

Algoritmi e Strutture dati

Federico Zhou

December 10, 2021

1 Introduzione e motivazioni

Un algoritmo è un insieme ordinato e finito di passi eseguibili non ambigui che definiscono un procedimento che termina. Le operazioni di analisi sugli algoritmi sono basati sull'analisi di *correttezza, complessità, efficienza*, basate sul *tempo, spazio, costo*.

Per analizzare gli algoritmi utilizziamo le notazioni asintotiche, che sono: O , Ω e Θ .

I parametri su cui facciamo le analisi sono:

- Complessità in tempo: tempo richiesto per terminare l'esecuzione
- Complessità in spazio: quantità di spazio richiesto per terminare l'esecuzione.

Dobbiamo tuttavia parlare prima del modello di calcolo: quando facciamo l'analisi del tempo e dello spazio, possiamo farlo contando la singola operazione come un singolo passo, indipendentemente dalla dimensione degli operandi. Questo tuttavia è irrealistico, l'algoritmo che calcola fibonacci cresce in maniera esponenziale. Per ovviare a questo problema è stato introdotto il modello di calcolo del costo logaritmico che assume che il costo di esecuzione delle istruzioni dipenda dalla dimensione degli operandi coinvolti.

- Criterio del costo uniforme
 - Tempo: ogni istruzione utilizza un'unità di tempo indipendentemente dalla grandezza degli operandi
 - Spazio: ogni variabile utilizza un'unità di spazio indipendentemente dalla grandezza del valore
- Criterio del costo logaritmico
 - Tempo: ogni operazione utilizza un tempo proporzionale alla lunghezza dei valori
 - Spazio: ogni variabile utilizza un'unità di spazio dipendente dalla rappresentazione del dato

Gli algoritmi con cui lavoriamo possono avere tempo:

- polinomiale, algoritmi che lavorano in un tempo limitato da un polinomio, ragionevoli e praticabili
- esponenziali, algoritmi che lavorano in tempo esponenziale, non praticabili.

2 Algoritmi di ordinamento

Gli algoritmi di ordinamento sono tantissimi, ma si possono suddividere principalmente in 2 categorie: ad ordinamento interno se l'ordinamento avviene su dati in memoria centrale, e a ordinamento esterno se l'ordinamento avviene su dati in memoria di massa.

Gli algoritmi di ordinamento che analizziamo noi vengono applicati su array:

- gli array sono strutture di dati statiche, con elementi omogenei
- gli elementi in un array sono salvati consecutivamente, e sono individuati in base alla loro posizione rispetto al primo elemento.
- l'accesso in un array è diretto, l'operazione di lettura e modifica ha complessità $O(1)$

Gli algoritmi che studieremo si basano sul confronto di chiavi, in quanto è l'operazione più costosa. Studieremo lo spazio: la memoria occupata nello stack, ed il tempo: la complessità di tempo

Stabilità in un algoritmo:

Un algoritmo è detto stabile se preserva l'ordine relativo tra i record con medesima chiave. I record che hanno una stessa chiave vengono lasciati nell'ordine originale.

Operabilità "sul posto":

Quando parliamo di un algoritmo che "opera sul posto" parliamo di un algoritmo che non ha bisogno di una struttura d'appoggio. Alcuni esempi sono il MergeSort, che utilizza un array d'appoggio

2.1 Algoritmi elementari

In particolare gli algoritmi elementari sono SelectionSort, InsertionSort, BubbleSort, sono algoritmi che utilizzano $\Theta(n^2)$ confronti. Iniziamo da SelectionSort e InsertionSort dato che lavorano in maniera simile:

- all'inizio di ciascun passo l'array contiene una parte ordinata, ed una parte da ordinare
- alla fine del passo il segmento ordinato contiene un elemento in più, alla fine dei passi totali l'array è ordinato.

- SelectionSort

Cerco l'elemento più piccolo nella porzione non ordinata e lo metto in coda alla porzione ordinata. L'indicatore della porzione ordinata avanza di uno, per $n - 1$ volte.

Codice 2.1 Ordinamento per selezione

```

Algoritmo selectionSort (array A[0..n - 1])
  for k ← 0 to n - 2 do
    // ricerca del minimo in A[k..n - 1]
    m ← k                                // m indica la posizione del minimo
    for j ← k + 1 to n - 1 do
      if A[j] < A[m] then m ← j
    scambia A[m] con A[k] // sistema il minimo nella sua posizione definita k
  
```

SelectionSort è un algoritmo stabile (perché utilizza < al posto di <=, la sua complessità in tempo è la medesima sia nel caso peggiore che nel caso migliore, ovvero $\Theta(n^2)$)

- InsertionSort

Prendo un elemento nella parte non ordinata e lo inserisco nella posizione giusta all'interno della parte ordinata. Sposto di avanti un indice la parte ordinata.

Codice 2.2 Ordinamento per inserimento

```

Algoritmo insertionSort (array A[0..n - 1])
  for k ← 1 to n - 1 do
    x ← A[k]                                // elemento da inserire in A[0..k - 1]
    // ricerca da destra la posizione in cui inserire x,
    // spostando man mano in avanti gli elementi maggiori
    j ← k - 1
    while j ≥ 0 and A[j] > x do
      A[j + 1] ← A[j]                      // sposta in avanti l'elemento A[j]
      j ← j - 1
    A[j + 1] ← x                            // inserisce x
  
```

InsertionSort è un algoritmo stabile, nel caso peggiore la complessità di tempo è $\Theta(n^2)$ ma nel caso migliore il numero di confronti da fare è $n - 1$

- BubbleSort

Scansiona l'array e confronto coppie adiacenti ripetutamente, scambio gli elementi se sono in ordine diverso. Alla fine dell'ultimo round l'elemento più grande finisce sul fondo.

Codice 2.4 Ordinamento “a bolle” (versione migliorata)

```

Algoritmo bubbleSort (array A[0..n - 1])
  i ← 1
  do
    scambiato ← false          // per ricordare se durante la scansione corrente
                                // è stato fatto almeno uno scambio
    for j ← 1 to n - i do
      if A[j] < A[j - 1] then
        scambia A[j - 1] con A[j]
        scambiato ← true
    i ← i + 1
  while scambiato and i < n
  
```

BubbleSort è un algoritmo stabile, nel caso peggiore la complessità di tempo è $\Theta(n^2)$ ma nel caso migliore il numero di confronti da fare è $n - 1$

2.2 Algoritmi avanzati

In particolare gli algoritmi avanzati sono MergeSort, QuickSort, HeapSort, sono algoritmi che utilizzano $\Theta(n \log n)$ confronti.

- MergeSort

Dividi l'array in porzioni unitarie, ordina ordina ricorsivamente le porzioni adiacenti fino a ricostruire l'array ordinato.

Codice 3.1 Schema dell'ordinamento per fusione

```
Algoritmo mergeSort (array A[0..n - 1])
  if n > 1 then
    m  $\leftarrow n/2$ 
    B  $\leftarrow A[0..m - 1]$ 
    C  $\leftarrow A[m..n - 1]$ 
    mergeSort(B)
    mergeSort(C)
    A  $\leftarrow \text{merge}(B, C)$ 
```

MergeSort è un algoritmo stabile, non opera sul posto ed è più efficiente degli algoritmi visti in precedenza.

Per funzionare l'algoritmo MergeSort ha bisogno di un metodo che ci permetta di fare la fusione tra 2 array, in particolare dati due array *B, C* dobbiamo avere un array *A* che sia uguale all'unione tra *B, C*.

```
Algoritmo merge (Array B[0, ..., n-1], Array C[0, ..., n-1])  $\rightarrow$  array
```

- QuickSort

1. Scegli un elemento *n* di *A* come perno
2. Inserisci tutti gli elementi $n \leq a$ in un array *B*
3. Inserisci tutti gli elementi $n > a$ in un array *C*
4. Ordina *B* e *C*
5. Unisci *B* e *C*

Codice 4.1 Quicksort: schema ad alto livello

Nello schema si utilizza impropriamente la notazione insiemistica per semplicità e per enfatizzare il fatto che l'ordine con cui vengono collocati gli elementi in ciascuna delle due parti B e C non è importante.

```
Algoritmo quickSort (array A)
if lunghezza di A > 1 then
    scegli un elemento di A come perno
    B ← { $y \in A \mid y \leq \text{perno}$ }
    C ← { $y \in A \mid y > \text{perno}$ }
    quickSort(B)
    quickSort(C)
    A ← concatenazione di B e C
```

QuickSort è il primo algoritmo instabile che incontriamo. Diversamente da MergeSort, QuickSort opera sul posto utilizzando due variabili aggiuntive.

Nel caso ottimo l'algoritmo ha un tempo

$$T(n) = O(n * \log n)$$

nel caso pessimo invece

$$T(n) = O(n^2)$$

Nel caso medio QuickSort opera in $T(n) = O(n * \log n)$. In media QuickSort è più efficiente di InsertionSort, SelectSort, BubbleSort e ha prestazioni peggiori di MergeSort solo nel caso peggiore. Da verifiche sperimentali possiamo concludere che QuickSort è l'algoritmo di ordinamento per array più efficiente. Possiamo inoltre concludere che in genere $T(n) = O(n * \log n)$ è il tempo migliore possibile per gli algoritmi di ordinamento basati sul confronto

3 Tecniche di risoluzione

3.1 Tecniche Divide-Et-Impera

La tecnica del divide et impera è una tecnica di progettazione degli algoritmi suddivisibile in 3 fasi:

- dividere l'istanza dei dati in ingresso in più sottoistanze disgiunte
- risolvere ricorsivamente i problemi in ciascuna sottoistanza separatamente
- combinare le soluzioni delle sottoistanze per ricavare la soluzione dell'istanza iniziale.

L'efficienza del metodo dipende dal metodo di decomposizione e ricomposizione delle soluzioni parziali. Questa tecnica adotta un approccio top-down, i principali esempi che vediamo noi sono MergeSort e QuickSort. Una generalizzazione della tecnica può essere la seguente:

```
Algoritmo risolviP (Instanza I) -> Soluzione
if (I <= C) then ove C è una costante qualsiasi
    risolvi P
    return soluzione
else
    dividì I in b Istanze con lunghezza minore
    sol1 <- risolviP(I1)
    ...
    soln <- risolviP(In)
    return combina(sol1, ..., soln)
```

Gli esempi che abbiamo visto durante il corso di applicazione di tecniche divide-et-impera sono:

- Algoritmo ricorsivo per calcolare il massimo ed il minimo subarray di un array di n interi
L'array iniziale è suddiviso in 3 subarray disgiunti di $n/2$ interi, e ricorsivamente calcoliamo il massimo o il minimo di ciascun subarray.
- Algoritmo di ricerca binaria in un array di n interi
L'algoritmo divide l'array in due subarray di grandezza $n/2$ ricorsivamente, procedendo solo in uno dei due subarray.
- Mergesort per array di n elementi
L'algoritmo divide l'array in due subarray di grandezza $n/2$ ricorsivamente, e riordina ciascuno dei subarray, riunendoli alla fine
- Quicksort per array di n elementi
L'algoritmo divide l'array in tre subarray disgiunti composti rispettivamente dagli elementi \leq , $=$ e \geq . Il primo ed il terzo array sono riordinati ricorsivamente, e vengono poi concatenati i subarrays.

3.2 Dynamic Programming — Programmazione dinamica

La programmazione dinamica è una tecnica di progettazione di algoritmi che diversamente dal divide-et-impera opera secondo una logica bottom-up; la risoluzione si basa sulla risoluzione di sottoproblemi progressivamente più grandi. Ne consegue che per tenere conto delle soluzioni ai sottoproblemi la tecnica di programmazione dinamica utilizza un'apposita tabella.

Possiamo dividere la tecnica di programmazione dinamica in quattro fasi:

- Identificazione dei sottoproblemi del problema iniziale
- Predisporre una tabella ove memorizzare dinamicamente le soluzioni ai sottoproblemi a partire dai sottoproblemi più semplici, la cui soluzione è immediata
- Si utilizzano le soluzioni dei sottoproblemi precedentemente risolti per risolvere sottoproblemi più difficili, con conseguente consultazione e aggiornamento della tabella
- Si costruisce la soluzione finale a partire dalle soluzioni dei sottoproblemi risolti.

La programmazione dinamica, come negli algoritmi greedy, rappresentano problemi di ottimizzazione e seguono il principio di ottimalità, che garantisce che se una soluzione è ottima, anche le porzioni che vanno a comporre la situazione sono ottime. Gli esempi che abbiamo visto durante il corso di applicazione di tecniche programmazione dinamica sono:

- Allgoritmo di Floyd-Warshall per un grafo di n vertici. Identifica come sottoproblemi quelli di calcolare i percorsi più brevi tra tutte le coppie di vertici del grafo che passano per vertici di indice $\leq k$ per ogni $1 \leq k < n$, predispone una tabella di $n + 1$ elementi (che sono delle matrici quadrate di ordine n) inizializzando il primo elemento in base alla matrice di adiacenza del grafo, avanza di un elemento alla volta determinandone il valore tramite una specifica formula applicata al valore dell'elemento precedente, e restituisce come risultato il valore dell'ultimo elemento. In pratica, non serve predisporre una tabella di matrici in quanto è sufficiente un'unica matrice per effettuare tutti i calcoli.
- Algoritmo per la ricerca di un sottovettore di somma massima/minima
L'algoritmo lavora creando due indici di supporto, che designano rispettivamente l'inizio e la fine del subarray che contiene il valore massimo. Viene creato un array V di supporto che calcola i valori su cui effettuare le verifiche.

```

Algoritmo sottovettoreMax(Array V[1, ..., n]) -> (intero, intero):
Sia S[1, ..., n] un vettore          V = [1, 2, -4, 7 -2, 3, -1], S[]
S[1] = V[1]                          S[1] = 1
max = S[1], fine = 1                max = 1, fine = 1
// popolazione dell'array di supporto
for i = 2 to n do
    if S[i-1] >= 0 then            S[1] >= 0
        S[i] = S[i-1] + V[i]       S[2] = S[1] + 2 = 3
    else
        S[i] = V[i]
    if S[i] > max then           S[2] > 1
        max = S[i]                 max = 3
        fine = i                   fine = 2
// ricerca a ritroso dell'inizio dell'array massimo
inizio = fine
while S[inizio] != V[inizio] do
    inizio = inizio - 1
return(inizio, fine)

```

Considero sottoproblemi $P(1), \dots, P(n)$ via via più difficili. Risolvo i sottoproblemi a partire dal più semplice $P(1)$, e da questi ricavo la soluzione di P . In questo caso $P(i)$ è trovare il sottovettore di somma massima che termina in posizione i , mentre la soluzione di P è la scelta tra le soluzioni di $P(1), \dots, P(N)$
Il tempo totale dell'algoritmo è dell'ordine di $T(n) = O(n)$

- Algoritmo per la ricerca di un cammino di valore minimo su matrice
Data una matrice $n * n$ di interi determinare un cammino dalla prima all'ultima colonna di valore minimo minimo tale che
 1. l'inizio sia una posizione qualsiasi sulla prima colonna
 2. per raggiungere la colonna successiva può muoversi sulla riga sopra, sulla stessa riga, o sulla riga sotto
 3. la fine sia una posizione qualsiasi sull'ultima colonna

Per impostare il problema con la programmazione dinamica identifichiamo:

- P : trovare il cammino di valore minimo in una matrice $n * n$
- $P(i, j)$: trovare il cammino di valore minimo che inizia nella colonna 1, e termina nella posizione (i, j)

La soluzione di P si ricava dalle soluzioni di $P(1, n), P(2, n), \dots, P(n, n)$.

Matrice C dei risultati dei problemi P(i,j)

Il valore $C[i, j]$ rappresenta il costo del cammino minimo che inizia nella posizione 1, e termina nella posizione (i, j) , in particolare la sua formula è:

$$C[i, j] = M[i, j] + \min\{C[i - 1, j - 1], C[i, j - 1], C[i + 1, j - 1]\}$$

Ovvero il valore della matrice iniziale sommata al valore minimo tra i valori raggiungibili.

```
Algoritmo camminoMinimo(matrice M[1 ... n, 1 ... n]) -> intero
Sia C[1 ... n, 1 ... n] una matrice
//popolazione della prima colonna con prima colonna iniziale
for i = 1 to n do
    C[i, 1] = M[i, 1]
//calcolo valore per ogni colonna > 1
for j = 2 to n do
    for i = 1 to n do
        min = C[i, j-1]
        if i > 1 && C[i-1, j-1] < min then min = C[i-1, j-1]
        if i < n && C[i+1, j-1] < min then min = C[i+1, j-1]
        C[i, j] = M[i, j] + min
//selezione del valore minimo
min = C[1, n]
for i = 2 to n do
    if C[i, n] < min then
        min = C[i, n]
return min
```

Il tempo totale dell'algoritmo è dell'ordine di $T(n) = O(n^2)$

3.3 Greedy Algorithms — Tecnica golosa

La tecnica golosa si applica ai problemi di ottimizzazione in cui la soluzione è costruita scegliendo, ad ogni passo, l'elemento migliore nel momento. La soluzione che ricaviamo non è sempre la soluzione ottima, è tuttavia la soluzione più veloce da computare. Gli esempi che abbiamo visto durante il corso di applicazione di tecniche greedy sono:

- Algoritmo di Kruskal per grafo ricoprente minimo
L'algoritmo di Kruskal costruisce la soluzione considerando come candidati i singoli archi del grafo, scegliendo ad ogni passo l'arco di peso minimo tra i rimasti, ritenendo i singoli archi ammissibili se i vertici appartengono a due alberi diversi tra quelli costruiti fino a quel momento
- Algoritmo di Prim per grafo ricoprente minimo
L'algoritmo di Prim costruisce la soluzione partendo da un grafo iniziale contenente un solo vertice s , considerando come candidati i rimanenti vertici del grafo scegliendo ad ogni passo quelli con distanza minima dall'albero libero costruito fino a quel momento.
- Algoritmo di Dijkstra per il calcolo di cammino minimo tra un vertice s e tutti gli altri
L'algoritmo prende come soluzione iniziale un insieme vuoto (con input iniziale un vertice s), considerando come candidati i singoli vertici del grafo, scegliendo tra quelli rimasti quelli con distanza minima dal vertice sorgente.

4 Tipi di dati

Il tipo di una variabile rappresenta una variabile che stabilisce i valori che il dato può assumere e le operazioni che essa può svolgere. Le varie strutture dati che andiamo ad analizzare sono riconducibili a quattro classi fondamentali, ovvero: *array*, *liste*, *alberi* e *grafi*. Le strutture dati possono essere caratterizzate dalla loro occupazione di memoria, che può essere statica, o dinamica.

4.1 Dizionario — HashMap

Il dizionario è una collezione di elementi ciascuno dei quali è caratterizzato da una **chiave**. Le operazioni che possiamo svolgere su un dizionario sono *la ricerca*, *l'inserimento* e *la cancellazione* di un elemento.

Un dizionario può essere implementato utilizzando un array:

- se l'array è ordinato in base la chiave la ricerca ha costo $\Theta(\log n)$ e l'inserimento costa $\Theta(n)$
- se l'array invece non è ordinato la ricerca costa $\Theta(n)$ e l'inserimento costa $\Theta(1)$

Un dizionario può essere anche implementato utilizzando collezioni, ovvero strutture statiche (con dimensione fissata all'inizializzazione), indicizzate ad esempio *vectors* e *arraylists*; oppure utilizzando strutture collegate, dinamiche ad esempio *liste* e *alberi*

4.2 Liste concatenate

Le liste concatenate sono un insieme di nodi collegati linearmente tra di loro. In particolare ogni nodo contiene:

- un dato della collezione
- informazioni per accedere al prossimo nodo
- informazioni per accedere al nodo precedente (nel caso delle doppiamente concatenate)

```
typedef struct elem_lista
{
    int             valore;
    struct elem_lista *succ_p;
} elem_lista_t;

typedef struct elem_lista_dc
{
    int             valore;
    struct elem_lista_dc *succ_p, *prec_p;
} elem_lista_dc_t;
```

I tipi di problemi che possiamo andare a svolgere sulle liste sono:

- problema della visita: data una lista attraversare tutti gli elementi una volta
- problema della ricerca, data una lista ed un valore, stabilire se il valore è contenuto all'interno della lista, e l'indirizzo in caso affermativo
- problema dell'inserimento, data una lista ed un valore, inserire il valore nella posizione appropriata
- problema della rimozione, data una lista ed un valore, rimuovere il valore dalla lista

4.3 Pila

La pila, o stack, è una struttura dati che organizza i dati secondo una logica LIFO, il che significa che gli inserimenti e le rimozioni avvengono presso l'estremità della lista. Come in una lista la pila è individuata dall'indirizzo per primo elemento, chiamato cima. L'implementazione può avvenire con un array, o delle liste lineari. Le operazioni fondamentali che vengono svolte su una pila sono:

- isEmpty: che mi restituisce true se la pila è vuota, e false se la pila non lo è
- push: dato un valore, lo aggiunge sulla pila
- pop: dato un valore, lo rimuove dalla pila
- peek: mi restituisce il valore in cima, senza modificarla

4.4 Alberi con radice

Gli alberi con radice (o solamente albero) sono una struttura dati simile agli array, ma dotati di una rappresentazione gerarchica dei dati. I nodi possono essere definiti come padre, nodo interno, foglia e radice, mentre i collegamenti tra i nodi sono chiamati archi. In particolare quando parliamo di alberi possiamo identificare:

- la profondità come il numero massimo di nodi da attraversare partendo dalla radice alla foglia più distante
- grado di un nodo come il numero massimo di figli che un nodo può avere

I problemi classici che trattiamo per gli alberi sono: visita, ricerca, inserimento e rimozione.

4.4.1 Alberi binari

Un albero è detto binario se ad ogni nodo sono associati al più 2 figli, chiamati rispettivamente figlio destro e figlio sinistro.

4.4.2 Alberi binari di ricerca

Un albero è detto binario di ricerca se:

- tutti gli elementi nel sottoalbero sinistro hanno valore < della chiave in n
- tutti gli elementi nel sottoalbero destro hanno valore \geq della chiave in n

Gli alberi binari di ricerca differiscono dagli alberi binari in quanto permettono una *ricerca più efficiente* degli alberi binari. In particolare per ogni nodo incontrato si verifica se il valore si maggior o minore della chiave contenuta nel nodo.

L'inserimento e la rimozione devono invece garantire che l'albero binario risultante sia ancora un albero binario di ricerca.

La complessità degli algoritmi di ricerca, inserimento e rimozione con n il numero di nodi, è h la sua altezza:

$$\text{Caso ottimo: } T(n) = O(1)$$

$$\text{Caso pessimo: } T(n) = O(n)$$

$$\text{Caso medio: } T(n) = (\log n)$$

4.4.3 Alberi perfettamente bilanciati

Un albero binario è detto perfettamente bilanciato se, per ogni nodo, il numero di nodi del suo sottoalbero sinistro, ed il numero di nodi del suo sottoalbero destro differiscono al più di 1.

Brutalmente: guardando sotto ogni sotto-nodo, la differenza tra il numero di nodi destro e

sinistro è al massimo 1

Siccome negli alberi perfettamente bilanciati, il bilanciamento dopo le operazioni di inserimento e rimozione sono costose sono stati introdotti una serie di alberi con forme di bilanciamento meno restrittive di quello perfetto.

4.4.4 Alberi di ricerca AVL — Alberi bilanciati in altezza

Un albero binario è detto bilanciato in altezza o AVL se, per ogni nodo, l'altezza del suo sottoalbero sinistro e l'altezza del suo sottoalbero destro differiscono al più di 1. Si noti che un albero perfettamente bilanciato anche bilanciato. Gli alberi AVL sono particolari in quanto hanno un tempo di ricerca, inserimento e cancellazione pari che lavorano a tempo logaritmico:

Per ricerca, inserimento e cancellazione:

$$T(n) = O(\log n)$$

4.4.5 Alberi binario di ricerca rosso-nero

Un albero binario di ricerca è detto rosso-nero se, dopo aver aggiunto un nodo sentinella come figlio di tutte le foglie, e dopo aver colorato ogni nodo di rosso o nero si ha che:

1. La radice è nera
2. La sentinella è nera
3. Se un nodo è rosso, tutti i suoi figli sono neri
4. per ogni nodo, tutti i percorsi dal nodo alla sentinella attraversano lo stesso numero di nodi neri

Un albero binario di ricerca rosso-nero è una buona approssimazione di un albero binario di ricerca perfettamente bilanciato.

```
typedef enum {rosso, nero} colore_t;

typedef struct nodo_albero_bin_rn
{
    int                     valore;
    colore_t                colore;
    struct nodo_albero_bin_rn *sx_p, *dx_p, *padre_p;
} nodo_albero_bin_rn_t;
```

4.4.6 Alberi 2-3

Un albero è detto 2-3 se ogni nodo interni ha 2-3 figli, e le foglie sono tutte allo stesso livello. I dati vengono inseriti all'interno delle foglie, mentre i nodi interni contengono solo informazioni di instradamento.

- Se il nodo interno ha 1 valore, esso compare come sottofiglio sinistro, ed il sottofiglio destro è maggiore
- Se il nodo interno ha 2 valori i sottofigli sono 3, i due valori compaiono come sottofiglio sinistro e centrale, mentre il sottofiglio destro è maggiore

4.4.7 B-albero

I B-alberi sono nati per rappresentare gli indici nelle basi di dati, l'obbiettivo è quello di minimizzare il numero di accessi. Le informazioni, a differenza degli alberi 2-3, sono presenti anche nei nodi interni. Dato un b-albero di ordine t :

- i nodi interni hanno al massimo $2t$ figli
- ogni nodo interno diverso dalla radice ha almeno t figli
- la radice ha almeno 2 figli
- ogni foglia contiene k chiavi ordinate

Poniamo $n = \# \text{ chiavi}$		
	Passi di calcolo (tempo)	Accessi a memoria di massa
Ricerca	$\theta(\log n)$	$\approx \log_t n$
Inserimento	$\theta(t * \log n)$	$\approx c * \log_t n$
Cancellazione	$\theta(t * \log n)$	$\approx c * \log_t n$

4.4.8 Alberi binari quasi completi

Un albero binario è detto *quasi completo* quando è completo fino al penultimo livello e quindi tutte le foglie si trovano all'ultimo o penultimo livello.

5 Heap

Lo heap è un albero binario quasi completo, in cui la chiave contenuta in ciascun nodo è maggiore o uguale alle chiavi contenute nei figli. In particolare:

- tutte le foglie si trovano all'ultimo o penultimo livello
- per ogni nodo, la chiave è \geq delle chiavi contenute nei figli

Procedura risistema

Consideriamo uno heap, se estraiamo un elemento è possibile sia necessario una *risistemazione* dell'albero, in quanto potrebbe perdere la proprietà di heap. Questa procedura è implementata come segue:

Codice 7.1 Risistema (fixHeap)

Si ricordi che nell'ordinamento di solito si devono ordinare dei record rispetto a un campo chiave. Nello pseudocodice sono indicati esplicitamente come "altri campi" tutti gli altri campi, che devono seguire la chiave durante gli spostamenti. Nel caso particolare di uno heap contenente sono numeri interi, vi sarà solo il campo chiave, mentre gli altri campi saranno assenti.

```
Procedura risistema (heap H)
  v  $\leftarrow$  H
  x  $\leftarrow$  v.chiave                                // chiave del nodo radice
  y  $\leftarrow$  v.altri campi                      // altri campi del nodo radice
  daCollocare  $\leftarrow$  true
  do
    if v è una foglia then
      | daCollocare  $\leftarrow$  false // la posizione appropriata per x è stata trovata
    else
      | u  $\leftarrow$  figlio di v di valore massimo
      | if u.chiave > x then
          |   | v.chiave  $\leftarrow$  u.chiave           // i dati in u risalgono: chiave
          |   | v.altri campi  $\leftarrow$  u.altri campi // i dati in u risalgono: altri campi
          |   | v  $\leftarrow$  u                           // si prosegue su u
        else
          |   | daCollocare  $\leftarrow$  false // posizione appropriata per x è stata trovata
    while daCollocare
    v.chiave  $\leftarrow$  x           // copia la ex-radice nella posizione trovata: chiave
    v.altri campi  $\leftarrow$  y // copia la ex-radice nella posizione trovata: altri campi
```

Dato un albero binario quasi completo, come lo trasformo in uno heap?

- con una tecnica divide-et-impera (top-down, ricorsiva)

Codice 7.2 Creazione dello heap (versione ricorsiva)

```
Procedura creaHeap (albero binario T)
  /* Trasforma l'albero binario T in uno heap
  if T  $\neq$  albero vuoto then
    | creaHeap(T.sx)
    | creaHeap(T.dx)
    | risistema(T)
```

- se l'albero è vuoto, è già uno heap e non dobbiamo fare niente
- se l'albero non è vuoto, trasformo ricorsivamente in uno heap i due sottoalberi, e chiamo la procedura risistema, trasformando l'albero in uno heap

- dai sottoalberi più piccoli a quelli più grandi (bottom-up, iterativa)

Codice 7.3 Creazione dello heap (versione iterativa)

Procedura <i>creaHeap (albero binario T)</i> <i>/* Trasforma l'albero binario T in uno heap</i> $h \leftarrow$ altezza di <i>T</i> for $p \leftarrow h$ downto 0 do foreach nodo <i>x</i> di profondità <i>p</i> do <i>risistema</i> (sottoalbero T_x di radice <i>x</i>)

A partire dall'ultima foglia:

- considero ciascun nodo di profondità h da destra a sinistra, trasformo in heap tutti i sottoalberi
- chiamo la procedura risistema
- ripeto sino ad arrivare alla radice

Abbiamo quindi tutti gli strumenti per finalmente implementare l'HeapSort

HeapSort

HeapSort è un algoritmo iterativo basato sulla struttura dati chiamata heap. Questa struttura dati è rappresentabile con un array. L'heap è un albero binario quasi completo, in cui la chiave contenuta in ciascun nodo è maggiore o uguale al contenuto dei figli. Tutte le foglie si trovano all'ultimo o penultimo livello, e per ogni nodo, la chiave è \geq delle chiavi nei figli.

Codice 7.4 Schema di Heapsort

Algoritmo <i>heapSort (albero binario A) \rightarrow lista</i> crea uno heap <i>H</i> da <i>A</i> $X \leftarrow$ lista vuota while <i>H</i> $\neq \emptyset$ do aggiungi all'inizio di <i>X</i> l'elemento presente nella radice di <i>H</i> colloca nella radice di <i>H</i> l'elemento che si trova nella foglia più in basso a destra rimuovi tale foglia <i>risistema(H)</i> return <i>X</i>

Per funzionare l'HeapSort ha bisogno di una serie di metodi:

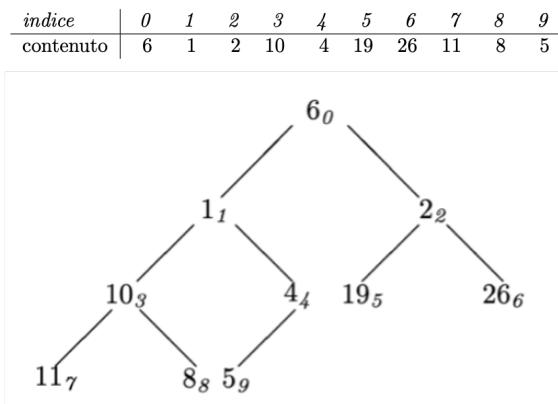
- risistema: che risistema un heap disordinato in un heap ordinato
- creaHeap: crea uno heap a partire da un albero binario

L'algoritmo di heapSort ha una complessità di

Per la *parte esterna* al *while*: $T(n) = n$
 Per la *parte interna* al *while*: $T(n) = n \log n$
 Complessivamente: $O(n \log n)$

HeapSort basato su Arrays

L'algoritmo HeapSort è basato su una struttura dati chiamata heap, per implementarlo tuttavia ci piacerebbe non dover utilizzare una struttura di supporto. Utilizziamo quindi l'array, predisponendo gli elementi all'interno nel seguente modo:



Gli elementi dell'array sono inseriti da sinistra verso destra, garantendo che l'albero risultante sia sempre un albero binario quasi completo. L'elemento in posizione 0 rappresenta la radice dell'array. A questo punto possiamo implementare l'algoritmo *HeapSort* direttamente sull'array, interpretandolo come un albero binario, senza l'utilizzo di puntatori.

Codice 7.5 Heapsort

```

Algoritmo heapSort (array A[0..n - 1])
  creaHeap(A)
  for  $\ell \leftarrow n - 1$  downto 1 do
    scambia A[0] e A[ℓ]
    risistema(A, 0,  $\ell$ )
  
```

Nuovamente abbiamo bisogno di due metodi:

- risistema: che risistema l'array disordinato in uno heap

- creaHeap: utilizziamo la soluzione bottom-up, che trasforma in uno heap ogni sottoalbero a partire dall'ultima foglia

Considerazioni finali

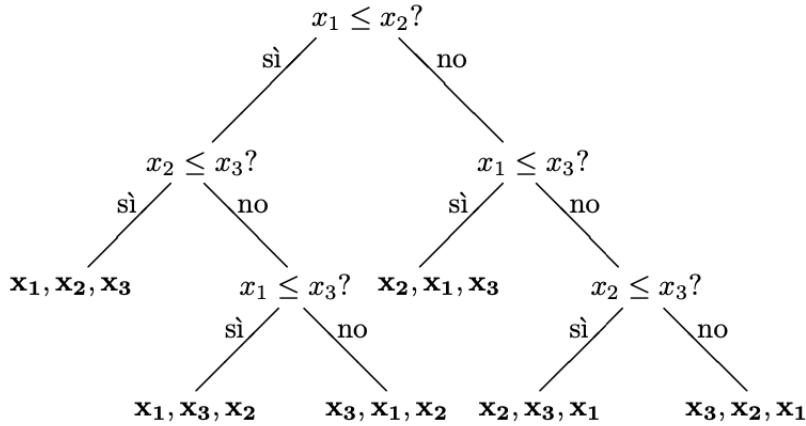
HeapSort è un algoritmo di ordinamento in loco, non stabile, che effettua $\theta(n \log n)$ confronti. Il tempo complessivo è $\theta(n \log n)$

Una panoramica sugli algoritmi di ordinamento

<i>Algoritmo</i>	<i>Numero confronti</i>	<i>Spazio</i>	<i>Note</i>	<i>Stabile</i>
selectionSort	$\Theta(n^2)$ sempre	$\Theta(1)$	in loco	no
insertionSort	$\Theta(n^2)$ nel caso peggiore $n - 1$ su array già ordinato	$\Theta(1)$	in loco	sì
bubbleSort	$\Theta(n^2)$ nel caso peggiore $n - 1$ su array già ordinato	$\Theta(1)$	in loco	sì
mergeSort	$\Theta(n \log n)$	$\Theta(n)$	spazio $\Theta(n)$ per array ausiliario più $\Theta(\log n)$ per stack ricorsione	sì
quickSort	$\Theta(n^2)$ nel caso peggiore $\Theta(n \log n)$ caso migliore $\approx 1.39n \log_2 n$ in media	$\Theta(n)$ $\Theta(\log n)$	in loco spazio $\Theta(1)$ più stack ricorsione: $\Theta(n)$ algoritmo base $\Theta(\log n)$ algoritmo migliorato	no
heapSort	$\Theta(n \log n)$	$\Theta(1)$	in loco	no

Numero minimo di confronti

Un qualsiasi algoritmo di ordinamento basato su confronti necessita, nel caso peggiore, un numero di confronti tra chiavi dell'ordine di $n \log n$. A tale scopo osserviamo un confronto tra due chiavi x_i e x_j , con risposta *si/no*. Per rappresentare la sequenza di decisioni percorribili per $n = 3$ elementi possiamo utilizzare un albero di decisione:



Indipendentemente dal numero di confronti, l'albero avrà un numero di foglie uguale al numero di possibili ordini tra le chiavi, ovvero il numero di permutazioni di n , ovvero $n!$. Il numero massimo di confronti è pari alla profondità dell'albero, di nuovo $n = 3$. Possiamo tuttavia verificare che la profondità dell'albero binario con k foglie è almeno logaritmico in k .

Per trovare il numero di confronti necessari nel caso peggiore stimiamo la profondità minima che si ottiene da un albero con $n!$ foglie, calcolando il logaritmo di $n!$. Utilizziamo l'approssimazione di Stirling:

$$\begin{aligned}
 \log(n!) &\approx \log\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\right) \\
 &= \log\sqrt{2\pi} + \frac{1}{2}\log n + n\log n - n\log e \\
 &= \Theta(n\log n)
 \end{aligned}$$

Concludiamo che un *qualsiasi* algoritmo di ordinamento basato su confronti richiede al minimo un numero di confronti tra chiavi dell'ordine $n\log n$ per ordinare n elementi.

Heap e Code con priorità

Gli heap hanno altre applicazioni oltre all'utilizzo nell'algoritmo HeapSort, in particolare servono ad implementare le code con priorità. Le code con priorità sono strutture dati utilizzati in molti algoritmi, innanzitutto estendiamo le operazioni sugli heap:

- *findMax* — trova l'elemento di chiave massima
L'elemento si trova in radice, l'operazione viene svolta in $O(1)$

- *deleteMax* — cancella l'elemento di chiave massima
Sostituiamo l'elemento in radice, con la foglia più a destra all'ultimo livello, per poi applicare *risistema*. Il numero di passi è proporzionale all'altezza dell'heap quindi $O(\log n)$
- *insert* — inserisci un nuovo elemento
L'inserimento avviene con l'aggiunta alla prima posizione libera più a sinistra dell'ultimo livello, o come prima foglia ad un nuovo livello. Non possiamo utilizzare la funzione *risistema*, ma dobbiamo utilizzare un approccio dal basso, con *risistemaDalBasso*. Il numero di passi è proporzionale all'altezza dell'heap quindi $O(\log n)$
- *delete* — cancella un elemento
Per cancellare un elemento lo sostituiamo con la foglia più a destra dell'ultimo livello, che viene rimossa, e risistemiamo l'heap. Sia x la chiave del nodo n da cancellare, e f la chiave della foglia più a destra dell'ultimo livello:
 - se $f < x$ utilizziamo *risistema*
 - se $f > x$ utilizziamo *risistemaDalBasso*
 In entrambi i casi il numero di passi è pari a $O(\log n)$
- *changeKey* — modifica la chiave di un elemento
Modifica la chiave associata ad un nodo n , se:
 - la chiave viene diminuita, utilizziamo *risistema*
 - la chiave viene aumentata, utilizziamo *risistemaDalBasso*

Code con priorità

Utilizzando gli heap e le operazioni descritte possiamo implementare strutture a coda, in cui gli elementi vengono estratti secondo un criterio di priorità. In particolare le operazioni di una coda con priorità sono:

- *findMin()* — restituisce l'elemento minimo della coda
- *deleteMin ()* — rimuovere l'elemento minimo della coda
- *insert(e, k)* — inserisce nella coda un elemento e associa una chiave
- *delete(e)* — rimuove l'elemento
- *changeKey(e, d)* — modifica la priorità dell'elemento

Algoritmi di ordinamento non basati su confronti

IntegerSort

E' il primo algoritmo di ordinamento non basato su confronti. Oltre all'array da ordinare dobbiamo fornire anche un intero che rappresenti il range degli elementi all'interno dell'array. Viene creato un array ausiliario di contatori, che conta le occorrenze all'interno all'array A .

Infine viene riempito A con i valori ordinati.

Codice 10.1 Integersort

```
Algoritmo integerSort (array A[0..n - 1], intero k)
/* Ordina un array di interi appartenenti all'intervallo [1,k] */ 
Sia Y[1..k] un array // array di contatori
for i ← 1 to k do Y[i] ← 0 // azzeri i contatori
// Conta le occorrenze nell'array A di ciascun intero appartenente a [1,k]
for i ← 0 to n - 1 do
    x ← A[i]
    Y[x] ← Y[x] + 1

// riempi A con i valori ordinati
j ← 0 // indice della prossima posizione di A da riempire
for i ← 1 to k do
    // assegna l'intero i alle successive Y[i] posizioni di A
    for h ← 1 to Y[i] do
        A[j] ← i
        j ← j + 1
```

BucketSort

E' il secondo algoritmo di ordinamento non basato su confronti che incontriamo, ed utilizza le code. Nuovamente oltre all'array l'algoritmo necessita del range dell'array A .

Codice 10.2 Bucketsort

```
Algoritmo bucketSort (array A[0..n - 1], intero k)
    /* Ordina un array di record in base a un campo chiave intero appartenente
       all'intervallo [1, k] */ 
    Sia Y[1..k] un array // array di code
    for i ← 1 to k do Y[i] ← coda vuota
    // colloca gli elementi di A in differenti code, in base alle chiavi
    for i ← 0 to n - 1 do
        x ← A[i].chiave
        Y[x].enqueue(A[i])
    // riempi A con i valori ordinati
    j ← 0 // indice della prossima posizione di A da riempire
    for i ← 1 to k do
        // colloca i record con chiave i nelle prossime posizioni di A
        while not Y[i].isEmpty() do
            A[j] ← Y[i].dequeue()
            j ← j + 1
```

Vengono create una serie di code per ogni $n < k$, che vengono popolate con gli elementi presenti nell'array A . Il BucketSort è un algoritmo stabile, principalmente utilizzato per riordinare liste, manipolando direttamente i puntatori.

RadixSort

Consideriamo di dover ordinare una serie di dati con chiavi intere. Possiamo applicare BucketSort prima sulle unità, poi decine, poi centinaia e così via. L'algoritmo riordina a partire dal *least significant digit*.

Codice 10.3 Radixsort in base B ($B > 1$ è una costante fissata, e.g., $B = 10$).

```

Algoritmo radixSort (array A[0..n - 1])
  /* Ordina l'array A secondo un campo chiave intero */ 
  t ← 0
  while esiste una chiave k in A con  $k/B^t \neq 0$  do
    | bucketSort(A, B, t)
    | t ← t + 1

Procedura bucketSort (array A[0..n - 1], intero b, intero t)
  /* Ordina l'array A secondo la cifra di posizione t nella rappresentazione
   * in base b della chiave */ 
  Sia Y[0..b - 1] un array // array di code
  for i ← 0 to b - 1 do Y[i] ← coda vuota
  // colloca gli elementi di A in differenti code, in base alle chiavi
  for i ← 0 to n - 1 do
    | c ← cifra di posizione t nella rappresentazione in base b di A[i].chiave
    | Y[c].enqueue(A[i])

  // riempি A con i valori ordinati
  j ← 0 // indice della prossima posizione di A da riempire
  for i ← 0 to b - 1 do
    | // colloca gli elementi che hanno in posizione t la cifra i nelle
      | prossime posizioni di A
    | while not Y[i].isEmpty() do
      | | A[j] ← Y[i].dequeue()
      | | j ← j + 1
  
```

Tecniche Union-Find

Gli algoritmi *union-find* lavorano su partizioni, e prendono il nome dalle due operazioni più svolte. Supponiamo quindi di avere una collezione S di n elementi distinti:

- Union(A, B): unisce gli insieme A, B in un unico insieme, il nuovo nome dell'insieme è A
- Find(x): dato un elemento, restituisce il nome dell'insiemi che lo contiene
- MakeSet(x): dato un elemento, crea un nuovo insieme con nome $x : x$

Vi sono vari tipi di soluzioni utilizzati, divisibili in soluzioni elementari, e soluzioni evolute. In tutte le soluzioni presentate ogni insieme è rappresentato da un albero con radice in cui:

- i nodi rappresentano gli elementi dell'insieme
- le radici rappresentano i nodi dell'insieme

I puntatori puntano verso l'alto, ed una partizione rappresenta una foresta di alberi.

Algoritmi QuickFind

Negli algoritmi QuickFind gli elementi dell'insieme rappresentano le foglie, mentre i nomi dell'insieme rappresentano la radice. Ricordiamo che i puntatori puntano verso l'alto.

- Union(A, B): spostiamo i puntatori che puntavano da B , ad A
Questa operazione lavora in tempo $T(n) = O(n)$
- Find(x): restituiamo l'elemento in tempo $T(n) = O(1)$
- MakeSet(x): crea un nuovo albero, con foglia e radice uguale, in tempo $T(n) = O(1)$

Possiamo migliorare gli algoritmi con il *QuickFind con bilanciamento*. In particolare notiamo che nell'operazione di **Union(A,B)**, se B ha più elementi di A , anziché collegare gli elementi di B ad A , ci conviene collegare gli elementi di A sotto B , e rinominare la radice. Miglioriamo l'efficienza, e al massimo dobbiamo spostare $n/2$ puntatori. Il tempo rimane dell'ordine di $T(n) = O(n)$

Algoritmi QuickUnion

Negli algoritmi QuickUnion gli elementi dell'insieme sono rappresentati dai nodi, mentre l'elemento che da il nome all'insieme è la radice. Gli alberi sono di varia altezza. La differenza con gli alberi QuickFind è che consentiamo agli alberi di avere altezza maggiore di 1.

- Union(A, B): spostiamo il puntatore di B , ad A Questa operazione lavora in tempo $T(n) = O(n)$
- Find(x): restituiamo l'elemento in tempo $T(n) = O(n)$, dato che dobbiamo risalire da x fino alla radice
- MakeSet(x): crea un nuovo albero, con foglia e radice uguale, in tempo $T(n) = O(1)$

Nuovamente consideriamo un'operazione di **Union(A,B)**, possiamo migliorare l'efficienza, attaccando la radice dell'albero più basso sotto a quella dell'albero più alto. Chiamiamo questa *union* migliorata *QuickUnion by rank*

Grafi

Si dice grafo diretto o orientato una coppia $G = (V, E)$ ove V è un insieme di vertice o nodi ed E è una relazione binaria su V

- I grafi possono essere non orientati/simmetrici, ovvero con archi senza freccia; oppure orientati/diretti, ovvero gli archi hanno frecce.
- un cammino da x a y è una sequenza di vertici che porti da x a y , in particolare un cammino è detto semplice se non contiene vertici ripetuti. Un ciclo è un cammino da vertice x a x
- un ciclo semplice è un ciclo in cui è ripetuto solo il vertice iniziale, alla fine
- una catena tra x e y è una sequenza di nodi collegati da archi orientati, ma di cui si ignora l'orientamento. Un circuito è una catena chiusa
- un sottografo è un grafo, formato da un sottoinsieme di vertici e archi del grafo iniziale
- un cammino Euleriano è un cammino che attraversa ogni arco del grafo una sola volta (è quindi un ciclo)
- un cammino Hamiltoniano è un cammino che attraversa ogni vertice una sola volta

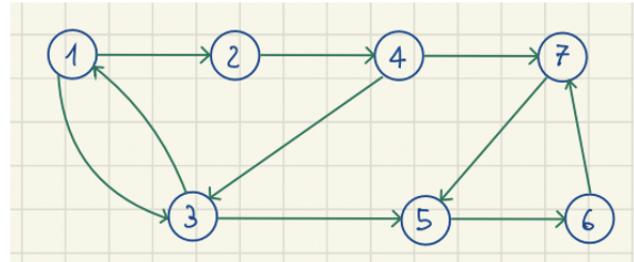
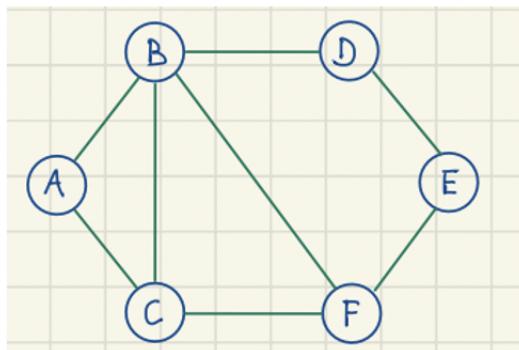
Alberi e foreste

Un albero è un grafo non orientato, connesso e privo di cicli. Una foresta è un insieme di alberi.

Un albero di supporto o ricoprimento (spanning tree) è un albero con gli stessi vertici, ma con gli archi che sono un sottoinsieme del grafo iniziale; un qualunque sottografo che contiene lo stesso numero di vertici, e che gli archi sia un sottoinsieme degli archi.

Rappresentazione dei grafi

Consideriamo i seguenti grafi non orientati, ed orientati



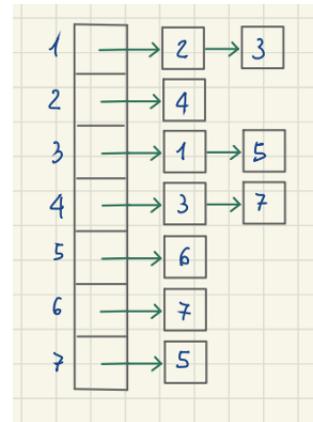
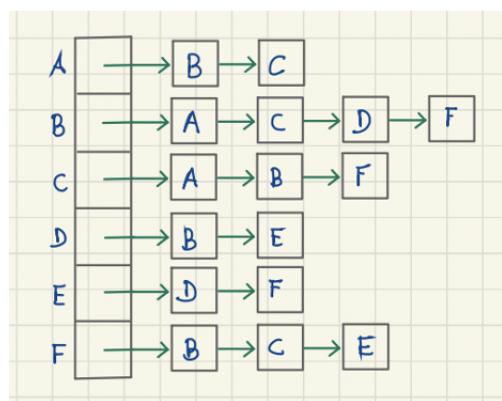
Un grafo può essere rappresentato con

- **Lista di archi**

Una lista di archi è una struttura dati formata da 2 strutture, una lista che rappresenta i vertici, e una che rappresenta gli archi. Gli archi vengono rappresentati come un elenco che contiene le coppie di vertici, e gli archi che l'arco collega. Questa rappresentazione è comoda per visualizzare i vertici di un arco, ma non per ricostruire la forma iniziale di un grafo. Lo spazio occupato è $O(m + n)$ ove m è il numero di vertici, e n il numero di archi

- **Lista di adiacenza**

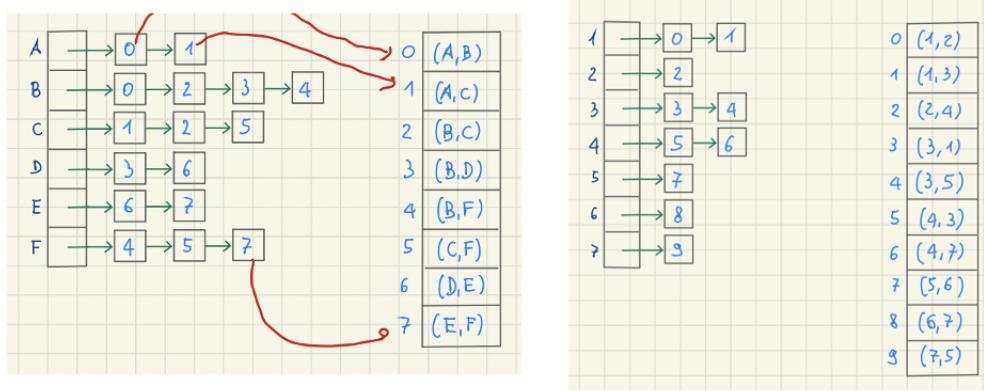
Una lista di adiacenza è una struttura dati formata da una lista primaria dei vertici e più liste secondarie degli archi. La lista primaria contiene un elemento per ciascun vertice del grafo, il quale contiene a sua volta la testa della relativa lista secondaria. La lista secondaria associata ad un vertice descrive tutti gli archi uscenti da quel vertice, in quanto contiene gli indirizzi di tutti i vertici adiacenti al vertice in questione.



L'accesso ad ognuno di essi necessita dell'indirizzo del primo elemento della lista primaria, l'accesso ad un vertice avviene scorrendo tutti i vertici che precedono il vertice in questione nella lista primaria.

- **Lista di incidenza**

Una lista di incidenza è una struttura dati formata sostituendo la lista dei vertici delle liste di adiacenza, con liste di archi.



Anche in questo caso lo spazio utilizzato è $O(m + n)$

- Matrice di adiacenza

Una matrice di adiacenza creata a partire da un grafo G è una matrice in cui il valore

$$M(u, v) = 1 \text{ sse } (u, v) \in E$$

Se una matrice non è orientato, la matrice generato è una matrice simmetrica. Lo spazio occupato da questa struttura è $O(n^2)$

$$\begin{aligned} & \left(\begin{array}{cccccc} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 \end{array} \right) \\ \Rightarrow & \left(\begin{array}{cccccc} 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{array} \right) \end{aligned}$$

- Matrice di incidenza

Una matrice di incidenza creata a partire da un grafo G è una matrice in cui:

- ogni riga rappresenta un vertice
- ogni colonna rappresenta un arco

	(A,B)	(A,C)	(B,C)	(B,D)	(B,F)	(C,F)	(D,E)	(E,F)
A	1	1	0	0	0	0	0	0
B	1	0	1	1	1	0	0	0
C	0	1	1	0	0	1	0	0
D	0	0	0	1	0	0	1	0
E	0	0	0	0	0	0	1	1
F	0	0	0	0	1	1	0	1

	(1,2)	(1,3)	(2,4)	(3,1)	(3,5)	(4,3)	(4,7)	(5,6)	(6,7)	(7,5)
1	1	1	0	-1	0	0	0	0	0	0
2	-1	0	1	0	0	0	0	0	0	0
3	0	-1	0	1	1	-1	0	0	0	0
4	0	0	-1	0	0	1	1	0	0	0
5	0	0	0	0	-1	0	0	1	0	-1
6	0	0	0	0	0	0	0	-1	1	0
7	0	0	0	0	0	0	-1	0	-1	1

Nel caso dei grafi orientati, il valore $M(u,v) = 1$ se l'arco è uscente, mentre è $M(u,v) = -1$ quando l'arco è entrante, $M(u,v) = 0$ quando non è incidente. Lo spazio occupato da questa struttura è $O(n * m)$

Attraversamento dei grafi

Data la generalità della loro struttura, nel caso dei grafi ci sono ancora più alternative per effettuare una visita di quelle viste nel caso degli alberi binari. Di solito i vertici di un grafo $G = (V, E)$ vengono attraversati in maniera sistematica procedendo in ampiezza (breadth-first search) o in profondità (depth-first search).

Visita in ampiezza — Breadth-first Search

La visita in ampiezza viene svolta nel seguente modo, a partire da un vertice s

- si visitano i vertici adiacenti ad S aggiungendo man mano in una coda gli elementi
- si visitano i vertici adiacenti ai vertici adiacenti ad S che non sono ancora stati visitati.

Codice 11.1 Visita in ampiezza (Breadth-First Search)

```

Algoritmo visitaInAmpiezza (grafo  $G = (V, E)$ , vertice  $s$ )  $\rightarrow$  albero
/* Visita in ampiezza di un grafo non orientato connesso a partire da un
   vertice  $s$  e costruzione di un albero ricoprente ottenuto selezionando
   gli archi secondo l'ordine della visita */
 $C \leftarrow$  coda vuota
 $T \leftarrow (\{s\}, \emptyset)$  // albero formato dal solo vertice  $s$ 
marca  $s$  come raggiunto
 $C.enqueue(s)$ 
while not  $C.isEmpty()$  do
     $u \leftarrow C.dequeue()$ 
    foreach  $(u, v) \in E$  do
        if  $v$  non è marcato come raggiunto then
             $T \leftarrow (vertici(T) \cup \{v\}, archi(T) \cup \{(u, v)\})$ 
            // aggiunge il vertice  $v$  e l'arco  $(u, v)$  a  $T$ 
            marca  $v$  come raggiunto
             $C.enqueue(v)$ 
    return  $T$ 

```

Visita in profondità — Depth-first Search

La visita in ampiezza viene svolta ricorsivamente, a partire da un vertice s

- si visitano i vertici non ancora visti, fermandoci ad un vertice privo di vicini non ancora visitati
- si ripete dal passo precedente, partendo dall'ultimo vertice sul cammino precedente che ha un vertice non raggiunto

si parte da un vertice, e cerchiamo di esplorare il più possibile fino a quando non posso più muovermi. A quel punto torno indietro fino a quando non trovo una strada che posso percorrere.

Codice 11.2 Visita in profondità (Depth-First Search)

```

Algoritmo visitaInProfondità (grafo  $G = (V, E)$ , vertice  $s$ )  $\rightarrow$  albero
/* Visita in profondità di un grafo non orientato connesso a partire da un
   vertice  $s$  e costruzione di un albero ricoprente ottenuto selezionando
   gli archi secondo l'ordine della visita */
 $T \leftarrow (\{s\}, \emptyset)$  // albero formato dal solo vertice  $s$ 
visitaRicorsiva( $G, s, T$ )
return  $T$ 

Procedura visitaRicorsiva(grafo  $G = (V, E)$ , vertice  $u$ , albero  $T$ )
marca  $u$  come visitato
foreach  $(u, v) \in E$  do
    if  $v$  non è marcato come visitato then
         $T \leftarrow (vertici(T) \cup \{v\}, archi(T) \cup \{(u, v)\})$ 
        // aggiunge il vertice  $v$  e l'arco  $(u, v)$  a  $T$ 
        visitaRicorsiva( $G, v, T$ )

```

Problemi di ottimizzazione e algoritmi Greedy

Dato un problema P di ottimizzazione:

- C l'insieme di soluzioni candidati
- Trovare $S \in C$ soluzione ottima

La tecnica golosa si applica ai problemi di ottimizzazione in cui la soluzione è costruita scegliendo, ad ogni passo, l'elemento migliore nel momento. La soluzione che ricaviamo non è sempre la soluzione ottima, è tuttavia la soluzione più veloce da computare.

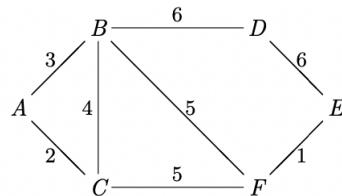
Grafo pesato

Consideriamo un grafo $G = (V, E)$, associamo una funzione $\omega : E \rightarrow R$ chiamata funzione peso.

Grafo pesato: $G = (V, E, \omega)$

Dato un grafo pesato possiamo analizzare una serie di problemi:

- problema dei cammini minimi
- problema degli alberi ricoprenti minimi (su grafi non orientati)



Albero ricoprente minimo

Sia $G = (V, E, \omega)$ un grafo pesato, trovare l'albero ricoprente minimo tra i possibili. Introduciamo quindi due algoritmi per trovare gli alberi ricoprenti minimi di grafi connessi, non orientati e pesati, chiamati *algoritmo di Kruskal*, e *algoritmo di Prim*, entrambi basati sulla strategia greedy

Algoritmo di Kruskal

L'algoritmo di Kruskal risolve il problema costruendo un grafo T che ha gli stessi vertici di G e, inizialmente, è privo di archi.

- L'algoritmo esamina gli archi di G in ordine di peso non decrescente.
- Un arco viene aggiunto a T se, insieme a quelli già scelti, non forma cicli, altrimenti viene scartato e non sarà più considerato.
- Ogni volta che si aggiunge un grafo, si connettono tra loro due alberi della foresta che diventano, con l'arco aggiunto, un unico albero.
- Alla fine, quando sono stati esaminati tutti gli archi, T contiene un unico albero che è un albero ricoprente di peso minimo per il grafo G dato.

Codice 12.1 Algoritmo di Kruskal

```

Algoritmo Kruskal (grafo connesso non orientato  $G = (V, E, \omega)$ )  $\rightarrow$  albero
  /* Costruzione di un albero ricoprente minimo */  

  ordina l'insieme  $E$  in base ai pesi in modo non decrescente  

   $T \leftarrow (V, \emptyset)$   

  foreach  $(x, y) \in E$  secondo l'ordine do  

    | if  $x$  e  $y$  non sono connessi in  $T$  then  

    | | aggiungi al grafo  $T$  l'arco  $(x, y)$   

  return  $T$ 

```

L'algoritmo di Kruskal è implementabile attraverso lo union-find, con tempo totale $O(m \log^* m)$

Algoritmo di Prim

Dato in ingresso un grafo connesso, non orientato con pesi sugli archi, l'algoritmo inizia costruendo un albero T formato da un unico vertice s qualsiasi del grafo.
 Ad ogni passo, l'albero T viene espanso scegliendo, tra tutti gli archi che hanno un vertice in T e l'altro non in T , un arco di peso minimo. Tale arco viene aggiunto a T (insieme al vertice che non era in T).
 L'algoritmo di Prim può essere implementato ricorrendo a una coda con priorità C , contenente un elemento per ogni vertice che deve essere ancora inserito nell'albero.

Codice 12.3 Algoritmo di Prim (schema ad alto livello)

```

Algoritmo Prim (grafo connesso non orientato  $G = (V, E, \omega)$ )  $\rightarrow$  albero
  /* Costruzione di un albero di ricoprimento di peso minimo */  

   $T \leftarrow (V_T \leftarrow \{s\}, E_T \leftarrow \emptyset)$  // albero formato da un solo nodo  $s$   

  while  $T$  ha meno di  $n$  nodi do //  $n = \#V$   

    | sia  $(x, y) \in E$  di peso minimo con  $x \in V_T$  e  $y \notin V_T$   

    |  $V_T \leftarrow V_T \cup \{y\}$   

    |  $E_T \leftarrow E_T \cup \{(x, y)\}$   

  return  $T$ 

```

L'algoritmo di Prim è implementabile con liste di adiacenza o incidenza, con code di priorità, e con code di priorità. Per l'analisi del tempo supponiamo che il grafo in ingresso sia rappresentato mediante una lista di adiacenza o incidenza. Il tempo totale è $O(m \log n)$

Problema dei cammini minimi

Consideriamo $G = (V, E, \omega)$ un grafo pesato orientato, il problema dei cammini minimi ci chiede di trovare un cammino, tra i cammini possibili, che sia il minimo possibile. In particolare

- se tutti i pesi sono positivi, allora il cammino è uno semplice

- se ci sono pesi negativi, ma non cicli negativi, allora tra una coppia di vertici esiste un cammino minimo semplice.

Per rappresentare i grafi pesati, come avevamo visto in precedenza, utilizziamo matrici dei pesi, e liste di adiacenza. I problemi che possiamo trovare sui cammini minimi sono:

1. trovare il cammino minimo tra 2 vertici
Non esiste un algoritmo diretto, ma possiamo applicare Dijkstra per risolverlo
2. trovare i cammini minimi tra un vertice S e tutti gli altri
 - Applicare Dijkstra se non ci sono pesi negativi, con tempo $T(n) = O(m + n \log n)$
 - Applicare Bellman Ford, anche con pesi negativi, con tempo $T(n) = O(m * n)$
3. trovare i cammini minimi tra ogni coppia di vertici
 - Applicare Floyd-Warshall, anche con pesi negativi, con tempo $T(n) = O(n^3)$

Algoritmo di Floyd & Warshall

L'algoritmo di Floyd & Warshall risolve il problema di cammino minimo tra due vertici attraverso una rappresentazione del grafo a matrice di adiacenza, basato sulla strategia della programmazione dinamica. Sia

- $V = v_1, \dots, v_n$ l'insieme dei vertici numerati
- d_{ij} il peso del cammino minimo tra v_i e v_j

Viene impostata una matrice delle distanze D , con valore d_{ij}^k , che cambia per $k = 0$ e $k > 0$

$$d_{ij}^{(0)} = \begin{cases} w(v_i, v_j) & \text{se } (v_i, v_j) \in E \text{ e } v_i \neq v_j \\ 0 & \text{se } v_i = v_j \\ \infty & \text{altrimenti} \end{cases}$$

$$d_{ij}^{(k)} = \min \{ d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \}$$

Codice 13.1 Calcolo della lunghezza dei cammini minimi tra ogni coppia di vertici

```

Algoritmo FloydWarshall (grafo pesato G = (V, E, ω)) → matrice distanze
/* La funzione ω assegna un peso a ciascun arco.
   Il grafo non deve contenere cicli di peso negativo. */
*/
```

Sia D una matrice $n \times n$, con $V = \{v_1, v_2, \dots, v_n\}$

```

// Per ogni coppia di vertici, calcola la minima lunghezza di un cammino
// che li congiunge, senza passare per vertici intermedi (cioè singolo nodo
// o singolo arco)
for i ← 1 to n do
    for j ← 1 to n do
        if i = j then  $D[i, j] \leftarrow 0$ 
        else if  $(v_i, v_j) \in E$  then  $D[i, j] \leftarrow \omega(v_i, v_j)$ 
        else  $D[i, j] \leftarrow \infty$ 
```

```

for k ← 1 to n do
    // per ogni coppia di vertici, calcola la minima lunghezza di un cammino
    // che li congiunge, i cui vertici intermedi appartengono a  $\{v_1, v_2, \dots, v_k\}$ 
    // cioè hanno indice ≤ k
    for i ← 1 to n do
        for j ← 1 to n do
            if  $D[i, k] + D[k, j] < D[i, j]$  then
                 $D[i, j] \leftarrow D[i, k] + D[k, j]$ 
```

return D

L'algoritmo lavora in tempo $T(n) = O(n^3)$

Algoritmo di Bellman & Ford

L'algoritmo di Bellman-Ford (1958-1962) consente la presenza di archi di peso negativo (ma non cicli negativi) e procede alla riduzione sistematica della stima della distanza minima calcolata per ciascun vertice.

ALGORITMO Bellman & Ford (*Grafo G, vertice s*) → *Vettore*

Sia $d[V]$ un vettore con indici in V

$d[s] \leftarrow 0$

$$d_v^{[0]} = \begin{cases} 0 & \text{se } v=s \\ \infty & \text{altrimenti} \end{cases}$$

FOR EACH $v \in V \setminus \{s\}$ DO $d[v] \leftarrow \infty$

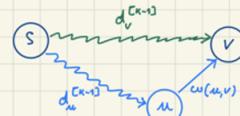
FOR $K \leftarrow 1$ TO $n-1$ DO

FOR EACH $(u, v) \in E$ DO

```

IF  $d[u] + \omega(u, v) < d[v]$  THEN
     $d[v] \leftarrow d[u] + \omega(u, v)$ 
```

RETURN d



Ad ogni vertice viene assegnato un valore iniziale ∞ , che viene aggiornato man mano partendo dal vertice iniziale. Una volta completato l'aggiornamento di tutti gli archi uscenti si passa al prossimo vertice, se la somma del valore è < il valore d_v è aggiornato, altrimenti d_v è scartato
L'algoritmo lavora in tempo $T(n) = O(m * n)$

Algoritmo di Dijkstra

L'algoritmo di Dijkstra è più efficiente di quello di Bellman-Ford, ma si applica solo a grafi privi di archi di peso negativo basato sulla strategia greedy.

- Inizialmente tutti i d_v che non sono il vertice iniziale hanno un valore assegnato a ∞ , 0 se invece solo il vertice iniziale; viene creato C insieme di vertici candidati inizialmente $C = V$
- preleva da C un vertice u con d_u minima
- d_u diventa definita e aggiorna $d[v]$ per ogni v adiacente a u

Codice 13.2 Calcolo della lunghezza dei cammini minimi da un vertice s a tutti gli altri

```

Algoritmo Dijkstra (grafo pesato  $G = (V, E, \omega)$ , vertice  $s$ )  $\rightarrow$  vettore distanze
/* La funzione  $\omega$  assegna un peso a ciascun arco.
   Il grafo non deve contenere archi di peso negativo. */
```

Sia D un vettore con insieme di indici V
 $D[s] \leftarrow 0$
foreach $v \in V \setminus \{s\}$ **do** $D[v] \leftarrow \infty$
 $C \leftarrow V$
while $C \neq \emptyset$ **do**
 $u \leftarrow$ elemento di C con $D[u]$ minima
 $C \leftarrow C \setminus \{u\}$
foreach $(u, v) \in E$ **do**
if $D[u] + \omega(u, v) < D[v]$ **then**
 $D[v] \leftarrow D[u] + \omega(u, v)$

return D

L'algoritmo lavora in tempo $T(n) = O(m + n \log n)$

Hash e tabelle hash

Vediamo prima di tutto una tabella riassuntiva dei tempi richiesti delle 3 operazioni nelle varie strutture che abbiamo affrontato:

operazioni	Array non ordinato	Array ordinato	Alberi di ricerca	Alberi AVL e alberi 2-3
Ricerca	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(\log n)$
Inserimento	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$
Cancellazione	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$

Esiste tuttavia una struttura dati che inserisce in disordine i dati di proposito, in particolare le tabelle hash.

- sia U = l'universo delle chiavi
- sia $0, \dots, n-1$ lo spazio delle chiavi

Una funzione di hash è una funzione che associa un elemento della chiave ad ogni elemento degli indici così definita:

$$h = U \rightarrow 0, \dots, n-1$$

Una funzione di hash è perfetta se è iniettiva, ovvero se $u = v$, $h(u) = h(v)$, più una funzione di hash sparpaglia gli elementi, meglio è. E' tuttavia possibile che due elementi abbiano una stessa chiave, in questo caso parliamo di *collisione*. Per gestire le collisioni possiamo adottare

- Strategie interne, ove le collisioni vengono gestite all'interno delle tabelle
- Strategie esterne, ove collisioni vengono gestite esternamente dalle tabelle.
Viene impiegata una cosiddetta lista di collisioni, all'inserimento di un elemento, essa non viene inserita nella tabella, ma viene collegata alla lettera corretta, ne consegue che la tabella deve essere dimensionata in maniera corretta da permettere il contenimento di tutte le chiavi.

- Inserimento: $O(1)$ (grazie all'inserimento in testa)
- Ricerca: $O(m)$
- Cancellazione: $O(n)$

$$\text{Tempo medio: } T_{AVG}(n) = O\left(1 + \frac{n}{m}\right) = O(1 + \alpha) \rightarrow \text{dipende dalla lunghezza delle liste}$$

Problema di agglomerazione primaria

Il problema di agglomerazione primaria si presenta quando abbiamo una zona in cui si concentrano molti dati. Questo succede quando la funzione di hashing non sparpaglia sufficientemente bene. Consideriamo un esempio pratico:

- sia $U = \text{stringhe di lettere}$
- sia $h(x) = \text{ord}(\text{1lettera di } x)$
eg. $\text{ord}(A) = 1, \text{ord}(Z) = 25$

Consideriamo la sequenza: Pippo, Paperino, Topolino, Qui, Pluto, Quo, Qua, Paperoga

...	
14	
15	Pippo
16	Paperino
17	Qui
18	Pluto
19	Topolino
20	Quo
21	Qua
22	Paperoga
...	

Per aggirare questo problema, possiamo implementare l'indirizzamento aperto. L'indirizzamento aperto è basato su una funzione ausiliaria $c(k, i)$, ove $k = \text{chiave}$ e $i = \text{intero}$ con $i = 1, \dots, n$ ordine della scansione. Definiamo scansione lineare:

$$c(k, i) = (h(k) + i) \bmod m$$

e scansione quadratica:

$$c(k, i) = [h(k) + c1i + c2i^2] \bmod m$$

Questa funzione ausiliaria ci permette di aggirare il problema di agglomerazione primaria, ma non interviene sull'agglomerazione secondaria

Hashing doppio

L'hashing doppio è applicabile quando all'inserimento di un valore nella tabella di hash, troviamo un posto occupato. In questo caso possiamo impiegare una seconda funzione di hashing, diversa. Il problema con questa implementazione è che la cancellazione crea qualche problema, infatti ad ogni cancellazione, dovremmo ristrutturare tutta la tabella perché potremmo perdere i legami che legano le celle, che vengono utilizzati nelle ricerche.

Risolviamo tale problema introducendo un flag booleano di cancellazione, al momento della cancellazione effettuiamo solo una cancellazione virtuale, e non effettiva.

Rehashing

Il rehashing avviene quando la tabella viene riempita oltre la soglia prestabilita. Viene creata una tabella più grande (di solito il doppio), in cui vengono trasferiti tutti gli elementi, per continuare l'hashing. In particolare dobbiamo andare a vedere la *funzione hashing della nuova tabella, e il costo in termini di tempo La funzione hash*

Solitamente la funzione hash è parametrica rispetto alla dimensione m della tabella, in particolare

$$h_m(k) = f(k) \bmod m$$

Ove $f(k)$ è un'altra funzione hash che restituisce un interno molto grande. Se f è uniforme, allora anche h_m è uniforme.

Complessità computazionale

A partire da un problema possiamo effettuare analisi classificando stimando rispetto alle risorse necessarie e sufficienti per risolvere il problema. Dato un problema da risolvere cerchiamo un algoritmo risolutivo, e mentre lo facciamo facciamo anche la stima delle risorse utilizzate dallo specifico algoritmo, che possono essere spazio, tempo, numero di confronti, etc.

Quando parliamo di *classe di complessità* facciamo riferimento all'insieme dei problemi che possono essere risolti utilizzando la stessa classe di quantità di una data risorsa.

$$\begin{aligned} r & \text{ risorsa computazionale} \\ \pi & \text{ problema risolvibile algoritmamente} \end{aligned}$$

quanta risorsa r è necessaria e sufficiente per risolvere π ?

Classe P

Identifichiamo quindi la classe P , ovvero la classe di problemi che possono essere risolti in tempo polinomiale rispetto alla lunghezza dell'output. In particolare questa classe è chiamata *PTIME*, ma non è l'unica, altre classi che ci interessano sono:

- PSPACE: classi di problemi che possono essere risolti in spazio polinomiale
- EXPTIME: classi di problemi che possono essere risolti in tempo esponenziale

Ne consegue che $P \subseteq EXPTIME$

Tipologie di problemi

Consideriamo:

I = universo delle possibili istanze di input

S = universo delle soluzioni

π = universo dei problemi

Di norma i problemi che affrontiamo rientrano in 3 macro-categorie:

- Ricerca: dato $x \in I$, trovare $s \in S$ t.c. $(x, s) \in I$
- Ottimizzazione: dato $x \in I$, trovare $s \in S$ t.c. $(x, s) \in I$ che soddisfi un criterio di ottimalità fissato (ad esempio minimo o massimo)
- Decisione: data un universo di soluzioni $S = 0, 1$:
 $(x, 1) \in \pi$ istanze positive ovvero $\pi(x) = 1$ $(x, 0) \in \pi$ istanze negative ovvero $\pi(x) = 0$

Dato un problema di ottimizzazione possiamo passare ad un problema di decisione, ad esempio:

- Ottimizzazione: dato un grafo $G(V, E, \omega)$ trovare l'albero ricoprente minimo
- Decisione: dato un grafo $G(V, E, \omega)$ con $K \in N$ esiste l'albero ricoprente di G di peso $\leq K$?

Relazione spazio/tempo

Consideriamo un tempo t (ovvero con tempo fissato), ogni istruzione elementare visita al massimo k celle di memoria. In particolare il numero di celle visitate (lo spazio) è limitato dal tempo t :

$$t \leq kt$$

Possiamo quindi concludere che un tempo polinomiale implica uno spazio polinomiale:

$$P \subseteq PSPACE$$

Consideriamo invece uno spazio s (ovvero con spazio fissato), se consideriamo s locazioni di memoria di k bit ciascuno, abbiamo al massimo 2^k configurazioni diverse. Rispetto allo spazio abbiamo 2^{ks} configurazioni diverse dello spazio.

$$t \leq p2^{ks}$$

Possiamo quindi concludere che uno spazio polinomiale implica un tempo al massimo esponenziale.

$$PSPACE \subseteq EXPTIME$$

Possiamo infine concludere che:

$$P \subseteq PSPACE \subseteq EXPTIME$$

Nessuno è riuscito a verificare che P e $EXPTIME$ sono propri. Si sa tuttavia che almeno una delle due è propria. Sappiamo infatti che $P \subset EXPTIME$

Problemi difficili

Problema Clique

- Istanza: dato un grafo non orientato $G = (V, E)$ ed un intero $k \geq 0$
- Problema: esiste un sottografo completo di G con k vertici?

Problema delle formule booleane — soddisfacibilità SODD

- Istanza: data una formula booleana ϕ in forma normale congiuntiva con un insieme di variabili V
- Problema: esiste un assegnamento delle variabili in V che rende vera ϕ , ovvero
- Una modalità per approcciare il problema sarebbe di verificare tutti i possibili assegnamenti di valori alle variabili x_1, \dots, x_n

$$\phi(f) = 1$$

Algoritmi non deterministici

Un algoritmo non deterministico è un algoritmo che, anche per lo stesso input, può esibire comportamenti diversi su corse diverse. In altre parole, è un algoritmo in cui il risultato di ogni algoritmo non è definito in modo univoco e il risultato potrebbe essere casuale.

La classe NP

La classe NP è la classe dei problemi di decisione risolvibili in tempo polinomiale da algoritmi non deterministici, oppure la classe dei problemi di decisione con certificati verificabili in tempo polinomiale. Gli algoritmi che risolvono problemi NP possono essere divisi in due fasi:

- fase non deterministica: che consiste nella costruzione del certificato
- fase deterministica: che consiste nella fase di verifica del certificato in tempo polinomiale.

In particolare possiamo formalizzare che

$$P \subseteq NP$$

Possiamo tuttavia dire che $P \neq NP$? Non lo sappiamo:

- in particolare se $P = NP$ ciò ci permettere di determinare se i problemi che possono essere verificati in tempo polinomiale possono essere anche risolti in tempo polinomiale
- invece se $P \neq NP$ ciò ci permetterebbe di dire che problemi in NP sono più difficili da computare che da verificare, ovvero non possono essere risolti in tempo polinomiale, ma la risolta è verificabile in tempo polinomiale