_2 in entrambi i membri).
$log(n + 1) = h + 1$ (porto il -1 a primo membro e scambio i due membri).
$h = log(n + 1) - 1$ (possiamo dire che $h \in \Theta(logn)$.

Preso un qualunque albero binario completo possiamo dire che:
$2^h \leq n \leq 2^{h+1} - 1$.
Quindi possiamo scrivere:
$2^h \leq n < 2^{h+1}$ (faccio il log_2 di tutti i membri).
$h \leq logn < h + 1$.

Ottenendo che:

$$\begin{cases} h \leq logn \\ h > logn - 1 \end{cases}$$

Concludiamo scrivendo $h = \lfloor logn \rfloor$.
Quando decidiamo di utilizzare un Heap è fondamentale conoscere gli algoritmi per l'estrazione o lettura del massimo e per l'inserimento di una chiave.

Inserimento

```
MAXHEAPINSERT (A, key)
  A.heapsize=A.heapsize+1;
  A[A.heapsize]=-Infinity;
  HEAPINCREASEKEY (A,A.heapsize,key);
  HEAPINCREASEKEY (A,i,key);
  if (key<A[i]) return "errore";
  A[i]=key;
  while (i>1 && A[parent(i)]<A[i]) {
    swap (A[i],A[parent(i)]);
    i=parent(i);
  }
```

Costo in tempo
$T(n) = O(logn)$.

Estrazione del massimo

```
HEAPEXTRACTMAX (A)
  if (A.heapsize<1) return "errore" || "coda vuota";
  max=A[i];
  A[1]=A[A.heapsize];
  A.heapsize=A.heapsize-1;
  MAXHEAPIFY (A,1);
  reutrn max;
  l=left(i);
  r=right(i);
  if (l<=A.heapsize && A[l]>A[i]) max=l;
  else max=i;
  if (r<=A.heapsize && A[r]>max) max=r;
  if (max!=i) {
    swap (A[max],A[i]);
    MAXHEAPIFY (A,max);
  }
```

Costo in tempo
$T(n) = O(logn)$.

Costruzione di un Heap

```
BUILDMAXHEAP (A)
  A.heapsize=A.length;
  for (i=A.length/2;i>=1;i--) MAXHEAPIFY (A,i);
```

Costo in tempo
$T(n) = O(n)$.

Invariante di ciclo: All'inizio dell'i-esima iterazione del for abbiamo che ogni nodo da i+1...n è radice di un Heap.

Heap sort

Possiamo vedere questa procedura di ordinamento come un "miglioramento" del Selection sort visto precedentemente e si basa su due "passi" fondamentali:

1. Costruzione dell'Heap.
2. - Selezione max e posizionamento finale.
   - Ristrutturazione dell'Heap.

```
HEAPSORT (A)
  BUILDMAXHEAP (A);
  for (i=A.length;i>1;i--) {
    swap (A[i],A[1]);
    A.heapsize=A.heapsize-1;
    MAXHEAPIFY (A,1);
  }
```

Costo in tempo
$T(n) = O(nlogn)$.

A volte è possibile ordinare senza fare confronti sfruttando eventuali proprietà note a priori degli oggetti da ordinare.

Counting sort

```
COUNTINGSORT (A,B,k)
  for (i=1;i<=k;i++) C[i]=0;
  for (j=1; j <= n; j++) C[A[j].v]+=1;
  for (i=2;i<=k;i++) C[i]=C[i]+C[i-1];
  for (j=n;j>=1;j--){
    B[C[A[j].v]]=A[j];
    C[A[j].v]-=1;
  }
```

Costo in tempo
$T(n) = \Theta(k+n)$.

Un algoritmo viene detto *stabile* se preserva l'ordine iniziale tra due elementi con la stessa chiave.
Insertion sort, Merge sort, Counting sort, per esempio, sono degli algoritmi stabili. Heap sort e Quick sort no.
Un altro algoritmo di sorting da conoscere è sicuramente Radix sort che si basa sull'idea di ordinare interi cifra per cifra partendo dalla cifra meno significativa.

Vediamo come funziona.

Radix sort

```
RADIXSORT (A,d)
  for (i=1;i<=d;i++) {
    STABLESORT (A); // sulla cifra i
  }
```

Costo in tempo
$T(n) = O(d(n+k))$.

In conclusione possiamo dire che quando vogliamo usare un algoritmo di sorting stabile conviene spesso utilizzare il Counting sort perchè ha un costo in tempo minore rispetto al Radix sort.

6 Condizioni logiche su Array e Matrici

Notazioni sugli intervalli

[a,b] → *intervallo chiuso* (a, b compresi).

[a,b) → *intervallo aperto a destra* (a compreso, b escluso).

(a,b] → *intervallo aperto a sinistra* (a escluso, b compreso).

(a,b) → *intervallo aperto* (a, b esclusi).

Proprietà su Array e quantificatori

$\forall i \in [0, N).P(A[i])$ → *P è valida per* ***tutti*** *gli elementi di A.*

$\exists i \in [0, N).P(A[i])$ → *P è valida per* ***almeno*** *un elemento di A.*

Vediamo adesso due esempi sul quantificatore $\forall$ e sul quantificatore $\exists$.

```
BELONGSTOARRAY (A,x) {
  i=0;
  trovato=false;
  while ((i<A.length) && (!trovato)) do {
    if (A[i]==x) then trovato=true;
    i++;
  }
  return trovato;
}
```

```
CHECKALLODDARRAY (A) {
  i=0;
  while ((i<A.length) && (A[i]%2==1)) do {
    i++;
  }
  if (i==A.length) return true;
  else return false;
}
```

Proprietà su Matrici e quantificatori
Vogliamo verificare se P vale:

- Per **tutti** gli elementi di **almeno** una colonna.
  - $\exists j \in [0, M).\forall i \in [0, N).P(A[i, j])$.
- Per **tutti** gli elementi di **almeno** una riga.
  - $\exists i \in [0, N).\forall j \in [0, M).P(A[j, i])$
- Per **almeno** un elemento di **ogni** riga.
  - $\forall i \in [0, N).\exists j \in [0, M).P(A[i, j])$.
- Per **almeno** un elemento di **ogni** colonna.
  - $\forall j \in [0, M).\exists i \in [0, N).P(A[i, j])$.
- Per **almeno** un elemento di **tutta** la matrice.
  - $\exists i \in [0, N).\exists j \in [0, M).P(A[i, j])$.

7 Strutture dati lineari

Sappiamo bene la definizione di algoritmo, essi gestiscono i vari insiemi dinamici di dati e ne usufruiamo per risolvere i nostri innumerevoli problemi. Le *strutture dati*, invece, organizzano i dati da manipolare per semplificarne l'accesso, la modifica e per eseguire in modo efficiente le operazioni sui dati.
È bene saper distinguere i due tipi principali di strutture dati:

- Lineari: dove gli elementi vengono disposti uno dopo l'altro (es. Array, liste, code e pile).
- **Non** lineari: dove un elemento può avere più di un "successivo" (es. Alberi e grafi).

Array
È una struttura dati lineare (SD) indicizzata ad accesso diretto.
Organizzazione logica: i valori si trovano in posizioni adiacenti.
Organizzazione fisica: i valori vengono memorizzati in indirizzi di memoria adiacenti.
L'ordine logico è determinato dall'*indice* e corrisponde (in genere) all'ordine fisico in memoria. L'accesso ad una qualsiasi *cella* a[i] è costante, ovvero $O(1)$.
Utilizzare gli array può essere vantaggioso o svantaggioso in base a quello di cui abbiamo bisogno, per gli *array statici* abbiamo i seguenti vantaggi e svantaggi:

- Vantaggi: Accesso diretto a tempo costante ($O(1)$).
- Svantaggi: Gli array sono strutture rigide poichè hanno una dimensione fissata al lato della costruzione e la cancellazione e l'inserimento sono poco efficienti.

Liste

Sono strutture dati lineare ad accesso sequenziale. Sono rappresentate da una sequenza di elementi che occupano delle posizioni non necessariamente adiacenti ed il metodo per accedere all'elemento successivo consiste nel memorizzare il valore e l'indirizzo dell'elemento successivo grazie all'ausilio della parola chiave *key*, che si riferisce al valore dell'elemento, e alla parola chiave *next*, che si memorizza l'indirizzo dell'elemento successivo.
L'ordine logico è determinato dagli indirizzi e non corrisponde in generale all'ordine fisico nella memoria.
La parola chiave *head* si riferisce all'indirizzo del primo elemento in lista, per accedere all'i-esimo elemento della lista occorre quindi partire dalla "testa" (head) e scandire tutti quelli che lo precedono. Il costo per svolgere questa operazione non sarà più costante ma avrà un costo $O(i)$.

Principalmente le liste si suddividono in due tipi:

- Lista semplice (Monodirezionale): come abbiamo visto è composta da *head*, il quale fornisce l'indirizzo del primo elemento in lista, e ogni altro elemento è costituito dal campo *key* (valore) e dal campo *next* (successivo). Per essere sicuri di essere arrivati alla fine della lista, ovvero aver trovato l'ultimo elemento x, basta verificare che $x.next = nil$.
- Lista doppia (Bidirezionale): la differenza rispetto alla lista semplice è che, nella lista doppia, ogni elemento è formato, oltre che dal campo *key* e dal campo *next*, anche dal campo *prev*. Al contrario della lista semplice, che può muoversi in un'unica direzione, quella doppia può muoversi in entrambe le direzioni. Preso un qualsiasi elemento x, $x.prev$ restituirà l'indirizzo dell'elemento precedente, $x.next$ quello del successivo e *head* corrisponderà all'elemento con $x.prev = nil$.

Operazioni

Le operazioni che possiamo svolgere sulle liste sono generalmente di due tipi:

- Modifica
- Interrogazioni (Query)

List insert

```
func singleListInsert (head,x) {
  x.next=head;
  head=x;
}
```

Costo in tempo
$T(n) = O(1)$.

```
func doubleListInsert (head,x) {
  x.next=head;
  if (head!=nil) head.prev=x;
  head=x;
  x.prev=nil;
}
```

Costo in tempo
$T(n) = O(1)$.

List search

```
func listSearch (head,k) {
  x=head;
  while (x!=nil && x.key!=k) x=x.next;
  return x;
}
```

Costo in tempo (caso pessimo)
$T(n) = \Theta(n)$.

List delete

```
func singleListDelete (head,k) {
  var temp, prev;
  if (head==nil) return;
  if (head.key==k) {
    head=head.next;
    return;
  }
  prev=head;
  temp=head.next;
  while (temp!=nil && temp.key!=k) {
    prev=temp;
    temp=temp.next;
  }
  if (temp==nil) return;
  else prev.next=temp.next;
}
```

Costo in tempo
$T(n) = O(n)$.

```
func doubleListDelete (head,x) {
  if (x.prev!=nil) x.prev.next=x.next;
  else head=x.next;
  if (x.next!=nil) x.next.prev=x.prev;
}
```

Costo in tempo
$T(n) = O(1)$.

Pile e code

Sono insiemi di dati dinamici dove l'elemento da rimuovere è "predeterminato" (cioè non scelgo io quale elemento rimuovere).
Organizzazione logica: i valori si trovano in posizioni adiacenti.
Sia le pile che le code vengono spesso implementate su array perchè a volte risulta più semplice svolgere le varie operazioni sfruttando le proprietà di cui godono gli array.
Una *pila* implementa lo schema L.I.F.O. (Last in first out) ovvero l'ultimo elemento ad essere stato inserito sarà il primo ad essere eliminato. Su di esse possiamo svolgere due tipi di operazioni: modifica e interrogazioni.

Operazioni di modifica

- push: inserisce un elemento in cima alla pila.
- pop: rimuove un elemento dalla cima della pilla.

Operazioni di interrogazioni (Query)

- isEmpty: verifica se la pila è vuota.
- top: restituisce il valore dell'elemento in cima alla pila senza rimuoverlo.

Implementazione su Array

Come sappiamo le pile o le code vengono spesso implementate su array, ma quando l'array preso in considerazione è una struttura statica corriamo il rischio di andare in contro a problemi di *overflow*. Proprio per questo a volte conviene sfruttare array dinamici utilizzando una tecnica nota come "dimezza-raddoppia".

Pila vuota (Query)

```
var top=-1;
func isEmpty (pila) {
  if (t<0) return true;
  else return false;
}
```

Costo in tempo
$T(n) = O(1)$.

Inserimento

```
func push (pila,x) {
  top++;
  if (top>=pila.length) return <error:overflow>;
  pila[top]=x;
}
```

Costo in tempo
$T(n) = O(1)$.

Cancellazione

```
func pop (pila,x) {
  if (isEmpty(pila)) return <error:underflow>;
  top--;
  return pila[top+1];
}
```

Costo in tempo
$T(n) = O(1)$.

Primo elemento (Query)

```
func top (pila) {
  if (isEmpty(pila)) return <error:underflow>;
  else return pila[top];
}
```

Costo in tempo
$T(n) = O(1)$.

Implementazione su Liste

Si fa in modo tale che la cima della pila corisponda alla testa della lista, gli inserimenti e le cancellazioni si fanno sempre in testa alla lista e si mantiene il riferimento alla testa della lista che equivale alla cima della pila (*topEl*).

Pila vuota (Query)

```
func isEmpty (topEl) {
  if (topEl==nil) return true;
  else return false;
}
```

Costo in tempo
$T(n) = O(1)$.

Inserimento

```
func push (topEl,x) {
  x.next=topEl;
  topEl=x;
}
```

Costo in tempo
$T(n) = O(1)$.

Cancellazione

```
func pop (topEl,x) {
  if (isEmpty(topEl)) return <error:underflow>;
  var v=topEl.key;
  topEl=topEl.next;
  return v;
}
```

Costo in tempo
$T(n) = O(1)$.

Primo elemento (Query)

```
func top (topEl) {
  if (isEmpty(topEl)) return <error:underflow>;
  else return topEl.key;
}
```

Costo in tempo
$T(n) = O(1)$.

In una *coda* l'elemento da rimuovere (predeterminato) è quello che è rimasto al suo interno per più tempo, quindi possiamo dire che una coda utilizza una struttura F.I.F.O. (First in first out). Gli inserimenti in una coda avvengono in fondo, le estrazioni avvengono in testa.
Anche per le code abbiamo due tipi di operazioni:

- Modifica
- Interrogazioni (Query)

Operazioni di modifica

- enqueue: inserisce un elemento in fondo alla coda.
- dequeue: rimuove un elemento in testa alla coda e lo restituisce.

Operazioni di interrogazioni

- isEmpty: verifica se la coda è vuota.
- first: restituisce il valore dell'elemento in testa alla coda senza rimuoverlo.

Implementazione su Array
Anche le code come le pile possono essere implementate su array. Gli elementi della coda sono gli elementi dell'array nelle posizioni $head, head + 1, ..., tail - 1$ dove *head* si riferisce all'indice dell'elemento in testa alla coda e *tail* è l'indice della locazione in cui inserire il prossimo elemento.

Una coda implementata in un array gode di una particolare proprietà, infatti i suoi elementi vengono organizzati seguendo una *gestione circolare* se prima di *head* ci sono delle celle vuote.
Quando la coda è *vuota* *head* e *tail* coincidono, invece, quando la coda è *piena* $head = (tail + 1)\%coda.length$

Dopo aver dichiarato $head = 0$ e $tail = 0$ come variabili globali possiamo introdurre alcune funzioni su delle code implementate su array.

Coda vuota (Query)

```
func isEmpty (coda) {
  return (head==tail);
}
```

Costo in tempo
$T(n) = O(1)$.

Primo elemento (Query)

```
func first (coda) {
  if (isEmpty (coda)) return <error:underflow>;
  else return coda[head];
}
```

Costo in tempo
$T(n) = O(1)$.

Inserimento

```
func enqueue (coda,x) {
  if (head==(tail+1)%coda.length)
    return <error:overflow>;
  else coda[tail]=x;
  tail=(tail+1)%coda.length;
}
```

Costo in tempo
$T(n) = O(1)$.

Cancellazione

```
func dequeue (coda) {
  if (isEmpty (coda)) return <error:underflow>;
  var x=coda[head];
  head=(head+1)%coda.length;
  return x;
}
```

Costo in tempo
$T(n) = O(1)$.

Implementazione su liste

Quando implementiamo una coda su una lista, come per gli array, abbiamo due riferimenti: *head* e *tail*. Anche sulle liste possiamo svolgere le operazioni che eravamo in grado di svoglere sugli array, quindi abbiamo a disposizione due operazioni di modifica e due query.

Coda vuota (Query)

```
func isEmpty (head) {
  return (head==nil);
}
```

Costo in tempo
$T(n) = O(1)$.

Primo elemento (Query)

```
func first (head) {
  if (isEmpty (head)) return <error:underflow>;
  else return head.key;
}
```

Costo in tempo
$T(n) = O(1)$.

Inserimento

```
func enqueue (head,tail,x) {
  if (isEmpty (head)) head=x;
  else tail.next=x;
  tail=x;
  x.next=nil;
}
```

Costo in tempo
$T(n) = O(1)$.

Cancellazione

```
func dequeue (head,tail) {
  if (isEmpty (head)) return <error:underflow>;
  var k=head.key;
  if (head==tail) tail=nil;
  head=head.next;
  return k;
}
```

Costo in tempo
$T(n) = O(1)$.

8 Dizionari

Un *dizionario* (S) è una struttura dati che memorizza un insieme dinamico e fornisce operazioni di modifica (Insert, delete...) o query (Search, verify...).
In un dizionario *statico* possiamo svolgere solo operazioni di modifica, in quello *dinamico* possiamo svolgere entrambe le operazioni.
$\forall x \in S$:

- $x.key \rightarrow$ Individua in modo univoco ogni elemento di S.
- x.dati satellite.

$\forall x_1, x_2 \in S$: $x_1.key \neq x_2.key$.
Da ora in avanti indicheremo il numero di elementi di un dizionario con la lettera n, quindi:

$$n = |S|$$

Operazioni di modifica

- Insert (S,x): x è il riferimento all'elemento da inserire (S $\leftarrow$ S $\cup$ $\{x\}$).
- Delete (S,x): x è il riferimento all'elemento da rimuovere (S $\leftarrow$ S $\setminus$ $\{x\}$).

Operazioni di interrogazioni

- Search (S,k): k è la chiave cercata.

$$Search(S,k) \rightarrow \begin{cases} x & \text{se S contiene un elemento } x \text{ t.c. } x.key = k. \\ nil & \text{se S non contiene un elemento } x \text{ t.c. } x.key = k. \end{cases}$$

Per quanto riguarda i *dizionari ordinati* abbiamo a disposizione più query:

- Min (S): trova il minimo in un dizionario.
- Max (S): trova il massimo in un dizionario.
- Predecessore (S,x): restituisce un riferimento all'elemento con la chiave più grande che sia minore di x.key.
- Successore (S,x): restituisce un riferimento all'elemento con la chiave più piccola che sia maggiore di x.key.

Nella Tabella 1 possiamo osservare il costo in tempo di tutte queste operazioni sui vari tipi di dizionari.

Struttura dati	Ricerca	Inserimento	Cancellazione
Array non ordinato	$O(n)$	$O(1)$	$O(n)$
Array ordinato	$O(logn)$	$O(n)$	$O(n)$
Lista non ordinata	$O(n)$	$O(1)$	$O(n) \leftrightarrow O(1)$
Lista ordinata	$O(n)$	$O(n)$	$O(n) \leftrightarrow O(1)$

Tabella 1: Costo in tempo su array e liste

Tabelle ad indirizzamento diretto

Idea: usare la chiave degli elementi come posizione negli array.

$U \subseteq N \longrightarrow T = |U|$.

$\forall x \in S \rightarrow$ viene memorizzato in $T[x.key]$.

Data un qualsiasi chiave $k \in U$:

$$T[k] = \begin{cases} x & \text{se S contiene un elemento } x \text{ t.c. } x.key = k \\ nil & \text{se S non contiene un elemento } x \text{ t.c. } x.key = k \end{cases}$$

Search

```
func search (T,k) {
  return T[k];
}
```

Costo in tempo

$\Theta(1)$.

Insert

```
func insert (T,x) {
  T[x.key]=k;
}
```

Costo in tempo

$\Theta(1)$.

Delete

```
func delete (T,x) {
  T[x.key]=nil;
}
```

Costo in tempo

$\Theta(1)$.

In sintesi il costo in tempo è sempre $\Theta(1)$ quindi costante, ma il *costo in spazio* non va bene perchè è $\Theta(|U|)$.

9 Tabelle hash

Come abbiamo visto le operazioni di Search, Insert e Delete hanno un costo in tempo costante ma un costo in spazio non coveniente ($\Theta(|U|)$). Per ovviare a questo problema possiamo utilizzare le *tabelle hash* che hanno come obbiettivo principale quello di ridurre l'occupazione in spazio da $\Theta(|U|)$ a $\Theta(|S|)$ senza pregiudicare il costo in tempo e usando le chiavi per calcolare la posizione degli elementi nella tabella.
Con $h : U \rightarrow [0, 1, ..., m-1]$ indicheremo la cosiddetta *funzione hash* dove $[0, 1, ..., m-1]$ sono tutte le posizioni di una tabella di dimensione m ed $m \simeq O(|S|)$.
$\forall k \in j$:

- $h(k) \in [0, m-1]$.
- $x, x.key = k$.
- $T[h(x.key)] = x$.

$$T[j] = \begin{cases} x & \text{se } x \in S \text{ e } h(x.key) = j \\ nil & \text{se } x \notin S \text{ quindi } h(x.key) \neq j \end{cases}$$

Problema delle collisioni

1. h non può essere *iniettiva* perchè il suo dominio è molto più grande del suo codominio, si verifica il cosiddetto *"problema delle collisioni"* ($|U| >> n$) e avviene una **collisione** dove avremo $k_1, k_2 \in U$ con $k_1 \neq k_2$ ma $h(k_1) = h(k_2)$.
2. h deve essere *suriettiva* (per usare tutte le posizioni di T).
3. h deve essere facile da calcolare (costo in tempo costante $O(1)$).
4. h deve distribuire uniformemente gli elementi di S nella tabella.
   - $\forall k \in U$: $h(k)$ deve assumere valore i, con $i \in [0, m-1]$ ed una probabilità pari a $\frac{1}{m}$.
5. h deve essere deterministica, dato k, $h(k)$ deve produrre sempre lo stesso valore.

Hashing "perfetto"

Se h è iniettiva sulle chiavi degli elementi di S, nessuna coppia genere *collisioni.*
$\forall x_1,\ x_2 \in S \rightarrow h(x_1.key) \neq h(x_2.key)$.
Dato che h è iniettiva si può procedere in seguito ad un inserimento o ad una cancellazione ed è possibile ripristinare in modo un pò complicato.

Osservazione

h non preserva l'ordine delle chiavi, infatti:

$$k_1 < k_2 \quad \text{non implica che} \quad h(k_1) < h(k_2)$$

Le tabelle hash vanno bene principalmente per dizionari non ordinati.
Vediamo adesso aclcuni metodi per il calcolo delle funzioni hash.

Metodo della divisione

Dato $U \subseteq N$, $k \in U$ e $m =$ dimensione di T.

$$h(k) = k \ \% \ m$$

Dove m è un numero primo non troppo vicino ad una potenza di 2 (o di 10).

Randomizzare le chiavi

In questo caso sceglieremo p (numero primo) molto più grande di m.

$$h(k) = (k \ \% \ p) \ \% \ m$$

N.B. m non è più "dannoso", possiamo scegliere anche $m = 2^t$ o $m = 10^t$.

Tabelle hash con concatenamento (Liste di trabocco)

Qui T: array di liste doppie.

$$T[j] = \begin{cases} head & \text{se } h(head.key) = j \text{ (vedi fondo pagina)}^2 \\ nil & \text{se } \forall x \in S,\ h(x.key) \neq j \end{cases}$$

Operazioni di dizionario

```
func search (T,k) {
  return listSearch (T[h(k)], k);
}
```

Costo in tempo

$O(n)$.

```
func insert (T,x) {
  // inserisci x in testa alla lista T[h(x.key)];
}
```

Costo in tempo

$O(1$.

```
func delete (T,x) {
  // rimuovi x dalla lista T[h(x.key)];
}
```

Costo in tempo

$O(1)$.

[2]Riferimento alla testa di una lista doppia che contiene tutti gli elementi di S la cui chiave ha valore hash pari a j.

Ipotesi di hashing uniforme semplice (HUS)

$\forall k \in U \rightarrow h(k)$ assume valore i, $0 \leq i < m$ con probabilità pari a $\frac{1}{m}$.

Teorema

In una tabella hash con concatenamento la ricerca senza successo richiede tempo $\Theta(1+\alpha)$ con $\alpha = \frac{n}{m}$ (fattore di carico), al caso medio, nell'ipotesi di hashing uniforme semplice (HUS).
Se α è costante allora $T(n) = O(1)$.

Dimostrazione

Dato che ci troviamo nel caso di una ricerca senza successo, k non è presente, quindi bisogna scorrere tutta la lista T[h(k)].

$$k \rightarrow h(k) \rightarrow \text{ricerca di } k \text{ nella lista } T[h(k)]$$

Ne segue che, sempre nell'ipotesi di HUS, ogni lista in media contiene $\alpha = \frac{n}{m}$ elementi, quindi faccio α accessi alla lista e $T_{medio}(n,m) = \Theta(1+\alpha)$ dove "1" è il costo in tempo che impiego per calcolare $h(k)$ e "α" rappresenta il numero medio di accessi alla lista $T[h(k)]$.
Se α è costante anche $T_{medio}(n,m) = O(1)$, nella pratica $\alpha \simeq \frac{1}{2}$.

Operazione	**Elementi esaminati**	**Costo in tempo**
Ricerca senza successo	α	$\Theta(1+\alpha)$
Ricerca con successo	$1 + \frac{\alpha}{2} - \frac{\alpha}{2n}$	$\Theta(1+\alpha)$

Tabella 2: Costo in tempo al caso medio su Tabelle hash

Teorema

In una tabella hash con concatenamento la ricerca con successo richiede, al caso medio, $1 + \frac{\alpha}{2} - \frac{\alpha}{2n}$ ispezioni nell'ipotesi di HUS.

Dimostrazione

Ipotizziamo che x, elemento di chiave k, sia l'i-esimo elemento inserito in T, quindi $n-i$ elementi sono stati inseriti dopo x e si sono distribuiti uniformemente nelle m liste.
L'ipotesi di HUS ci permette di concludere che di questi $n-i$ elementi, soltanto $\frac{n-i}{m}$ elementi hanno lo stesso valore hash $h(k)$ e sono finiti nella stessa lista (in testa) che contiene x.
Il numero medio di ispezioni sarà $1 + \frac{n-i}{m}$ dove "1" rappresenta il costo in tempo per accedere ad x e "$\frac{n-i}{m}$" è il numero di elementi che precedono x nella lista $T[h(k)]$. Occore però fare la media su tutti i valori di i, assumendo di cercare con pari probabilità tutte le chiavi presenti nel dizionario.

Idea: costo al caso medio $\rightarrow \frac{1}{n} \cdot \sum_{i=1}^{n} (1 + \frac{n-i}{m})$.

Otteniamo questo:

$$= \frac{1}{n} \cdot \sum_{i=1}^{n}(1 + \frac{n-i}{m}) = \frac{1}{n} \cdot \sum_{i=1}^{n} 1 + \frac{1}{n} \cdot \sum_{i=1}^{n} \frac{n-i}{m} = \frac{1}{n} \cdot n + \frac{1}{n \cdot m} \cdot \sum_{i=1}^{n}(n-i) =$$

$$1 + \frac{1}{n \cdot m} \cdot [(n-1) + (n-2) + (n-3) + ... + 2 + 1 + 0] = 1 + \frac{1}{n \cdot m} \cdot \sum_{j=0}^{n-1} j =$$

$$1 + \frac{1}{n \cdot m} \cdot \frac{n(n-1)}{2} = 1 + \frac{n}{m \cdot 2} - \frac{1}{2n} = 1 + \frac{\alpha}{2} - \frac{\alpha}{2n}.$$

Tabelle hash con indirizzamento aperto (Open hash)

Idea: $\forall x \in S \rightarrow$ memorizzo x in $T[h(x.key)]$ se la cella è libera, se è occupata si cerca una posizione alternativa nella tabella seguendo un ordine (che dipende dalla chiave) per esaminare le celle.
In questa situazione la *funzione hash* $h : U \times [0, m-1] \rightarrow [0, m-1]$ opera in base alla seguente proprietà:

$\forall k \in U :$ si associa una *sequenza di ispezione, scansione o probing.*

Dove $< h(k,0), h(k,1), h(k,2), ..., h(k,m-1) >$ rappresenta l'ordine con cui si esaminano le posizioni della tabella durante le operazioni di dizionario relative all'elemento di chiave k e deve essere una permutazione di {0,1,2,...,m-1}.

Operazioni di dizionario

Sono le tre operazioni principali di dizionario che abbiamo già visto.

Inserimento

```
INSERT (T,k)
  i=0;
  repeat {
    j=h(k,i);
    if (T[j]==nil) {
      T[j]=k;
      return j;
    }
    else i++;
  }
  until i==n;
  error "overflow della tabella";
```

Costo in tempo
$T(n,m) = O(1)$ (caso ottimo).
$T(n,m) = \Theta(n) = O(m)$ (caso medio).
$T(n,m) = O(\frac{1}{1-\alpha})$ (caso pessimo).

Ricerca

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

Costo in tempo
$T(n,m) = O(1)$ (caso ottimo).
$T(n,m) = \frac{1}{1-\alpha}$ (caso medio, ricerca senza successo).
$T(n,m) = \frac{1}{\alpha} \cdot ln \cdot \frac{1}{1-\alpha}$ (caso medio, ricerca con successo).
$T(n,m) = \Theta(n)$ (caso pessimo).

Cancellazione

È virtuale/logica altrimenti sarebbe impossibile ritrovare una chiave k inserita quando la cella che si vuole cancellare era già occupata.
Se volessimo riusare la SEARCH rimarrebbe inalterata dato che ignora i valori con la marcatura *deleted* e si arresta solo se trova la chiave o nil. Se invece volessimo riusare INSERT dopo aver fatto delle cancellazioni il codice cambierebbe leggermente.

```
INSERT (T,k)
  i=0;
  repeat {
    j=h(k,i);
    if (T[j]==nil or T[j]=="deleted") {
      T[j]=k;
      return j;
    }
    else i++;
  }
  until i==n;
  error "overflow della tabella";
```

Osservazione

La presenza di molte celle marcate *deleted* provoca un degrado delle prestazioni dell'operazione di ricerca.

Analisi al caso medio

Abbiamo tre ipotesi:

1. $\alpha = \frac{n}{m} < 1$.
2. Non ci sono cancellazioni.
3. Hashing uniforme (HU).
   - La sequenza di ispezioni di ogni chiave è una qualsiasi permutazione degli interi {0,1,2,...,m-1} con pari probabilità.

Teorema

Nell'ipotesi di HU, data una tabella hash ad indirizzamento aperto con fattore di carico $\alpha < 1$, il numero di ispezioni al caso medio in una ricerca senza successo è al massimo $\frac{1}{1-\alpha}$.

Dimostrazione

Dato x = numero di accessi alla tabella, E[x] si dice *valore atteso di* x ed è uguale a $\sum_{i=1}^{+\infty} i \cdot Prob[x = i] = \sum_{i=1}^{+\infty} i \cdot Prob[x \geq i]$.

α	**Occupazione in (%)**	**E[x]** ($\leq \frac{1}{1-\alpha}$)
$\frac{1}{2}$	50%	2
$\frac{1}{10}$	10%	1.11
$\frac{9}{10}$	90%	10

Tabella 3: Ricerca senza successo Open hash

L'inserimento equivale alla ricerca senza successo dato che entrambi "toccano" le stesse celle e si arresta sulla prima cella vuota in cui inserisce il nuovo elemento.
Gli accessi al caso medio sono $\leq \frac{1}{1-\alpha}$.
Nel caso in cui avessimo una ricerca con successo le cose cambierebbero.

Teorema

In una tabella hash ad indirizzamento aperto, nell'ipotesi di HU, una ricerca con successo richiede al più $\frac{1}{\alpha} \cdot ln \cdot \frac{1}{1-\alpha}$ accessi in media.

Dimostrazione

La ricerca con successo segue la stessa sequenza di ispezione seguita quando l'elemento di chiave k (k è la chiave cercata) era stato inserito. Supponiamo che la chiave cercata sia relativa all'$(i+1) - esimo$ elemento inserito in T.
Quando l'elemento di chiave k è stato inserito, T conteneva i elementi ed il *fattore di carico* era $\frac{i}{m}$.

Segue che il numero di accessi a T nella ricerca di k al caso medio è uguale al numero di accessi fatti quando l'elemento di chiave k è stato inserito ed il fattore di carico era $\frac{i}{m}$, quindi $\leq \frac{1}{1-\frac{i}{m}}$.
Scritto in formula:

$$E[x] \leq \tfrac{1}{n} \cdot \sum_{i=0}^{n-1} \frac{1}{1-\frac{i}{m}} = \frac{1}{n} \cdot \sum_{i=0}^{n-1} \frac{m}{m-i} = \frac{m}{n} \cdot \sum_{i=0}^{n-1} \frac{1}{m-i} = \frac{1}{\alpha} \cdot [\frac{1}{m} + \frac{1}{m-1} + \frac{1}{m-2} + ... + \frac{1}{m-(n-1)}] = \frac{1}{\alpha} \cdot \sum_{k=m-(n-1)}^{m} \frac{1}{k} \leq \frac{1}{\alpha} \cdot \int_{m-n}^{m} \frac{1}{x} dx = \frac{1}{\alpha} \cdot (ln(m) - ln(m-n)) = \frac{1}{\alpha} \cdot ln \cdot \frac{m}{m-n} = \frac{1}{\alpha} \cdot ln \cdot \frac{1}{1-\frac{n}{m}} = \frac{1}{\alpha} \cdot ln \cdot \frac{1}{1-\alpha}$$

α	**Occupazione in (%)**	**# ispezioni RSS**	**# ispezioni RCS**
		$\frac{1}{1-\alpha}$	$\frac{1}{\alpha} \cdot ln \cdot \frac{1}{1-\alpha}$
$\frac{1}{10}$	10%	≤ 1.11	≤ 1.05
$\frac{1}{2}$	50%	≤ 2	≤ 1.38
$\frac{9}{10}$	90%	≤ 10	≤ 2.55

Tabella 4: Ricerca con e senza successo Open hash

Calcolo della sequenza di ispezione

Ispezione lineare

Per fare un' ispezione lineare abbiamo bisogno della nostra nota *funzione hash* $h(k,i) : U \times [0, m-1] \to [0, m-1]$ e di una *funzione ausiliaria* $h' : U \to [0, m-1]$ che definisce $h(k,i) = (h'(k) + i)\%m$.
La sequenza di ispezione sarà $< h'(k)\%m, (h'(k)+1)\%m, ..., (h'(k)+m-1)\%m >$ ed è una permutazione delle m posizioni della tabella (m di solito è un numero primo).
Il punto di partenza dell'ispezione lineare è sempre "casuale" poichè gestito dalla *funzione hash ausiliaria* $h'(k)$. Il passo di scansione è costante (+1) e non dipende nè da k nè da i.
Il numero di sequenze diverse generate sarà m che è molto minore di $m!$.
Un problema delle ispezioni lineari è la formazione di agglomerati primari, ovvero lunghi tratti di celle adiacenti occupate che causano un degrado delle prestazioni in quanto devono essere attraversati interamente prima di trovare una cella libera.

Ispezione quadratica

In questo caso scegliamo (di solito) m numero primo, $c_1,\ c_2$ costanti con $c_2 \neq 0$.

Grazie alla *funzione hash ausiliaria* $h' : U \times [0, m-1]$ otteniamo la seguente *funzione hash*:

$$h(k,i) = (h'(k) + c_1 \cdot i + c_2 \cdot i^2)\%m$$

Il punto di partenza di un'ispezione quadratica è "casuale" dato che dipende da $h'(k)$ mentre il punto di scansione non è più costante ma dipende da i.
Inoltre $c_1,\ c_2$ ed m devono essere scelti in modo tale che la sequenza ottenuta sia una permutazione, anche in questo caso il numero delle sequenze diverse generate è di nuovo m, infatti la sequenza dipende solo dal punto di partenza (che pu essere uno degli m interi in $[0, m-1]$).

Doppio hash

Offre prestazioni confrontabili con quelle teoriche del caso di HU.
In questo caso abbiamo due *funzioni hash ausiliarie* $h_1,\ ,h_2$ e definiamo la nostra funzione, che determina la sequenza di ispezione, in questo modo:

$$h(k,i) = (h_1(k) + i \cdot h_2(k))\%m$$

Il punto di partenza è "casuale" e dipende da k tramite $h_1(k)$ e il passo dipende da k tramite $h_2(k)$.
La sequenza sarà $< h_1(k)\%m, (h_1(k) + h_2(k))\%m, ..., (h_1(k) + i \cdot h_2(k))\%m >$.
Per generare effettivamente una permutazione di $[0, m-1]$, quindi toccare tutte quante le posizioni, la richiesta che facciamo è che $MCD(h_2(k), m) = 1$. Per garantire che $MCD(h_2(k), m) = 1$ abbiamo due modi per farlo:

1. Prendiamo m potenza di 2 e $h_2(k)$ assume solo valori dispari.
2. Prendiamo m numero primo e $h_2(k)$ è sempre positiva ma $< m$.

Nel caso (2) spesso si usa il metodo della divisione scegliendo $h_1(k) = k\%m$ e $h_2(k) = 1 + k\%m'$ con $m' < m$ (di solito $m' = m-1$), quindi possiamo ridefinire la nostra *funzione hash* in questo modo:

$$h(k,i) = (k\%m + i \cdot (1 + k\%(m-1)))\%m$$

Il numero "1" sommato a $k\%(m-1)$... garantisce che il passo non sia nullo, infatti ha valori compresi tra 1 e $m-1$.
Tra le particolarità abbiamo che non ci sono *agglomerati primari* o *secondari* e chiavi diverse hanno la stessa sequenza di ispezione soltanto se abbiamo una *doppia collisione*, cioè solo se $k_1,\ k_2$ sono tali che $h_1(k_1) = h_1(k_2)$ e allo stesso tempo $h_2(k_1) = h_2(k_2)$.
Segue che il numero di sequenze di ispezione diverse generate è uguale al numero di coppie diverse $(h_1(k), h_2(k))$ che è uguale a $\binom{n}{2} = \Theta(n^2)$. Quindi abbiamo $\Theta(n^2)$ sequenze (sempre molto minore di $m!$).

10 Alberi binari

Un albero binario è un insieme finito di nodi. L'insieme può essere vuoto (in tal caso diciamo che l'albero è vuoto) oppure non vuoto. Un nodo di un albero binario si dice *nodo esterno* o, più semplicemente, **foglia** se non ha figli (entrambi i sottoalberi di cui è radice sono vuoti), si dice, invece, *interno* se ha almeno un figlio. In un albero binario, la profondità di un nodo è la lunghezza di un cammino dalla radice al nodo (cioè il numero di archi tra la radice ed il nodo). La **profondità** massima di un nodo all'interno di un albero è detta **altezza** dell'albero.
Un albero non vuoto deve soddisfare le seguenti proprietà:

- Contenere un nodo speciale detto *radice*.
- Ad ogni nodo possiamo associare (al più) due figli detti rispettivamente figlio sinistro e figlio destro. Se un nodo n è figlio di un nodo m, allora diciamo che m è padre di n.
- Ogni nodo, tranne la radice, ha esattamente un padre. La radice è l'unico nodo che non ha un padre.

È importante sottolineare che con n indicheremo un qualunque nodo di un albero, con $n.sx$ indicheremo il figlio sinistro di un nodo n, con $n.dx$ indicheremo il figlio destro di un nodo n e infine con $n.key$ indicheremo il valore (o etichetta) di un nodo n. Tra le varie operazioni che si possono fare su un *albero binario* la più comune è sicuramente la cosiddetta *visita*, esistono tre tipi di visita.

Visita anticipata

```
VISITA–ANT (bt)
  if (bt!=nil) then {
    print (bt.key);
    VISITA–ANT (bt.sx);
    VISITA–ANT (bt.dx);
  }
```

Visita simmetrica

```
VISITA–SIM (bt)
  if (bt!=nil) then {
    VISITA–SIM (bt.sx);
    print (bt.key);
    VISITA–SIM (bt.dx);
  }
```

Visita posticipata

```
VISITA–POST (bt)
  if (bt!=nil) then {
    VISITA–SIM (bt.sx);
    VISITA–SIM (bt.dx);
    print (bt.key);
  }
```

La complessità in tutti e tre i casi è di $\Theta(n)$.

Contafoglie

```
CONTAFOGLIE (bt)
  if (bt==nil) then return 0;
  else {
    if (bt.sx==nil && bt.dx==nil) then return 1;
    else return CONTAFOGLIE(bt.sx) + CONTAFOGLIE(bt.dx);
  }
```

Costo in tempo
$T(n) = \Theta(1)$.

Vediamo adesso un altro modo di rappresentare gli alberi binari.
Con n indicheremo il numero di nodi di un albero, con $m = n - 1$ il numero di archi e con r la radice. A sua volta la radice avrà tre campi fondamentali, $r.key$ rappresenta il valore della radice, $r.left$ il sottoalbero sinistro e $r.right$ il sottoalbero destro. La dimensione di un albero binario è uguale a n (numero di nodi), scritto formalmente:

$$dim(bt) = \begin{cases} dim(\lambda) = 0 \\ dim(bt) = 1 + dim(left) + dim(right) \end{cases}$$

Dimensione

```
DIM (u)
  if (u==nil) return 0;
  else {
    dims=DIM(u.left);
    dimd=DIM(u.right);
    return 1+dims+dimd;
  }
```

Costo in tempo
$T(n) = \Theta(n)$.

Altezza

```
ALTEZZA (u)
  if (u==nil) return -1;
  else {
    altsx=ALTEZZA(u.left);
    altdx=ALTEZZA(u.right);
    return 1+max{altsx,altdx};
  }
```

Costo in tempo
$T(n) = \Theta(n)$.

Definizione

Si dice *valore decomponibile* quel valore da calcolare, per un albero binario, come combinazione dei valori dei suoi sottoalberi.

Valore decomponibile

```
DECOMP (u)
  if (u==nil) return DECOMP (0);
  else {
    sx=DECOMP(u.left);
    dx=DECOMP(u.right);
    return RICOMBINA(sx,dx);
  }
```

Costo in tempo
$T(n) = \Theta(n)$ (DECOMP).
$T(n) = \Theta(1)$ (RICOMBINA).

Albero binario completo

Un albero binario si dice *completo* se ogni nodo ha esattamente due figli ($\neq nil$), eccetto le foglie. Un albero binario è un albero binario *completamente bilanciato* (ABCB) se, oltre ad essere completo, a tutte le foglie alla stessa profondità. Avremo che, se n è il numero di nodi e h è l'altezza, in un ABCB ho $2^h - 1$ nodi interni, 2^h foglie, $2^{h+1} - 1$ nodi e $h = log(n+1) - 1$.
Un ABCB è tale che $h = \Theta(logn)$ dove n è il numero di nodi. Un albero è *bilanciato* $\Leftrightarrow h = \Theta(logn)$, se un albero è un ABCB allora questo sarà bilanciato (il viceversa non vale). Se (u.left) è bilanciato e (u.right) è bilanciato allora tutto u sarà bilanciato.

Nodo cardine

Preso un qualunque nodo u, indichiamo con P_u la sua profondità e con h_u la sua altezza. Esso si dice *nodo cardine* $\Leftrightarrow P_u = h_u$.

Nodo centrale

Un nodo di un albero bilanciato si dice *centrale* se la dimensione del sottoalbero di cui è radice è pari alla somma delle chiavi dei nodi che appartengono al percorso dalla radice al nodo stesso.

Nodi centrali

```
CENTRALI (u,somma)
  if (u==nil) then return 0;
  else {
    dims=CENTRALI (u.left ,somma+u.key);
    dimd=CENTRALI (u.right ,somma+u.key);
    dim=dims+dimd+1;
    if (dim==somma+u.key) then printf (u.key);
    return dim;
  }
```

11 Alberi binari di ricerca

Un albero binario di ricerca (ABR), in informatica, è un particolare tipo di struttura dati. Permette di effettuare in maniera efficiente operazioni come: ricerca, inserimento e cancellazione di elementi. Intuitivamente, un albero binario di ricerca ha le seguenti proprietà:

- Il sottoalbero sinistro di un nodo x contiene soltanto i nodi con chiavi minori della chiave del nodo x.
- Il sottoalbero destro di un nodo x contiene soltanto i nodi con chiavi maggiori della chiave del nodo x.
- Il sottoalbero destro e il sottoalbero sinistro devono essere entrambi due alberi binari di ricerca.

Tra le più comuni operazioni su ABR vedremo la ricerca di una chiave k o del minimo e del massimo, il successore e il predecessore, l'inserimento e la cancellazione.

Ricerca

```
ABRSEARCH (x,k) {
  if (x==nil || x.key==k) return x;
  if (x.key>k) then {
    return ABRSEARCH (x.left ,k);
  else return ABRSEARCH (x.right ,k);
  }
}
```

Costo in tempo
$T(n) = \Theta(1)$ (caso ottimo).
$T(n) = \Theta(logn)$ (caso medio).
$T(n) = O(h)$ (caso pessimo).

Minimo

```
ABRMIN (x) {
  if (x==nil) return nil;
  while (x.left!=nil) x=x.left;
  return x;
}
```

Costo in tempo
$T(n) = O(h)$.

Successore

```
ABRSUCCESSOR (x) {
  if (x==nil) return nil;
  if (x.right!=nil) return ABRMIN (x.right);
  y=x.p;
  while (y!=nil && x==y.right) {
    x=y;
    y=y.p;
  }
  return y;
}
```

Costo in tempo
$T(n) = O(h)$.

Inserimento

Per capire al meglio come funziona l'inserimento in un ABR ecco alcuni semplici passi da seguire per non commettere errori:

- Si inserisce la nuova chiave nelle foglie ma non in un posto *qualsiasi*.
- Si simula una ricerca della chiave stessa e quando si trova *nil* si attacca il nuovo nodo.
- Si tiene il *puntatore* "inseguitore" per settare il padre del nuovo nodo.

```
ABRINSERT (T,z) {
  y=nil; x=T.root;
  while (x!=nil) {
    y=x;
    if (z.key<x.key) x=x.left;
    else x=x.right;
  }
  z.p=y;
  if (y==nil) T.root=z;
  else if (z.key<y.key) y.left=z;
  else y.right=z;
}
```

Costo in tempo
$T(n) = O(h)$.

Cancellazione

Esistono tre casi diversi.

1. z non ha figli. (molto facile)
2. z ha un figlio. (facile)
3. z ha due figli. (difficile)

Anche per rimuovere un elemento da un ABR ci sono delle regole ben precise da seguire. Di seguito sono elencate in modo sintetico.

- Trovo il successore di z che ha al massimo un figlio (dato che siamo in *z.right* e il suo figlio sinistro sarebbe più piccolo).
- Metto y al posto di z ed elimino il vecchio nodo di y.

```
ABRDELETE (T,z) {
  if (z.left==nil && z.right==nil) y=z;
  else y=ABRSUCCESSOR (z);
  if (y.left!=nil) x=y.left;
  else x=y.right;
  if (x!=nil) x.p=y.p;
  if (y.p==nil) T.root=x;
  else {
    if (y==y.p.left) y.p.left=x;
    else y.p.right=x;
  }
  if (y!=z) z.key=y.key;
  return y;
}
```

Costo in tempo
$T(n) = O(h)$.

12 Alberi 2-3

Un albero 2-3 è un albero in cui ogni nodo interno ha due o 3 figli e tutti i cammini dalla radice ad una foglia hanno la stessa lunghezza (tutte le foglie stanno allo stesso livello). Le chiavi sono memorizzate nelle foglie in ordine crescente se conto da sinistra verso destra.
Indicheremo l'altezza con h ($h \in \Theta(logn)$), il numero dei nodi con n ed il numero delle foglie con f.

Lemma

Dato T (albero 2-3) con n nodi, f foglie e altezza h, abbiamo le seguenti proprietà:

1. $2^{h+1} - 1 \leq n \leq \frac{3^{h+1}-1}{2}$
2. $2^h \leq f \leq 3^h$

In entrambi i casi, tutto quello che si trova alla destra del secondo $\leq$ rappresenta il *caso estremo di albero ternario completamente bilanciato*. Allo stesso modo, tutto quello che si trova alla sinistra del primo $\leq$ rappresenta il *caso estremo di albero binario completamente bilanciato*.

Dimostrazione

Preso un albero T' di altezza h, n' nodi e f' foglie, andiamo ad aggiungere un altro livello (tutti figli) e un'altezza $h+1$. La dimostrazione avviene per *induzione*.

Ipotesi induttiva: (1) e (2) valide fino ad altezza h.
Caso base: $h = 0$, $n = 1$, $f = 1$ (T ha solo la radice).

$$1 = 2^{0+1} - 1 \leq 1 \leq \frac{3^{0+1} - 1}{2} = \frac{2}{2} = 1 \rightarrow (1) \textit{ verificata}$$

$$1 = 2^0 \leq 1 \leq 3^0 = 1 \rightarrow (2) \textit{ verificata}$$

Passo induttivo: $h \rightarrow h+1$, ogni foglia di T' ha 2 o 3 figli.

Segue che $f \leq 3f' \leq 3 \cdot 3^h = 3^{h+1}$ e che $f \geq 2f' \geq 2 \cdot 2^h = 2^{h+1}$, quindi $2^{h+1} \leq f \leq 3^{h+1}$ e questo verifica la (2).

Inoltre $n = n' + f \leq \frac{3^{h+1}-1}{2} + 3^{h+1} = 3^{h+1} \cdot (\frac{1}{2} + 1) - \frac{1}{2} = \frac{3 \cdot 3^{h+1}}{2} - 1 = \frac{3^{h+2}-1}{2}$ e $n = n' + f \geq 2^{h+1} - 1 + 2^{h+1} = 2^{h+1} \cdot 2 - 1 = 2^{h+2} - 1$, quindi concludiamo dicendo che $2^{h+2} - 1 \leq n \leq \frac{3^{h+2}-1}{2}$ e questo verifica la (1).

Corollario

Dato T (albero 2-3) con n nodi, la sua altezza sarà $h \in O(logn)$.

Dimostrazione

In base a quanto abbiamo dimostrato nel Lemma precedente, sappiamo che $2^{h+1} - 1 \leq n \leq \frac{3^{h+1}-1}{2}$.
Prendendo solo $2^{h+1} - 1 \leq n$ e aggiungendo 1 ad entrambi i membri per poi farne il logaritmo otterremo $h \leq log(n+1) + 1$ che $\in O(logn)$, prendendo $n \leq \frac{3^{h+1}-1}{2}$ e aggiungendo 1 e moltiplicando per 2 entrambi i membri per poi farne il logaritmo otterremo $2n + 1 \leq 3^{h+1} = log_3(2n+1) \leq h$ che $\in O(logn)$.
Abbiamo dimostrato che $h \in O(logn)$.

Ricerca

```
SEARCH (v,k) {
  if ((v.left==nil) && (v.right==nil)) {
    if (v.key==k) return v;
    else return nil;
  }
  if (k<=v.s) return SEARCH (v1,k);
  else {
    if (v ha due figli || k<=v.m) return SEARCH (v2,k);
    else return SEARCH (v3,k);
  }
}
```

Costo in tempo
$T(n) = \Theta(h) = \Theta(logn)$.

Inserimento

Il procedimento per inserire un nuovo nodo all'interno di un albero 2-3 deve seguire questi passaggi:

- Creo un nodo u con dati e chiave k.
- Si localizza la sua corretta posizione come nuova foglia tra le "vecchie" foglie facendo una *search* di k.
- Si identifica quindi il nodo v che dovrebbe diventare il padre di u.

A questo punto ci troviamo davanti a due casi:

1. v ha due figli ($v.l$, $v.r$): aggiungo u come terzo figlio, confrontando k con $v.s$ e lo colloco correttamente tra $v'.l$ e $v'.r$ e aggiorno $v.m$.
2. v ha tre figli: aggiungo u come quarto figlio, se il padre di v aveva due figli ho finito (perchè devo fare lo *split* dei due figli del padre di v sempre rispettando le proprietà degli alberi 2-3), altrimenti faccio lo *split* anche del padre di v (al massimo fino alla radice).

Complessità di Insert

- Creo un nodo u con dati e chiave k. $\longrightarrow \Theta(1)$
- Lo aggiungo come nuova foglia $\longrightarrow \Theta(1)$, dopo aver trovato la sua collocazione. $\longrightarrow \Theta(logn)$
- Se è il terzo figlio aggiorno $v.s$ e $v.m$ e termino. $\longrightarrow \Theta(1)$
- Se è il quarto figlio faccio lo *split* $\longrightarrow O(h)$, se il padre ha due figli faccio lo *split* dei figli e così via al massimo fino alla radice. $\longrightarrow O(logn)$

```
SPLIT (v) {
  // w: nuovo nodo con due figli;
  // vi: i-esimo figlio di v con 1<=i<=4;
  // <v1 e v2 figli di w> e <v3 e v4 figli di v>;
  // aggiorno w.s e v.s;
  w.key=v.key; w.s=max(v1);
  v.s=v3.key; v.s=max(v3);
  if (v.p==nil) {creo un nuovo nodo radice r con
    solo due figli w e v e aggiorno r.s;}
  else {
    aggiungo w come figlio di v.p
    immediatamente precedente a v;
    if (v.p ha quattro figli) SPLIT (v.p);
  }
}
```

Costo in tempo
$T(n) = O(h) \in \Theta(logn)$.

Cancellazione

La DELETE (v) ha principalmente tre casi:

1. v è la radice: la rimozione di $v \Rightarrow T = 0$.
2. $v.p$ ha tre figli: rimuovo v, aggiorno $v.s$ e rimuovo $v.m$.
3. $v.p$ ha due figli
   - v.p è la radice: $v.suopadre$ è la nuova radice.
   - v.p non è la radice: uso la *fuse*.

13 Grafi

Il termine *grafo* in informatica viene usato per indicare una particolare figura geometrica, costituita da un insieme finito di punti, detti **nodi** (V) e da **archi** (E), che congiungono coppie di nodi.
Formalmente vengono indicati con $G = (V, E)$ dove V è l'insieme dei nodi e $E \subseteq V \times V$ è l'insieme degli archi, useremo $n = |V|$ per indicare il numero dei nodi (ordine del *grafo*) e $m = |E|$ per indicare il numero degli archi.

Esistono due tipi principali di grafi:

- Orientati: dove E è un insime di coppie ordinate.
- Non orientati: dove E è un insieme di coppie non ordinate.

Dimensione del Grafo

La dimensione di un grafo è $|V| + |E| = n + m$ dove $0 \leq n \leq \binom{n}{2} = \frac{n(n-1)}{2}$, quando $m = O(n)$ allora si tratta di un *grafo sparso* mentre se $m = \Theta(n^2)$ allora avremo un *grafo denso*.

Grafi non orientati

Dati $u, v \in V$, l'arco $(u,v) \in E$ connette i nodi u e v (che si dicono adiacenti) e si dice incidente su u e v.

$\forall v \in V$:

- Il **grado** di v è $\delta(v)$ = numero di archi incidenti su v.
- v è un **vertice isolato** se $\delta(v) = 0$

$\sum_{v \in V} \delta(v) = 2|E| \longrightarrow$ Ogni arco incrementa di un fattore 1 il grado dei suoi estremi e contribuisce per un fattore 2 alla somma.

Un **cammino** $u \to v$ in un grafo inizia con un nodo u e termina in un nodo v, chiamati gli estremi del cammino. In particolare corrisponde ad un' alternanza di nodi e archi, nell'ordine in cui vengono incontrati, e viene rappresentato dalla sequenza di nodi adiacenti $x_0, x_1, x_2, ..., x_{k-1}, x_k$ tale che $x_0 = u$ e $x_k = v$.

$\forall i$ con $1 \leq i \leq k$, $(x_{i-1}, x_i) \in E$ dove k è la lunghezza del cammino (ovvero il numero di archi che lo compongono).

Un cammino si dice **cammino semplice** se passa per ogni suo nodo e arco una sola volta ed è composto da soli nodi distinti.
Un cammino semplice si dice **ciclo** se è *chiuso* (ovvero i suoi estremi sono uguali, quindi $u = v$).
La *distanza* $\delta(u, v)$ è il numero minimo di archi da dover percorrere per spostarsi da u a v.
Un grafo si dice **completo** (Clique) se ogni coppia di nodi distinti è adiacente. Formalmente: $\forall u, v \in V,\ u \neq v,\ (u, v) \in E$.
Dati $u, v \in V$ u e v si dicono *connessi* se esiste un cammino da u a v tale che $\delta(u, v) < +\infty$. G si dice **connesso** se ogni coppia di vertici è connessa.
Dato $G = (V, E)$, $G' = (V', E')$ si dice **sottografo** di G se $V' \subseteq V$, $E' \subseteq V' \times V'$ e $E' \subseteq E$.
Un sottografo G' di G si dice **massimale** quando non può essere ulteriormente esteso (ovvero non esistono altri nodi di G connessi ai nodi di G'). Le *componenti connesse* di un grafo sono le classi di equivalenza della relazione "*è raggiungibile da...*".

Grafi orientati

In questo caso avremo che la *direzione* di un arco che va da un nodo ad un altro conta, infatti nel caso dei grafi orientati scrivere (u, v) o (v, u) non è la stessa cosa. Nel primo caso stiamo indicando quell'arco che esce dal nodo u ed entra nel nodo v, nel secondo caso esattamente l'opposto, quindi $(u, v) \neq (v, u)$.

$\forall v \in V$:

- Si dice grado uscente di v ($\delta_u(v)$) il numero di archi che escono da v.
- Si dice grado entrante di v ($\delta_e(v)$) il numero di archi che entrano in v.

Il grado di un nodo v si indica con $\delta(v) = \delta_u(v) + \delta_e(v)$, mentre la sommatoria dei gradi entrati e uscenti di un nodo v è pari alla cardinalità di E.
Formalmente:

$$\sum_{v \in V} \delta_e(v) = \sum_{v \in V} \delta_u(v) = |E|$$

Ogni arco incrementa il grado di un solo estremo e contribuisce alla somma per un fattore.
Per i grafi orientati valgono le stesse definizioni di cammino e ciclo ma devono soddisfare la proprietà principale, ovvero che il cammino e il ciclo devono essere orientati.
Dati u,$v \in V$:

- u,v si dicono *connessi* se esiste un cammino orientato da verso v.
- $G = (V, E)$ si dice **fortemente connesso** se ogni coppia ordinata di nodi è connessa.

  Formalmente, $\forall v \in V$:

  - $\exists\ u \rightsquigarrow v$ cammino orientato.
  - $\exists\ v \rightsquigarrow u$ cammino orientato.
  - u e v sono **mutuamente raggiungibili**.

I sottografi fortemente connessi e massimali di un grafo orientato sono le classi di equivalenza della relazione "*sono mutuamente raggiungibili...*".

Grafi aciclici

Un grafo (orientato o non orientato) si dice *aciclico* se al suo interno non contiene cicli.
Un **albero** è:

- Un grafo non orientato, connesso e aciclico.
- Un grafo non orientato, connesso e tale che $|E| = |V| - 1$.
- Un grafo non orientato, aciclico e tale che $|E| = |V| - 1$.

Si definisce inoltre *foresta* un grafo non orientato e aciclico le cui componenti connesse sono alberi.

La rappresentazione più immediata per un grafo $G = (V, E)$ con $n = |V|$ nodi è quella tabellare.
La *matrice di adiacenza* per G è una tabella A con n righe e n colonne, numerate da 0 a $n - 1$, dove la casella che si trova all'incrocio tra la riga i e la colonna j viene chiamata cella $A_{[i,j]}$ e può assumere un valore binario (0,1).

$$A_{[i,j]} = \begin{cases} 0 & \text{se } (i,j) \notin E \\ 1 & \text{se } (i,j) \in E \end{cases}$$

Quando un grafo è orientato la matrice di adiacenza relativa ad esso si dice *simmetrica* ($A_{[i,j]} = A_{[j,i]}$) ma per entrambi i tipi di grafo il costo in spazio è:

$$S(n, m) = \Theta(n^2)$$

Operazione	**Costo in tempo**
adiacenti (u,v)	O(1) return A[u,v]
grado (v)	$\Theta(n)$ scorre tutta la riga v di A
aggiungiArco (u,v)	O(1) $A[u, v] = 1$
rimuoviArco (u,v)	O(1) $A[u, v] = 0$

Tabella 5: Costo in tempo delle relative operazioni su matrici

Un altro modo per rappresentare i grafi (orientati o meno) è attraverso le *liste di adiacenza*, l'idea è quella di associare ad ogni nodo una lista di nodi a lui adiacenti. Le liste di adiacenza per G sono un array A di $n = |V|$ liste che rappresentano il vicinato per ogni nodo $u \in V$:

- Se G non è orientato, $A[u]$ rappresenta il vicinato $\delta(u)$.
- Se G è orientato, $A[u]$ rappresenta il vicinato in uscita $\delta_u(u)$.

$$Adj[u] = \begin{cases} nil & \text{se } \delta(u) = 0 \\ \text{lista dei nodi } v \text{ adiacenti ad } u \text{ tali che } (u,v) \in E \end{cases}$$

$\forall u$, la lunghezza della lista $Adj[u]$ è pari al grado di u, nei grafi non orientati, ed al grado uscente di u nei grafi orientati.
L'occupazione in spazio (costo) in entrambi i casi è $S(|V|, |E|) = S(n, m) = \Theta(n + m)$ dove n è un array di n riferimenti alle liste ed m è la lunghezza complessiva di tutte le liste.

$\forall v \in V$:

$$|Adj[v]| = \delta(v) \quad \Leftrightarrow \quad \sum_{v \in V} |Adj[v]| = \sum_{v \in V} \delta(v) = 2|E|$$

$$|Adj[v]| = \delta_u(v) \quad \Leftrightarrow \quad \sum_{v \in V} |Adj[v]| = \sum_{v \in V} \delta_u(v) = |E|$$

Operazione	**Costo in tempo**
adiacenti (u,v)	$O(\delta(u))$ cerca v nella lista $Adj[u]$
grado (v)	$\Theta(\delta(v))$ scorre la lista $Adj[v]$
aggiungiArco (u,v)	$O(1)$ se le liste non sono ordinate $O(\delta(u))$ grafo non orientato $O(\delta(u) + \delta(v))$ grafo orientato
rimuoviArco (u,v)	$O(\delta(u) + \delta(v))$ grafo non orientato $O(\delta(u))$ grafo orientato

Tabella 6: Costo in tempo delle relative operazioni su liste

Esistono due principali metodi per "visitare" un grafo.

- BFS (Breadth First Search): visita in ampiezza.
- DFS (Depth First Search): visita in profondità.

Quando utilizziamo la BFS, G viene rappresentato mediante liste di adiacenza. L'algoritmo relativo a questo tipo di visita agisce sul grafo, partendo da una *sorgente*.
La BFS scopre quindi tutti i nodi raggiungibili da s (tutto V solo se G è connesso), in ordine di distanza crescente da essa, usando una *coda* di appoggio. Calcola inoltre la distanza da s di tutti i nodi raggiungibili dalla sorgente e se G è connesso, la BFS lo esplora completamente, altrimenti esplora solo la componente connessa che contiene s o la posizione di G raggiungibile da s.
$\forall v \in V$:

- $v.color$ rappresenta indica lo "stato" nel quale si trova il nodo v.
- $v.d$, alla fine della visita, memorizza la distanza di v dalla sorgente.
- $v.\pi$ rappresenta il predecessore di v ($v.\pi = u$ se u è il predecessore di v).

B	**G**	**N**
v non è ancora stato scoperto	v è stato scoperto	tutti gli archi uscenti da v sono stati visitati

Tabella 7: Tre colorazioni di un nodo

Vediamo adesso l'algoritmo della BFS.

BFS

```
BFS (G,s) {
  for all v in V \ {s} {
    v.color=B;
    v.d=+Infinity;
    v.pigreco=nil;
  }
  s.color=G;
  s.d=0;
  s.pigreco=nil;
  Q=nuova Coda ();
  Enqueue (Q,s);
  while (Q!=0) {
    u=Dequeue (Q);
    for all v in Adj[u] {
      if (v.color==B) {
        v.color=G;
        v.pigreco=u;
        v.d=u.d+1;
        Enqueue (Q,v);
      }
    }
    u.color=N;
  }
}
```

Costo in tempo
$T(n) = O(|V| + |E|)$.

Analisi di complessità

1) I nodi entrano ed escono dalla coda al più una volta ($B \to G$).

2) Estrazioni ed inserimenti nella coda costano $O(1)$ (costo complessivo delle operazioni sulla coda $\to O(|V|)$).

3) La fase iniziale (color, π, d) ha un costo $\Theta(|V|)$.

4) La lista di adiacenza di un nodo viene esaminata al più una volta, quando il nodo viene estratto dalla coda.

Costo del while

$$\sum_{v \text{ estratti da } Q} |Adj[v]| \quad = \quad \sum_{v \text{ estratti da } Q} \delta(v) \quad = \quad O(|E|)$$

$T_{BFS}(|V|, |E|) = O(|V| + |E|)$ $\quad (O \to \Theta$ se G è connesso).
$S_{BFS}(|V|, |E|) = \Theta(|V|)$.

Lemma 22.1

Dato $G = (V, E)$ e $s \in V$, per ogni arco $(u, v) \in E$ vale che $\delta(s, v) \leq \delta(s, u) + 1$.

Lemma 22.2 (limite superiore)

Dato $G = (V, E)$ e $s \in V$, al termine della BFS (G,s), per ogni nodo $v \in V$ vale che $v.d \geq \delta(s, v)$.

Dimostrazione (per induzione)

Ipotesi induttiva: $v.d \geq \delta(s, v)$ per ogni $v \in V$.
Caso base: Dopo la prima operazione di Enqueue l'ipotesi è vera, infatti abbiamo che $s.d = 0 = \delta(s, s)$ e $v.d = \infty \geq \delta(s, v)$ per ogni $v \in V \setminus \{s\}$.
Passo induttivo: Sia v un nodo bianco scoperto a partire da u (dunque $(u, v) \in E$). Per ipotesi induttiva $u.d \geq \delta(s, u)$, l'algoritmo pone:

$$\begin{array}{rcl} v.d & = & u.d + 1 \\ & \geq & \delta(s, u) + 1 \quad \text{(ipotesi induttiva)} \\ & \geq & \delta(s, v) \quad \text{(Lemma 22.1)} \end{array}$$

v viene inserito nella coda, diventa G, e non sarà mai più reinserito. $v.d$ non cambia più e l'ipotesi induttiva resta valida.

Lemma 22.3 (proprietà dei vertici della coda)

Durante l'esecuzione della BFS se $Q = [v_1, v_2, ..., v_r]$, allora:

- $v_r.d \leq v_1.d + 1$.
- $v_i.d \leq v_{i+1}.d$ con $i = 1, 2, ..., r - 1$.

In ogni istante, nella coda ci sono al massimo due valori distinti della distanza dalla sorgente, si dimostra per induzione sul numero di operzioni sulla coda Q.

Corollario 22.4

I valori delle distanze dalla sorgente dei nodi inseriti nella coda sono *monotoni* crescenti, se v_i è inserito nella coda prima di v_j allora $v_i.d \leq v_j.d$.

Teorema 22.5 (correttezza della BFS)

Dato $G = (V, E)$ e $s \in V$, la BFS scopre tutti i nodi $v \in V$ che sono raggiungibili da s e alla fine dell'esecuzione $v, d = \delta(s, v)$ per ogni nodo $v \in V$. Per ogni nodo $v \neq s$ che è raggiungibile da s, uno dei *cammini minimi* da s a v è un cammino minimo da s a $v.\pi$ seguiti dall'arco $(v.\pi, v)$.

Dimostrazione

Supponiamo *per assurdo* che ci sia almeno un nodo v che riceva un valore d diverso dalla sua distanza dalla sorgente.
Sia v il nodo più vicino alla sorgente che riceve un valore errato (con $v \neq s$):

- Per il Lemma 22.2 avremo che $v.d \geq \delta(s, v)$, quindi $v.d > \delta(s, v)$.
- v deve essere raggiungibile da s altrimenti $\delta(s, v) = \infty \geq v.d$.

Sia u il nodo che precede immediatamente v in un cammino minimo da s a v:

- $\delta(s, v) = \delta(s, u) + 1$.
- $\delta(s, u) < \delta(s, v)$ e $u.d = \delta(s, u)$.

Dunque risulta che $v.d > \delta(s, v) = \delta(s, u) + 1 = u.d + 1$.
Consideriamo il momento in cui u viene estratto dalla coda Q, v può essere solamente B, G o N.

- Se v è B, l'algoritmo assegna $v.d = u.d + 1$.
- Se v è N, allora v era già stato rimosso dalla coda e, per il Corollario 22.4, $v.d \leq u.d$.
- Se v è G, allora è stato scoperto da un vertice w estratto da Q prima di u e si ha $v, d = w.d + 1$, per il Corollario 22.4 $w.d \leq u.d$ quindi $v.d = w.d + 1 \leq u.d + 1 \implies v.d = \delta(s, v)$.

Tutti i nodi raggiungibili da s devono essere scoperti, se non lo fossero $\infty = v.d > \delta(s, v)$.
Osserviamo che, se $v.\pi = u$, allora $v.d = u.d + 1$, quindi possiamo ottenere un cammino minimo da s a v prendendo un cammino minimo da s a $v.\pi$ e poi aggiungendo l'arco $(v.\pi, v)$.

Albero BF

$G\pi = (V\pi, E\pi)$.

$V\pi = \{v \in V \mid v.\pi \neq nil\} \cup \{s\}$.

$E\pi = \{(v.\pi, v) \mid v \in V\pi \setminus \{s\}\}$.

$G\pi$ è un *albero di cammini minimi* che contiene tutti i nodi raggiungibili dalla sorgente, è connesso e $|E\pi| = |V\pi| - 1$.

Lemma 22.6 (albero BF)

La BFS applicata ad un grafo $G = (V, E)$, orientato o non orientato, costruisce π in modo tale che il sottografo dei predecessori $G\pi = (V\pi, E\pi)$ sia un albero BF.
La BFS imposta $v.\pi = u$ se e solo se $(u, v) \in E$, e se v è raggiungibile da s, dunque $V\pi$ è formato dai nodi raggiungibili dalla sorgente. $G\pi$ è un albero, quindi contiene un unico cammino semplice da s a ciascun nodo in $V\pi$, appplicando il Teorema 22.5 in modo induttivo si conclude che ciascuno di questi cammini è un cammino minimo.

Cammino minimo

```
CamminoMinimo (G,s,v) {
  BFS (G,s);
  PrintPath (G,s,v);
}

PrintPath (G,s,v) {
  if (s==v) print s;
  else if (v.pigreco==nil) {
    print "v non e' raggiungibile da s";
  }
  else {
    PrintPath (G,s,v.pigreco);
    print v;
  }
}
```

Costo in tempo
$T_{PP}(|V|,|E|) = \Theta(\delta(s,v)) = O(|V|)$.

Nella DFS abbiamo un pò di cambiamenti...
$\forall v \in V$:

- $v.\pi$ indica il predecessore.
- $v.color$ indica il colore o lo "stato" del nodo.
- $v.d$ (discover) indica il tempo di "scoperta" o inizio visita di ogni nodo.
- $v.f$ indica il tempo di fine visita di ogni nodo.

DFS

```
DFS (G) {
  for all v in V {
    v.color=B;
    v.pigreco=nil;
  }
  time=0;
  for all v in V {
    if (v.color==B) DFS-visit (G,v);
  }
}
```

Costo in tempo
$T(|V|,|E|) = \Theta(|V|+|E|)$.
$S(|V|,|E|) = \Theta(|V|)$.

N.B. DFS-visit viene chiamata esattamente una volta su ogni nodo quando il esso viene scoperto e diventa G.

```
DFS-visit (G,u) {
  time++;
  u.d=time;
  u.color=G;
  for all v in Adj[u] {
    // ispeziono l'arco (u,v);
    if (v.color==B) {
      v.pigreco=u;
      DFS-visit (G,v);
    }
  }
  u.color=N;
  time++;
  u.f=time;
}
```

N.B. Al di fuori delle chiamate ricorsive spendo tempo $\Theta(|Adj[u]|)$, il costo totale di tutte le chiamate ricorsive è $\Theta(\sum_{v \in V} |Adj[v]|) = \Theta(|E|)$.

Foresta DF

$G\pi = (V, E\pi)$.

$E\pi = \{(v.\pi, v) \mid v \in V,\ v.\pi \neq nil\}$

Contiene un albero per ogni nodo scelto come sorgente in DFS (G) e gode delle seguenti proprietà:

- u è il padre di v in un albero della *foresta* DF se e solo se v è stato scoperto esaminando $Adj[u]$ ($v.\pi = u$).
- v è un discendente di u in un albero della *foresta* DF se e solo se v è stato scoperto quando u era G.

Teorema delle parentesi 22.7

$\forall\ u, v \in V$, $I_u = [u.d,\ u.f]$, $I_v = [v.d,\ v.f]$ è soddisfatta una ed una sola delle seguenti condizioni.

1. $I_u \cap I_v = \emptyset$.
2. $I_v \subset I_u$ v è discendente di u.
3. $I_u \subset I_v$ u è discendente di v.

Dimostrazione

Dati $u, v \in V$ supponiamo che $u.d < v.d$, abbiamo due casi possibili:

1. Se $u.f < v.d \longrightarrow u.d < u.f < v.d < v.f$, segue che $I_u \cap I_v = \emptyset$ quindi nessun nodo è stato scoperto mentre l'altro era grigio (non ci sono relazioni di discendenza).

2. Se $u.f > v.d$, allora v è stato scoperto quando u è G. v sarà discendente di u e la visita di v termina prima della visita di u, segue che $I_v \subset I_u$.

Corollario (annidamento degli intervalli)

v è discendente di u nella *foresta* DF se e solo se $I_v \subset I_u$ (con $v \neq u$).

Teorema del cammino bianco 22.9

v è discendente di u nella *foresta* DF di un grafo G se e solo se al tempo $u.d$, v può essere raggiunto da un cammino formato solo da nodi B.

Dimostrazione

Per poter dimostrare il Teorema 22.9 dobbiamo dimostrare le due "ipotesi" in entrambi i "sensi di percorrenza" $(\Rightarrow, \Leftarrow)$.

Caso $(\Rightarrow)$

- $v = u$: Il cammino da u a v contiene solo il nodo u e u è B al tempo $u.d$.
- $v \neq u$: Sia v un generico discendete di u, allora $I_v \subset I_u$ (per il corollario di annidamenti degli intervalli), quindi $u.d < v.d$ e v è B al tempo $u.d$.

Caso $(\Leftarrow)$

Per assurdo: esiste un cammino B ($u \rightsquigarrow v$) al tempo $u.d$, ma v non diventa discendente di u (con $v \neq u$). Sia v il nodo sul cammino B più vicino ad u, che non diventa discendente di u e sia w il nodo che precede v sul cammino B (w potrebbe essere u), w diventa discendente di u quindi $I_w \subseteq I_u$.
Formalmente scriviamo che $u.d < v.d < w.f \leq u.f$. $v_d \in I_u$ e per il Teorema 22.7 $\rightarrow v.f < u.f$ e $I_v \subset I_u$ quindi v diventa discendente di u.

Classificazione degli archi (Grafi orientati)

1. Archi d'albero (archi della *foresta* DF): (u,v) è un *arco d'albero* se $u = v.\pi$ quando ispeziono (u,v), v è B.
2. Archi all'indietro: (u,v) è un *arco all'indietro* se v è un antenato di u in un albero DF, quando ispeziono (u, v), v è G.
3. Archi in avanti: (u, v) è un *arco in avanti* se non è un arco d'albero e v è un discendente di u.
4. Archi trasversali: (u, v) è un *arco trasversale* se u e v non sono discendenti uno dell'altro, quando ispeziono (u, v) , v è N e $v.d < u.d$

Classificazione degli archi (Grafi non orientati)

(u, v) e (v, u) sono lo stesso arco, la classificazione si fa la prima volta che si ispeziona l'arco e la si mantiene.

Teorema

In una DFS su un grafo non orientato gli archi sono solo *archi d'albero* o *archi all'indietro*.

Dimostrazione

Dato $(u, v) \in E$ con $u.d < v.d$, v diventa G e poi N mentre u è G. Abbiamo anche qui due casi:

- 1° caso: (u, v) lo esploro la prima volta da u verso v (v è B) e (u, v) diventa un *arco d'albero*.
- 2° caso: (u, v) lo esploro la prima volta da v verso u (u è G) e (u, v) è un *arco all'indietro*.

Teorema (grafo orientato o non orientato)

Dato G un grafo orientato o non orientato, G è ciclico se e solo se G contiene almeno un arco all'indietro.

Dimostrazione

Anche in questo caso bisogna "spezzare" la dimostrazione in due parti per poter affrontare il $\Leftrightarrow$.

Caso ($\Leftarrow$)

Dato (u, v) arco all'indietro, v è un antenato di u se $\exists$ un cammino da v a u di archi d'albero.

Caso ($\Rightarrow$)

In questo caso per verifcare la presenza di un arco all'indietro abbiamo bisogno di separare il caso in cui il grafo sia orientato ed il caso in cui non lo sia.

- Grafo non orientato: Se G contiene un ciclo gli archi del ciclo non possono essere tutti archi d'albero, quindi $\exists$ un arco all'indietro.
- Grafo orientato: G contiene un ciclo, sia v il primo nodo del ciclo ad essere scoperto e diventare G. Sia u il nodo che precede v nel ciclo, al tempo $v.d$ tutti i nodi del ciclo sono B. Per il Teorema 22.9 u sarà discendete di v e l'arco (u, v) va da u verso un antenato, quindi è un arco all'indietro.

Orientamento topologico di grafi diretti e aciclici (DAG)

Formalmente $O.T.$ (ordinamento topologico): $\forall\ (u, v) \in E \rightarrow u$ precede v nell'ordinamento.
$O.T.$ si ottiene ordinando tutti i nodi del DAG in ordine decrescente in base ai tempi di fine visita.

OT(G)

- Si usa la DFS (G) per calcolare i tempi di fine visita.
- Quando termina la visita di un nodo lo si inserisce in testa ad una lista e la si restituisce.

Analisi di complessità

$T(|V|,|E|) = \Theta(|V|+|E|) + \Theta(|V|) = \Theta(|V|+|E|)$, dove $\Theta(|V|+|E|)$ è il costo della visita DFS e $\Theta(|V|)$ è il costo dei $|V|$ inserimenti in lista ciascuno di costo costante.

Teorema di correttezza:

L'algoritmo OT (G) produce un ordinamento topologico di un DAG.

Dimostrazione:

Dobbiamo dimostrare che $\forall\ (u,v) \in E$, u precede v nell'ordinamento. La visita di u finisce dopo la visita di v, quindi devo dimostrare che $u.f > v.f$. Quando ispeziono (u,v) ci sono due casi possibili:

1. v è B: (u,v) arco d'albero. v discendente di u, la sua visita finisce prima della visita di u $(v.f < u.f)$.

2. v è N: la sua visita è già terminata mentre la visita di u è ancora in corso quindi $(v.f < u.f)$.

3. v è G: ??? $\longrightarrow$ v non può essere G! (Altrimenti (u,v) sarebbe un arco all'indietro, ma G è un DAG e non ha cicli).:?"

Grafi pesati

Dato $G = (V,E)$, la *funzione peso* $w : E \to \mathbb{R}$ associa un *peso* a ciascun arco (rappresentano costi, perdite, distanze, quantità che occorre minimizzare).
Preso un qualunque cammino $p =< v_0, v_1, ..., v_k >$, indicheremo che $w(p)$ il peso del cammino p, formalmente avremo che:

$$w(p) = w(v_0,v_1) + w(v_1,v_2) + ... + w(v_{k-1},v_k) = \sum_{i=1}^{k} w(v_{i-1},v_i)$$

Con $\delta(u,v)$ indichiamo il peso di un *cammino minimo* e avremo che:

$$\delta(u,v) = \begin{cases} min\{w(p) : u \rightsquigarrow v\} & \text{se esiste un cammino da } u \text{ a } v \\ \infty & \text{altrimenti} \end{cases}$$

Un *cammino minimo* da u a v è un cammino di peso $w(p) = \delta(u,v)$ e ne esistono diverse varianti in base alle diverse "esigenze".

- Cammini minimi con sorgente unica $s \in V$: l'obbiettivo è quello di trovare il cammino di peso minimo dal nodo s a ciascun nodo $s \in V$.

- Cammini minimi con destinazione unica $t \in V$: l'obbiettivo è quello di trovare il cammino di peso minimo da ciascun nodo $v \in V$ a un nodo "destinazione" t. Invertendo la direzione degli archi il problema si riconduce a quello dei cammini minimi da sorgente unica.

- Cammino minimo per una coppia di nodi: l'obbiettivo è quello di trovare un cammino minimo da u a v, con u e v fissati. Si può risolvere con l'algoritmo per i cammini minimi con sorgente unica ($s = u$), asintoticamente questa soluzione ha lo stesso costo al caso pessimo di algoritmi specifici per questo problema.
- Cammini minimi tra tutte le coppie di nodi: l'obbiettivo è quello di trovare un cammino minimo per ogni coppia per ogni coppia di nodi del grafo. Si può risolvere con l'algoritmo per i cammini minimi con sorgente unica scegliendo ogni nodo come sorgente (esistono metodi più efficaci).

Struttura ottima

I sottocammini di cammini minimi sono cammini minimi (un cammino minimo contiene cammini minimi al suo interno).

Lemma 1

Dato un grafo orientato pesato $G = (V, E)$ con funzione peso $w : E \to \mathbb{R}$, sia $p =< v_0, v_1, ..., v_k >$ un cammino minimo $v_0 \rightsquigarrow v_k$, e per qualsiasi i e j tali che $1 \leq i \leq j \leq k$, sia $p_{ij} =< v_i, v_{i+1}, ..., v_j >$ il sottocammino di p da v_i a v_j, allora p_{ij} è un cammino minimo da v_i a v_j.

Dimostrazione

Supponiamo che il cammino p sia $v_1 \overset{p_{1j}}{\rightsquigarrow} v_i \overset{p_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$ e $w(p) = w(p_{1j}) + w(p_{ij}) + w(p_{jk})$ sia il suo peso. Per assurdo, esista un cammino p'_{ij} da v_i a v_j tale che $w(p'_{ij}) < w(p_{ij})$, allora possiamo costruire un cammino da v_i a v_j di peso minore di $w(p)$.

$v_1 \overset{p_{1j}}{\rightsquigarrow} v_i \overset{p'_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k \Rightarrow w(p') = w(p_{1j}) + w(p'_{ij}) + w(p_{jk}) < w(p)$.

Proprietà dei cammini minimi

Premessa: supponiamo che tutti i pesi siano non negativi (il problema è ben definito anche per pesi negativi a patto che non esistano cicli di peso negativo raggiungibili dalla sorgente).

I cammini minimi sono cammini semplici, privi di cicli.

- Se il ciclo ha peso > 0, eliminandolo dal cammino se ne ottiene uno di costo minore.
- Se il ciclo ha peso < 0, eliminandolo dal cammino se ne ottiene uno di costo uguale.

Un cammino semplice può contenere al più $|V|$ nodi e dunque $|V| - 1$ archi.
Rappresentiamo i cammini minimi usando l'attributo π (predecessore), la catena dei predecessori a partire da v descrive (all'indietro) il cammino minimo dalla sorgente s a v e ogni cammino minimo può essere stampato con PrintPath.

Algoritmo di Dijkstra

L'algoritmo di Dijkstra risolve il problema dei cammini minimi da sorgente unica in un grafo orientato pesato $G = (V, E)$ con pesi negativi, $\forall\ (u, v) \in E$, $w(u, v) \geq 0$. Mantiene inoltre, per ogni nodo v, un attributo $v.d$ (limite superiore per il peso di un cammino minimo) tale che $v.d \geq \delta(s, v)$.

$v.d$ ci serve per fare una *stime del cammino minimo* ed è inizializzato su ∞.
Tutti i nodi sono inseriti in una coda PQ di Min-priorità [priorità: $v.d$]. L'algoritmo seleziona ripetutamente il nodo u con la minor *stima* del cammino minimo (estraendolo da PQ) ed esamina gli archi uscenti da u. Per ogni $v \in Adj[u]$ controlla se passando da u si trova un cammino minimo meno costoso da $s \rightsquigarrow v$, in questo caso aggiorna $v.d$ e $v.\pi$ (rilassamento dell'arco (u, v)).

Dijkstra

```
DIJKSTRA (G,w,s) {
  PQ=nuova coda di Min-proprieta';
  s.d=0;
  s.pigreco=nil;
  INSERT (PQ,s);
  for all v in V \ {s} {
    v.d=Infinity;
    v.pigreco=nil;
    INSERT (PQ,v);
  }
  while (PQ!=0) {
    u=EXTRACT-MIN (PQ);
    for all v in Adj[u] {
      dnew=u.d+w(u,v);
      if (dnew<v.d) {
        v.d=dnew;
        v.pigreco=u;
        DECREASE-KEY (PQ,v,dnew);
      }
    }
  }
}
```

Costo in tempo
$T(|V|, |E|) = \Theta(|V|) + O(|V|log|V| + |E| + log|V|) = O((|V| + |E|)log|V|)$.

N.B. L'algoritmo viene diviso in due parti: la prima (dall'inizio fino alla fine del primo *for*) viene chiamata "*fase di inizializzazione*", la seconda "*rilassamento degli archi*".

Teorema (correttezza dell'algoritmo di Dijkstra)

L'algoritmo di Dijkstra, eseguito su un grafo orientato pesato $G = (V, E)$ con funzione peso non negativa w e sorgente s, termina con $v.d = \delta(s, v)$ per tutti i nodi $v \in V$.

Dimostrazione (sketch)

Possiamo dimostrare il precedente Teorema in due passi:

(1) Osserviamo che per ogni vertice $v \in V$, $v.d \geq \delta(s, v)$. All'inizio $v.d = \infty \geq \delta(s, v)$ e $v.d$ si aggiorna tramite le operazioni di rilassamento.

Il rilassamento pone $v.d$ uguale al peso di un qualche cammino $s \rightsquigarrow v$, che è certamente maggiore o uguale al peso $\delta(s, v)$ di un cammino minimo. Sulla sorgente si pone invece $s.d = 0 = \delta(s, s)$ e $s.d$ non viene più aggiornato.

Sia S l'insieme dei vertici già visitati ed estratti da PQ. Il teorema si dimostra per induzione sul numero di vertici in S.

Ipotesi induttiva: per ogni $v \in S$, $v.d = \delta(s, v)$.
Caso base: per $v = s$, risulta che $s.d = 0 = \delta(s, v)$.
Passo induttivo: sia v il prossimo nodo estratto da PQ e inserito in S. P è un cammino $s \rightsquigarrow v$ di peso minimo (dove s può coincidere con x e y può coincidere con v) e $(x, y) \in E$ è il primo arco del cammino P che esce da S ($x \in S$ e y è ancora in PQ).

(2) Osserviamo che $y.d = \delta(s, y)$. Quando x è stato estratto da PQ, l'arco (x, y) è stato rilassato, $y.d \leq x.d + w(x, y) = \delta(s, x) + w(x, y) = \delta(s, y)$ (induzione su x, sottostruttura ottima dei cammini minimi) e $y.d \geq \delta(s, y)$, $y.d \leq \delta(s, y) \Rightarrow y.d = \delta(s, y)$ (per il passo (1)).
Segue che $v.d \geq \delta(s, v) \geq \delta(s, y) = y.d \geq v.d$, quindi $v.d \geq \delta(s, v)$ e $v.d \leq \delta(s, y) \Rightarrow v.d = \delta(s, v)$.
Al termine dell'algoritmo $PQ = \emptyset$ e $S = V$, per ogni nodo $v \in V$, $v.d = \delta(s, v)$.

Analisi di complessità

La costruzione della coda di MIN-priorità si fa in tempo $\Theta(|V|)$, ciascuna inserzione richiede tempo costante (solo due valori: 0 e ∞).
Si eseguono $|V|$ operazioni EXTRACT-MIN, la lista di adiacenza di ciascun nodo viene esaminata esattamente una volta, ci sono in totale $|E|$ iterazioni del ciclo *for* e al più $|E|$ chiamate di DECREASE-KEY.

Il tempo di esecuzione dipende dall'implementazione dalla coda di MIN-priorità.

Array indicizzato su nodi

- Le operazioni INSERT e DECREASE-KEY hanno costo $O(1)$.
- EXTRACT-MIN ha costo $O(|V|)$ (occorre ispezionare l'intero array per trovare il nodo con la stima di distanza minima).
- Il costo complessivo è $T(|V|, |E|) = \Theta(|V|) + O(|V|^2 + |E|) = O(|V|^2)$.

MIN-heap binario

- La costruzione di PQ richiede tempo $\Theta(|V|)$.
- EXTRACT-MIN ha costo $O(log|V|)$.
- DECREASE-KEY ha costo $O(log|V|)$ ma occorre prima cercare il nodo v in PQ. I nodi devono mantenere dei riferimenti ai corrispondenti elementi dell'heap PQ e viceversa: in questo modo si accede in tempo costante a v in PQ e si esegue DECREASE-KEY in tempo $O(log|V|)$.
- Il costo complessivo è:
  $T(|V|, |E|) = \Theta(|V|) + O(|V|log|V| + |E|log|V|) = O((|V| + |E|)log|V|)$

14 Programmazione Dinamica

La programmazione dinamica è un paradigma per la costruzione di algoritmi alternativo alla ricorsione. Esso si usa nei casi in cui esiste una definizione ricorsiva del problema, ma la trasformazione diretta di tale definizione in un algoritmo genera un programma di complessità esponenziale a causa del calcolo ripetuto, sugli stessi sottoinsiemi di dati, da parte delle diverse chiamate ricorsive.
A differenza della ricorsione (che opera in modo *top down*) risolve prima i sottoproblemi, ne memorizza le soluzioni e da queste costruisce la soluzione del problema originale attraverso soluzioni di sottoproblemi via via sempre più grandi, in modo *bottom up*.
Per comprendere questo punto consideriamo i due algoritmi ricorsivi RICERCA BINARIA e MERGESORT. Il primo comprende due chiamate ricorsive nella sua formulazione, ma ogni esecuzione dell'algoritmo ne richiede una sola a seconda che il dato da ricercare sia minore o maggiore di quello incontrato nel passo della ricerca: in tal modo il calcolo si ripete su un solo sottoinsieme grande $\frac{n}{2}$, poi su un sottoinsieme di questo grande $\frac{n}{4}$, e così via. Dividere il sottoinsieme per due fino a ridurlo a un singolo elemento implica, come abbiamo visto, l'esecuzione di $O(logn)$ passi (si noti l'ordine O e non Θ: infatti l'algoritmo potrebbe terminare prima se il dato cercato si incontra in uno dei primi tentativi). MERGESORT comprende anch'esso due chiamate ricorsive nella sua formulazione e ogni esecuzione dell'algoritmo le richiede entrambe per ordinare le metà "destra" e "sinistra" dell'insieme. Anche in questo caso in $logn$ passi i sottoinsiemi si riducono a un singolo elemento, ma poichè entrambi i sottoinsiemi vengono esaminati e il tempo per la loro fusione è lineare, il tempo richiesto, come già visto, è nel complesso $\Theta(nlogn)$. La grande efficienza di questi due algoritmi rispetto ad altri realizzati in modo meno sofisticato è dovuta alla loro struttura ricorsiva, formulata in modo che chiamate ricorsive diverse agiscano su dati disgiunti, cioè l'algoritmo non è chiamato a ripetere operazioni già eseguite sugli stessi dati. Se ciò non si verifica, l'effetto della ricorsività sulla complessità di un algoritmo può essere disastroso.
Due esempi elementari sono il calcolo dei numeri di *Fibonacci* e il calcolo dei *coefficienti binomiali*. I primi sono definiti come:

$$F_0 = 0,\ F_1 = 1,\ F_i = F_{i-1} + F_{i-2}$$

E generano la successione:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, ...$$

La funzione che ci permette di calcolare i numeri Fibonacci è la seguente:

$$F_n = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ F_{n-1} + F_{n-2} & n \geq 2 \end{cases}$$

Calcolo ricorsivo dell'ennesimo numero di Fibonacci

```
FIB–REC (n) {
  if (n==0) return 0;
  if (n==1) return 1;
  return FIB–REC (n−1) + FIB–REC (n−2);
}
```

Costo in tempo
L'algoritmo ricorsivo per il calcolo dell'ennesimo numero di Fibonacci ha complessità esponenziale in n, quindi:

$$T(n) = \begin{cases} \Theta(1) & n = 0,\ 1 \\ T(n-1) + T(n-2) + \Theta(1) & n \geq 2 \end{cases}$$

Il calcolo di F_n si innesca con la chiamata estserna FIB-REC (n), come abbiamo già visto $T(n)$ rappresenta il tempo richiesto dall'algoritmo e avremo che:

$T(0) = c_0,\ T(1) = c_1$, con c_0 e c_1 costanti.

$T(n) = T(n-1) + T(n-2) + c_2$, con c_2 costante, per $n > 1$.

Possiamo semplificare il calcolo notando che $T(n) > 2T(n-2)$ da cui otteniamo:

$$\begin{aligned} T(n) &> 2T(n-2) > 2(2T(n-4)) = 2^2T(n-4) \\ &> 2^2(2T(n-6)) = 2^3T(n-6) > ... \\ &> 2^kT(n-2k) > ... \end{aligned}$$

Segue che, per n pari, otteniamo $T(n) > 2^{\frac{n}{2}}T(0) = 2^{\frac{n}{2}}c_0$, e per n dispari otteniamo $T(n) > 2^{\frac{n-1}{2}}T(1) = 2^{\frac{n-1}{2}}c_1$. In ogni caso $T(n)$ è limitato inferiormente da una funzione esponenziale in n, quindi l'algoritmo è esponenziale.
Vediamo adesso come "trasformare" l'algoritmo ricorsivo in algoritmo di programmazione dinamica per calcolare l'ennesimo numero di Fibonacci.

```
FIB–DP (n) {
  if (n==0) return 0;
  if (n==1 || n==2) return 1;
  a=b=1;
  for (i=3 to n) {
    c=a+b;
    a=b;
    b=c;
  }
  return b;
}
```

Costo in tempo
In questo caso l'algoritmo per il calcolo dell'ennesimo numero di Fibonacci ha costo lineare.

$$T(n) = \begin{cases} \Theta(1) & n = 0,\ 1,\ 2 \\ \Theta(n) + \Theta(1) = \Theta(n) & n \geq 3 \end{cases}$$

A questo punto sarebbe utile chiederci: quando si applica la programmazione dinamica? O ancora meglio, quando, per il nostro problema, si deve cercare una soluzione che impiega la programmazione dinamica?

Risposta: quando il problema ha una sottostruttura ottima (la soluzione ottima di un problema contiene le soluzioni ottime dei sottoproblemi) e ha sottoproblemi sovrapposti.
Un algoritmo di programmazione dinamica agisce seguendo questi quattro semplici passaggi:

1. Identificare i sottoproblemi "interessanti" del problema.
2. Identificare i sottoproblemi elementari e la loro soluzione diretta.
3. Definire la regola (ricorsiva) per il calcolo delle soluzioni dei sottoproblemi a partire da quelle dei valori già calcolati (dei sottoproblemi ancora più piccoli).
4. Identificare (nella Tabella) e restituire la soluzione del problema iniziale.

Edit distance

Date X e Y due stringhe su un alfabeto Σ ($|X| = n$ e $|Y| = m$), la *edit distance*[3] tra X e Y si indica con $d_E(X, Y)$ ed è il numero minimo di operazioni di editing (sostituzione, cancellazione, inserimento...) su una lettera che trasformano X in Y. Notiamo che *edit distance* ha sottostruttura ottima:
Prese $X = x_1,\ x_2, ...,\ x_i, ...,\ x'_i, ...,\ x_n$ e $Y = y_1,\ y_2, ...,\ y_j, ...,\ y'_j, ...,\ y_m$ due stringhe (dove $|X| = n$ e $|Y| = m$), notiamo che entrambe formano un *allineamento ottimo* di X e Y. Per dimostrare che la *edit distance* ha sottostruttura ottima dobbiamo far vedere che l'allineamento ottimo precedentemente scritto contiene le soluzioni ottime dei suoi sottoporoblemi.
Prese $x_i, ...,\ x'_i$ e $y_j, ...,\ y'_j$ due sottostringhe di X e Y (con $1 \leq i \leq i' \leq n$ e $1 \leq j \leq j' \leq m$), vediamo che entrambe formano un allineamento ottimo di $X[i, ..., i']$ e $Y[j, ..., j']$. Per assurdo potremmo immaginare che quest'ultimo allineamento non sia ottimo, ma questo vuol dire che esiste un altro allineamento $X[i, ..., i']$, $Y[j, ..., j']$ che ha costo minore (ovvero è migliore) e avrei un allineamento migliore di quello iniziale, ma questo contraddice l'ipotesi che sostiene che il primo allineamento fosse ottimo.

Calcolo della Edit distance con Programmazione Dinamica

1. L'allineamento X,Y ha tre possibili modi per terminare:
   - Abbiamo l'ultima lettera di X, cioè x_n, cancellata. In questo caso avremo che il nostro allineamento tra X e Y avrà $X[1, ..., n-1]x_n$ con tutto Y al di sotto e sarà un allineamento ottimo.

[3] La edit distance è una metrica del tipo ↓
$d_E(X, Y) \geq 0,\ d_E(X, X) = 0,\ d_E(X, Y) = d_E(Y, X),\ d_E(X, Y) \leq d_E(X, Z) + d_E(Z, Y)$

- Abbiamo l'ultima lettera di Y, cioè y_m, inserita. In questo caso avremo che il nostro allineamento tra X e Y avrà tutto X con $Y[1, ..., m-1]y_m$ al di sotto e sarà un allineamento ottimo.
- In questo caso, se termino con x_n (ultima lettera di X) sopra y_m (ultima lettera di Y), devo avere allineate nel migliore dei modi (per sottostruttura ottima) la sottostringa $X[1, ..., n-1]x_n$ di X e la sottostringa $Y[1, ..., m-1]y_m$ di Y.

2. Per allineare una stringa non vuota come ad esempio X_i[4] con la stringa vuota ε[5], bisogna fare i cancellazioni sulla stringa X per renderla vuota. Nel caso contrario in cui bisogna allineare la stringa vuota ε con una stringa Y_j, bisonga fare j inserimenti nella stringa vuota.

3. X_i (prefisso lungo i di X) con al di sotto Y_j (prefisso lungo j di Y) è un allineamento ottimo del prefisso X_i di X e del prefisso Y_j di Y.

4. A questo punto notiamo che $0 \leq i \leq n$ e $0 \leq j \leq m$. Per capire al meglio il procedimento di questo punto bisogna allocare una matrice che rappresenta tutti i passagi (vedi Matrice 1).

Costruiremo una matrice così formata $M(n+1) \times (m+1)$, la casella $M[i, j]$ conterrà la *edit distance* tra X_i e Y_j e la indichiamo con $d_E(X_i, Y_j)$.

Vediamo ora come riempire la prima riga e la prima colonna della nostra matrice.

Inizzializzazione: $M[0, j] = j$, $M[i, 0] = i \ \rightarrow \forall\ 1 \leq i \leq n$ e $1 \leq j \leq m$.

$$M[i,j] = \text{MIN} \rightarrow \begin{cases} M[i-1,j]+1 \\ M[i,j-1]+1 \\ M[i-1,j-1]+1 \quad \text{se } x_i \neq y_j \\ M[i,j-1] \quad \text{altrimenti} \end{cases}$$

È importante sottolineare che la casella nella quale si trova il valore della *edit distance* tra X e Y è $M[n, m]$, ovvero l'ultima in basso a destra nella matrice, quindi avremo che $d_E(X, Y) = M[n, m]$.

$$\begin{bmatrix} 0 & 1 & 2 & . & . & j & . & . & m \\ 1 & & & & & . & & & . \\ 2 & & & & & . & & & . \\ . & & & & & . & & & . \\ . & & & & & . & & & . \\ i & . & . & . & . & d_E(X_i, Y_j) & & & . \\ . & & & & & & & & . \\ . & & & & & & & & . \\ n & . & . & . & . & . & . & . & d_E(X, Y) \end{bmatrix}$$

Matrice 1: Edit distance tra X e Y

[4] $X[1...i]$ è il *prefisso* di X lungo $i \equiv X_i$

[5] ε è la stringa vuota, $|\varepsilon| = 0$

Problema dello zaino 0-1 (Knapsack problem)

Il *problema dello zaino*, o in inglese Knapsack problem, è un problema di ottimizzazione combinatoria posto nel modo seguente.
Sia dato uno zaino che possa sopportare un determinato peso W e siano dati in input n oggetti, ognuno dei quali caratterizzato da un *peso* e un *valore*. Il problema si propone di scegliere quali di questi oggetti mettere nello zaino per ottenere il maggior valore senza eccedere il peso sostenibile dallo zaino stesso.
Indichiamo con $A = a_1, ..., a_n$ l'insieme che contiene gli n oggetti e ad ogni oggetto viene associato un *valore* $v_1, ..., v_i, ..., v_n \in \mathbb{Z}^+$ ed un *peso* $w_1, ..., w_i, ..., w_n \in \mathbb{Z}^+$. Come output avremo un insieme $S \subseteq A$ il quale conterrà il carico con il valore più alto trasportabile dallo zaino.
Formalmente avremo $\longrightarrow max \sum_{a_i \in S} v_i$ e tale che $\sum_{a_i \in S} w_i \leq W$.

Esempio
Input: $n = 3$, $W = 8$.

	a_1	a_2	a_3
w_i	5	4	4
v_i	10	5	6

Tabella 8: Esempio di un generico problema dello zaino

Se scegliessimo $S = \{a_1\}$ avremo peso $= 5 \leq W$ e $v = 10$. Questa però sarebbe una scelta ottima solo *localmente*, infatti se scegliessimo $S = \{a_2, a_3\}$ avremo peso $= 8 \leq W$ e $v = 11$, quindi avremmo fatto una scelta migliore della precedente (quindi ottima).
Esistono due "strategie" Greedy, la prima sceglie l'oggetto con valore maggiore, la seconda sceglie l'oggetto con rapporto valore-peso maggiore. In entrambi i casi attua una scelta ottima solo localmente.

	a_1	a_2	a_3
w_i	5	4	4
v_i	10	5	6
$\frac{v}{w}$	2	1,25	1,5

Tabella 9: Strategia #2 Greedy

Differenze tra algoritmi Greedy e Programmazione dinamica

Le differenze principali tra un algoritmo Greedy e la programmazione dinamica stanno nelle scelte che l'uno e l'altra attuano ad ogni passo. Un algoritmo

Greedy fa la scelta localmente ottima ad ogni suo passaggio, riducendosi ad un sottoproblema che ne è la conseguenza. Continuando a fare scelte di tipo Greedy l'algoritmo arriva al caso limite.
La programmazione dinamica invece si basa su scelte che dipendono dalla soluzione di sottoprobelmi e tutti quelli che ci servono sono già stati risolti.

Problema dello zaino con Programmazione dinamica:

1. Definizione di sottoproblemi: Dato un insieme $|A| = n$ e W come peso massimo del nostro zaino, definiamo $\forall\ 0 \leq i \leq n$ il sottoinsieme di dimensione i $S_i = \{a_1, a_2, ..., a_i\}$ che contiene i primi i oggetti di A. Consideriamo anche, $\forall\ 0 \leq j \leq W$ un *peso parziale* che ci permette di definire il sottoproblema $S_{ij} \subseteq S_i = \{a_1, a_2, ..., a_i\}$ (soluzione ottima con $A = S_i$ e $W = j$) che rappresenta il carico con il valore più alto $\leq S_i$ e di peso $\leq j$. Indicheremo con $c[i, j]$ il valore totale degli oggetti in S_{ij}.

2. Sottoproblemi elementari: Abbiamo due sottoproblemi elementari, uno per le righe e uno per le colonne. Nel primo caso avremo $i = 0$ (nessun oggetto) quindi $S_0 = \emptyset$, nella matrice in posizione $c[0, j]$ metteremo la somma dei valori di tutti gli oggetti presi dall'insieme S_0 (senza sforare il peso j), quindi $c[0, j] = 0\ \forall j$. Nel secondo caso avremo $j = 0$ (nessun peso trasportabile) quindi $W = 0$ e nella matrice in posizione $c[i, 0] = 0\ \forall i$.

3. (De)composizione ricorsiva: Nel calcolo di $c[i, j]$ ci sono due casi possibili, nel primo il peso di a_i non è trasportabile, cioè $w_i > j$ e dato che $a_i \notin S_{ij}$, $c[i, j] = c[i-1, j]$. Nel secondo caso invece a_i è trasportabile, cioè $w_i \leq j$ e resta solo da verificare se aggiungerlo è la soluzione migliore. Tale scelta

$$c[i, j] = \text{MAX} \rightarrow \begin{cases} c[i-1, j] & S_{ij} \text{ se non prendo } a_i \\ v_i + c[i-1, j-w_i] & S_{ij} \text{ se prendo } a_i \end{cases}$$

4. Soluzione: $c[n, m]$ è il massimo valore trasportabile $S \subseteq S_n = A$ di peso $\leq j = W$.

$$\begin{bmatrix} & 0 & . & . & j & . & . & W \\ 0 & 0 & . & . & 0 & . & . & 0 \\ . & . & & & . & & & \\ . & . & & & . & & & \\ i & 0 & . & . & ? & & & \\ . & . & & & & & & \\ . & . & & & & & & \\ n & 0 & & & & & & \end{bmatrix}$$

Matrice 2: Problema dello zaino

Problema dello zaino

```
Zaino (n, v[], w[], W) {
  C: matrice (n+1)x(W+1);
  for (j=0 to W) Do C[0,j]=0;
  for (i=1 to n) Do C[i,0]=0;
  for (i=1 to n) Do {
    for (j=1 to W) Do {
      C[i,j]=C[i-1,j];
      if (j>=w[i]) then {
        m=C[i-1,j-w[i]]+v[i];
        if (m>C[i,j]) then C[i,j]=m;
      }
    }
  }
  return C[n,m];
}
```

Costo in tempo

$T(n, W) = \Theta(n * W)$.

N.B. La dimensione in input equivale a $n + W$ e la dimensione dell'input di W é $log_2 W$, quindi, dato che la complessitá dell'algoritmo era di $\Theta(n * W)$, avremo che W sará esponenziale in $log_2 W$ (input size di W). Possiamo avere due casi:

- $W \in O(n^c)$ quindi W si dice polinomiale in n e questo implica che, anche $T(n, W) = \Theta(n * W)$ é polinomiale in n.
- $W \in 2^n$ quindi $T(n, W) = \Theta(n * W) = O(n * 2^n)$ é esponenziale in n.

Entrambi i casi ci permettono di concludere che l'algoritmo proposto (Problema dello zaino) é *pseudopolinomiale*.

Se volessimo invece dimostrare che tale algoritmo abbia struttura ottima dovremmo procedere nel seguente modo.
Dati n oggetti con valore v e peso w ipotizziamo che $S \subseteq A$, con peso di $S \leq W$, sia la *soluzione ottima*.
A questo punto possono verificarsi due casi:

- $a_n \in S \Rightarrow S \setminus \{a_n\}$ con valore max e con peso $\leq W - w_n$.
- $a_n \notin S \Rightarrow S$ é la soluzione ottima di $A \setminus \{a_n\}$ e peso W.

Nel secondo punto stiamo ipotizzando per assurdo che esista una soluzione S'' migliore in $A \setminus \{a_n\}$ e $W - w_n$ e quindi migliore della soluzione S iniziale e ció contraddice l'ipotesi che S sia la soluzione ottima (migliore).

Esempio

	a_1	a_2	a_3
w_i	1 kg	2 kg	3 kg
v_i	€60	€100	€120

Tabella 10: Esempio di un generico problema dello zaino

Il peso trasportabile dal nostro zaino é $W = 5\ kg$ e dobbiamo costruire una tabella $C(n+1) \times (W+1)$.

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	60	60	60	60	60
2	0	60	100	160	160	160
3	0	60	100	160	180	220

Tabella 11: Tabella C risultante

Risposta: $S = \{a_1, a_2, a_3\}$ e $v = 220$.

Problema dello zaino frazionario

Anche in questo caso avremo lo stesso input dello zaino 0-1 ma possiamo prendere delle *frazioni* degli oggetti anziché fare la scelta binaria 0-1. Qui l'algoritmo Greedy # 2 funziona, infatti ordineremo in base al valore del rapporto $\frac{v}{w}$ e prenderemo dal $\frac{v}{w}$ piú grande a quello piú piccolo finché $\leq W$, eventualmente frazionando l'ultimo oggetto inserito.

Soluzione enumerativa per Zaino 0-1

Genero tutti i possibili sottoinsiemi di A ($2^|A| = 2^n$), calcolo peso e valore di ogni sottoinsieme e prendo il sottoinsieme di valore massimo tra tutti quelli di peso $\leq W$ ($O(n * 2^n)$.
Segue che $T(n) = O(n * 2^n)$.

Problema della scacchiera

Data una scacchiera $n \times n$ ed una pedina che si muove in questo modo:

$$(i,j) = \begin{cases} (i+1, j) & \text{se } i \leq n-1 \\ (i, j+1) & \text{se } j \leq n-1 \end{cases}$$

Calcolare il numero totale di possibili percorsi per spostare la pedina da **START** $(0,0)$ a **END** (n,n) e valutare la complessitá dell'algoritmo.

Indicheremo con L_{ij} il numero di modi per arrivare alla casella (i, j) da **START**. $L_{0,j} = 1 \ \ \forall\ j = 1, ..., n$ e $L_{i,0} = 1 \ \ \forall\ i = 1, ..., n$, $L_{ij} = L_{i-1,j} + L_{i,j-1}$.

15 Teoria della calcolabilitá

Si occupa delle questioni fondamentali circa la potenza e le limitazioni dei sistemi di calcolo. L'origine risale alla prima metà del ventesimo secolo, quando i logici matematici iniziarono ad esplorare i concetti di:

- Computazione
- Algoritmo
- Problema risolvibile per via algoritmica

Quindi dimostrarono l'esistenza di problemi che non ammettono un algoritmo di risoluzione.

Problemi computazionali

Sono problemi formulati matematicamente di cui cerchiamo una soluzione algoritmica. Sono classificati nel seguente modo:

- Problemi non decidibili
- Problemi decidibili
  - Problemi trattabili (costo polinomiale)
  - Problemi intrattabili (costo esponenziale)

Calcolabilitá e complessitá

Calcolabilità: nozioni di algoritmo e di problema non decidibile.
Complessità: nozione di algoritmo efficiente e di problema intrattabile.
La calcolabilità ha lo scopo di classificare i problemi risolvibili e non risolvibili, mentre la complessità in "facili" e "difficili".

Esistenza di problemi indecidibili

Insiemi numerabili

Due insiemi A e B hanno lo stesso numero di elementi $\iff$ si può stabilire una corrispondenza biunivoca tra i loro elementi. Un insieme è numerabile (possiede una infinità numerabile di elementi) $\iff$ i suoi elementi possono essere messi in corrispondenza biunivoca con i numeri naturali.

Enumerazione delle sequenze

Si vogliono elencare in un ordine ragionevole le sequenze di lunghezza finita costruite su un alfabeto finito. Le sequenze non sono in numero finito, quindi non si potrà completare l'elenco.

Scopo: raggiungere qualsiasi sequenza σ arbitrariamente scelta in un numero finito di passi. σ deve dunque trovarsi ad una distanza finita dall'inizio dell'elenco.
Ad una sequenza arbitraria corrisponde il numero che ne indica la posizione nell'elenco, ad un numero naturale n corrisponde la sequenza che occupa la posizione n nell'elenco.

Osservazione

La numerazione delle sequenze è possibile perché esse sono di lunghezza finita anche se illimitata, cioè per un qualunque intero d scelto a priori esistono sequenze di lunghezza maggiore di d. Per sequenze di lunghezza infinita la numerazione non è possibile.

Ordinamento canonico
Si stabilisce un ordine tra i caratteri dell'alfabeto, si ordinano le sequenze in ordine di lunghezza crescente e, a pari lunghezza, in "ordine alfabetico". Una sequenza s si troverà tra quelle di $|s|$ caretteri, in posizione alfabetica tra queste.

Insiemi non numerabili

Sono tutti gli insiemi non equivalenti ad N.

Per esempio:

- Insieme dei numeri reali
- Insieme dei numeri reali compresi nell'intervallo aperto $(0, 1)$
- Insieme dei numeri reali compresi nell'intervallo chiuso $[0, 1]$
- Insieme di tutte le linee nel piano
- Insieme delle funzioni in una o più variabili

L'insieme dei problemi computazionali non è numerabile

Problemi e funzioni
Un problema computazionale può essere visto come una funzione matematica che associa ad ogni insieme di dati, espressi da k numeri interi, il corrispondente risultato, espresso da j numeri interi.

$$f : N^k \to N^j$$

L'insieme delle funzioni $f : N^k \to N^j$ non è numerabile.

Diagolanizzazione
$F = \{funzioni\ f | f : N \to \{0, 1\}\}$
Ogni $f \in F$ può essere rappresentata da una sequenza infinita:
x 0 1 2 3 4 ... n ...
$f(x)$ 0 1 0 1 0 ... 0 ...
O, se possibile, da una regola finita di costruzione:

$$f(x) = \begin{cases} 0 & \text{se } x \text{ é pari} \\ 1 & \text{se } x \text{ é dispari} \end{cases}$$

Teorema

L'insieme F non è numerabile.

Dimostrazione

Supponiamo per assurdo che F sia numerabile, quindi possiamo enumerare ogni funzione. Assegnamo ad ogni $f \in F$ un numero progressivo nella numerazione, e costruiamo una tabella (infinita) di tutte le funzioni.

$\boldsymbol{x}$	0	1	2	3	4	5	6	7	8	...
$f_0(x)$	1	0	1	0	1	0	0	0	1	...
$f_1(x)$	0	0	1	1	0	0	1	1	0	...
$f_2(x)$	1	1	0	1	0	1	0	0	1	...
$f_3(x)$	0	1	1	0	1	0	1	1	1	...
$f_4(x)$	1	1	0	0	1	0	0	0	1	...
...					...					...

Tabella 12: Teorema della Diagonalizzazione

Consideriamo la funzione $g \in F$

$$g(x) = \begin{cases} 0 & \text{se } f_x(x) = 1 \\ 1 & \text{se } f_x(x) = 0 \end{cases}$$

g non corrisponde ad alcuna delle f_i della tabella e differisce da tutte nei valori posti sulla diagonale principale.

Per assurdo $\exists\ j$ t.c. $g(x) = f_j(x)$, allora $g(j) = f_j(j)$ ma per definizione:

$$g(j) = \begin{cases} 0 & \text{se } f_j(j) = 1 \\ 1 & \text{se } f_j(j) = 0 \end{cases}$$

Quindi $g(x) \neq f_j(x)$ e siamo di fronte ad una *contraddizione.*

Per qualunque numerazione scelta, esiste sempre almeno una funzione esclusa quindi F non è numerabile. Si possono considerare linee arbitrarie che attraversano la tabella toccando tutte le righe e tutte le colonne esattamente una volta, e definire funzioni che assumono in ogni punto un valore opposto a quello incontrato sulla linea. Esistono infinite funzioni di F escluse da qualsiasi numerazione.

Conclusione

$F = \{f : N \to \{0, 1\}\}$ non è numerabile. A maggior ragione, non sono numerabili gli insiemi delle funzioni:

- $f : \mathbb{N} \to \mathbb{N}$

- $f : \mathbb{N} \to \mathbb{R}$
- $f : \mathbb{R} \to \mathbb{R}$
- $f : N^k \to N^j$

Il problema della rappresentazione
L'informatica rappresenta tutte le sue entità (quindi anche gli algoritmi) in forma digitale, come sequenze finite di simboli di alfabeti finiti.

Il concetto di algoritmo
Un Algoritmo è una sequenza finita di operazioni, completamente e univocamente determinate.

La formulazione di un algoritmo dipende dal modello di calcolo utilizzato.

- Programma per un modello matematico astratto, come una Macchina di Turing
- Algoritmo per in pseudocodice per RAM
- Programma in linguaggio C per un PC

Qualsiasi modello si scelga, gli algoritmi devono esservi descritti, ossia rappresentati da sequenze finite di caratteri di un alfabeto finito ⇒ Gli gli algoritmi sono possibilmente infiniti, ma numerabili e possono essere elencati (messi in corrispondenza biunivoca con linsieme dei numeri naturali).

Drastica perdita di potenza

Gli algoritmi sono numerabili e sono meno dei problemi computazionali, che hanno la potenza del continuo $|\{Algoritmi\} \mid < < \mid \{Problemi\}|$. Esistono problemi privi di un corrispondente algoritmo di calcolo.

Problema indecibile (Problema dell'arresto)

Come abbiamo giá detto esistono problemi non calcolabili, i problemi che si presentano spontaneamente sono tutti calcolabili. Non è stato facile individuare un problema che non lo fosse: **Turing** (1930): Problema dell'arresto.

Problema dell'arresto

È un problema posto in forma decisionale $Arresto : \{Istanze\} \to \{0, 1\}$. Per i problemi decisionali, la calcolabilità è in genere chiamata *decidibilità*. Presi ad arbitrio un algoritmo A e i suoi dati di input D, decidere in tempo finito se la computazione di A su D termina o no.
È un algoritmo che indaga sulle proprietà di un altro algoritmo, trattato come dato di input. È legittimo: possiamo usare lo stesso alfabeto per codificare algoritmi e i loro dati di ingresso (sequenze di simboli dell'alfabeto). Una stessa sequenza di simboli può essere quindi interpretata sia come un programma, sia come un dato di ingresso di un altro programma.
Un algoritmo A, comunque formulato, può operare sulla rappresentazione di un altro algoritmo B Possiamo calcolare A(B). In particolare può avere senso calcolare A(A).

Consiste nel chiedersi se un generico programma termina la sua esecuzione oppure "va in ciclo", ovvero continua a ripetere la stessa sequenza di istruzioni all'infinito (supponendo di non avere limiti di tempo e memoria).

Congettura di Goldbach

"Ogni numero intero pari $n \geq 4$ è la somma di due numeri primi".

Congettura falsa $\rightarrow$ Goldbach() si arresta.
Congettura vera $\rightarrow$ Goldbach() NON si arresta.

Teorema

Turing ha dimostrato che riuscire a dimostrare se un programma arbitrario si arresta e termina la sua esecuzione non è solo un'impresa ardua, ma in generale è IMPOSSIBILE!
Segue che il problema dell'arresto è *indecidibile*.

Dimostrazione

Se il problema dell'arresto fosse decidibile, allora esisterebbe un qualsiasi algoritmo ARRESTO che presi A e D come dati di input determina in tempo finito le risposte ARRESTO(A,D) = 1 se A(D) termina ARRESTO(A,D) = 0 se A(D) non termina.

Osservazione

L'algoritmo ARRESTO non può consistere in un algoritmo che simuli la computazione A(D). Se A non si arresta su D, ARRESTO non sarebbe in grado di rispondere "NO" (0) in tempo finito.

Dimostrazione

In particolare possiamo scegliere $D = A$, cioè considerare la computazione A(A), $ARRESTO(A, A) = 1 \iff A(A)$ termina.
Se esistesse l'algoritmo ARRESTO, esisterebbe anche il seguente algoritmo.

```
PARADOSSO(A) {
   while (ARRESTO(A,A)) {
      ;
   }
}
```

L'ispezione dell'algoritmo PARADOSSO mostra che $PARADOSSO(A)$ termina $\iff x = ARRESTO(A, A) = 0 \iff A(A)$ non termina.

Ma cosa accadrebbe se calcolassimo PARADOSSO(PARADOSSO)?

$PARADOSSO(PARADOSSO)$ termina $\iff$
$x = ARRESTO(PARADOSSO, PARADOSSO) = 0 \iff$
$PARADOSSO(PARADOSSO)$ non termina, ma questa é una contraddizione!
L'unico modo di risolvere la contraddizione è che l'algoritmo PARADOSSO non possa esistere dunque non può esistere nemmeno l'algoritmo ARRESTO.
Segue che il problema dell'arresto è *indecidibile*.

Osservazione

Non può esistere un algoritmo che decida in tempo finito se un algoritmo arbitrario termina la sua computazione su dati arbitrari ciò non significa che non

si possa decidere in tempo finito la terminazione di algoritmi particolari. Il problema è indecidibile su una coppia $< A, D >$ scelta arbitrariamente.
L'algoritmo ARRESTO costituirebbe uno strumento estremamente potente, permetterebbe infatti di dimostrare congetture ancora aperte sugli interi (esempio: la congettura di Goldbach).

Lezione di Turing

Non esistono algoritmi che decidono il comportamento di altri algoritmi esaminandoli dall'esterno, cioè senza passare dalla loro simulazione.

Il decimo problema di Hilbert

Esistono risultati di non calcolabilità relativi ad altre aree della matematica, tra cui la teoria dei numeri e l'algebra.
Un'**equazione diofantea** è un'equazione della forma $p(x_1, x_2, ..., x_n) = 0$ dove p é un polinomio a coefficienti interi. Data un'arbitraria equazione diofantea, di grado arbitrario e con un numero arbitrario di incognite $p(x_1, x_2, ..., x_n) = 0$ stabilire se p ammette soluzioni intere. Il decimo problema di Hilbert non è calcolabile.

Modelli di calcolo

I linguaggi di programmazione esistenti sono tutti equivalenti? Ce ne sono alcuni più potenti e/o più semplici di altri? Ci sono algoritmi descrivibili in un linguaggio, ma non in un altro? È possibile che problemi oggi irrisolvibili possano essere risolti in futuro con altri linguaggi o con altri calcolatori? La teorie della calcolabilità e della complessità dipendono dal modello di calcolo?

La tesi di Church-Turing

Tutti i (ragionevoli) modelli di calcolo risolvono esattamente la stessa classe di problemi dunque si equivalgono nella possibilità di risolvere problemi, pur operando con diversa efficienza.
La decidibilità è una proprietà del problema. Incrementi qualitativi alla struttura di una macchina, o alle istruzioni di un linguaggio di programmazione, servono solo ad abbassare il tempo di esecuzione e rendere più agevole la programmazione.

16 Macchine di Turing

La Macchina di Turing è una macchina ideale che consente di calcolare funzioni, data una Macchina di Turing che calcola una funzione f è possibile specificare un input ed ottenere, al termine della computazione, l'output.
La tesi di Church afferma che tutte le funzioni effettivamente calcolabili siano esprimibili con una Macchina di Turing.

Una MdT è definita da:

- Nastro
- Testina

- Stato interno
- Programma
- Stato iniziale

Il nastro

Il nastro é infinito e suddiviso in celle, in una cella può essere contenuto un simbolo preso da un alfabeto opportuno e una cella deve contenere un simbolo che può appartenere all'alfabeto oppure essere un simbolo speciale, un alfabeto è semplicemente un insieme di simboli.

Lo stato interno e la testina

La macchina è dotata di una testina di lettura/scrittura, che è in grado di leggere e scrivere il contenuto della cella del nastro su cui si trova. É dotata uno stato interno, che è un elemento appartenente all'insieme degli stati.

Il programma di una MdT

Il comportamento della macchina è determinato da un insieme di regole. Una regola ha la forma seguente: (A, a, B, b, dir). Una regola viene applicata se lo stato corrente della macchina è A e il simbolo letto dalla testina è *a*. L'applicazione della regola cambia lo stato in B, scrive sul nastro b ed eventualmente sposta la testina di una cella a sinistra o a destra *(dir)*.

Il funzionamento di una Mdt

La macchina opera come segue:

1. Determina la regola da applicare in base allo stato interno e al simbolo corrente (quello letto dalla testina).
2. Se esiste una tale regola cambia lo stato, scrive il simbolo sulla cella corrente si sposta come indicato dalla regola.
3. Se non esiste la regola l'esecuzione termina

In questo modello non può esistere più di una regola per uno stato ed un simbolo corrente.

La notazione mediante le regole non è l'unica possibile per esprimere un programma, una notazione più compatta utilizza i grafi: un insieme di nodi collegati da archi opportunamente collegati. Ad ogni stato della macchina si associa un nodo del grafo, se una regola provoca la transizione in un altro stato esiste un arco tra i due nodi annotato col simbolo letto, quello scritto e la direzione.

Algoritmo

Il procedimento utilizzato per esprimere il calcolo di una funzione è anche detto algoritmo. Esso specifica le operazioni necessarie per risolvere un problema ma non è necessariamente espresso in un linguaggio formale (es. Un programma per la MdT), dato un algoritmo è possibile scrivere un programma che sia eseguibile da una macchina. Un problema è normalmente specificato in modo parametrico

nell'input, detto istanza del problema. Un algoritmo specifica i passi necessari per risolvere un'istanza del problema.

Riutilizzare un programma

Un problema complesso spesso può essere scomposto in sottoproblemi che vengono risolti separatamente. Capita spesso che esistano già algoritmi per i sottoproblemi identificati. È quindi importante individuarli per non "reinventare la ruota". Se si dispone anche della sua codifica, questa può essere inclusa risparmiando tempo e sforzi per la realizzazione del programma.

MdT Universale

Il comportamento della macchina è specificato ed è a tutti gli effetti un algoritmo. È possibile programmare una macchina di Turing in una Macchina di Turing? La risposta è affermativa e la macchina interprete è nota come **MdT Universale** che prende come input la specifica di una macchina e l'input per questa e calcola la funzione della macchina input sul dato specificato.

Stati, simboli e direzione

I simboli della macchina sono: "0", "1". Il blank è rappresentato con "N", gli stati invece sono rappresentati come sequenze di "S" (es. S, SS, SSS,...). Utilizziamo tre simboli per codificare la direzione: "0", "1" e "N", come vedremo l'uso degli stessi caratteri per rappresentare i simboli e le direzioni non crea ambiguità.

La codifica delle regole

Una regola viene codificata semplicemente giustapponendo i simboli che la compongono, le regole vengono semplicemente elencate una di seguito all'altra e prima di esse viene inserito il simbolo "B". Al termine delle regole, prima dell'input, è presente il simbolo "I".

Codifica dell'input

Il nastro della macchina è semi-infinito a destra (è equivalente ad un nastro infinito). Dopo il simbolo "I" segue il contenuto del nastro eccezion fatta per la testina indicata col simbolo "T" che legge/scrive il simbolo immediatamente a destra. Un possibile nastro di input è: I 100T01101010N

Il ciclo della macchina

Lo stato iniziale è lo stato "S". Durante il calcolo lo stato corrente e il simbolo letto saranno alla sinistra del simbolo "B", il *ciclo* della macchina è il seguente:

- Si legge il simbolo corrente
- Si cerca una regola applicabile
- Se la regola esiste la si applica scrivendo un simbolo e spostando la testina sul nastro

Numeriamo le MdT

Per associare un numero in modo univoco ad ogni Macchina basta dare un valore numerico ad ogni simbolo del programma: $\{S \to 2, 0 \to 0, 1 \to 1, N \to 3, B \to$

4, I $\rightarrow$ 5}. Per esempio il programma B S0S11 S1S01 I si puo scrivere come il numero 430311313015.

Goëdelizzazione

È possibile realizzare codifiche che associano ad ogni numero una macchina o un input sempre invertibili. Il procedimento di enumerare tutte le Macchine di Turing è noto col termine di Göedelizzazione. ATTENZIONE! Le macchine di Turing sono tante quante i numeri naturali: sappiamo ottenere una macchina dato un numero e viceversa.

Funzioni non calcolabili

Sappiamo che un sottoinsieme delle funzioni da $\mathbb{N}$ a $\mathbb{N}$ non è numerabile, quindi le funzioni sono più dei numeri naturali. Ne consegue che esistono funzioni che non sono calcolabili poiché le macchine di Turing sono quanti i numeri naturali.

17 Problemi intrattabili

```
TorriHanoi(n,p,t,s) {
  if (n==1) print p->t;
  else {
    TorriHanoi(n-1,p,s,t);
    print p->t;
    TorriHanoi(n-1,s,t,p);
  }
}
```

Costo in tempo

$$M(n) = \begin{cases} 1 & n = 1 \\ 2M(n-1)+1 & n > 1 \end{cases}$$

Dimostrazione

Vogliamo dimostrare per induzione che $M(n) = 2M(n-1)+1 = 2^n - 1$.

Procediamo per *induzione* su n.

Ipotesi induttiva: $M(n-1) = 2^{n-1} - 1$.
Caso Base: $n = 1 \Longrightarrow M(1) = 1 \Longrightarrow M(1) = 2^1 - 1 = 1$.
Passo induttivo: $M(n) = 2M(n-1)+1 = 2(2^{n-1}-1)+1 = 2^n - 2 + 1 = 2^n - 1$.

Generazione disequenze binarie

Prendiamo un insieme qualsiasi $A = \{a_0, a_1, a_2, ..., a_{n-1}\}$ con n elementi e un suo sottoinsieme $S \subseteq A$, $S = \{a_1, a_2, a_4\}$. Questo sottoinsieme si può descrivere tramite il vettore caratteristico B_s con la stessa dimensione n.

$B_S =$

0	1	1	0	1	0	0	0

Avremo che:

$$B_S[i] = \begin{cases} 1 & a_i \in S \\ 0 & a_i \notin S \end{cases}$$

Esisteranno quindi 2^n sequenze binarie di n bit.

```
GeneraBinarie (B,i,n) {
  if (n==1) Elabora (B,n);
  else {
    B[i]=0;
    GeneraBinarie(B, i+1,n);
    B[i]=1;
    GeneraBinarie(B, i+1,n);
  }
}
```

Algoritmo enumerativo per il problema dello zaino

```
Zaino (v,w,W,n) {
  vmax=0;
  sol=nuovo array di dim n;
  B=nuovo array di dim n;
  GeneraBINARIE (B,0,n,v,w,W,vmax,sol);
  print vmax;
  print_array (sol);
}

Elabora (B,n,v,w,W,vmax,sol) {
  peso=0;
  valore=0;
  for (i=0; i<n; i++) {
    peso=peso+B[i]*w[i];
    valore=valore+B[i]*w[i];
  }
  if (peso<=W && valore>vmax) {
    vmax=valore;
    "copio B in sol";
  }
}
```

Costo in tempo

$T(n) = \Theta(n)$.

La differenza tra il costo del problema dello zaino con la programmazione dinamica e la soluzione enumerativa é la seguente.
(PD) $T_{Zaino}(n, W) = \Theta(n \cdot W) \Rightarrow$ lineare in n ma esponenziale nel numero di cifre binarie di W.
(Sol. Enumerativa) $T_{Zaino}(n, W) = \Theta(n \cdot 2^n) \Rightarrow$ esponenziale in n.

Generazione di permutazioni

```
GeneraPermutazioni (P,i,n) {
  if (i==n-1) Elabora (P,n);
  else {
    for (j=1; j<n; j++) {
      scambia P[i] e P[j];
      GeneraPermutazioni (P,i+1,n);
      scambia P[i] e P[j];
    }
  }
}
```

Costo in tempo

$T(n) = \Theta(n! \cdot$ Costo di Elabora$)$.

Algoritmo enumerativo per C.H. (cammino hamiltoniano)

```
CH (Ag) {
  P=nuovo array che contiene gli n vertici di G;
  cammino=false;
  GeneraPermutazioni (P,0,n,Ag,cammino);
  if (cammino) {
    print "esiste un CH";
    print_array(P);
  }
  else print "G non contiene un CH";
}

GeneraPermutazioni (P,i,n,Ag,cammino) {
  if (i==n-1) {
    Elabora (P,n);
    if (cammino) return;
  }
  else {
    for (j=1; j<n; j++) {
      scambia P[i] e P[j];
      GeneraPermutazioni (P,i+1,n,Ag,cammino);
      if (cammino) return;
      scambia ... ;
    }
  }
}
```

```
Elabora (P,n,Ag,cammino) {
  for (i=0; i<n-1; i++) {
    if (Ag[P[i],P[i+1]]==0) return;
  }
  cammino=true;
}
```

Costo in tempo

$T(n) = O(n)$.

18 Teoria della complessitá

Esistono dei problemi (problema dell'arresto) che non possono essere risolti da nessun calcolatore, indipendentemente dal tempo a disposizione e prendono il nome di *problemi indecidibili*. Allo stesso modo ci sono problemi decidibili che possono richiedere tempi di risoluzione esponenziali nella dimensione dell'istanza (torri di Hanoi, generazione delle sequenze binarie e delle permutazioni) e prendono il nome di *problemi intrattabili*.
Esistono inoltre dei problemi che possono essere risolti mediante algoritmi di costo polinomiale (problemi trattabili "facili") e problemi di cui non conosciamo lo stato (problemi presumibilmente intrattabili). Per risolvere gli ultimi disponiamo solo di algoritmo di costo esponenziale nonostante nessuno sia mai stato in grado di dimostrare che non esistano algoritmi di costo polinomiale.

Algoritmi polinomiali ed esponenziali

Studiamo adesso la dimensione dei dati trattabili in funzione all'incremento della velocità dei calcolatori. Presi C_1 e C_2 (k volte più veloce di C_1) due qualsiasi calcolatori e t come tempo a disposizione, avremo che:

- n_1 = dati trattabili nel tempo t su C_1
- n_2 = dati trattabili nel tempo t su C_2

Osservazione

Usare C_2 per un tempo t, equivale a usare C_1 per un tempo $k * t$.

L'algoritmo **polinomiale** risolve il problema in $c \cdot n^s$ secondi (c, s costanti), e avremo che:

- $C_1 \Longrightarrow c \cdot n_1^s = t \rightarrow n_1 = (\frac{t}{c})^{\frac{1}{s}}$
- $C_2 \Longrightarrow c \cdot n_2^s = k \cdot t \rightarrow n_2 = (\frac{k \cdot t}{c})^{\frac{1}{s}} = k^{\frac{1}{s}} \cdot (\frac{t}{c})^{\frac{1}{s}}$

Segue che $n_2 = k^{\frac{1}{s}} \cdot n_1$ e abbiamo un miglioramento di un *fattore moltiplicativo* $k^{\frac{1}{s}}$.

L'algoritmo **esponenziale** risolve il problema in $c \cdot 2^n$ secondi (c costante) e avremo che:

- $C_1 \Longrightarrow c \cdot 2^{n_1} = t \rightarrow 2^{n_1} = \frac{t}{c}$
- $C_2 \Longrightarrow c \cdot 2^{n_2} = k \cdot t \rightarrow 2^{n_2} = \frac{k \cdot t}{c} = k \cdot 2^{n_1}$

Segue che $n_2 = n_1 + log_2 k$ e abbiamo un miglioramento di un *fattore moltiplicativo* $log_2 k$.

Tipologie di problemi

Dato un qualsiasi problema Π indicheremo con "I" l'insieme delle istanze in ingresso (INPUT) e con "S" l'insieme delle soluzioni.
Esistono molti tipi di problemi, tra i piú importanti abbiamo:

- Problemi decisionali
- Problemi di ricerca
- Problemi di ottimizzazione

Problemi decisionali

La teoria della complessità computazionale è definita principalmente in termini di problemi di decisione. Essendo la risposta binaria, non ci si deve preoccupare del tempo richiesto per restituire la soluzione e tutto il tempo è speso esclusivamente per il calcolo e la difficoltà di un problema è già presente nella sua versione decisionale. Molti problemi di interesse pratico sono però problemi di ottimizzazione, é possibile esprimere un problema di ottimizzazione in forma decisionale, chiedendo l'esistenza di una soluzione che soddisfi una certa proprietà. Il problema di ottimizzazione è quindi almeno tanto difficile quanto il corrispondente problema decisionale, caratterizzare la complessità di quest'ultimo permette quindi di dare almeno una limitazione inferiore alla complessità del primo.

Classi di complessità

Dato un problema decisionale P ed un algoritmo A, diciamo che A risolve P se, data un'istanza di input x $A(x) = 1 \Longleftrightarrow \Pi(x) = 1$. A risolve P in tempo $T(n)$ e spazio $S(n)$ se il tempo di esecuzione e l'occupazione di memoria di A sono rispettivamente $T(n)$ e $S(n)$.
Data una qualunque funzione $f(n)$, $Time(f(n))$ rappresenta l'insieme dei problemi decisionali che possono essere risolti in tempo $O(f(n))$ e $Space(f(n))$ rappresenta l'insieme dei problemi decisionali che possono essere risolti in spazio $O(f(n))$.

Algoritmo polinomiale (tempo)

Esistono due costanti c, $n_0 > 0$ t.c. il numero di passi elementari è al più $n \cdot c$ per ogni input di dimensione n e per ogni $n > n_0$.

La **Classe P** è la classe dei problemi risolvibili in spazio polinomiale nella dimensione n dell'istanza di ingresso.

Algoritmo polinomiale (spazio)

Esistono due costanti c, $n_0 > 0$ t.c. il numero di celle di memoria utilizzate è al più $n \cdot c$ per ogni input di dimensione n e per ogni $n > n_0$.
La **Classe PSPACE** è la classe dei problemi risolvibili in spazio polinomiale nella dimensione n dell'istanza di ingresso.

La **Classe Exp Time** è la classe dei problemi risolvibili in tempo esponenziale nella dimensione n dell'istanza di ingresso.

Relazioni tra classi

$P \subseteq PSPACE$ infatti un algoritmo polinomiale può avere accesso al più ad un numero polinomiale di locazioni di memoria diverse (in ordine di grandezza).
$PSPACE \subseteq EXPTIME$

Altri problemi interessanti

- Clique
- Cammino Hamiltoniano
- Soddisfacibilità di formule booleane (SAT)

Aloritmo per Clique

Si considerano tutti i sottoinsiemi di nodi, in ordine di cardinalità decrescente, e si verifica se formano una *clique* di dimensione almeno k. Se n è il numero di nodi, quanti diversi sottoinsiemi esamina l'algoritmo al caso peggiore? $\Rightarrow 2^n$.
Segue che $CLIQUE \in EXPTIME$ ma l'algoritmo polinomiale non é noto.

Aloritmo per Cammino Hamiltoniano

Si considerano tutte le permutazioni di nodi, e si verifica se i nodi in quell'ordine sono a due a due adiacenti. Se n è il numero di nodi, quante diverse permutazioni esamina l'algoritmo al caso peggiore? $\Rightarrow n!$.
Segue che $CAMMINOHAMILTONIANO \in EXPTIME$, ma l'algoritno polinomiale non è noto.

SAT

Dato un insieme V di variabili Booleane.

- Letterale: variabile o sua negazione.
- Clausola: disgiunzione (OR) di letterali.

Un'espressione Booleana su V si dice in forma normale congiuntiva (FNC) se è espressa come congiunzione di clausole (AND di OR di letterali).

Algoritmo per SAT

Si considerano tutti i 2^n assegnamenti di valore alle n variabili, e per ciascuno si verifica se la formula è vera.
Segue che $SAT \in EXPTIME$ ma l'algoritno polinomiale non è noto.

Non possiamo sapere se per questi tre algoritmi la *ricerca esaustiva* sia necessaria, l'unico modo per esserne certi é avere a disposizione un "metodo" che ci permette di risolvere il problema senza sfruttare tale ricerca.

Problemi decisionali e certificati

In un problema decisionale siamo interessati a verificare se una istanza del problema soddisfa una certa proprietà: esiste una clique di k nodi, un cammino hamiltoniano o un assegnamento di valori che rende vera la formula? Per alcuni problemi, per le istanze accettabili (positive) x è possible fornire un certificato y che possa convincerci del fatto che l'istanza soddisfa la proprietà e dunque è un'istanza accettabile.

Certificati

- Certificato per CLIQUE: sottoinsieme di k nodi, che forma la clique
- Certificato per Cammino Hamiltoniano: permutazione degli n nodi che definisce un cammino semplice
- Certificato per SAT: Un'assegnazione di verità alle variabili che renda vera l'espressione

Un certificato è un attestato breve di esistenza di una soluzione con determinate proprietà. Si definisce solo per le istanza accettabili e in generale, non è facile costruire attestati di non esistenza.

Verifica

IDEA: utilizzare il costo della verifica di un certificato (una soluzione) per un'istanza accettabile (positiva) per caratterizzare la complessità del problema stesso.
Un problema Π è verificabile in tempo polinomiale se:

- Ogni istanza accettabile x di Π di lunghezza n ammette un certificato y di lunghezza polinomiale in n.
- Esiste un algoritmo di verifica polinomiale in n e applicabile a ogni coppia $< x, y >$, che permette di attestare che x è accettabile.

Classe NP

NP è la classe dei problemi decisionali verificabili in tempo polinomiale dove P sta per polinomiale ed N sta per "non". La classe NP è la classe dei problemi risolvibili in tempo polinomiale *non deterministico*.

Osservazioni

Un certificato contiene un'informazione molto prossima alla soluzione, quindi qual è l'interesse di questa definizione? La teoria della verifica è utile per far luce sulle gerarchie di complessità dei problemi e non aggiunge nulla alla possibilità di risolverli efficientemente. Chi ha una soluzione può verificare in tempo

polinomiale che l'istanza è accettabile, mentre chi non ha una soluzione (certificato), può individuarla in tempo esponenziale considerando tutti i casi possibili con una *ricerca esaustiva*.

Riduzioni polinomiali

Dati $\Pi_1 =$ e $\Pi_2 =$ problemi decisionali e I_1 e $I_2 =$ insiemi delle istanze di input di $\Pi_1 =$ e Π_2.
Π_1 si riduce in tempo polinomiale a Π_2

$$\Pi_1 \leq_p \Pi_2$$

Se esiste una funzione $f : I_1 \to I_2$ calcolabile in tempo polinomiale tale che, per ogni istanza x di Π_1 x è un'istanza accettabile di $\Pi_1 \Leftrightarrow f(x)$ è un'istanza accettabile di Π_2. Se esistesse un algoritmo per risolvere Π_2 potremmo utilizzarlo per risolvere Π_1.

$$\Pi_1 \leq_p \Pi_2 \text{ e } \Pi_2 \in P \Rightarrow \Pi_1 \in P$$

Problemi NP-completi

Sono i problemi più difficili all'interno della classe NP. Se esistesse un algoritmo polinomiale per risolvere uno solo di questi problemi, allora tutti i problemi in NP potrebbero essere risolti in tempo polinomiale, e dunque $P = NP$. Quindi tutti i problemi NP-completi sono risolvibili in tempo polinomiale oppure nessuno lo è.

Un problema Π si dice **NP-arduo** se per ogni $\Pi' \in NP$, $\Pi' \leq_p \Pi$.

Un problema decisionale Π si dice **NP-completo** se $\Pi \in NP$ e per ogni $\Pi' \in NP$, $\Pi' \leq_p \Pi$.

Per dimostrare che un problema è in NP può essere facile basta esibire un certificato polinomiale, non è altrettanto facile dimostrare che un problema P è NP-arduo o NP-completo. Bisogna dimostrare che tutti i problemi in NP si riducono polinomialmente a P ma in realtà la prima dimostrazione di NP-completezza aggira il problema.

Teorema di Cook (1971)

SAT è NP completo.

IDEA: Cook ha mostrato che dati un qualunque problema Π in NP ed una qualunque istanza x per Π. Si può costruire una espressione Booleana in forma normale congiuntiva che descrive il calcolo di un algoritmo per risolvere Π su x. L'espressione è vera se e solo se l'algoritmo restituisce 1.

Un problema decisionale P è NP-completo se:

$$\Pi \in N \text{ e } SAT \leq_p \Pi$$

Infatti, per ogni $\Pi' \in NP$, $\Pi' \leq_p SAT$ e $SAT \leq_p \Pi$, quindi $\Pi' \leq_p \Pi$.

CLIQUE è NP completo

Sapendo che $SAT \leq_p CLIQUE$, data un'espressione booleana F in forma normale congiuntiva con k clausole costruire in tempo polinomiale un grafo G che contiene una clique di k vertici se e solo se F è soddisfacibile.

Riduzione

G contiene una *clique* $\Rightarrow$ F è soddisfacibile. Si dà valore 1 (true) ai k letterali che corrispondono ai nodi della clique, tutte le clausole corrispondenti diventano di valore 1 (true) e $F = 1$ (true) è soddisfacibile.
F è soddisfacibile $\Rightarrow$ G contiene una clique. Esiste almeno un letterale vero per ogni clausola ed i corrispondenti vertici in G formano una clique.
La riduzione da F a $G = (V, E)$ si esegue in tempo polinomiale:

- $n = \#variabili$
- $k = \#clausole$
- $|V| \leq n \cdot k$

L'esistenza di un arco si stabilisce in tempo costante ($|E| \leq O((n \cdot k)^2)$).

Problemi NP equivalenti

$SAT \leq_p CLIQUE \Rightarrow CLIQUE$ è NP completo
SAT è NP completo $\Rightarrow CLIQUE \leq_p SAT$
SAT e CLIQUE sono NP equivalenti. Tutti i problemi NP completi sono tra loro NP equivalenti. Sono tutti facili, o tutti difficili.

Problemi di ottimizzazione NP- hard

Se la soluzione ottima è troppo difficile da ottenere, una soluzione quasi ottima ottenibile facilmente forse è buona abbastanza. A volte, avere una soluzione esatta non è strettamente necessario e ci si accontenta di una soluzione che non si discosti troppo da quella ottima e si possa calcolare in tempo polinomiale.

Gerarchia delle classi