
ALGORITMI
E
STRUTTURE DATI

DISPENSE NON UFFICIALI

LEONARDO SOLARI



UNIVERSITÀ DEGLI STUDI DI MILANO
DIPARTIMENTO DI INFORMATICA

Indice dei contenuti

1	Introduzione	1
1.1	Algoritmica	1
1.2	Pseudocodice	1
1.3	Analisi e progettazione di algoritmi	1
1.4	Notazioni asintotiche	2
1.4.1	Limitazione superiore	2
1.4.2	Limitazione inferiore	2
1.4.3	Stesso ordine di grandezza	2
2	Criterio di costo	3
2.1	Costo uniforme	3
2.2	Costo logaritmico	3
3	Ricerca in un array	4
3.1	Ricerca sequenziale	4
3.2	Ricerca binaria o dicotomica	5
3.2.1	Alcune note sullo pseudocodice	6
4	Algoritmi di ordinamento o sorting	7
5	Selection Sort	8
6	Insertion Sort	9
7	Bubble Sort	10
7.1	Una considerazione su confronti e spostamenti	12
8	Merge Sort	13
9	Quick Sort	17
9.1	Partiziona	17
10	Strutture dati	21
11	Liste concatenate lineari	22
11.1	Operazioni	22
12	Stack (Pila)	24
13	Queue (Coda)	25
14	Alberi	27
14.1	Rappresentazione di alberi	27
14.1.1	Vettore dei padri	27
14.1.2	Rappresentazioni collegate: puntatori ai figli e lista dei fratelli	28
14.2	Visite di alberi	28

15 Alberi binari di ricerca	30
15.1 Operazioni	30
16 Altri tipi di alberi	34
16.1 Alberi perfettamente bilanciati	34
16.2 Alberi bilanciati in altezza o AVL	34
16.3 Alberi 2-3	34
16.3.1 Operazioni	35
16.3.2 Costo operazioni	37
16.4 B-Alberi	37
17 Heapsort	38
17.1 La struttura dati <i>Heap</i>	38
17.2 Sistemare uno heap	38
17.3 Creazione di uno heap	40
17.4 Schema di <code>heapSort</code>	41
17.5 Ordinamento in loco di array tramite <code>heapSort</code>	42
17.6 Spazio	44
17.7 Costo operazioni su heap	44
17.8 Riassumendo	44
18 Riassunto ordinamento	45
18.1 Numero minimo di confronti	45
19 Code con priorità	47
20 Ordinamento senza confronti	48
20.1 IntegerSort	48
20.2 BucketSort	49
20.3 RadixSort	50
21 Union-Find	51
21.1 Operazioni QUICKFIND	51
21.2 Operazioni QUICKUNION	52
21.3 Algoritmo QUICKFIND bilanciato	52
21.4 Algoritmo QUICKUNION bilanciato	52
21.5 Compressione di cammino	53
21.6 Riepilogo costi operazioni	53
22 Grafi	54
22.1 Rappresentazione di grafi	56
22.1.1 Lista di archi	56
22.1.2 Lista di adiacenza	57
22.1.3 Lista di incidenza	57
22.1.4 Matrice di adiacenza	58
22.1.5 Matrice di incidenza	59
22.2 Attraversamento di grafi	60

22.2.1	Visita in ampiezza	60
22.2.2	Visita in profondità	61
23	Ottimizzazione	62
23.0.1	Grafi pesati	62
23.0.2	Problemi di ottimizzazione	62
23.0.3	Tecnica greedy	62
23.0.4	Programmazione dinamica	63
23.1	Albero ricoprente minimo	64
23.1.1	Algoritmo di Kruskal	65
23.1.2	Algoritmo di Prim	67
23.2	Cammini minimi	70
23.2.1	Algoritmo di Floyd-Warshall	71
23.2.2	Algoritmo di Bellman e Ford	73
23.2.3	Algoritmo di Dijkstra	74
24	Dizionari e tabelle hash	76
24.1	Funzioni hash	76
24.2	Fattore di carico	76
24.3	Gestione delle collisioni	76
24.4	Gestione esterna	77
24.5	Gestione interna	78
24.5.1	Scansione quadratica	78
24.5.2	Hashing doppio	78
24.5.3	Operazioni	78
24.5.4	Numero di confronti	79
24.5.5	Re-hashing	79
25	Complessità Computazionale	80
25.1	Classi di complessità	80
25.2	Problemi NP-completi	81
25.2.1	Problema soddisfacibilità (SODD)	81
25.3	Relazioni tra classi di complessità	82

Prefazione

Le seguenti dispense nascono con lo scopo di fornire ai colleghi una fonte contenente un riassunto di tutti gli argomenti trattati nel corso di algoritmi e strutture dati tenuto dal professor Pighizzini. Per la creazione sono stati utilizzati i documenti PDF forniti dal professore dal quale sono stati estratti i frammenti di codice degli algoritmi e la loro spiegazione. Tali informazioni sono poi state integrate con appunti presi personalmente durante le lezioni e da materiale condiviso da altri colleghi.

Queste dispense non sostituiscono il libro di testo e i materiali ufficiali forniti dal professore, bensì sono da considerarsi unicamente uno strumento integrativo.

Parte del contenuto di queste dispense è stato riprodotto dai documenti del professor Pighizzini, che ne detiene la proprietà, per scopi istituzionali, come indicato dalla dicitura di copyrigth che viene qui riportata per intero.

©2022 Giovanni Pighizzini

Il contenuto di queste pagine è protetto dalle leggi sul copyright e dalle disposizioni dei trattati internazionali. Il titolo ed i copyright relativi alle pagine sono di proprietà dell'autore. Le pagine possono essere riprodotte ed utilizzate liberamente dagli studenti, dagli istituti di ricerca, scolastici e universitari afferenti al Ministero dell'Istruzione e al Ministero dell'Università e della Ricerca, per scopi istituzionali, non a fine di lucro. Ogni altro utilizzo o riproduzione (ivi incluse, ma non limitatamente a, le riproduzioni a mezzo stampa, su supporti magnetici o su reti di calcolatori) in toto o in parte è vietata, se non esplicitamente autorizzata per iscritto, a priori, da parte dell'autore. L'informazione contenuta in queste pagine è ritenuta essere accurata alla data della pubblicazione. Essa è fornita per scopi meramente didattici e non per essere utilizzata in progetti di impianti, prodotti, ecc. L'informazione contenuta in queste pagine è soggetta a cambiamenti senza preavviso. L'autore non si assume alcuna responsabilità per il contenuto di queste pagine (ivi incluse, ma non limitatamente a, la correttezza, completezza, applicabilità ed aggiornamento dell'informazione). In ogni caso non può essere dichiarata conformità all'informazione contenuta in queste pagine. In ogni caso questa nota di copyright non deve mai essere rimossa e deve essere riportata anche in utilizzi parziali.

1 Introduzione

Un algoritmo è una strategia o un procedimento per risolvere un problema, uno schema o un procedimento sistematico di calcolo. Formalmente:

Un algoritmo è un insieme ordinato e finito di passi eseguibili e non ambigui che definiscono un procedimento che termina

Matematicamente un algoritmo può essere visto come una funzione

$$f_a : D_I \rightarrow D_S$$

dove D_I rappresenta il **dominio delle istanze** e D_S il **dominio delle soluzioni**.

1.1 Algoritmica

L'algoritmica si occupa di:

- Risoluzione di problemi → **Sintesi**
- Trovare una strategia buona per risolvere i problemi → **Analisi** efficienza
- Stabilire se un problema è facile o difficile → **Classificazione** della complessità dei problemi
- Studio delle strutture dati utilizzate
- Definizione di nuovi modelli di calcolo

L'algoritmica viene studiata per scrivere programmi. Ha due aspetti:

- **Pratico:** un computer è inutile senza algoritmi e programmi
- **Teorico:** gli algoritmi sono la base dell'informatica, sono uno strumento mentale e metodologico per risolvere i problemi.

1.2 Pseudocodice

Per scrivere gli algoritmi useremo uno pseudocodice con strutture di controllo "Algol-like"

Algorithm 1: <i>moltiplicazione</i>
Algoritmo <i>moltiplicazione</i> (intero a , intero b) → intero
return $a \cdot b$

1.3 Analisi e progettazione di algoritmi

Esistono varie metodologie per progettare algoritmi. In base al tipo di utilizzo e alle operazioni che dovrò effettuare utilizzo strutture dati differenti. L'analisi e la progettazione di algoritmi si basano fondamentalmente su due fattori:

- **Correttezza:** dato un algoritmo a e un problema P , dimostrare che a risolve P

- **Efficienza:** valutare la complessità di un algoritmo e la quantità di risorse utilizzate(tempo, spazio, energia, rete, ecc...)

Per eseguire l'analisi di un algoritmo posso:

1. Far girare il programma (*testing*) → **valutazione a posteriori**

Questo approccio ha alcuni problemi:

- Possono esistere infiniti ingressi possibili
- costo della codifica elevato

2. Stima in fase di progettazione → **valutazione a priori**

Per stimare il consumo di tempo di un programma assumo che ogni linea di codice costi tempo unitario.

1.4 Notazioni asintotiche

Siano f e g due funzioni:

$$f, g : \mathbb{N} \rightarrow \mathbb{R}^+$$

1.4.1 Limitazione superiore

$f(n)$ è O-grande di $g(n)$ se $\exists c > 0, n_0 \in \mathbb{N} \mid \forall n > n_0: f(n) \leq c \cdot g(n)$

1.4.2 Limitazione inferiore

$f(n)$ è Ω -grande di $g(n)$ se $\exists c > 0, n_0 \in \mathbb{N} \mid \forall n > n_0: f(n) \geq c \cdot g(n)$

1.4.3 Stesso ordine di grandezza

$f(n)$ è Θ -grande di $g(n)$ se $\exists c, d > 0, n_0 \in \mathbb{N} \mid \forall n > n_0: c \cdot g(n) \leq f(n) \leq d \cdot g(n)$

2 Criterio di costo

Suppongo di avere un algoritmo che trova il minimo in una sequenza di dati.

Se la sequenza è lunga n elementi allora vengono fatti $n-1$ confronti ed un numero di assegnamenti compreso tra n e $2n$.

Assumendo che queste operazioni vengano effettuate in tempo costante il tempo è $O(n)$, quindi posso dire che è $\Theta(n)$.

2.1 Costo uniforme

Ogni istruzione elementare utilizza un'unità di tempo indipendentemente dalla grandezza degli operandi.

Ogni variabile elementare utilizza un'unità di spazio indipendentemente dal valore contenuto.

2.2 Costo logaritmico

Il tempo di calcolo di ciascuna operazione è proporzionale alla lunghezza dei valori coinvolti.

3 Ricerca in un array

Input: array A , elemento x

Output:

- indice i t.c. $A[i] = x$
- -1 se A non contiene x

3.1 Ricerca sequenziale

Codice 1.1 Ricerca sequenziale

```
Algoritmo ricercaSequenziale (array  $A[0..n - 1]$ , elemento  $x$ ) → indice
   $i \leftarrow 0$ 
  while  $i < n$  and  $A[i] \neq x$  do
     $i \leftarrow i + 1$ 
  if  $i = n$  then return -1
  else return  $i$ 
```

Posso rendere l'algoritmo più "intelligente" cercando a partire dal fondo. In questo modo se l'elemento non è nell'array l'indice diventa automaticamente -1.

Codice 1.2 Ricerca sequenziale

Ricercando dal fondo si evita la selezione finale presente nell'Algoritmo 1.1.

```
Algoritmo ricercaSequenziale (array  $A[0..n - 1]$ , elemento  $x$ ) → indice
   $i \leftarrow n - 1$ 
  while  $i \geq 0$  and  $A[i] \neq x$  do
     $i \leftarrow i - 1$ 
  return  $i$ 
```

Tempo: $\Theta(n)$

3.2 Ricerca binaria o dicotomica

Se ho un array ordinato posso usare un algoritmo di ricerca binaria.

Codice 1.3 Ricerca binaria o dicotomica ricorsiva in un array ordinato

La funzione ricorsiva effettua la ricerca nella parte dell'array A che inizia all'indice sx e termina all'indice che *precede* dx (pertanto sx è l'indice del primo elemento della porzione di array da considerare, mentre dx è l'indice del primo elemento, tra quelli successivi, da non considerare).

Algoritmo *ricercaBinaria* (array $A[0..n - 1]$, elemento x) \rightarrow indice
return *ricercaRic*($A, 0, n, x$)

Funzione *ricercaRic* (array A , indice sx , indice dx , elemento x) \rightarrow indice

```

if  $dx \leq sx$  then return  $-1$ 
else
   $m \leftarrow (sx + dx)/2$ 
  if  $x = A[m]$  then return  $m$ 
  else if  $x < A[m]$  then
    return ricercaRic( $A, sx, m, x$ )
  else
    return ricercaRic( $A, m + 1, dx, x$ )
  
```

Codice 1.4 Ricerca binaria o dicotomica iterativa in un array ordinato

Algoritmo *ricercaBinaria* (array $A[0..n - 1]$, elemento x) \rightarrow indice

```

 $sx \leftarrow 0$ 
 $dx \leftarrow n$ 
 $pos \leftarrow -1$ 
while  $sx < dx$  and  $pos = -1$  do
   $m \leftarrow (sx + dx)/2$ 
  if  $x = A[m]$  then  $pos \leftarrow m$ 
  else if  $x < A[m]$  then  $dx \leftarrow m$ 
  else  $sx \leftarrow m + 1$ 
return  $pos$ 
  
```

3.2.1 Alcune note sullo pseudocodice

- *Indici degli array*

Quando si definisce un array (in questo caso nei parametri degli algoritmi o della funzione), il range di indici viene indicato qualora sia rilevante per la scrittura dell'algoritmo. Quando non sia rilevante o sia chiaro dal contesto, il range viene omesso (come, in questo esempio, per il parametro A della funzione ricercaRic).

- *Operatori logici*

Assumiamo che per congiunzione (*and*) e disgiunzione (*or*) sia utilizzata la *lazy-evaluation*. Pertanto in una condizione della forma *a and b* la condizione *b* viene valutata solo se *a* è vera, mentre in una condizione della forma *a or b* la seconda viene valutata solo se *a* è falsa.

- *Passaggio di parametri*

Assumiamo che per i tipi semplici il passaggio di parametro avvenga sempre *per valore* mentre per i tipi strutturati avvenga il passaggio *per riferimento*.

4 Algoritmi di ordinamento o sorting

In questa sezione vediamo alcuni algoritmi che servono per ordinare vettori di strutture complesse come oggetti o record. Un particolare campo è scelto come **chiave** per l'ordinamento. Studieremo principalmente algoritmi di ordinamento basati su confronti tra chiavi e stimeremo la complessità di questi algoritmi in funzione della lunghezza del vettore da ordinare, calcolando prima di tutto il numero di confronti eseguiti.

Un algoritmo di ordinamento è detto **stabile** se preserva l'ordine relativo tra record con la medesima chiave. Esistono due tipologie di ordinamento:

1. *Ordinamento interno:*

I dati da ordinare sono in memoria centrale → accesso diretto agli elementi

2. *Ordinamento esterno:*

I dati da ordinare sono in memoria di massa → accesso ai blocchi di dati con possibile lentezza dovuta dall'hardware delle periferiche.

Vedremo principalmente tecniche di ordinamento interno, tra cui troviamo tecniche

1. **Elementari**

Utilizzano nel caso peggiore un numero quadratico di confronti

- Per selezione (*SelectionSort*)
- Per inserimento (*InsertionSort*)
- A bolle (*BubbleSort*)

2. **Avanzate**

Utilizzano un numero di confronti dell'ordine di $n \log n$ (tranne *QuickSort*, il cui caso peggiore risulta però molto raro)

- Per fusione (*MergeSort*)
- Veloce (*QuickSort*)
- Basato su heap (*HeapSort*)

5 Selection Sort

1. Prima del passo principale k , con $k = 0, \dots, n - 1$, i primi k elementi dell'array sono al loro posto definitivo, cioè sono ordinati tra loro e sono minori o uguali degli elementi successivi
2. Si seleziona l'elemento che andrà collocato in posizione k , cioè il minimo della parte non ordinata (quindi il minimo tra $A[k], \dots, A[n - 1]$)
3. Lo si colloca in posizione k , scambiandolo con l'elemento ivi presente
4. In questo modo, dopo il passo principale k , i primi k elementi risultano collocati nella loro posizione definitiva.

Dopo il passo $n - 2$ la parte non ordinata contiene solo un elemento e, in base al punto 1, questo è maggiore o uguale dei precedenti, e dunque si trova nella sua posizione definitiva. Pertanto non è necessario eseguire il passo $n - 1$

Codice 2.1 Ordinamento per selezione

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

Numero di confronti

Nell'iterazione k del ciclo principale viene ricercato il minimo della porzione di vettore da posizione k a posizione $n - 1$, effettuando quindi $n - k - 1$ confronti. Sommando su tutte le iterazioni k del ciclo principale otteniamo il numero totale di confronti $\frac{n(n-1)}{2} = \Theta(n^2)$, che vengono eseguiti sempre indipendentemente dal contenuto dell'array.

Spazio

L'algoritmo, oltre all'array da ordinare, utilizza un numero costante di variabili. Pertanto la quantità di spazio aggiuntivo è costante.

6 Insertion Sort

1. Si memorizza l'elemento $A[k]$ da sistemare in una variabile x
2. Si ispeziona la porzione di array $A[0..k - 1]$ *da destra verso sinistra*, spostando avanti di una posizione ogni elemento maggiore di x , in modo da "fare posto" all'elemento da inserire
3. Individuata la posizione in cui inserire x (quindi quando si raggiunge un elemento che non è maggiore di x o quando si è ispezionata tutta la porzione iniziale di array), si inserisce x (gli elementi successivi sono già stati spostati durante il passo 3)

Codice 2.2 Ordinamento per inserimento

```

Algoritmo insertionSort (array A[0..n - 1])
  for  $k \leftarrow 1$  to  $n - 1$  do
     $x \leftarrow A[k]$                                 // 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
     $j \leftarrow k - 1$ 
    while  $j \geq 0$  and  $A[j] > x$  do
       $A[j + 1] \leftarrow A[j]$                       // sposta in avanti l'elemento  $A[j]$ 
       $j \leftarrow j - 1$ 
     $A[j + 1] \leftarrow x$                           // inserisce  $x$ 
  
```

Numero di confronti

Nel caso peggiore ogni elemento dell'array viene confrontato con ogni altro elemento dell'array. Pertanto vengono effettuati $\frac{n(n-1)}{2} = \Theta(n^2)$ confronti. Il caso peggiore si verifica nel caso in cui l'array sia ordinato al contrario, mentre nel caso migliore, ovvero quello in cui l'array è già ordinato, vengono effettuati $n - 1$ confronti.

Spazio

L'algoritmo, oltre all'array da ordinare, utilizza un numero costante di variabili. Pertanto la quantità di spazio aggiuntivo è costante.

7 Bubble Sort

L'idea di base è quella di scandire ripetutamente l'array dal primo all'ultimo elemento scambiando tra loro gli elementi adiacenti che non risultino ordinati. L'array sarà ordinato quando si riuscirà ad effettuare una scansione senza alcuno scambio.

Codice 2.3 Ordinamento “a bolle” (versione base)

```

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

Esempio di esecuzione versione base

<i>indici</i>	0	1	2	3	4	5
contenuto iniziale	7	2	4	5	3	1
dopo la 1a iterazione	2	4	5	3	1	7
dopo la 2a iterazione	2	4	3	1	5	7
dopo la 3a iterazione	2	3	1	4	5	7
dopo la 4a iterazione	2	1	3	4	5	7
dopo la 5a iterazione	1	2	3	4	5	7
dopo la 6a iterazione	1	2	3	4	5	7
						nessuno scambio effettuato!

Dopo la *i*-esima iterazione, gli ultimi *i* elementi dell'array sono al loro posto definitivo e dunque non è più necessario esaminarli. Per la stessa ragione dopo *n* - 1 scansioni, gli *n* - 1 elementi più grandi hanno raggiunto la loro posizione e di conseguenza l'elemento più piccolo deve trovarsi nell'unica posizione che resta, ovvero quella di indice 0. Pertanto, dopo aver effettuato *n* - 1 iterazioni l'algoritmo si può fermare anche se nell'ultima scansione ci sono stati scambi. Possiamo quindi scrivere una versione migliorata dell'algoritmo.

Versione migliorata

Codice 2.4 Ordinamento “a bolle” (versione migliorata)

```

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

Esempio di esecuzione versione migliorata

<i>indici</i>	0	1	2	3	4	5	
contenuto iniziale	7	1	4	3	5	6	
dopo la 1a iterazione	1	4	3	5	6	7	
dopo la 2a iterazione	1	3	4	5	6	7	
dopo la 3a iterazione	1	3	4	5	6	7	nessuno scambio effettuato!

Numero di confronti

Nell’iterazione *i* del ciclo principale si effettuano esattamente $n - 1$ confronti. Il ciclo principale viene eseguito a partire da *i* = 1, incrementando fino al più a *n* − 1. Pertanto sommando su tutte le iterazioni il numero di confronti è al massimo $\frac{n(n-1)}{2} = \Theta(n^2)$ nel caso peggiore, che si ha quando l’array è ordinato al contrario. Se invece l’array è già ordinato il numero di confronti è *n* − 1

Spazio

L’algoritmo, oltre all’array da ordinare, utilizza un numero costante di variabili. Pertanto la quantità di spazio aggiuntivo è costante.

7.1 Una considerazione su confronti e spostamenti

Abbiamo detto che la stima del tempo di calcolo di questi algoritmi può avvenire a partire da quella del numero di confronti, moltiplicando il numero di confronti per il tempo necessario per effettuare ciascun confronto. Questo è vero a patto che i confronti tra chiavi siano le operazioni più costose effettuate dagli algoritmi. Potremmo calcolare anche il numero di spostamenti di elementi. Da questo calcolo possiamo scoprire che tra i tre algoritmi presentati sopra, l'ordinamento per selezione è quello che effettua un numero di spostamenti più basso. Ma quanto costano gli spostamenti in termini di tempo? Come per i confronti, se stiamo ordinando numeri interi di grandezza fissata, come i valori dei tipi `int` e `long`, gli spostamenti sono effettuati mediante assegnamenti che copiano un numero fissato di bit, e quindi avvengono in tempo costante. Se tuttavia, come avviene nella pratica, stiamo ordinando rispetto a un campo chiave dei record di grandi dimensioni, la copia di interi record diventa costosa in termini di tempo e il numero di spostamenti può essere un parametro critico, anche più importante del numero di confronti, per valutare il tempo impiegato da un algoritmo. Questo problema può essere evitato utilizzando i puntatori: anziché memorizzare negli elementi dell'array i record da ordinare, possiamo memorizzare i puntatori ad essi. Quindi, ogni cella dell'array conterrà il puntatore a un record, memorizzato altrove. In questo modo, per effettuare uno spostamento, non è necessario copiare l'intero record, ma solo copiare dei puntatori ai record. La dimensione dei puntatori può essere considerata costante (dipende dalla grandezza delle memorie che può essere indirizzata).

8 Merge Sort

L'algoritmo di ordinamento per fusione si basa sul seguente schema:

- Un array di un solo elemento è già ordinato (*caso base*)
- Per ordinare un array A contenente $n > 1$ elementi possiamo:
 1. suddividere A in due array B e C di $n/2$ elementi ciascuno, corrispondenti alla prima e alla seconda metà dell'array A (Nel caso n sia dispari le due metà saranno di $\lfloor n/2 \rfloor$ e $\lceil n/2 \rceil$)
 2. ordinare separatamente gli array
 3. "fondere" gli array ordinati B e C nell'array A , in modo da ottenere un array ordinato contenente gli elementi di B e C .

L'operazione di fusione (`merge`) di due array ordinati in un array ordinato è più semplice rispetto all'ordinamento di un array.

Studiamo come effettuare il merge. Disponiamo di due vettori B e C ordinati in modo non decrescente, e vogliamo ottenere un vettore X ordinato che contenga gli stessi elementi di B e C .

1. Creiamo un vettore X la cui lunghezza sia la somma delle lunghezze di B e C
2. Ispezioniamo B e C iniziando a considerare gli elementi minimi, ovvero quelli nella prima posizione dei due vettori
3. Confrontiamo i due elementi e scegliamo il minimo, copiandolo nella prima posizione libera di X . Inoltre, nel vettore da cui abbiamo preso l'elemento, possiamo considerare quello di posizione successiva.
4. Ripetiamo le operazioni precedenti fino a raggiungere la fine di uno dei due array
5. Copiamo in X tutti gli elementi rimanenti dell'altro array

Numero di confronti

Indicando con $C(n)$ il numero di confronti effettuato da `mergeSort` possiamo scrivere la seguente equazione di ricorrenza.

$$C(n) = \begin{cases} C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil) + C_{\text{merge}}(n) & \text{se } n > 1 \\ 0 & \text{altrimenti} \end{cases}$$

Nel caso peggiore $C_{\text{merge}}(n) = n - 1$. Risolvendo per sostituzione otteniamo $C(n) = \Theta(n \log n)$

Tempo di calcolo

Ci sono varie operazioni costose sia in termini di tempo che in termini di spazio: dobbiamo creare due array B e C , copiarvi gli elementi di A e, dopo il merge, ricopiare tutti gli elementi nell'array iniziale. Indicando con $T(n)$ il tempo utilizzato dall'algoritmo per ordinare un array di lunghezza n osserviamo che:

- Se l'array contiene al più 1 elemento, l'algoritmo usa tempo costante. Indichiamo tale tempo con a
- Se l'array contiene $n > 1$ elementi, allora $T(n)$ è la somma dei seguenti tempi:
 - Tempo per la creazione dei due array: $\Theta(n)$
 - Tempo per ordinare i due array: $T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil)$
 - Tempo per il merge: $\Theta(n)$

Possiamo quindi scrivere l'equazione di ricorrenza

$$T(n) = \begin{cases} 2T(n/2) + bn + c & \text{se } n > 1 \\ a & \text{altrimenti} \end{cases}$$

dove b e c sono due costanti.

Risolvendo per sostituzione ottengo $T(n) = \Theta(n \log n)$

Implementazione

Come abbiamo visto, l'implementazione di una versione base di `mergeSort` richiede l'uso di tempo e spazio per gli array B e C . Possiamo implementare l'algoritmo in maniera differente, servendoci direttamente dell'array A da ordinare e di due indici che delimitano la parte da ordinare. Al contrario, per effettuare la procedura `merge` ci serviremo di un array ausiliario, che per evitare sprechi verrà creato preliminarmente e verrà usato da tutte le chiamate di `merge`. La precedente analisi relativa al numero di confronti non cambia e il tempo rimane dell'ordine di $n \log n$.

Codice 3.3 Ordinamento per fusione

Al posto degli array ausiliari B e C (che richiedono uso di memoria e di tempo per il trasferimento dei dati) si utilizza l'array A stesso con indici che delimitano le parti da ordinare. Si utilizza un array ausiliario X per le operazioni di merge. X potrebbe essere definito localmente alla procedura $merge$, in quanto viene utilizzato solo da essa. In questo modo tuttavia occorrerebbe un nuovo array ausiliario ad ogni chiamata di $merge$. Per evitare ciò, l'array ausiliario viene definito a livello globale.

Algoritmo $mergeSort$ (array $A[0..n - 1]$)

Sia $X[0..n - 1]$ un array
 $mergeSort(A, 0, n, X)$

Procedura $mergeSort$ (array A , indice i , indice f , array X)

/* Ordina $A[i..f - 1]$ utilizzando X come array ausiliario */

```
if  $f - i > 1$  then
     $m \leftarrow (i + f)/2$ 
     $mergeSort(A, i, m, X)$ 
     $mergeSort(A, m, f, X)$ 
     $merge(A, i, m, f, X)$ 
```

Procedura $merge$ (array A , indice i , indice m , indice f , array X)

/* Merge tra $A[i..m - 1]$ e $A[m..f - 1]$ utilizzando X come array ausiliario */

```

 $i_1 \leftarrow i$           // Prima parte: merge di  $A[i..m - 1]$  e  $A[m..f - 1]$  in  $X[0..f - i - 1]$ 
 $i_2 \leftarrow m$ 
 $k \leftarrow 0$ 
while  $i_1 < m$  and  $i_2 < f$  do
    if  $A[i_1] \leq A[i_2]$  then
         $X[k] \leftarrow A[i_1]$ 
         $i_1 \leftarrow i_1 + 1$ 
    else
         $X[k] \leftarrow A[i_2]$ 
         $i_2 \leftarrow i_2 + 1$ 
     $k \leftarrow k + 1$ 
if  $i_1 < m$  then
    for  $j \leftarrow i_1$  to  $m - 1$  do
         $X[k] \leftarrow A[j]$ 
         $k \leftarrow k + 1$ 
else
    for  $j \leftarrow i_2$  to  $f - 1$  do
         $X[k] \leftarrow A[j]$ 
         $k \leftarrow k + 1$ 
for  $k \leftarrow 0$  to  $f - i - 1$  do      // Seconda parte: copia il risultato in  $A[i..f - 1]$ 
     $A[i + k] \leftarrow X[k]$ 
```

Spazio

L'algoritmo non è in loco, in quanto utilizza un array ausiliario per effettuare il merge. L'array è di n elementi, quindi usa spazio $\Theta(n)$. Dobbiamo inoltre considerare lo spazio utilizzato dallo stack per gestire la ricorsione. In ciascun record di attivazione di `mergeSort` devono essere memorizzati gli indici i ed f che servono a delimitare la porzione di array da delimitare e la variabile m . Pertanto la dimensione di ogni record è costante. Per calcolare l'altezza dello stack utilizziamo un'equazione di ricorrenza.

- Se $n \leq 1$ (caso base) non viene effettuata alcuna chiamata ricorsiva. Pertanto viene utilizzato solo il record di attivazione corrente e $H(n) = 1$
- Se $n > 1$ viene effettuata una prima chiamata ricorsiva su un array di lunghezza $\lfloor n/2 \rfloor$, che dunque utilizzerà altezza $H(\lfloor n/2 \rfloor)$. Terminata tale chiamata si effettua una seconda chiamata sull'altra parte di array, quindi con altezza $H(\lceil n/2 \rceil)$. Poiché al termine di ciascuna chiamata ricorsiva lo stack viene riportato all'altezza che aveva prima della chiamata, la parte di stack utilizzata dalla prima chiamata viene riutilizzata per la seconda. Pertanto l'altezza dello stack utilizzata dalle due chiamate è il massimo tra $H(\lfloor n/2 \rfloor)$ e $H(\lceil n/2 \rceil)$.

Otteniamo dunque

$$H(n) = \begin{cases} \max(H(\lfloor n/2 \rfloor), H(\lceil n/2 \rceil)) + 1 & \text{se } n > 1 \\ 1 & \text{altrimenti} \end{cases}$$

Questo ci permette di concludere che l'altezza dello stack è logaritmica rispetto a n , ed è in particolare $\Theta(\log n)$.

9 Quick Sort

Supponiamo di dover ordinare la sequenza di numeri

44 55 12 42 94 6 18 67

Scegliamo all'interno di essa un qualunque elemento, ad esempio 42 (che chiameremo *perno* o *pivot*), e costruiamo due sequenze nelle quali collichiamo rispettivamente tutti gli elementi minori o uguali al perno e tutti quelli maggiori, in qualunque ordine:

12 42 6 18 44 55 94 67

Ordinando separatamente le due sequenze e concatenandole otteniamo la sequenza ordinata:

6 12 18 42 44 55 67 94

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  $\leftarrow \{y \in A \mid y \leq \text{perno}\}$ 
    C  $\leftarrow \{y \in A \mid y > \text{perno}\}$ 
    quickSort(B)
    quickSort(C)
    A  $\leftarrow$  concatenazione di B e C
```

9.1 Partiziona

Per creare la partizione dell'array procediamo nel seguente modo:

1. Scegliamo come perno l'elemento più a sinistra dell'array
2. Scansioniamo l'array da destra verso sinistra fino al primo elemento minore o uguale al perno
3. Scansioniamo l'array da sinistra verso destra fino al primo elemento maggiore del perno
4. Se le due scansioni non si sono incontrate, scambiamo i due elementi individuati e proseguiamo le scansioni ai passi 2 e 3
5. Quando ogni elemento è stato confrontato con il perno, scambiamo il perno con l'elemento su cui si è arrestata la scansione da destra

Per le scansioni da destra e da sinistra utilizziamo due indici di nome *dx* e *sx*, che indicano gli elementi correntemente ispezionati dalle due scansioni. Ad ogni passo tutti gli elementi a sinistra dell'indice *sx* risultano minori o uguali al perno, mentre quelli a destra di *dx* maggiori del perno. Quando i due indici si incontrano o $sx \geq dx$ tutti gli elementi sono stati ispezionati. Inoltre,

l'elemento di indice dx è minore o uguale al perno. A questo punto è sufficiente scambiare questo elemento con il perno per ottenere la partizione.

contenuto iniziale	12 33 15 7 19 9 56 2	perno = 12
prima coppia da scambiare	12 33 15 7 19 9 56 2 ↑ _{sx}	↑ _{dx}
seconda coppia da scambiare	12 2 15 7 19 9 56 33 ↑ _{sx}	↑ _{dx}
gli indici si incontrano	12 2 9 7 19 15 56 33 ↑↑ _{sx} ^{dx}	
partizione	7 2 9 12 19 15 56 33 ≤12 >12	

Codice 4.2 Partizionamento di un array

```

Algoritmo partiziona (array A, indice i, indice f) → indice
/* Riorganizza gli elementi all'interno di  $A[i..f-1]$  e restituisce un
   indice  $j$  in modo tale che tutti gli elementi di  $A[i..j-1]$  siano minori o
   uguali ad  $A[j]$  e tutti gli elementi di  $A[j+1..f-1]$  siano maggiori
   di  $A[j]$  */
perno ←  $A[i]$ 
sx ← i
dx ← f
while sx < dx do
  do dx ← dx - 1 while  $A[\text{dx}] > \text{perno}$ 
  do sx ← sx + 1 while sx < dx and  $A[\text{sx}] \leq \text{perno}$ 
  if sx < dx then scambia  $A[\text{sx}]$  con  $A[\text{dx}]$ 
scambia  $A[i]$  con  $A[\text{dx}]$ 
return dx

```

Numero di confronti

Codice 4.3 Quicksort

```

Algoritmo quickSort (array A[0..n - 1])
    quickSort(A, 0, n)
Procedura quickSort (array A, indice i, indice f)           /* Ordina A[i..f - 1] */
    if f - i > 1 then
        m  $\leftarrow$  partiziona(A, i, f)
        quickSort(A, i, m)
        quickSort(A, m + 1, f)
    
```

Per effettuare la partizione ogni elemento dell'array deve essere confrontato con il perno (eccetto il perno stesso). Pertanto vi sono almeno $n - 1$ confronti. Per semplicità di calcolo utilizzeremo solo n

Caso peggiore

Nel caso peggiore $C_w(n)$ quickSort esegue il seguente numero di confronti:

$$C_w(n) = \begin{cases} n + \max C_w(n) + C_w(n - k - 1) & |0 \leq k \leq n| \\ 0 & \text{se } n > 1 \\ & \text{altrimenti} \end{cases}$$

Il secondo addendo della somma rappresenta il numero di confronti nelle chiamate ricorsive nell'ipotesi che, dopo la partizione, vi siano k elementi a sinistra del perno e $n - k - 1$ a destra. Dato che stiamo studiando il caso peggiore consideriamo il valore di k che massimizza la somma. Svolgendo i calcoli otteniamo $C_w(n) = \Theta(n^2)$. Pertanto nel caso peggiore (molto raro) quickSort effettua lo stesso numero di confronti degli algoritmi elementari che abbiamo studiato.

Caso migliore

Abbiamo visto che il caso peggiore si ottiene quando ad ogni livello della ricorsione la partizione risulta sbilanciata. Se, al contrario, l'array viene sempre suddiviso in due parti circa della stessa lunghezza, il numero di confronti diminuisce drasticamente.

$$C_b(n) = \begin{cases} n + 2C_b(n/2) & \text{se } n > 1 \\ 0 & \text{altrimenti} \end{cases}$$

Svolgendo i calcoli otteniamo $C_b(n) = n \log_2 n$

Caso medio

Il numero di confronti effettuato da quickSort dipende dalla distribuzione dei valori all'interno dell'array. Si può calcolare che il caso medio $C(n) \leq 1.39n \log_2 n$, molto vicino al caso migliore e a mergeSort, motivo per cui quickSort viene utilizzato molto spesso.

Spazio di lavoro

L'algoritmo è in loco ma utilizza spazio aggiuntivo per la ricorsione. Ogni record di attivazione deve contenere i parametri i ed f che delimitano la parte di array da ordinare, oltre alla variabile m . Dunque la grandezza di ciascun record di attivazione è costante. La quantità di memoria utilizzata è proporzionale all'altezza raggiunta dallo stack, che nel caso peggiore è n . Si può modificare l'algoritmo in modo che l'altezza dello stack sia sempre $O(\log n)$ eliminando una chiamata ricorsiva e ordinando prima la parte destra dell'array e poi la sinistra.

Alcune osservazioni

Possiamo osservare che le prestazioni di `quickSort`, su uno stesso array possono variare notevolmente in base alla strategia utilizzata per scegliere il perno. Spesso, per evitare il caso peggiore (array già ordinato), si utilizzano strategie "randomizzate". Una possibilità è quella di disordinare in modo casuale gli elementi dell'array prima di eseguire l'algoritmo, un'altra può essere scegliere un elemento casuale dell'array da usare come perno e scambiarlo con il primo elemento, applicando poi la strategia di partizione che abbiamo visto. Si può osservare che questo metodo di ordinamento non è stabile.

10 Strutture dati

Le strutture dati consistono in una specifica organizzazione delle informazioni, che permette di realizzare ed implementare un determinato tipo di dati. La scelta della corretta struttura dati dipende dall'utilizzo che bisogna fare dei dati.

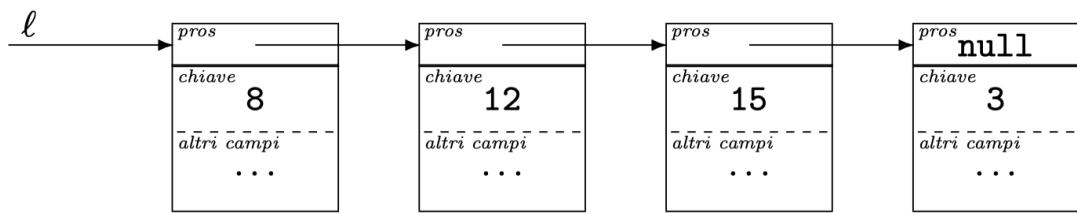
Il tipo di una variabile stabilisce i valori e le operazioni che possono essere eseguite. In generale quando parliamo del tipo non parliamo della rappresentazione del dato ma del "cosa". La rappresentazione influenza però sull'efficienza delle operazioni.

Consideriamo un esempio classico: il dizionario. Si tratta di una collezione di elementi ciascuno dei quali è caratterizzato da una chiave. Un esempio particolare di dizionario può essere quello della lingua italiana in cui ogni elemento ha due campi, *parola* e *definizione*, oppure la registrazione di uno studente, in cui ogni elemento ha tanti campi e la chiave è la *matricola*. Le chiavi in genere sono valori ordinabili.

In un dizionario dobbiamo poter svolgere le operazioni di **ricerca**, **inserimento** e **cancellazione**. A seconda del tipo di struttura dati e di implementazione che si sceglie alcune operazioni possono essere più facili da svolgere rispetto ad altre. Vediamo ora alcune strutture dati.

11 Liste concatenate lineari

Una lista concatenata lineare è una struttura composta da una collezione di nodi collegati linearmente tra loro tramite puntatori. Ogni **nodo** è contiene dei campi. Tra questi troviamo il campo *chiave*, rispetto al quale vengono effettuate le operazioni di ricerca, e il campo *pros*, che contiene un riferimento al nodo successivo. Nel caso delle liste ordinate, il campo *chiave* viene utilizzato per determinare l'ordine tra i nodi. Si accede alla lista tramite un riferimento al primo nodo. Le liste possono essere implementate tramite array o tramite strutture e puntatori. Noi studieremo il secondo tipo di implementazione.



11.1 Operazioni

Vediamo ora l'implementazione di alcune operazioni che si possono effettuare sulle liste concatenate lineari ordinate e non.

Codice 5.1 Ricerca in base alla posizione

```

Funzione elemento (Lista ℓ, intero i) → Nodo
  /* Restituisce il riferimento all'i-esimo nodo della lista o null se la
   lista è più corta (i nodi sono contati a partire da 0). */ 
  p ← ℓ
  while p ≠ null and i > 0 do
    p ← p.pros
    i ← i - 1
  return p
  
```

Codice 5.2 Ricerca in base alla chiave

```

Funzione trova (Lista ℓ, tipoChiave k) → Nodo
  /* Restituisce il riferimento al primo nodo della lista la cui chiave
   coincide con il valore fornito tramite il parametro k.
   Se tale chiave non è presente restituisce null. */ 
  p ← ℓ
  while p ≠ null and p.chiave ≠ k do
    p ← p.pros
  return p
  
```

Codice 5.3 Ricerca in una *lista ordinata* in base alla chiave

Funzione *trova* (*ListaOrdinata* ℓ , *tipoChiave* k) \rightarrow *Nodo*

```

/* Restituisce il riferimento al primo nodo della lista la cui chiave
   coincide con il valore fornito tramite il parametro  $k$ .
   Se tale chiave non è presente restituisce null. */
```

```

 $p \leftarrow \ell$ 
while  $p \neq \text{null}$  and  $p.\text{chiave} < k$  do
   $\quad p \leftarrow p.\text{pros}$ 
if  $p = \text{null}$  or  $p.\text{chiave} > k$  then
  | return null
else
   $\quad \text{return } p$ 
```

Codice 5.4 Inserimento in una *lista ordinata*

Funzione *inserisci* (*ListaOrdinata* ℓ , *elemento* d) \rightarrow *ListaOrdinata*

```

/* Inserisce nella lista ordinata un nuovo nodo e restituisce il
   riferimento alla lista così modificata. */
```

```

 $k \leftarrow d.\text{chiave}$ 
 $p \leftarrow \ell$           // riferimento all'elemento in esame (a partire dal primo)
 $prec \leftarrow \text{null}$  // riferimento all'elemento precedente quello in esame
while  $p \neq \text{null}$  and  $p.\text{chiave} < k$  do
   $\quad prec \leftarrow p$ 
   $\quad p \leftarrow p.\text{pros}$ 
 $r \leftarrow$  riferimento a nuovo nodo
 $r.\text{chiave} \leftarrow k$ 
 $r.\text{altri campi} \leftarrow d.\text{altri campi}$ 
 $r.\text{pros} \leftarrow p$ 
if  $prec = \text{null}$  then
  |  $\ell \leftarrow r$ 
else
   $\quad prec.\text{pros} \leftarrow r$ 
return  $\ell$ 
```

Codice 5.5 Cancellazione da una *lista ordinata*

Funzione *cancella* (*ListaOrdinata* ℓ , *tipoChiave* k) \rightarrow *ListaOrdinata*

```

/* Cancella dalla lista ordinata il primo nodo di chiave  $k$  e restituisce il
   riferimento alla lista così modificata. */
```

```

 $p \leftarrow \ell$ 
 $prec \leftarrow \text{null}$ 
while  $p \neq \text{null}$  and  $p.\text{chiave} < k$  do
   $\quad prec \leftarrow p$ 
   $\quad p \leftarrow p.\text{pros}$ 
if  $p \neq \text{null}$  and  $p.\text{chiave} = k$  then
  | if  $prec = \text{null}$  then
  | |  $\ell \leftarrow \ell.\text{pros}$ 
  | else
  | |  $prec.\text{pros} \leftarrow p.\text{pros}$ 
return  $\ell$ 
```

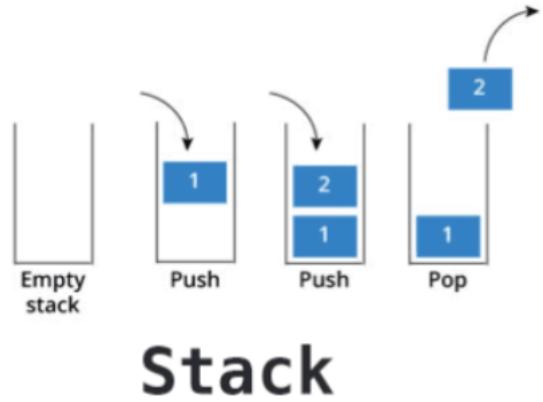
12 Stack (Pila)

Le pile sono delle strutture dati con organizzazione **LIFO** (Last-In-First-Out).

Possono essere implementate tramite array o tramite liste lineari. Sono preferibili le liste concatenate singolarmente.

Le operazioni che possono essere eseguite su una pila sono:

- `isEmpty()` → boolean restituisce `true` se la pila è vuota, `false` altrimenti
- `push(elemento)` aggiunge un elemento alla pila
- `pop()` → `elemento` rimuove il primo elemento dalla pila e lo restituisce
- `top()` → `elemento` restituisce il primo elemento della pila



Algorithm 2: isEmpty

```
Funzione isEmpty() → boolean
  if top = null then
    return true
  else
    return false
```

Algorithm 3: push

```
Procedura push(elemento x)
  r ← riferimento ad un nuovo nodo
  r.dato ← x
  r.pros ← top
  top ← r
```

Algorithm 4: top

```
Funzione top() → elemento
  return top.dato
```

Algorithm 5: pop

```
Funzione pop() → elemento
  x ← top.dato
  top ← top.pros
  return x
```

13 Queue (Coda)

Le code sono delle strutture dati con organizzazione **FIFO** (First-In-First-Out). Possono essere implementate tramite array o tramite liste concatenate. Sono preferibili le liste doppiamente concatenate.

Le operazioni che possono essere eseguite su una coda sono:

- **isEmpty()** → boolean restituisce true se la coda è vuota, false altrimenti
- **enqueue(elemento)** aggiunge un elemento alla coda
- **dequeue()** → elemento rimuove il primo elemento dalla coda e lo restituisce
- **first()** → elemento restituisce il primo elemento della coda

**Algorithm 6:** isEmpty

```
Funzione isEmpty() → boolean
  if primo = null then
    | return true
  else
    | return false
```

Algorithm 7: first

```
Funzione first() → elemento
  return primo.dato
```

Algorithm 8: dequeue

```
Funzione dequeue() → elemento
  x ← primo.dato
  primo ← primo.pros
  if primo = null then
    └ ultimo ← null
  return x
```

Algorithm 9: enqueue

```
Procedura enqueue(elemento x)
  r ← riferimento ad un nuovo nodo
  r.dato ← x
  r.pros ← null
  if primo = null then
    └ primo ← r
    └ ultimo ← r
  else
    └ ultimo.pros ← r
    └ ultimo ← r
```

14 Alberi

La definizione formale di albero sarà data quando tratteremo i grafi. Per ora diciamo che gli alberi sono strutture formate da nodi, simili alle liste, ma con una rappresentazione gerarchica dei dati. La *radice* è il nodo che sta in cima alla gerarchia.

Ogni nodo ha un solo nodo *padre* ma può avere un qualsiasi numero di *figli*. La radice non ha un nodo padre. I nodi che si trovano al livello più basso della gerarchia (i nodi che non hanno figli) sono detti *foglie*. I collegamenti tra nodi sono detti *archi*.

Un albero in cui ogni nodo può avere al massimo due figli è detto *albero binario*. Possiamo dare una definizione ricorsiva di albero:

Un **albero binario** è:

- una struttura vuota
- oppure
- un nodo (radice) con associati due alberi binari detti *sottoalbero sinistro* e *sottoalbero destro*.

La radice di un albero ha *profondità* pari a 0, i nodi di profondità k hanno profondità $k + 1$.

Si definisce *altezza* di un albero la massima profondità dei nodi.

Il *grado* di un nodo è il massimo di figli che può avere quel nodo.

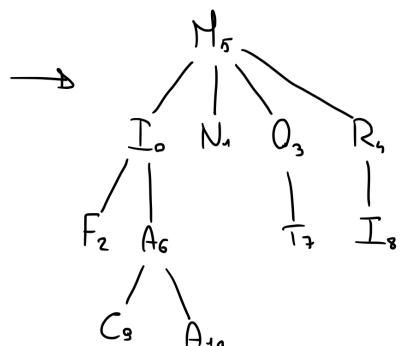
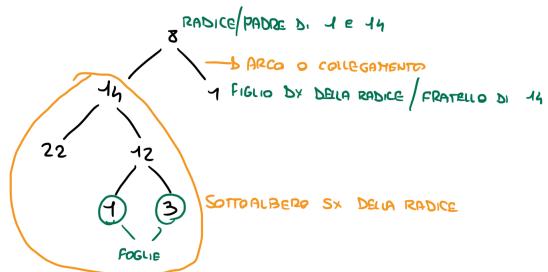
Alcuni esempi di dati rappresentati tramite alberi possono essere l'indice di un libro, uno schema del regno animale ma anche operazioni aritmetiche e in informatica le chiamate ricorsive.

14.1 Rappresentazione di alberi

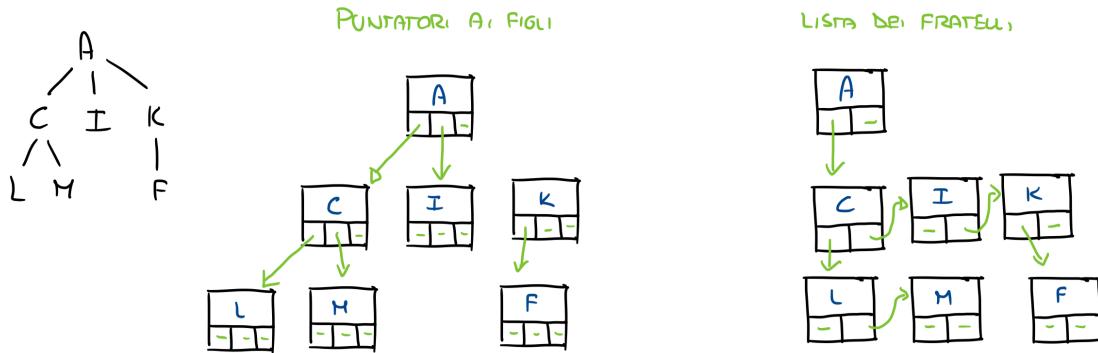
14.1.1 Vettore dei padri

	0	1	2	3	4	5	6	7	8	9
info	I	N	F	O	R	M	A	T	I	C
padre	5	5	0	5	5	-1	0	3	4	6

Figure 1: Esempio di albero binario



14.1.2 Rappresentazioni collegate: puntatori ai figli e lista dei fratelli



14.2 Visite di alberi

Vediamo ora alcune strategie per attraversare tutti i nodi di un albero.

Algorithm 10: Visita generica

```
Algoritmo visitaGenerica(AlberoBinario r)
  S  $\leftarrow \{r\}
  while S  $\neq \emptyset$  do
    | preleva un nodo v da S
    | visita v
    | S  $\leftarrow S \cup \{ \text{figli di } v \}$$ 
```

Algorithm 11: Visita in ampiezza

```
Algoritmo visitaAmpiezza(AlberoBinario r)
  C  $\leftarrow$  coda vuota
  C.enqueue(r)
  while not C.isEmpty() do
    | n  $\leftarrow C.dequeue
    | if n  $\neq null$  then
    |   | visita il nodo associato a n
    |   | C.enqueue(n.sx)
    |   | C.enqueue(n.dx)$ 
```

Algorithm 12: Visita in profondità**Algoritmo** *visitaProfondità(AlberoBinario r)*

```

 $P \leftarrow$  pila vuota
 $P.push(r)$ 
while not  $P.isEmpty()$  do
     $n \leftarrow P.pop$ 
    if  $n \neq null$  then
        visita il nodo associato a  $n$ 
         $P.push(n.sx)$ 
         $P.push(n.dx)$ 
    
```

Algorithm 13: Visita in ordine anticipato**Algoritmo** *visitaPreOrder(AlberoBinario r)*

```

if  $r \neq null$  then
    visita la radice
    visitaPreOrder(r.sx)
    visitaPreOrder(r.dx)
    
```

Algorithm 14: Visita in ordine simmetrico**Algoritmo** *visitaInOrder(AlberoBinario r)*

```

if  $r \neq null$  then
    visitaPreOrder(r.sx)
    visita la radice
    visitaPreOrder(r.dx)
    
```

Algorithm 15: Visita in ordine posticipato**Algoritmo** *visitaPostOrder(AlberoBinario r)*

```

if  $r \neq null$  then
    visitaPostOrder(r.sx)
    visitaPostOrder(r.dx)
    visita la radice
    
```

Algorithm 16: Numero nodi di un albero**Funzione** *numeroNodi(AlberoBinario r) → intero*

```

if  $r = null$  then
    return 0
else
     $nsx \leftarrow numeroNodi(r.sx)$ 
     $ndx \leftarrow numeroNodi(r.dx)$ 
    return 1 +  $nsx + ndx$ 
    
```

15 Alberi binari di ricerca

Gli alberi binari di ricerca sono alberi in cui per ogni nodo n :

1. Il valore di ogni chiave contenuta nel sottoalbero sinistro di n è minore o uguale alla chiave di n
2. Il valore di ogni chiave contenuta nel sottoalbero destro di n è maggiore della chiave di n

Una visita in ordine simmetrico di un A.B.R. produce un elenco ordinato per chiave.

Se devo trovare il nodo con chiave massima scendo tutto a destra, per quello di chiave minima tutto a sinistra.

Il costo di inserimento, ricerca e cancellazione è $O(\text{altezza})$. Il massimo numero di nodi di un albero di altezza h è $2^{h+1} - 1$, quindi:

- $h + 1 \leq n \leq 2^{h+1} - 1$
- $\log_2(n + 1) - 1 \leq h \leq n - 1$

Vogliamo fare in modo che l'albero rimanga più bilanciato possibile in modo da evitare il caso peggiore.

15.1 Operazioni

Vediamo ora l'implementazione di alcune operazioni eseguibili sugli alberi binari di ricerca.

Algorithm 17: Ricerca del massimo

```
Funzione massimo (AlberoRicerca r) → Nodo
  if r = null then
    | return null
  else
    | n ← r
    | while n.dx ≠ null do
      | | n ← n.dx
    | return n
```

Algorithm 18: Ricerca del minimo

```
Funzione minimo (AlberoRicerca r) → Nodo
  if r = null then
    | return null
  else
    | n ← r
    | while n.sx ≠ null do
      | | n ← n.sx
    | return n
```

Codice 6.1 Ricerca (versione ricorsiva)

```
Funzione trova (AlberoRicerca r, tipoChiave k) → Nodo
  if r = null then return null
  else if k < r.chiave then return trova(r.sx, k)
  else if k > r.chiave then return trova(r.dx, k)
  else return r
```

Codice 6.2 Ricerca (versione iterativa)

```
Funzione trova (AlberoRicerca r, tipoChiave k) → Nodo
  while r ≠ null and r.chiave ≠ k do
    if k < r.chiave then r ← r.sx
    else r ← r.dx
  return r
```

Codice 6.3 Inserimento (versione ricorsiva)

```
Funzione inserisci (AlberoRicerca r, elemento d) → AlberoRicerca
  /* Inserisce nell'albero di binario di ricerca riferito dal parametro r un
   nuovo nodo contenente il dato d, nella posizione opportuna.
   Restituisce il riferimento alla radice dell'albero così modificato. */
  k ← d.chiave
  if r = null then
    r ← riferimento a nuovo nodo
    r.chiave ← k
    r.altri campi ← d.altri campi
    r.sx ← null
    r.dx ← null
  else if k < r.chiave then r.sx ← inserisci(r.sx, d)
  else r.dx ← inserisci(r.dx, d)
  return r
```

Codice 6.4 Inserimento (versione iterativa)

Funzione inserisci (*AlberoRicerca r, elemento d*) → *AlberoRicerca*

```

/* Inserisce nell'albero di binario di ricerca riferito dal parametro r un
   nuovo nodo contenente il dato d, nella posizione opportuna.
   Restituisce il riferimento alla radice dell'albero così modificato. */

```

```

k ← d.chiave
// Preparazione del nodo da inserire
t ← riferimento a nuovo nodo
t.chiave ← k
t.altri campi ← d.altri campi
t.sx ← null
t.dx ← null

// Ricerca la posizione dove inserire
padre ← null
n ← r
while n ≠ null do
    padre ← n
    if k < n.chiave then n ← n.sx
    else n ← n.dx
// in questo punto padre si riferisce al nodo sotto il quale inserire
// (contiene null se l'inserimento va fatto alla radice - albero vuoto)

// Inserisci
if padre = null then r ← t
else if k < padre.chiave then padre.sx ← t
else padre.dx ← t
return r

```

Codice 6.5 Cancellazione

```

Funzione cancella (AlberoRicerca r, tipoChiave k) →AlberoRicerca
/* Elimina dall'albero di ricerca un nodo di chiave k, se presente.
   Restituisce il riferimento alla radice dell'albero così modificato. */

// Ricerca il nodo da cancellare e il suo nodo padre
padre ← null
n ← r
while n ≠ null and n.chiave ≠ k do
    padre ← n
    if k < n.chiave then n ← n.sx
    else n ← n.dx

// Cancella il nodo se è stato trovato
if n ≠ null then
    if n.sx = null then                                // manca figlio sinistro: sostituisce il
        if padre ≠ null then                          // nodo con il suo sottoalbero destro
            if n.chiave < padre.chiave then padre.sx ← n.dx
            else padre.dx ← n.dx
        else r ← r.dx                                // caso particolare: cancellazione radice
    else if n.dx = null then                            // manca figlio destro: sostituisce il
        if padre ≠ null then                          // nodo con il suo sottoalbero sinistro
            if n.chiave < padre.chiave then padre.sx ← n.sx
            else padre.dx ← n.sx
        else r ← r.sx                                // caso particolare: cancellazione radice
    else                                                 // ci sono entrambi i figli
        // ricerca il nodo contenente la chiave più grande
        // del sottoalbero sinistro
        t ← n
        m ← n.sx
        while m.dx ≠ null do
            t ← m
            m ← m.dx

        // in questo punto m si riferisce al nodo contenente la chiave più
        // grande del sottoalbero sinistro, t si riferisce al padre di m
        n.chiave ← m.chiave      // copia la chiave presente in m nel nodo n
        n.altri campi ← m.altri campi // e copia anche tutti gli altri dati

        // elimina il nodo riferito da m
        if t = n then n.sx ← m.sx // se il nodo m è figlio sinistro del nodo n
        else t.dx ← m.sx

return r

```

16 Altri tipi di alberi

16.1 Alberi perfettamente bilanciati

Un albero è *perfettamente bilanciato* quando per ogni nodo la differenza tra il numero di nodi del sottoalbero sinistro e il numero di nodi del sottoalbero destro è al massimo 1.

16.2 Alberi bilanciati in altezza o AVL

Un albero è *bilanciato in altezza* o *AVL* quando per ogni nodo la differenza in valore assoluto tra l'altezza del sottoalbero destro e l'altezza del sottoalbero sinistro è al massimo 1.

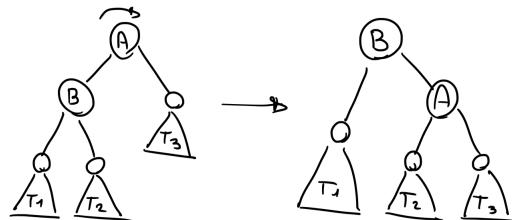
N.B bilanciato in altezza \Rightarrow bilanciato, ma non viceversa.

Numero massimo di nodi: $2^{h+1} - 1$

Numero minimo di nodi:

$$\begin{cases} 1 & \text{se } h = 0 \\ 2 & \text{se } h = 1 \\ 1 + n_{h-1} + n_{h-2} & \text{se } h > 1 \end{cases}$$

Un albero AVL con il minimo numero di nodi è detto *albero di Fibonacci*. Nel caso l'albero risulti sbilanciato devo eseguire delle operazioni per sistemarlo.



Costo operazioni

Dato un albero di ricerca di n nodi

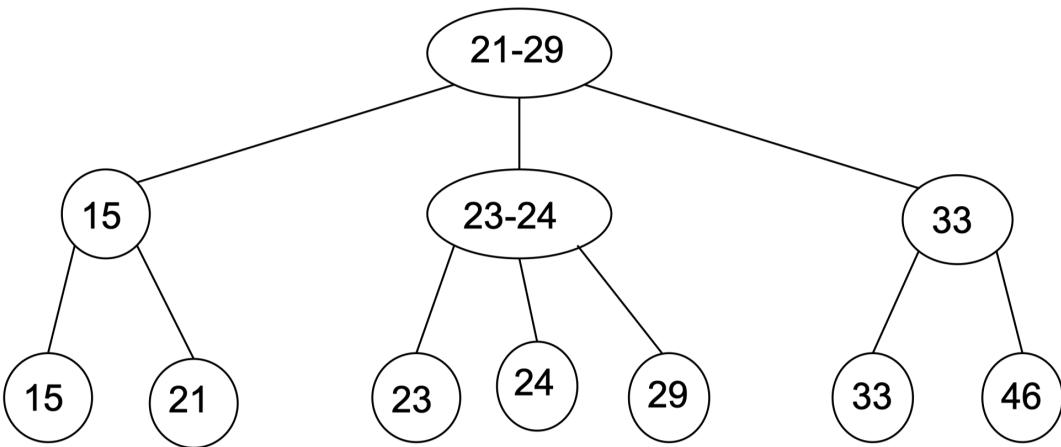
- Ricerca $O(\log_2 n)$
- Inserimento $O(\log_2 n)$
- Cancellazione $O(\log_2 n)$

Questa per ora è la struttura con prestazioni migliori per i dizionari, almeno finché non vedremo le tabelle hash più avanti.

16.3 Alberi 2-3

Gli *alberi 2-3* sono alberi in cui ogni nodo interno ha 2 o 3 figli e le foglie sono tutte allo stesso livello. I dati sono memorizzati solo nelle foglie e i nodi interni contengono solo informazioni di instradamento.

- Se la chiave di un nodo interno contiene solo un valore, significa che il nodo ha 2 figli, e quel valore è il maggiore del sottoalbero sinistro
- Se la chiave di un nodo interno contiene 2 valori significa che il nodo ha 3 figli, e i due valori corrispondono rispettivamente al massimo valore contenuto nel sottoalbero sinistro e al massimo valore contenuto nel sottoalbero centrale



	min	max
Numero nodi	$2^{h+1} - 1$	$\frac{3^{h+1} - 1}{2}$
Numero foglie	2^h	3^h

16.3.1 Operazioni

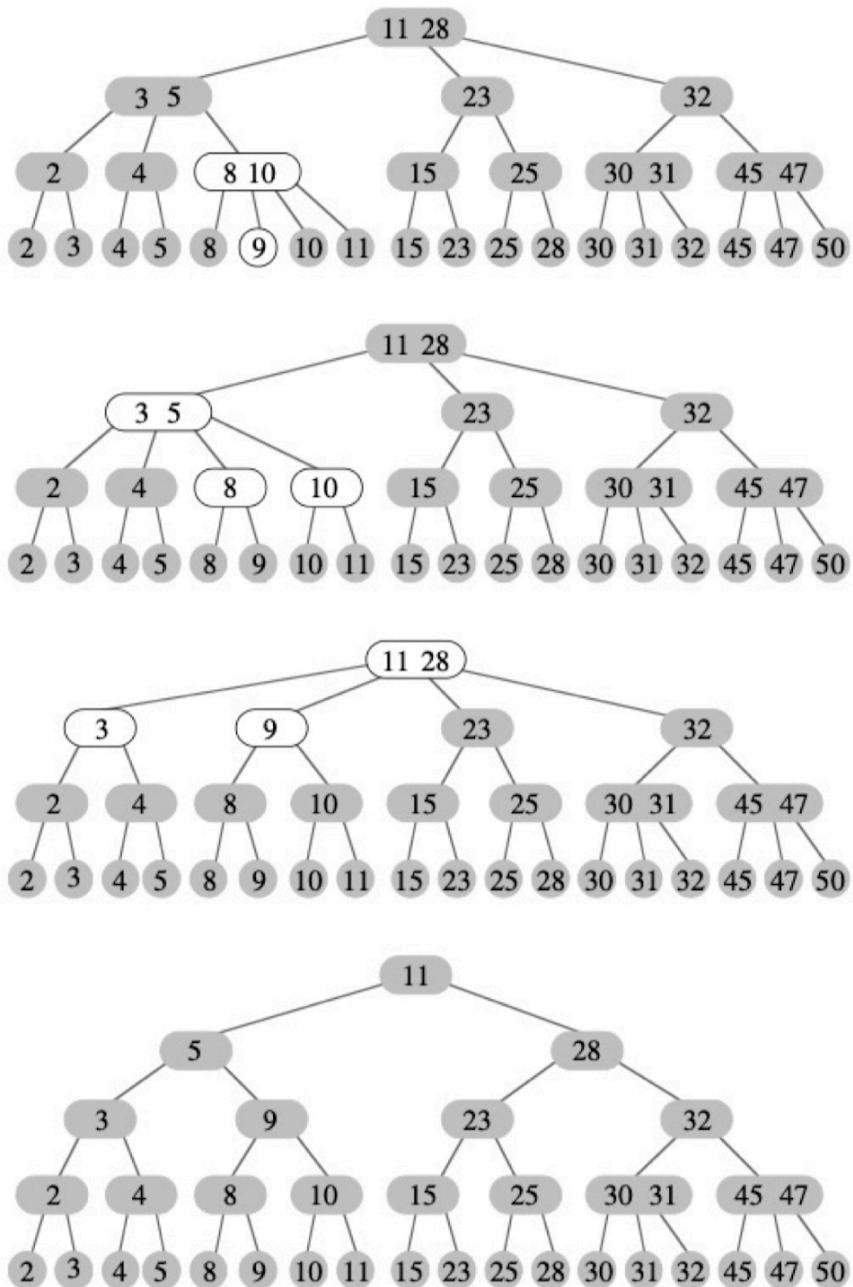
Algorithm 19: Ricerca di un dato

Funzione $trova(albero2-3 r, tipoChiave k) \rightarrow foglia$

```

 $n \leftarrow r$ 
while  $n$  si riferisce ad un nodo interno do
  if  $k \leq n.s$  then
     $| n \leftarrow n.sx$ 
  else
     $| n \leftarrow n.dx$ 
  if  $n.chiave = k$  then
     $| \text{return } n$ 
  else
     $| \text{return } null$ 
  
```

Per inserimenti e cancellazioni è utile tenere in ogni nodo un puntatore al nodo padre. Quando un nodo ha già 3 figli e devo inserirne un altro, faccio uno *split*.



16.3.2 Costo operazioni

- **Ricerca:** $O(\log n)$
- **Inserimento:** $O(\log n)$
- **Cancellazione:** $O(\log n)$

Come gli alberi AVL.

16.4 B-Alberi

Sono un modello nato per rappresentare gli indici delle basi di dati, quando i dati sono troppo grandi per stare in memoria centrale. L'obiettivo non è più quello di fare l'albero più basso possibile, ma quello di fare il minor numero possibile di accessi al disco. A differenza degli alberi 2-3 le informazioni non sono solo nelle foglie ma anche nei nodi interni. Diamo una definizione formale di *B-albero* di ordine t (dove t) è il grado minimo:

- Ogni nodo interno ha al massimo $2t$ figli
- Ogni nodo interno diverso dalla radice ha almeno t figli
- La radice ha almeno 2 figli
- Tutte le foglie hanno la stessa profondità
- Ogni foglia contiene k chiavi ordinate dove $t - 1 \leq k \leq 2t - 1$
- Ogni nodo interno con $k + 1$ figli e sottoalberi $T_0 \dots T_k$ contiene k chiavi ordinate tali che per ogni chiave c_i nell'albero T_i (con $i = 0 \dots k$) si ha:

$$c_0 \leq a_1 \leq c_1 \leq a_2 \leq \dots \leq a_{k-1} \leq c_{k-1} \leq a_k \leq c_k$$

Numero minimo di chiavi in un albero di altezza h : $2t^h - 1$

Altezza massima n chiavi: $2t^h - 1$

Passi totali ricerca: $\Theta(h \cdot \log t)$

Costo operazioni

	Passi di calcolo(tempo)	Accessi a memoria di massa
Ricerca	$\Theta(\log n)$	$\log_t n$
Inserimento	$\Theta(t \cdot \log n)$	$c \cdot \log_t n$
Cancellazione	$\Theta(t \cdot \log n)$	$c \cdot \log_t n$

- n = numero di chiavi
- c = costante piccola (dipende dall'implementazione, di solito è circa 4)

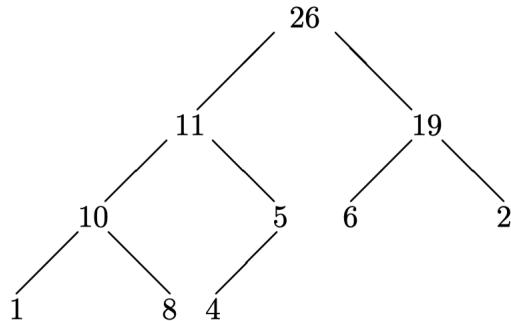
17 Heapsort

HeapSort è un algoritmo di ordinamento che utilizza la struttura dati *heap*. Vedremo innanzitutto in cosa consiste uno heap, per poi trattare l'algoritmo e la sua complessità in termini di numero di confronti. Vedremo poi come uno heap possa essere rappresentato nell'array stesso da ordinare, in modo tale da avere un implementazione in loco.

17.1 La struttura dati *Heap*

Uno *heap* è un *albero binario quasi completo*, ovvero completo almeno fino al penultimo livello, tale che la chiave contenuta in ogni suo nodo è maggiore o uguale alla chiave contenuta nei figli. Poichè un albero binario di altezza h contiene $2^{h+1} - 1$ nodi, possiamo affermare che in uno heap di altezza h il numero n di nodi soddisfa $2^h \leq n \leq 2^{h+1}$, da cui otteniamo $h \leq \log_2 n \leq h + 1$ e dunque $h = \lfloor \log_2 n \rfloor$.

La radice di uno heap contiene sempre la chiave maggiore. Pertanto, disponendo di uno heap contenente le chiavi che dobbiamo ordinare, possiamo prelevare l'elemento che si trova nella radice e collocarlo, come unico elemento, nella sequenza ordinata che dobbiamo produrre come risultato, che costruiremo a partire dal fondo. Una volta fatto ciò possiamo modificare la struttura in modo da riottenere uno heap ed applicare lo stesso procedimento.



17.2 Sistemare uno heap

Per risistemare uno heap applichiamo la seguente strategia.

Sostituiamo la chiave contenuta nella radice con quella contenuta nell'ultima delle foglie, cioè quella che si trova più a destra nell'ultimo livello, rimuovendo tale foglia. Tutti i nodi rispettano la condizione di heap, tranne la radice che potrebbe contenere una chiave inferiore rispetto a uno o entrambi i figli. In questo caso facciamo "scendere" il dato presente nella radice, scambiandolo con quello di chiave maggiore tra i figli. Se la condizione di heap non è rispettata dal figlio in cui abbiamo spostato il dato, iteriamo lo stesso procedimento su di esso.

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

```

Numero di confronti

Il numero di confronti usato da **risistema**, nel caso peggiore, $\Theta(h)$, dove h è l'altezza dello heap. Infatti il valore presente nella radice viene fatto scendere lungo un cammino fino a raggiungere la posizione corretta che, nel caso peggiore, potrebbe essere una foglia a distanza massima dalla radice. In questo processo, ad ogni passo viene ispezionato un nodo lungo il cammino, determinando la chiave massima tra i figli e confrontandola con la chiave ispezionata. Pertanto per ogni nodo del cammino ho 2 confronti.

17.3 Creazione di uno heap

Supponiamo di disporre di un albero binario quasi completo le cui chiavi non rispettino però la condizione di heap. Studieremo due soluzioni per trasformarlo in uno heap. La seconda soluzione è meno dispendiosa in termini di memoria.

Soluzione ricorsiva

Strategia *divide-et-impera*:

- Se l'albero è vuoto non devo fare nulla
- Se l'albero non è vuoto trasformiamo ricorsivamente ciascuno dei due sottoalberi sinistro e destro in heap; a questo punto tutti i nodi, eccetto la radice, soddisfano la condizione di heap. Applicando la procedura **risistema** possiamo trasformare l'albero in uno heap

Codice 7.2 Creazione dello heap (versione ricorsiva)

Procedura *creaHeap (albero binario T)*

```
/* Trasforma l'albero binario T in uno heap */  
if T ≠ albero vuoto then  
    creaHeap(T.sx)  
    creaHeap(T.dx)  
    risistema(T)
```

Soluzione iterativa

Anzichè costruire lo heap in maniera top-down, possiamo procedere in maniera bottom-up partendo dalle foglie dell'albero. Ispezioniamo cioè l'albero a partire dall'ultima foglia, trasformando ogni sottoalbero in uno heap. Quindi:

- Iniziamo a considerare ciascun nodo di profondità h , da destra verso sinistra, e trasformiamo in heap il sottoalbero che ha tale nodo come radice (questi nodi sono foglie, quindi i relativi sottoalberi sono già heap e per essi non occorre fare nulla).
- Passiamo a considerare ciascun nodo di profondità $h - 1$ (sempre da destra verso sinistra) e trasformiamo in heap il sottoalbero che ha radice in esso.
- Ripetiamo lo stesso procedimento considerando man mano profondità inferiori sino ad arrivare alla radice. A questo punto l'intero albero è uno heap.

Poichè i sottoalberi sono trasformati in heap a partire dal basso, quando in questo procedimento dobbiamo trasformare in heap il sottoalbero T_x che ha come radice un nodo x di profondità p , i sottoalberi di x , avendo profondità $p - 1$, sono già stati trasformati in heap in passi precedenti. Dunque l'unico nodo di T_x che potrebbe non rispettare la condizione di heap è radice x . Quindi è sufficiente applicare **risistema** per trasformare T_x in uno heap.

Codice 7.3 Creazione dello heap (versione iterativa)

```

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

Numero di confronti

creaHeap chiama *risistema* un certo numero di volte, per sottoalberi di altezze differenti. Il numero di confronti per trasformare in heap tutti i sottoalberi di profondità *p* è $\Theta(h - p)2^p$. Nel ciclo esterno *p* varia su tutte le profondità, cioè da 0 ad *h*. Sommando su di esse otteniamo che il numero di confronti è $2^{h+1} - 2 - h$.

Essendo l'albero completo la sua altezza è logaritmica rispetto al numero di nodi. Questo permette di concludere che il numero di confronti di *creaHeap* è $\Theta(n)$, cioè lineare rispetto al numero di chiavi.

17.4 Schema di heapSort**Codice 7.4** Schema di Heapsort

```

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

Numero di confronti

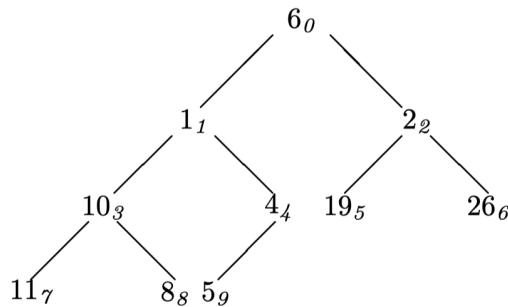
creaHeap effettua $\Theta(n)$ confronti. Segue poi la parte iterativa in cui, ad ogni passo, si preleva la radice e si risistema lo heap. Queste operazioni vengono ripetute fino a svuotare lo heap, quindi *n* volte. Risistemare lo heap utilizza, nel caso peggiore, un numero di confronti proporzionale alla sua altezza, che è logaritmica. Dunque il numero di confronti, nel caso peggiore, è $\Theta(n \log n)$.

17.5 Ordinamento in loco di array tramite `heapSort`

Si può implementare l'algoritmo in modo semplice senza ricorrere a strutture aggiuntive, servendosi di una corrispondenza tra alberi binari quasi completi e array. Supponiamo di disporre del seguente array:

<i>indice</i>	0	1	2	3	4	5	6	7	8	9
contenuto	6	1	2	10	4	19	26	11	8	5

Immaginiamo di collocare gli elementi dell'array nell'ordine in cui compaiono in un albero binario, riempiendo ciascun livello da sinistra verso destra a partire dalla radice, come in una visita in ampiezza. L'albero che otteniamo è il seguente:



L'albero è quasi completo, con le foglie dell'ultimo livello più a sinistra possibile.

Osserviamo che i figli del nodo che nell'array ha indice i hanno, se esistono, indice $2i + 1$ e $2i + 2$. L'array che rappresenta un albero binario quasi completo è detto *vettore posizionale*.

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  $A[\ell]$ 
    [ risistema(A, 0,  $\ell$ )
  ]
  ]

```

Si possono modificare anche `creaHeap` e `risistema` affinchè lavorino direttamente con l'array.

Codice 7.6 Creazione di uno heap rappresentato implicitamente in un array (confrontare con il Codice 7.3)

```
Procedura creaHeap (array A[0..n - 1])
  for i  $\leftarrow \lfloor n/2 \rfloor$  downto 0 do
    | risistema(A, i, n)
```

Codice 7.7 Risistema (*fixHeap*) su array posizionale (confrontare con il Codice 7.1)

Per semplicità qui stiamo schematizzando l'ordinamento di interi e, a differenza della versione ad alto livello presentata nel Codice 7.1, non indichiamo esplicitamente gli altri campi. Pertanto qui il singolo elemento dell'array è direttamente la chiave.

```
Procedura risistema (array A[0..n - 1], intero r, intero l)
  /* A è un array i cui primi l elementi formano uno heap. Risistema la
   parte di heap la cui radice si trova alla posizione di indice r di A.
   Quando viene richiamata all'interno di heapSort (Codice 7.5), il
   parametro l (numero di elementi nello heap, cioè ancora da ordinare)
   coincide con l'indice di A in cui inizia la parte già ordinata. */
  v  $\leftarrow r$ 
  x  $\leftarrow A[v]$  // dato da "far scendere" nella posizione appropriata
  daCollocare  $\leftarrow$  true
  do
    if  $2 * v + 1 \geq l$  then // v è l'indice di una foglia
      | daCollocare  $\leftarrow$  false // la posizione appropriata per x è stata trovata
    else
      | u  $\leftarrow 2 * v + 1$  // indice figlio sinistro
      | if  $u + 1 < l$  and A[u + 1] > A[u] then // c'è figlio destro ed è maggiore
        | | u  $\leftarrow u + 1$  // del sinistro
      | // ora u contiene l'indice del figlio di v di valore massimo
      | if A[u] > x then
        | | A[v]  $\leftarrow A[u]$  // il dato in posizione u risale
        | | v  $\leftarrow u$  // si prosegue sul figlio selezionato
      | else
        | | daCollocare  $\leftarrow$  false // posizione appropriata per x è stata trovata
    while daCollocare
  A[v]  $\leftarrow x$  // copia x nella posizione trovata
```

17.6 Spazio

Utilizzando la versione iterativa di `creaHeap` e l'implementazione in loco, l'algoritmo utilizza spazio costante oltre all'array da ordinare.

17.7 Costo operazioni su heap

- Trovare elemento di chiave massima $\rightarrow O(1)$ passi
- Cancellare elemento di chiave massima $\rightarrow \Theta(\log 1)$ passi
- Inserire un nuovo elemento $\rightarrow \Theta(\log n)$ passi
- Cancellare elemento di chiave $x \rightarrow \Theta(\log n)$ passi
- Modificare la chiave di un elemento $\rightarrow \Theta(\log n)$ passi

17.8 Riassumendo

`HeapSort` è un algoritmo di ordinamento in loco che, per ordinare n elementi effettua $\Theta(n \log n)$ confronti. Pertanto, se ciascun confronto viene effettuato in tempo $O(1)$, il tempo complessivo è $\Theta(n \log n)$.

Si può verificare che questo metodo non è stabile.

18 Riassunto ordinamento

Il problema dell'ordinamento può essere definito in questo modo:

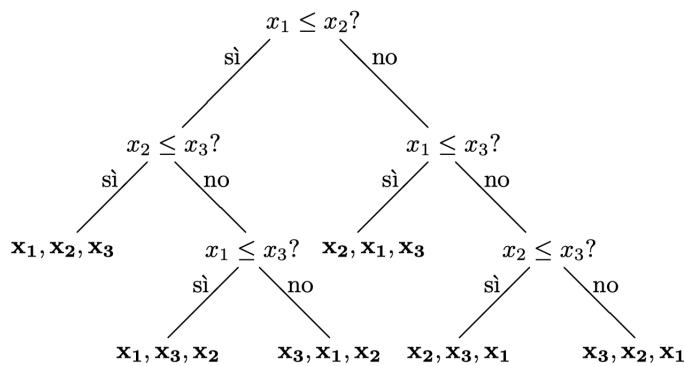
Input: n elementi x_1, x_2, \dots, x_n appartenenti a un dominio D su cui è definita una relazione \leq di ordine totale.

Output: Sequenza $x_{j1}, x_{j2}, \dots, x_{jn}$ dove $(j_1 \dots j_n)$ è una permutazione di $(1, 2, \dots, n)$ tale che $x_{j1} \leq \dots \leq x_{jn}$.

Algoritmo	Numero confronti	Spazio	Note	Stabile
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

18.1 Numero minimo di confronti

Dimostreremo ora che qualsiasi algoritmo di ordinamento basato su confronti richiede, nel caso peggiore, un numero di confronti almeno dell'ordine di $n \log n$. Le possibili computazioni di un algoritmo di ordinamento su sequenze di n elementi possono essere rappresentate mediante un *albero di decisione*, cioè un albero binario in cui ciascun nodo interno rappresenta un operazione di confronto, con associati due sottoalberi, che dipendono dall'esito di tale operazione, mentre ogni foglia rappresenta una risposta dell'algoritmo, cioè un possibile ordine tra le chiavi.



Indipendentemente dalla strategia utilizzata per eseguire i confronti, l'albero dovrà avere un numero di foglie pari almeno al numero dei possibili ordini tra le chiavi, cioè al numero di possibili permutazioni di n elementi, che è $n!$. Il numero massimo di confronti utilizzato da una strategia è pari alla profondità dell'albero. Si può verificare che la profondità di un albero binario con k foglie è almeno logaritmica in k .

Per trovare il numero di confronti necessari nel caso peggiore stimiamo quindi la profondità minima che deve avere un albero con $n!$ foglie, calcolando il logaritmo di $n!$. Utilizzando l'approssimazione di Stirling $n! \approx \sqrt{2\pi n(\frac{n}{e})^n}$ si ottiene $\Theta(n \log n)$.

Possiamo concludere che *ogni* algoritmo di ordinamento basato su confronti richiede nel caso peggiore un numero di confronti tra chiavi dell'ordine di $n \log n$ per ordinare n elementi.

19 Code con priorità

Utilizzando gli heap e le operazioni su di essi descritte in precedenza, si possono implementare delle strutture a coda in cui gli elementi vengono prelevati con un criterio di priorità. Solitamente la priorità è indicata da una chiave numerica con la convenzione che *Chiavi inferiori indicano priorità più alta*. Pertanto prelevare il primo elemento, cioè quello con priorità più alta, equivale a prelevare quello con chiave minima (numero più basso).

Consideriamo le seguenti operazioni:

- `findMin()`

Restituisce l'elemento minimo della coda (senza rimuoverlo)

- `deleteMin()`

Rimuove l'elemento minimo della coda e lo restituisce.

- `insert(elem e, chiave k)`

Inserisce nella coda un elemento e con associata una chiave (priorità) k .

- `delete(elem e)`

Cancella l'elemento e dalla coda.

- `changeKey(elem e, chiave d)`

Modifica la priorità dell'elemento e , assegnando come nuovo valore d .

Le code con priorità possono essere implementate utilizzando dei *Min-heap*. Come nell'implementazione di `heapSort`, lo heap può essere rappresentato mediante un array (o meglio la prima parte di un array, lasciando spazio nella seconda per eventuali inserimenti). Se la coda contiene n elementi e assumendo il criterio di costo uniforme, l'operazione di prelevare il primo elemento può essere effettuata in tempo costante, mentre le altre operazioni `deleteMin` e `insert` in tempo $O(\log n)$. Anche le operazioni `delete` e `changeKey` possono essere effettuate in $O(\log n)$, ma solo se è nota nello heap la posizione dell'elemento da cancellare o modificare. Per evitare di cercare tale posizione, si può tenere una struttura ausiliaria che fornisca, per ogni elemento, la sua posizione all'interno dello heap. Ogni volta che si manipola lo heap la struttura va aggiornata.

20 Algoritmi di ordinamento non basati su confronti

20.1 IntegerSort

È un algoritmo di ordinamento che si basa sulla conoscenza a priori dell'intervallo in cui sono compresi i valori da ordinare. L'algoritmo conta il numero di occorrenze di ciascun valore presente nell'array da ordinare, memorizzando questa informazione in un array temporaneo di dimensione pari all'intervallo di valori. Il numero di ripetizioni dei valori indica la posizione del valore immediatamente successivo.

- Si calcolano il valore massimo e il valore minimo, $\max(A)$ e $\min(A)$
- Si prepara un array ausiliario C di dimensione pari all'intervallo di valori con entrate $C[i]$ che rappresentano la frequenza dell'elemento $i + \min(A)$
- Si visita l'array A aumentando l'elemento di C corrispondente.
- Si visita l'array C in ordine e si scrivono su A $C[i]$ copie del valore $i + \min(A)$

Complessità

L'algoritmo esegue 3 iterazioni, 2 di lunghezza n per individuare massimo e minimo e per il calcolo delle occorrenze dei valori, e una di lunghezza $k = (\max(A) - \min(A) - 1)$.

La complessità totale è quindi $O(n + k)$.

Conviene utilizzarlo quando il valore di k è $O(n)$.

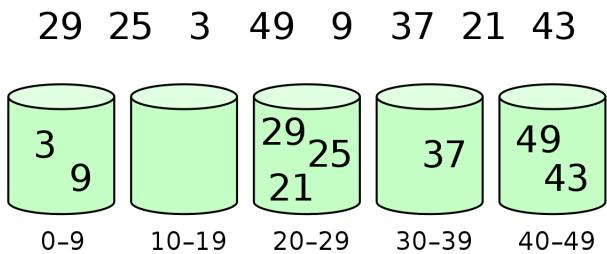
Codice 10.1 Integersort

```
Algoritmo integerSort (array A[0..n - 1], intero b)
  /* Ordina un array di interi appartenenti all'intervallo [0, b - 1] */ 
  Sia  $Y[0..b - 1]$  un array // array di contatori
  for  $i \leftarrow 0$  to  $b - 1$  do  $Y[i] \leftarrow 0$  // azzera i contatori
  // Conta le occorrenze nell'array  $A$  di ciascun intero appartenente
  // a  $[0, b - 1]$ 
  for  $i \leftarrow 0$  to  $n - 1$  do
     $x \leftarrow A[i]$ 
     $Y[x] \leftarrow Y[x] + 1$ 
  // riempie  $A$  con i valori ordinati
   $j \leftarrow 0$  // indice della prossima posizione di  $A$  da riempire
  for  $i \leftarrow 0$  to  $b - 1$  do
    // assegna l'intero  $i$  alle successive  $Y[i]$  posizioni di  $A$ 
    for  $h \leftarrow 1$  to  $Y[i]$  do
       $A[j] \leftarrow i$ 
       $j \leftarrow j + 1$ 
```

20.2 BucketSort

È un algoritmo di ordinamento per valori numerici che si assume siano distribuiti uniformemente in un intervallo $[0, 1)$

Se n è il numero di elementi da ordinare, l'intervallo $[0, 1)$ è diviso in n intervalli di uguale lunghezza, detti *bucket*. Ciascun valore dell'array è quindi inserito nel bucket a cui appartiene, i valori all'interno di ogni bucket vengono ordinati e l'algoritmo di conclude con la concatenazione dei valori contenuti nei bucket.



Complessità

La complessità di `bucketSort` è $O(n)$ per tutti i cicli, a parte l'ordinamento dei singoli bucket. Date le premesse sull'input, utilizzando `insertionSort` l'ordinamento di ogni bucket è $\Theta(1)$, quindi la complessità media è $O(n)$ per tutto l'algoritmo. La complessità complessiva nel caso migliore è $O(n + m)$ dove m è il massimo valore nell'array.

Codice 10.2 Bucketsort

```
Algoritmo bucketSort (array A[0..n - 1], intero b)
  /* Ordina un array di record in base a un campo chiave intero appartenente
   * all'intervallo  $[0, b - 1]$  */  

  Sia Y[0..b - 1] un array // array di code
  for i  $\leftarrow 0$  to b - 1 do Y[i]  $\leftarrow$  coda vuota
    // colloca gli elementi di A in differenti code, in base alle chiavi
    for i  $\leftarrow 0$  to n - 1 do
      x  $\leftarrow$  A[i].chiave
      Y[x].enqueue(A[i])
    // riempie A con i valori ordinati
    j  $\leftarrow 0$  // indice della prossima posizione di A da riempire
    for i  $\leftarrow 0$  to b - 1 do
      // colloca i record con chiave i nelle prossime posizioni di A
      while not Y[i].isEmpty() do
        A[j]  $\leftarrow$  Y[i].dequeue()
        j  $\leftarrow j + 1$ 
```

20.3 RadixSort

È un algoritmo che esegue degli ordinamenti per posizione della cifra, partendo dalla cifra meno significativa. Questo affinchè l'algoritmo non si trovi a dovere operare ricorsivamente su sottoproblemi di dimensione non valutabile a priori.

Complessità

L'algoritmo ha complessità computazionale pari a $O(n \cdot k)$ dove n è il numero di elementi da ordinare e k è la media del numero di cifre degli n elementi. Se k risulta essere minore di n non si ha guadagno rispetto a `integerSort` che opera in tempo lineare. Se $k > n$ l'algoritmo può risultare peggiore anche rispetto agli algoritmi basati su confronti.

1° PASSO	2° PASSO	3° PASSO	4° PASSO		
253	10	5	5		5
346	253	10	10		10
1034	1034	127	1034		127
10	5	1034	127		253
5	346	346	253		346
127	127	253	346		1034

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  $\lfloor k/B^t \rfloor \neq 0$  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. Le posizioni sono contate da 0, posizione della
   cifra meno significativa. */                                     */
  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
    x ← cifra di posizione t nella rappresentazione in base b di A[i].chiave
    Y[x].enqueue(A[i])
  // riempie 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
  
```

21 Rappresentazione di partizioni (UNION-FIND)

Dato un insieme \mathcal{A} , una partizione è una famiglia di sottoinsiemi $\mathcal{A}_{1\dots k}$ tali che

- $\mathcal{A}_i \neq \emptyset$
- $\mathcal{A}_i \cap \mathcal{A}_j = \emptyset$
- $\mathcal{A}_1 \cup \dots \cup \mathcal{A}_k = \mathcal{A}$

Vogliamo rappresentare una collezione di insiemi disgiunti mediante le operazioni:

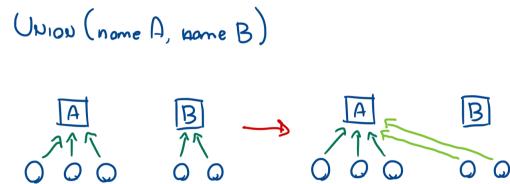
- UNION(A, B) unisce gli insiemi A e B in un unico insieme A
- FIND(X) restituisce il nome dell'insieme che contiene l'elemento x
- MAKESET(X) crea un nuovo insieme $\{x\}$ di nome X (x nuovo elemento)

Ogni insieme è rappresentato da un albero con radice con puntatori verso l'alto, dove i nodi sono gli elementi dell'insieme e la radice è il nome dell'insieme. Una partizione è quindi una foresta di alberi. In base a come impostiamo il nostro sistema di partizioni possiamo velocizzare le UNION oppure le FIND.

21.1 Operazioni QUICKFIND

Considero alberi di altezza 1 dove gli elementi dell'insieme sono le foglie e il nome dell'insieme è dato dalla radice. Quando $n(A) > n(B)$ conviene spostare gli elementi di B sotto ad A e cambiare nome alla radice. Per ottimizzare, durante Makeset memorizzo nella radice il numero di elementi dell'insieme. Quando poi faccio union sommo il numero di elementi. Lo spazio è lineare rispetto a n quindi è $O(n)$.

Effettuando una sequenza di n makeset e $O(n)$ union e find ottengo un costo ammortizzato $O(\log n)$



21.2 Operazioni QUICKUNION

Gli alberi non sono più vincolati ad avere altezza 1 e la radice contiene il nome dell'insieme. Al contrario delle operazioni QUICKFIND queste favoriscono in termini di complessità l'implementazione della funzione UNION.

- **MAKESET (elemento e)**

$$\textcircled{e} \quad T(n) = O(1)$$

- **UNION (name A, name B)**



- **FIND (elemento x)**

RISALGO DA X FINO ALLA RADICE

21.3 Algoritmo QUICKFIND bilanciato

L'utilizzo della rappresentazione QUICKFIND penalizza l'operazione di UNION. È possibile eseguire alcuni miglioramenti al fine di migliorare la complessità di tale operazione.

Gli accorgimenti che si possono introdurre sono:

1. Memorizzare all'interno di ogni albero la cardinalità dell'insieme, ovvero il numero di foglie dell'albero.
2. Nella realizzazione dell'operazione UNION(A, B):
 - (a) Spostare le foglie dell'albero rappresentante l'insieme di cardinalità minore verso l'albero rappresentante l'insieme di cardinalità maggiore;
 - (b) Memorizzare l'etichetta da associare al nuovo insieme all'interno della radice dell'albero rappresentante l'insieme unione.

Il tempo utilizzato dalla UNION di questo algoritmo bilanciato è logaritmico rispetto al numero di MAKESET effettuate, ovvero rispetto al numero di elementi contenuti nella foresta di alberi.

21.4 Algoritmo QUICKUNION bilanciato

In maniera speculare rispetto al QUICKFIND bilanciato, è possibile adottare alcuni accorgimenti per controllare l'altezza dell'albero rappresentante l'insieme e quindi migliorare l'esecuzione di FIND.

Union by rank

È una variante della rappresentazione QUICKUNION che, al fine di evitare che l'altezza dell'albero cresca senza alcun controllo, adotta i seguenti accorgimenti:

1. Memorizza all'interno di ogni radice l'altezza dell'albero
2. Nella realizzazione dell'operazione di UNION(A, B):
 - (a) La radice dell'albero avente altezza maggiore diventa padre della radice dell'albero avente altezza minore
 - (b) memorizza l'etichetta da associare al nuovo insieme all'interno del nodo diventato radice dell'albero unione

Lemma:

Ogni albero QUICKUNION bilanciato in altezza con radice x contiene almeno $2^{rank(x)}$ nodi.

21.5 Compressione di cammino

Sempre nell'ambito della rappresentazione QUICKUNION è possibile introdurre ulteriori accorgimenti volti a migliorare la complessità dell'operazione di FIND. La compressione di cammino si serve dell'algoritmo di FIND facendo leva sul movimento che esso esegue nella ricerca dell'etichetta posta alla radice. L'idea della compressione di cammino è quella di assegnare un ulteriore compito al FIND, ovvero quello di ristrutturare l'albero ponendo il padre di ogni nodo incontrato uguale alla radice dell'albero. Eseguiamo in tal modo una compressione dell'altezza dell'albero lungo tutto il cammino che dal nodo contenente l'elemento da trovare termina nella radice.

21.6 Riepilogo costi operazioni

	MAKESET	UNION	FIND
QUICKFIND	$O(1)$	$O(n)$	$O(1)$
QUICKFIND bilanciato	$O(1)$	$O(\log n)$	$O(1)$
QUICKUNION	$O(1)$	$O(1)$	$O(n)$
QUICKUNION bilanciato	$O(1)$	$O(1)$	$O(\log n)$

22 Grafi

I grafi sono una formalizzazione della connessione e relazione tra oggetti. Un grafo G è una coppia V, E dove V è un insieme finito di *vertici* (*o nodi*) ed E è un sottoinsieme di $V \cdot V$ segmenti detti *archi*, *lati* o *spigoli*.

$$G = (V, E) \quad E \subseteq V \cdot V$$

I grafi possono essere *orientati* o *non orientati*. Nel primo caso gli archi rappresentano una relazione simmetrica, cioè valida tra due nodi in entrambe le direzioni, nel secondo caso solo in una direzione.

Vediamo ora una serie di termini legati ai grafi. Dato un generico arco $(x, y) \in E$ in un grafo con vertici V :

- Un arco è **incidente** su due vertici
- Se un arco *esce* da x ed *entra* in y , allora y è **adiacente** ad x
- I **vicini** di un vertice sono i vertici adiacenti ad esso
- Il **grado** di un vertice è il numero di archi incidenti al vertice
- Un **cammino** da x a y è una sequenza di vertici collegati da archi appartenenti al grafo in cui il vertice di partenza è x e quello di arrivo y
- La **lunghezza del cammino** è il numero di archi del cammino
- y è **raggiungibile** da x se esiste un cammino da x a y
- Un **cammino semplice** non contiene vertici ripetuti
- Un **ciclo** è un cammino da x a x
- In un **ciclo semplice** è ripetuto solo il vertice iniziale, alla fine
- Una **catena** tra x e y è una sequenza in cui non rispetto l'orientamento degli archi
- Un **circuito** è una catena da x a x
- Un grafo è **connesso** quando per ogni coppia di vertici esiste una catena
- Un grafo è **fortemente connesso** quando per ogni coppia di vertici esiste un cammino
- Un **sottografo** è un grafo in cui prendo solo alcuni vertici e alcuni archi
- Un **sottografo indotto** è un grafo in cui prendo solo alcuni vertici e tutti i loro archi incidenti
- Una **componente fortemente connessa** è un sottografo indotto fortemente connesso massimale
- Un **circuito hamiltoniano** è un circuito che passa per ogni vertice del grafo una e una sola volta
- Un **circuito euleriano** è un circuito che attraversa ogni arco del grafo una e una sola volta

- Un **multigrafo** è un grafo in cui 2 vertici sono sollegati da più di un arco

A questo punto possiamo dare la definizione formale di albero:

Un albero è un grafo non orientato, connesso e privo di cicli.

Alcuni teoremi riguardanti i grafi:

1. esiste un circuito euleriano se e solo se ogni vertice ha grado pari
2. è sempre possibile suddividere un grafo in componenti fortemente connesse
3. Se un grafo è un albero allora il numero di vertici è uguale al numero di archi +1
4. Se un grafo è non orientato e connesso, allora, se il numero di vertici è = al numero di archi +1, è un albero
5. Un albero

Albero di supporto o ricoprente (Spanning tree)

Dato un grafo $G = (V, E)$ orientato non connesso, un albero ricoprente di G è un albero $G' = (V', E')$ con $V' = V$ ed $E' \subseteq E$.

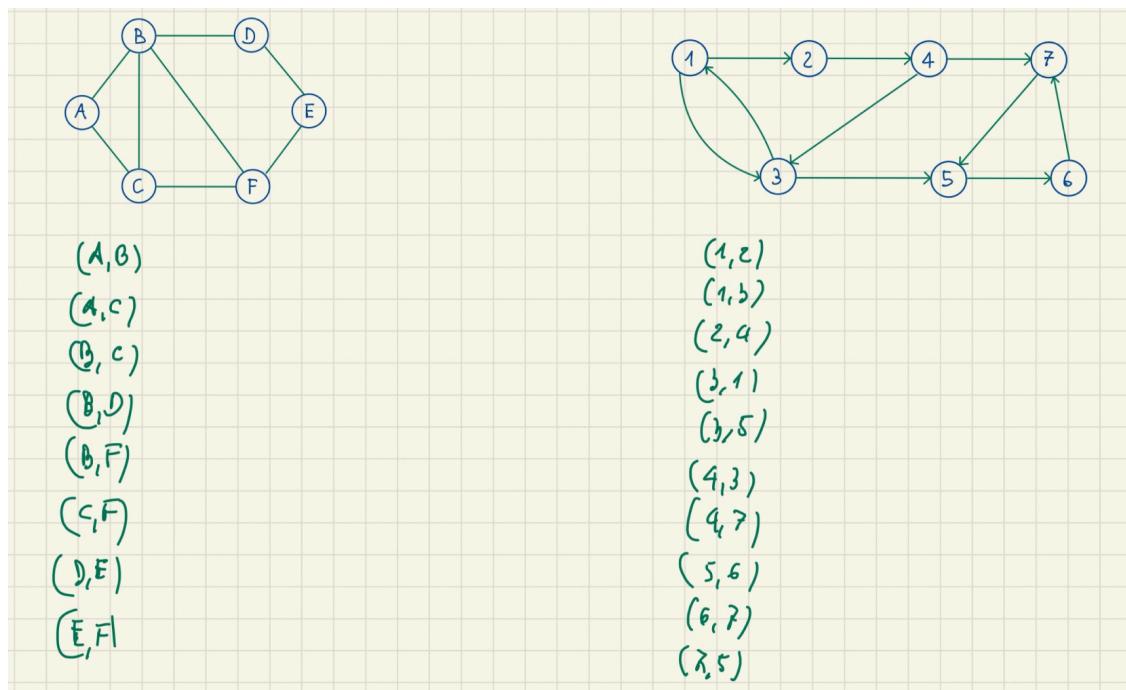
Una **cricca** è un grafo non orientato completo, ovvero in cui c'è un arco per ogni coppia di vertici

22.1 Rappresentazione di grafi

Vediamo ora alcuni metodi per rappresentare i grafi. La rappresentazione migliore dipende dai casi di utilizzo.

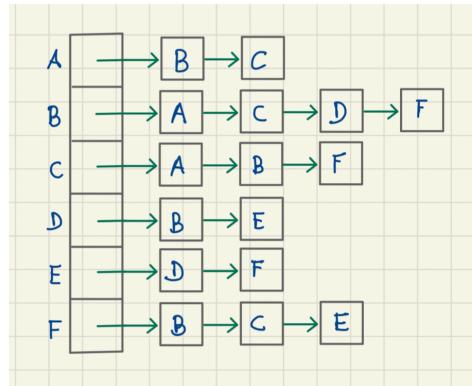
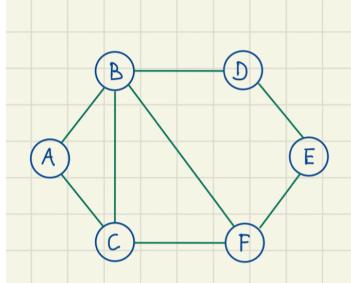
22.1.1 Lista di archi

Possiamo rappresentare gli archi come un elenco contenente le coppie di vertici che l'arco collega. Vale anche per i grafi orientati, ricordando che la posizione del nodo all'interno della coppia rappresenta l'orientamento dell'arco. Questa struttura è comoda per vedere i vertici di un arco ma è scomoda per ricostruire la forma del grafo, per seguire un cammino o se voglio sapere a cosa è collegato direttamente un vertice. In quest'ultimo caso infatti dovrei attraversare tutta la struttura. Lo spazio complessivo utilizzato è $O(n + m)$



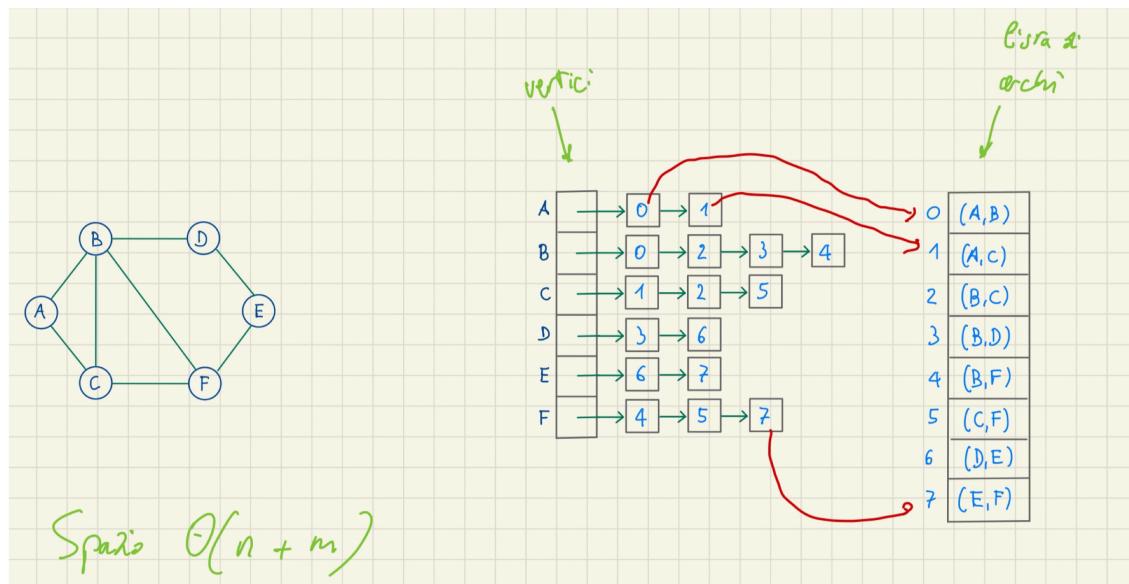
22.1.2 Lista di adiacenza

Struttura principale basata sui vertici. Per ogni vertice esiste la lista dei vertici adiacenti. Ogni arco è rappresentato due volte, quindi lo spazio occupato è $2m$ (solo dai nodi). Questa struttura è comoda per gli archi uscenti da ogni nodo ma se devo trovare gli archi entranti ad un nodo devo passare tutta la struttura. Inoltre non abbiamo informazioni esplicite sugli archi. Lo spazio complessivo utilizzato è $O(n + m)$



22.1.3 Lista di incidenza

Rimpiazziamo le liste dei vertici delle liste di adiacenza con delle liste di archi, tornando a usare strutture come nella lista di archi. Rimane il problema citato precedentemente sugli archi entranti. Lo spazio complessivo utilizzato è $O(n + m)$



22.1.4 Matrice di adiacenza

Si tratta di una matrice quadrata di 0 e 1 dove gli indici sono i vertici del grafo.

$M[u, v] = 1$ se e solo se $(u, v) \in E$. Un grafo non orientato genera una matrice simmetrica.

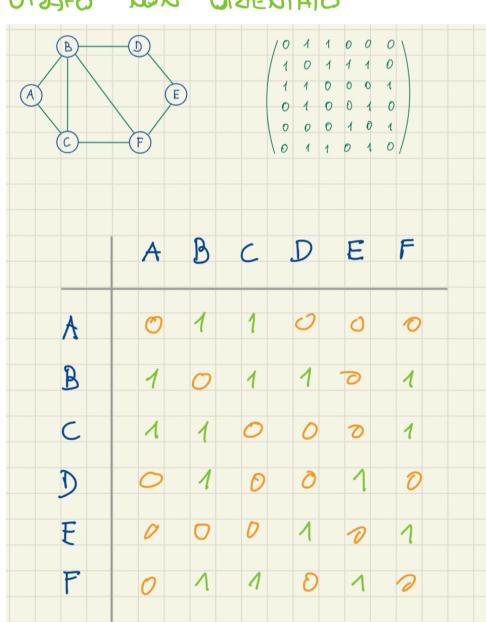
Osservando la matrice è possibile notare che possiamo vedere anche gli archi entranti leggendo le colonne. Lo spazio complessivo utilizzato è $O(n^2)$. Tale spazio è molto diverso da $O(n + m)$?

Dipende dal numero di archi. Si può dimostrare che, per ogni $k > 0$:

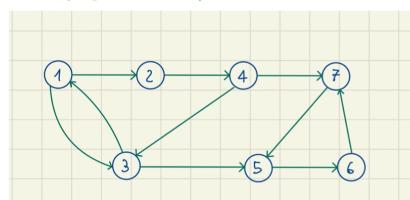
$$M^k[u, v] = 1 \text{ sse } V_{k=0}^{n-1} M^k \text{ "sommatoria" di OR}$$

Nella matrice risultante, se c'è un 1 in una determinata posizione significa che esiste un cammino. Quindi, in un grafo fortemente connesso, la matrice risultante sarà composta solo da 1.

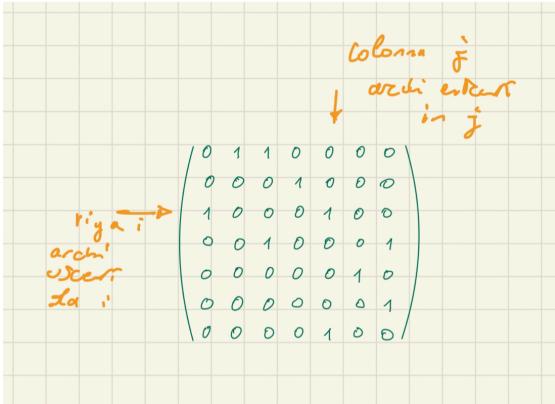
GRAFO NON ORIENTATO



GRAFO ORIENTATO



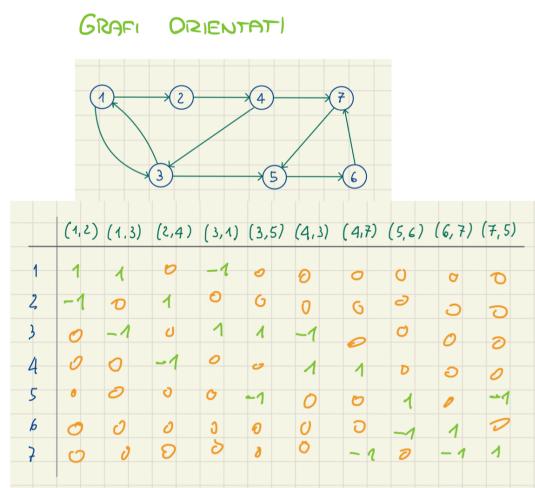
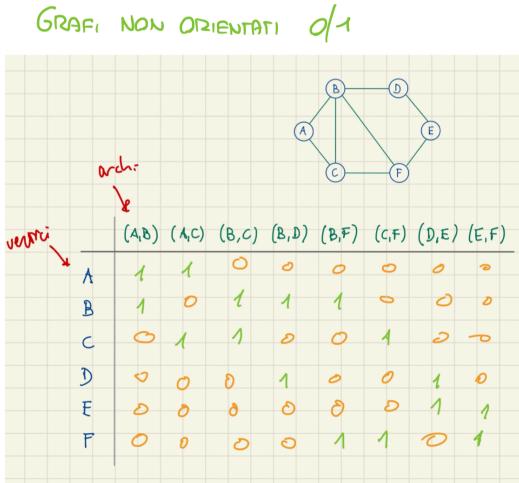
Colonna j
↓ archi entranti in j



22.1.5 Matrice di incidenza

Abbiamo una riga per ogni vertice e una colonna per ogni arco. Nei grafi non orientati metto 1 quando c'è un collegamento diretto, nei grafi orientati ho 1 quando c'è un arco uscente e -1 quando c'è un arco entrante. Questo sistema ci permette di risparmiare un po' di spazio mantenendo l'informazione su archi uscenti ed entranti per grafi orientati. Lo spazio complessivo è $O(n \cdot m)$

N.B. Ogni colonna contiene un 1 e un -1, quindi la somma algebrica di ogni colonna è pari a 0.



22.2 Attraversamento di grafi

Esistono diverse strategie per attraversare un grafo. Noi vedremo le visite in ampiezza e in profondità per grafi connessi e non orientati. I concetti di visità in ampiezza e in profondità sono gli stessi visti per gli alberi con radice. Il tempo impiegato dall'algoritmo dipende dalla struttura dati utilizzata per rappresentare il grafo.

$G = (V, E)$ grafo connesso non orientato,
 $s \in V$ vertice di partenza.

22.2.1 Visita in ampiezza

Questo algoritmo visita il grafo in ampiezza e crea un albero di supporto basato sul grafo dato in ingresso.

Codice 11.1 Visita in ampiezza (Breadth-First Search)

```

Algoritmo visitaInAmpiezza (grafo G = (V, E), vertice s) → 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 ← coda vuota
  T ← ( $\{s\}$ ,  $\emptyset$ )                                // albero formato dal solo vertice s
  marca s come raggiunto
  C.enqueue(s)
  while not C.isEmpty() do
    u ← C.dequeue()
    foreach  $(u, v) \in E$  do
      if v non è marcato come raggiunto then
        T ← (vertici(T) ∪ {v}, archi(T) ∪ {(u, v)})           // aggiunge il vertice v e l'arco (u, v) a T
        marca v come raggiunto
        C.enqueue(v)
  return T
```

Lista di archi	$O(n \cdot m)$
Lista di adiacenza	$O(n + m)$
Lista di incidenza	$O(n + m)$
Matrice di adiacenza	$O(n^2)$
Matrice di incidenza	$O(n \cdot m)$

22.2.2 Visita in profondità

Si parte da un vertice e si cerca di esplorare il più possibile partendo da ogni nodo in cui entriamo di volta in volta, finché non posso più muovermi. A quel punto torno indietro finché non trovo la prima strada che posso percorrere. Si va avanti fino a che non ho attraversato tutto il grafo. L'implementazione avviene tramite una pila o ricorsivamente (meglio la seconda alternativa). I tempi sono gli stessi visti per la visita in ampiezza.

Codice 11.2 Visita in profondità (Depth-First Search)

```
Algoritmo visitaInProfondità (grafo G = (V, E), vertice s) → albero
  /* Visita in profondità di un grafo non orientato连通的 a partire da un
   vertice s e costruzione di un albero ricoprente ottenuto selezionando
   gli archi secondo l'ordine della visita */ 
  T ← ({s}, ∅) // 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) ∈ E do
    if v non è marcato come visitato then
      T ← (vertici(T) ∪ {v}, archi(T) ∪ {(u, v)})
      // aggiunge il vertice v e l'arco (u, v) a T
      visitaRicorsiva(G, v, T)
```

23 Problemi di ottimizzazione e algoritmi greedy

23.0.1 Grafi pesati

Sono grafi in cui associo delle informazioni agli archi o ai nodi.

$G = (V, E)$ grafo,

$w : E \rightarrow \mathbb{R}$ funzione peso.

Alcuni esempi di problemi che utilizzano i grafi pesati sono:

- Cammini minimi
- Commesso viaggiatore
- Albero ricoprente minimo

23.0.2 Problemi di ottimizzazione

Tra tutte le soluzioni *ammissibili* per un problema voglio determinarne una *ottima* rispetto ad un dato criterio.

23.0.3 Tecnica greedy

P = problema di ottimizzazione C = insieme di candidati Voglio trovare $S^* \subseteq C$ ottima.

- In una sequenza di passi costruisco, a partire dall'insieme vuoto, una soluzione ammissibile $S \subseteq C$
- Ad ogni passo si espande una soluzione parziale già ottenuta
- L'algoritmo termina quando non è più possibile espandere la soluzione parziale

L'espansione della soluzione può essere vista in questo modo:

- **Soluzione ammissibile**

La soluzione parziale soddisfa i vincoli del problema

- **Scelta dell'ottimo locale**

Tra i candidati disponibili si sceglie quello che, al momento, appare migliore

- **Scelta irrevocabile**

Le scelte effettuate non vengono più messe in discussione

Algorithm 20: Schema tecnica greedy
--

Algoritmo <i>greedy</i> (insieme C) \rightarrow soluzione

$S \leftarrow \emptyset$ while $C \neq \emptyset$ do <ul style="list-style-type: none"> $x \leftarrow \text{seleziona}(C)$ /* elemento considerato "migliore" al momento */ $C \leftarrow C - \{x\}$ if $S \cup \{x\}$ è ammissibile then <li style="padding-left: 20px;">$S \leftarrow S \cup \{x\}$ return S
--

23.0.4 Programmazione dinamica

Si tratta di un approccio bottom-up. A differenza del divide-et-impera i sottoproblemi vengono risolti prima e le soluzioni parziali vengono salvate. Vediamo alcuni esempi.

Esempio 1: Dato un vettore V di interi in \mathbb{Z} trovare un sottovettore di somma massima.

$V[1...n]$ vettore in input

Sottovettore con

- Indice di inizio i con $1 \leq i \leq n$
- Indice di fine f con $1 \leq f \leq n$

Algorithm 21: Sottovettore di somma massima

```
Algoritmo sottovettoreMax(Array V[1...n]) → intero, intero
    max ← V[1], inizio ← 1, fine ← 1
    for f ← 1 to n do
        somma ← somma di V[1...f]
        if somma > max then
            max ← somma
            inizio ← i
            fine ← f
    return (inizio, fine)
```

Esempio 2: trovare il cammino di valore minimo in una matrice $n \cdot n$

$C[i, j] =$ costo cammino minimo che inizia nella colonna 1 e termina nella posizione (i, j)

La prima colonna è uguale alla prima colonna della matrice di partenza.

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

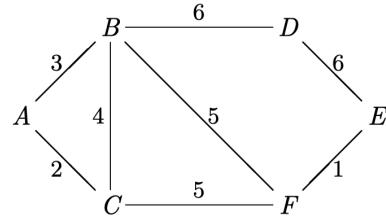
Anche se mi fermo prima di risolvere il problema ho comunque una soluzione ottima per il sottoproblema.

Algorithm 22: Cammino minimo in una matrice

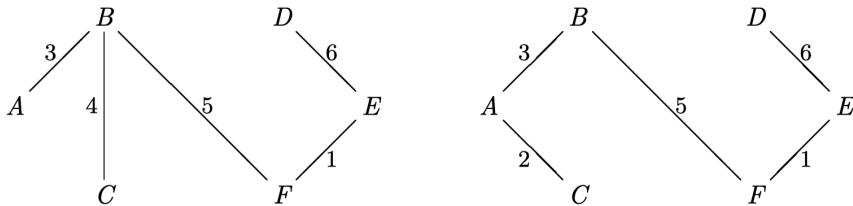
```
Algoritmo camminoMinimo(Matrice M[1...n, 1...n]) → intero
    Sia C[1...n, 1...n] una matrice
    for i ← 1 to n do
        C[i, 1] = M[i, 1] /* riempio la prima colonna */
    for j ← 2 to n do
        for i ← 1 to n do
            min ← C[i, j - 1]
            if i > 1 and C[i - 1, j - 1] < min then
                min ← C[i - 1, j - 1]
            if i < n and C[i + 1, j - 1] < min then
                min ← C[i + 1, j - 1]
        C[i, j] = M[i, j] + min
    for i ← 2 to n do
        if C[i, n] < min then
            min ← C[i, n]
    return min
```

23.1 Albero ricoprente minimo

Ricordiamo che dato un grafo non orientato e pesato, un *albero ricoprente minimo* del grafo è un albero ricoprente il cui peso sia minimo tra tutti gli alberi ricoprenti del grafo. Vediamo ora due



Nella figura seguente sono rappresentati due alberi ricoprenti di G . Quello di sinistra ha peso 19, quello di destra ha peso 17. Si può verificare che non esistono alberi ricoprenti di peso inferiore a 17. Pertanto l'albero di destra è un albero ricoprente minimo per il grafo G . Si può anche osservare che esistono altri alberi ricoprenti minimi, cioè con peso 17.



□

algoritmi per trovare un albero ricoprente minimo di un grafo连通的, non orientato e pesato. In entrambi gli algoritmi viene costruito in modo incrementale utilizzando una strategia greedy.

23.1.1 Algoritmo di Kruskal

Il primo algoritmo risolve il problema costruendo un grafo T che ha gli stessi vertici di G e, inizialmente, è privo di archi. L'algoritmo esamina 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. Pertanto, ad ogni passo, il grafo T è una foresta di alberi. Ogni volta che si aggiunge un arco 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 è un unico albero ricoprente che, come dimostreremo, è di peso minimo per il grafo G dato. Si può dimostrare che l'algoritmo trova sempre la soluzione ottima, cioè trova sempre un albero ricoprente di peso minimo.

Studiamo ora una possibile implementazione dell'algoritmo di Kruskal. È utile rappresentare il

Codice 12.1 Algoritmo di Kruskal

```
Algoritmo Kruskal (grafo connesso non orientato  $G = (V, E, \omega)$ ) → 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  $T$ 
```

grafo come lista di archi. La lista può essere rappresentata direttamente in un array, sul quale applicare uno degli algoritmi di ordinamento (in base ai pesi degli archi). Insieme al grafo T che viene costruito, utilizziamo una struttura che permetta, quando si ispeziona un arco (x, y) , di decidere facilmente se i vertici x e y sono già connessi in T . A tale scopo possiamo considerare partizioni dell'insieme dei vertici V , in cui due vertici appartengono allo stesso elemento della partizione se e solo se sono connessi in T . In altre parole ogni elemento della partizione rappresenta una componente连通的 di T .

- Inizialmente ogni vertice di V costituisce un singolo insieme della partizione (T non contiene archi e dunque non ci sono archi connessi tra loro).
- Quando esaminiamo un arco ci sono due possibilità:
 - Se x e y appartengono allo stesso elemento della partizione significa che sono già connessi in T . In tal caso l'arco (x, y) non viene aggiunto a T perché creerebbe un ciclo
 - Se x e y appartengono ad elementi diversi della partizione allora non sono connessi: aggiungendo l'arco (x, y) a T rendiamo ciascun vertice dell'elemento a cui appartiene x connesso con ciascun vertice dell'elemento a cui appartiene y , cioè rendiamo le due componenti connesse a cui appartengono x e y un'unica componente连通的.

La partizione può essere rappresentata mediante le strutture Union-Find. Per verificare se x e y appartengono allo stesso insieme della partizione confronto i risultati di $\text{FIND}(x)$ e $\text{FIND}(y)$. Per unire due elementi della partizione utilizziamo UNION .

Stimiamo ora il tempo di calcolo in funzione del numero n di vertici e m di archi del grafo G in input. Assumendo il criterio di costo uniforme, supponiamo che i confronti tra i pesi degli archi avvengano in tempo costante. Dobbiamo tenere conto dei seguenti tempi:

- Ordinamento di E :

Utilizziamo `heapSort` e ordiniamo in tempo $O(m \log m)$

- Operazioni Union/Find:

Supponiamo di usare QuickUnion con bilanciamento in altezza in cui ciascuna operazione `MAKESET` viene effettuata in tempo costante, `FIND` in tempo $O(\log n)$ e `UNION` in tempo costante, dove n è il numero di elementi presenti complessivamente negli insiemi della partizione. L'algoritmo effettua queste operazioni:

- n operazioni di `MAKESET`: tempo $O(n)$
- $2m$ operazioni di `FIND`: tempo $O(m \log n)$
- $n - 1$ operazioni di `UNION`: tempo $O(n)$

Sommando i vari tempi otteniamo $O(m \log n)$ approssimabile a $O(m \log m)$. Se i pesi sono interi si potrebbe ridurre il costo dell'ordinamento usando `radixSort`.

Codice 12.2 Algoritmo di Kruskal (mediante Union-Find)

```

Algoritmo Kruskal (grafo connesso non orientato  $G = (V, E, \omega)$ ) → albero
/* Costruzione di un albero di ricoprimento di peso minimo */ 
ordina l'insieme  $E$  in base ai pesi in modo non decrescente
 $T \leftarrow (V, \emptyset)$ 
Sia  $P$  una partizione inizialmente vuota
foreach  $v \in V$  do  $P.makeSet(v)$ 
foreach  $(x, y) \in E$  secondo l'ordine do
     $t_x \leftarrow P.find(x)$ 
     $t_y \leftarrow P.find(y)$ 
    if  $t_x \neq t_y$  then
         $P.union(t_x, t_y)$ 
        aggiungi a  $T$  l'arco  $(x, y)$ 
return  $T$ 

```

23.1.2 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). Si può dimostrare che, come l'algoritmo di Kruskal, anche questo trova sempre una soluzione ottima.

L'algoritmo di Prim può essere implementato ricorrendo ad una coda con priorità C , contenente

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 vertice  $s$   

while  $T$  ha meno di  $n$  vertici 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  $T$ 
```

un elemento per ogni vertice che deve ancora essere inserito nell'albero, secondo la tecnica che descriviamo ora:

- Ad ogni passo, per ogni vertice v non ancora in T consideriamo le seguenti informazioni:
 - $d[v]$: minimo peso di un arco tra un vertice appartenente all'albero T già costruito e v ,
 - $vicino[v]$: un vertice u nell'albero T già costruito con distanza minima da v
- La coda con priorità C contiene ciascun vertice v non ancora inserito in T con priorità $d[v]$
- Inizialmente l'albero è vuoto. Pertanto, per ogni vertice v , si pone $d[v] = \infty$, mentre il valore di $vicino[v]$ non è definito. Ogni vertice viene inserito in C
- Ad ogni passo si sceglie un vertice y corrispondente al minimo in C . (Al primo passo se ne sceglie uno qualsiasi)
- Nei passi successivi al primo si considera il "vicino" x in T del vertice y scelto. L'arco (x, y) è pertanto un arco di peso minimo con un vertice x in T e l'altro vertice y non in T . Il vertice y e l'arco (x, y) vengono aggiunti all'albero.
- Si ricalcolano le priorità dei vertici, tenendo conto del nuovo vertice y inserito in T . Per ogni arco (y, z) uscente da y con z non in T , nel caso il peso $w(y, z)$ risulti minore di $d[z]$, si modifica $d[z]$ e si aggiorna la coda con priorità e l'informazione relativa al vicino di z .
- Queste operazioni vengono ripetute fino a svuotare la coda. A quel punto si può restituire T

Tempo di calcolo

Prima di tutto assumiamo che il grafo in ingresso sia rappresentato mediante liste di adiacenza o di incidenza. Questo permette di trovare facilmente tutti gli archi entranti incidenti su un vertice. La coda con priorità può essere rappresentata come un array di n elementi e riempita in $O(n)$. Verranno eseguite in totale n operazioni `deleteMin()`, ciascuna delle quali impiega tempo al più $O(\log n)$, per un tempo complessivo pari a $O(n \log n)$. Anche tempo complessivo utilizzato dalle operazioni `changeKey` è $O(m \log n)$. Sommando questi tempi otteniamo $O(m \log n)$, come per Kruskal. Con una implementazione basata sugli *heap di Fibonacci* è possibile ottenere tempo $O(m + n \log n)$ che è meglio del precedente in quanto il numero di archi nel grafo è alto.

Codice 12.4 Algoritmo di Prim (mediante code con priorità)

```

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

  Sia  $C$  una coda con priorità inizialmente vuota  

  Siano  $vicino[V]$  e  $d[V]$  due array con insieme di indici  $V$   

  foreach  $v \in V$  do  

     $d[v] \leftarrow \infty$   

     $C.insert(v, \infty)$   

   $T \leftarrow (V_T \leftarrow \emptyset, E_T \leftarrow \emptyset)$  // albero vuoto  

  do  

     $y \leftarrow C.deleteMin()$   

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

    if  $d[y] \neq \infty$  then // condizione falsa solo nella prima iterazione  

       $x \leftarrow vicino[y]$   

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

    foreach  $(y, z) \in E$  do  

      if  $z \notin V_T$  and  $\omega(y, z) < d[z]$  then  

         $d[z] \leftarrow \omega(y, z)$   

         $C.changeKey(z, \omega(y, z))$   

         $vicino[z] \leftarrow y$   

  while  $C \neq \emptyset$   

  return  $T$ 
```

Codice 12.5 Algoritmo di Prim (mediante code con priorità, vertice di partenza fissato)

```

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

Sia C una coda con priorità inizialmente vuota

Siano $vicino[V]$ e $d[V]$ due array con insieme di indici V

```

 $d[s] \leftarrow 0$ 
 $C.insert(s, 0)$ 
foreach  $v \in V \setminus \{s\}$  do
   $d[v] \leftarrow \infty$ 
   $C.insert(v, \infty)$ 
```

$T \leftarrow (V_T \leftarrow \emptyset, E_T \leftarrow \emptyset)$ // albero vuoto

```

while  $C \neq \emptyset$  do
   $y \leftarrow C.deleteMin()$ 
   $V_T \leftarrow V_T \cup \{y\}$ 
  if  $y \neq s$  then // condizione falsa solo nella prima iterazione
     $x \leftarrow vicino[y]$ 
     $E_T \leftarrow E_T \cup \{(x, y)\}$ 
  foreach  $(y, z) \in E$  do
    if  $z \notin V_T$  and  $\omega(y, z) < d[z]$  then
       $d[z] \leftarrow \omega(y, z)$ 
       $C.changeKey(z, \omega(y, z))$ 
       $vicino[z] \leftarrow y$ 
```

return T

23.2 Cammini minimi

Siano:

- $G(V, E)$ un grafo orientato
- w funzione peso
- $\pi = \langle V_0 \dots V_k \rangle$ un cammino da V_0 a V_k
- $w(\pi)$ peso del cammino

Un cammino minimo tra due vertici è il cammino che ha peso minore tra tutti i cammini tra i due vertici.

Alcune proprietà dei cammini minimi:

1. Se π è un cammino minimo tra x e y che passa per un vertice v allora:
 - La parte da x a v è un cammino minimo
 - La parte da v a y è un cammino minimo
2. Se tutti i pesi sono positivi allora ogni cammino minimo è semplice
3. Se ci sono pesi negativi ma non ci sono cicli di peso negativo allora tra ogni coppia di vertici esiste un cammino minimo semplice

Per rappresentare grafi pesati posso usare liste di adiacenza con associate ad ogni arco le informazioni riguardanti il peso, oppure una matrice dei pesi in cui se tra due vertici c'è un arco scrivo il suo peso, altrimenti scrivo ∞ .

23.2.1 Algoritmo di Floyd-Warshall

Questo algoritmo calcola le lunghezze dei cammini minimi tra ogni coppia di vertici.

Gli elementi d_{ij} sono uguali a:

$$\begin{cases} w(V_i, V_j) & \text{se } (V_i, V_j) \in E \text{ e } V_i \neq V_j \\ 0 & \text{se } V_i = V_j \\ \infty & \text{altrimenti} \end{cases}$$

Lavora correttamente anche con pesi negativi purchè non ci siano cicli negativi.

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

```

Algoritmo FloydWarshall (grafo pesato  $G = (V, E, \omega)$ )  $\rightarrow$  matrice distanze
/* La funzione  $\omega$  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  $i \leftarrow 1$  to  $n$  do
  for  $j \leftarrow 1$  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  $k \leftarrow 1$  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  $\leq k$ 
  for  $i \leftarrow 1$  to  $n$  do
    for  $j \leftarrow 1$  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  $D$ 
```

Esiste anche un'altra versione in cui viene calcolata anche una matrice P che può essere utilizzata per ricavare i cammini minimi. Dopo l'iterazione k , l'elemento $D[i, j]$ contiene la lunghezza del cammino minimo i cui vertici intermedi hanno indice al più k . L'elemento $P[i, j]$ contiene il massimo indice di tali vertici intermedi. Pertanto, se alla fine dell'esecuzione $P[i, j] = 0$, significa che il cammino minimo da v_i a v_j non passa per vertici intermedi, se $P[i, j] = h > 0$ significa che il cammino minimo da v_i a v_j passa per v_h ed è costituito dal cammino da v_i a v_h seguito dal cammino da v_h a v_j .

Codice 13.2 Calcolo della lunghezza dei cammini minimi tra ogni coppia di vertici, tenendo traccia di informazioni sui vertici intermedi

Algoritmo FloydWarshall (*grafo pesato* $G = (V, E, \omega)$) \rightarrow *matrice distanze*

/ La funzione ω assegna un peso a ciascun arco.
Il grafo non deve contenere cicli di peso negativo. */*

Siano D e P due matrici $n \times n$, 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] ← 0
        else if (vi, vj) ∈ E then D[i, j] ← ω(vi, vj)
        else D[i, j] ← ∞
        P[i, j] ← 0

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 {v1, v2, ..., vk}
    // cioè hanno indice ≤ k
    // Nella matrice P si tiene traccia del massimo indice dei vertici
    // intermedi
    // lungo tale cammino minimo
    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] ← D[i, k] + D[k, j]
                D[i, j] ← k

return D

```

23.2.2 Algoritmo di Bellman e Ford

Supponiamo di avere un grafo privo di cicli negativi.

- $d_v^{[k]}$ = lunghezza del cammino minimo da s a v che visita al più k archi.
- Allora la lunghezza del cammino minimo da s a v è $d_v^{[n-1]}$
- $d_v^{[0]} = \begin{cases} 0 & \text{se } v = s \\ \infty & \text{altrimenti} \end{cases}$
- $d_v^{[k]} = \min(d_v^{[k-1]}, d_u^{[k-1]} + w(u, v) \text{ t.c. } u \in V)$

Codice 13.3 Calcolo della lunghezza dei cammini mimimi da un vertice s a tutti gli altri

Algoritmo BellmanFord (*grafo pesato* $G = (V, E, \omega)$, vertice s) \rightarrow vettore distanze

/* La funzione ω assegna un peso a ciascun arco.

Il grafo non deve contenere cicli di peso negativo.

*/

Sia D un vettore con insieme di indici V

$D[s] \leftarrow 0$

foreach $v \in V \setminus \{s\}$ **do** $D[v] \leftarrow \infty$

for $k \leftarrow 1$ **to** $n - 1$ **do**

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

23.2.3 Algoritmo di Dijkstra

Supponiamo di avere pesi non negativi.

- **Distanze provvisorie vettore $d[v]$**

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

- **$C \subseteq V$ insieme dei vertici candidati** Inizialmente $c = V$

- **Ad ogni passo strategia greedy**

1. Preleva da C il vertice u con $d[u]$ minima
2. $d[u]$ diventa definitiva
3. Aggiorna $d[v]$ per ogni v adiacente a u

È implementabile utilizzando liste di adiacenza o di incidenza per rappresentare il grafo e code con priorità. Il tempo di esecuzione è $O(m \log n)$

Codice 13.4 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

Codice 13.5 Calcolo della lunghezza dei cammini mimimi da un vertice s a tutti gli altri

Algoritmo Dijkstra (grafo pesato $G = (V, E, \omega)$, vertice s) \rightarrow vettore distanze

/* La funzione ω 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** $D[v] \leftarrow \infty$

Sia C una coda con priorità inizialmente vuota

foreach $v \in V$ **do** $C.insert(v, D[v])$

while $C \neq \emptyset$ **do**

$u \leftarrow C.deleteMin()$

foreach $(u, v) \in E$ **do**

if $D[u] + \omega(u, v) < D[v]$ **then**

$D[v] \leftarrow D[u] + \omega(u, v)$

$C.changeKey(v, D[v])$

return D

24 Dizionari e tabelle hash

I *dizionari* sono una collezione di elementi identificati da una chiave. Le operazioni che vogliamo poter eseguire sono ricerca, inserimento e cancellazione. Le possibili implementazioni che abbiamo visto finora sono array e alberi. Vediamo un riepilogo dei tempi richiesti dalle tre operazioni nelle varie strutture.

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

Non è sempre vero che tenere le cose in ordine ci permette di trovarle più velocemente. Infatti esistono alcune strutture che disordinano i dati di proposito, ovvero le *tabelle hash*.

24.1 Funzioni hash

Siano

- $U =$ universo delle chiavi
- $\{0 \dots m - 1\}$ spazio degli indici

Funzioni hash $h: U \rightarrow \{0 \dots m - 1\}$ trasformazioni di chiavi in indici

24.2 Fattore di carico

$$\alpha = \frac{n}{m}$$

- $n =$ numero di elementi memorizzati nella tabella
- $m =$ posizioni disponibili nella tabella
- Se $\alpha = 1$ la tabella è piena
- Se $\alpha = 0$ la tabella è vuota

Quando mi avvicino allo 0 sto "sprecando" la tabella. Una *funzione hash perfetta* (*o iniettiva*) è una funzione hash tale che:

$$\text{se } m \neq v \Rightarrow h(m) \neq h(v)$$

Nella pratica, salvo in casi particolari:

- Il numero di chiavi possibili è molto più grande del numero di chiavi attese
- La dimensione della tabella è scelta paragonabile al numero di chiavi attese

24.3 Gestione delle collisioni

Supponiamo di voler catalogare 20 persone in una tabella di 26 posizioni e che la nostra funzione di hash sia la prima lettera del cognome. È una funzione perfetta? No, perché esistono lettere più diffuse di altre per i cognomi e rischiamo che due persone vadano a finire nella stessa posizione della tabella, creando una *collisione*. Dobbiamo quindi fare in modo che le collisioni avvengano

raramente e, nel caso avvengano, avere una strategia per gestirle. Per quanto riguarda il fare in modo che le collisioni avvengano il meno possibile introduciamo i concetti di *sparpagliamento* e *uniformità*. Siano:

- $h: U \rightarrow \{0 \dots m - 1\}$ una funzione hash
- $P(x)$ la probabilità che scegliendo a caso una chiave da U si scelga x
- $Q(i) = \sum_{x|h(x)=i} P(x)$ probabilità che una chiave scelta a caso da U abbia valore hash i

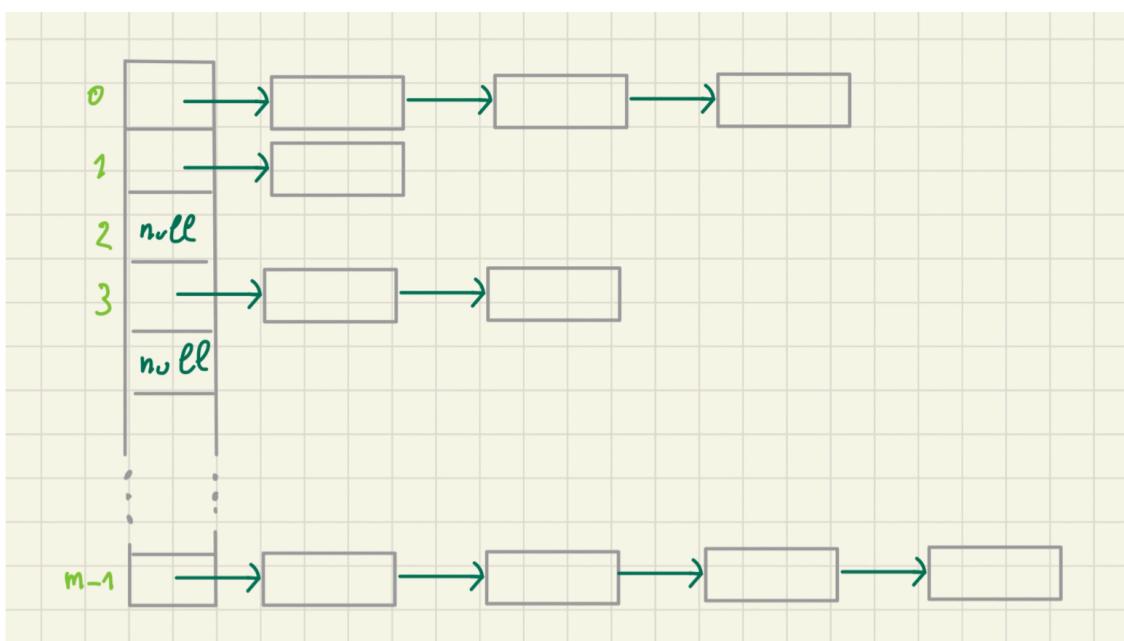
La funzione hash è uniforme se $Q(i)$ è la stessa per ogni i , cioè $Q(i) = \frac{1}{m}$

Alcuni esempi di funzioni hash sono il *metodo della divisione* e il *metodo del ripiegamento*.

Esistono due categorie di tecniche per gestire le collisioni: *interne* ed *esterne*.

24.4 Gestione esterna

Una tecnica di gestione esterna delle collisioni sono le *liste di collisione*.



In posizione i troviamo ogni record la cui chiave x ha valore hash i . La struttura consiste in un array di liste di coppie <elemento, chiave>. Ogni volta che un nuovo elemento deve essere inserito viene messo esterno alla tabella hash vera e propria ma collegato alla posizione corretta. Nel momento in cui arriva un nuovo elemento che collide con uno già presente viene aggiunto alla lista in testa e senza ordinamento. I tempi delle operazioni sono:

- **Inserimento:** $O(1)$
- **Ricerca:** $O(m)$
- **Cancellazione:** $O(n)$

Il tempo medio è $O(1 + \alpha)$, dipende dalla lunghezza della lista.

Quando una lista si riempie troppo si parla di *agglomerazione* ed è un problema che dobbiamo

cercare di evitare. Se si verifica significa che la funzione scelta non è adeguata dal punto di vista dell'uniformità.

24.5 Gestione interna

Esistono diverse metodologie interne per la gestione delle collisioni. Vedremo l'indirizzamento aperto. A grandi linee, possiamo dire che memorizziamo tutto nella tabella e in caso di collisione troviamo un altro posto libero scegliendo una delle possibili strategie. La prima strategia è cercare il primo posto vuoto disponibile e se arrivo in fondo riparto dalla cima. Questo sistema è afflitto dal tipo peggiore di agglomerazione, detta agglomerazione primaria, che si verifica quando ho valori con chiavi di diversi valori di hash che si mescolano. Formalmente questa metodologia si chiama *funzione ausiliaria*, nello specifico *scansione lineare* $C(k, i)$, dove k è la chiave, $i \geq 0$, $C(k, i) = (h(k) + i) \bmod m$.

24.5.1 Scansione quadratica

$$C(k, i) = \lfloor h(k) + C_1 i + C_2 i^2 \rfloor \bmod m$$

Questa funzione ausiliaria ci permette di evitare l'agglomerazione primaria ma non interviene su quella secondaria (meno grave ma comunque fastidiosa).

24.5.2 Hashing doppio

$C(k, i) = [h(k) + ih'(k)] \bmod m$ dove h' è una seconda funzione hash. La situazione ideale sarebbe $h(k_1) = h(k_2) \Rightarrow h'(k_1) \neq h'(k_2)$

In parole povere significa che se trovo un posto occupato provo ad usare una differente funzione (la stessa cosa con l'incremento di i).

24.5.3 Operazioni

Algorithm 23: Inserimento di un elemento nella tabella

```
Algoritmo inserimento(elemento i, chiave k)
    i ← 0
    while i < m and v[C(k, i)] è occupata do
        i ← i + 1
    if i < m then
        | v[C(k, i)] ← (e.k)
    else
        | errore! tabella piena
```

Per quanto riguarda la cancellazione, questa è più insidiosa perchè, ogni volta che si cancella un elemento, andrebbe ristrutturata la tabella, in quanto si perderebbero i legami impliciti tra le celle che vengono usati nelle ricerche. Per questo motivo non avviene mai una cancellazione vera e propria ma una "virtuale": introduciamo un flag booleano che indichi se il dato contenuto in quella posizione è cancellato o meno. Quando ci sarà un nuovo inserimento il dato vecchio sarà sovrascritto e il flag riportato a "non cancellato".

Algorithm 24: Ricerca di un elemento nella tabella

```

Algoritmo ricerca(chiave k) → elemento
    i ← 0
    while i < m and v[C(k, i)] è occupata and v[C(k, i)].chiave ≠ k do
        i ← i + 1
    if i = m or v[C(k, i)] è libera then
        return null
    else
        return v[C(k, i)].elemento

```

24.5.4 Numero di confronti

	Scansione lineare	Scansione quadratica e hashing doppio
Chiave trovata	$\frac{1}{2} + \frac{1}{2(1-\alpha)}$	$\frac{1}{\alpha} \log \log_2(1 - \alpha)$
Chiave non trovata	$\frac{1}{2} + \frac{1}{2(1-\alpha)^2}$	$\frac{1}{1-\alpha}$

Ricordiamo che α è il fattore di carico della tabella. Se $\alpha < 1$ i confronti saranno sempre in numero molto limitato. Questo significa che le funzioni ausiliarie da noi esposte sono efficienti più la tabella è vuota. Questa considerazione ci porterà alla ricerca di alcuni metodi (*re-hashing*) atti al mantenimento di un minimo di posti liberi ridimensionando la tabella.

24.5.5 Re-hashing

Si tratta della sostituzione della tabella con una nuova. È un'operazione molto dispendiosa in termini di tempo. Occorre inoltre una funzione hash adatta alla nuova tabella. Per quanto riguarda lo spostamento degli elementi dalla tabella vecchia a quella nuova occorre scandire l'intera tabella e inserire ogni suo elemento nella nuova tabella, calcolandone la posizione in base alla nuova funzione di hash e risolvendo eventuali collisioni. Pertanto il re-hashing richiede un numero minimo di passi pari almeno alla dimensione della vecchia tabella. Sebbene appaia molto dispendioso in termini di tempo, se gestito bene ha un costo ammortizzato basso. Supponiamo di procedere come segue:

- Fissiamo il valore massimo del fattore di carico ad $\alpha_{max} = \frac{1}{2}$
- Ogni volta che il fattore di carico raggiunge il valore massimo al momento del successivo inserimento effettueremo il re-hashing sostituendo la tabella con una nuova di capacità doppia.

Immaginiamo di avere una tabella T_0 di dimensione m e di effettuare una serie di inserimenti, sostituendo, quando necessario, una tabella T_i con una tabella $T_i + 1$ mediante re-hashing. Il

tabella	T_0	→	T_1	→	T_2	→	...	→	T_k
capacità	m		$2m$		2^2m		...		$2^k m$
numero max elementi	$m/2$		m		$2m$...		$2^{k-1}m$

numero totale di operazioni di inserimento che si effettuano a causa dei re-hashing è $\frac{m}{2}(2^k - 1)$. Possiamo concludere che se effettuiamo N operazioni di inserimento in una tabella hash, a causa del re-hashing effettuiamo in totale $O(N)$ ulteriori inserimenti. Se ogni operazione di inserimento utilizza un numero di passi costante, il numero di passi totali tenendo conto anche del re-hashing è $O(n)$. Pertanto, dividendo per il numero N di inserimenti "effettivi", otteniamo che il tempo ammortizzato è $\frac{O(n)}{N} = O(1)$. Quindi, anche effettuando il re-hashing nel modo indicato sopra, il costo medio delle operazioni di inserimento resta costante.

25 Classificazione dei problemi e complessità computazionale

Abbiamo un problema e vogliamo progettare uno o più algoritmi per risolverlo. Quando analizzo l'algoritmo e sono certo che funzioni vado a fare una stima delle risorse, solitamente tempo e spazio, ma non solo. Useremo il simbolo π per riferirci ad un problema.

- **Limitazione superiore** (upper bound) $f: \mathbb{N} \rightarrow \mathbb{N}$

$f(n)$ risorsa r è *sufficiente* per risolvere π se esiste un algoritmo \mathcal{A} che risolve π utilizzando su ogni input di lunghezza n al più $f(n)$ risorsa r .

- **Limitazione inferiore** (lower bound) $g: \mathbb{N} \rightarrow \mathbb{N}$

$g(n)$ risorsa r è *necessaria* per risolvere π se per ogni algoritmo \mathcal{A} che risolve π esiste un input di lunghezza n su cui \mathcal{A} utilizza almeno $g(n)$ risorsa r .

Per fare un esempio, tra gli algoritmi di ordinamento $\Theta(n^2)$ rappresenta un upper bound (`insertionSort`) e $\Theta(n \log n)$ rappresenta un lower bound.

Un algoritmo è ottimale quando lower bound ed upper bound coincidono.

25.1 Classi di complessità

Siano

$s, t: \mathbb{N} \rightarrow \mathbb{N}$ due funzioni.

Una *classe di complessità* è l'insieme dei problemi che possono essere risolti utilizzando la "stessa" quantità di una determinata risorsa. (stessa è tra virgolette perché non ci basiamo su un valore preciso ma su una categoria più ampia). Abbiamo innanzitutto la classe **P**, che è una classe di problemi π che ammettono un algoritmo risolutivo che utilizza tempo polinomiale ($n^{O(1)}$). Un esempio di un problema che non appartiene alla classe **P** è quello del commesso viaggiatore, ma la maggior parte di quelli che abbiamo visto appartiene a questa classe. Questo ci fa capire che una classe può contenere problemi di tipologie molto diverse. Troviamo per esempio problemi di ricerca (albero ricoprente), problemi di ottimizzazione (albero ricoprente minimo), e problemi di decisione. Un problema di decisione si può risolvere tramite un problema di ottimizzazione. I problemi di decisione ci permettono di confrontare in maniera molto semplice problemi apparentemente diversi. Spesso poi il problema di decisione ci permette di risolvere senza troppa fatica il problema di ottimizzazione associato.

Per esempio:

- Ottimizzazione: dato un grafo trovare l'albero ricoprente minimo
- Decisione: dato un grafo, esiste l'albero ricoprente del grafo di peso $\leq K$?

Se π è un problema di decisione e \mathcal{A} è un algoritmo, \mathcal{A} risolve π quando su input x \mathcal{A} restituisce 1 se e solo se $\pi(x) = 1$.

\mathcal{A} risolve π in tempo $t(n)$ e spazio $s(n)$ se e solo se \mathcal{A} risolve π utilizzando al più tempo $t(n)$ e al

più spazio $s(n)$ su ogni input di lunghezza n .

Possiamo formalizzare questi concetti e vedere alcune classi di complessità:

- **TIME**($t(n)$) = classe di problemi di decisione risolvibili in tempo $O(t(n))$
- **SPACE**($t(n)$) = classe di problemi di decisione risolvibili in spazio $O(s(n))$
- **CLASSE P** = $\bigcup_{c=0}^{\infty}$ **TIME**(n^c) (classe considerata risolvibile a tutti gli effetti)
- **PSPACE** = $\bigcup_{c=0}^{\infty}$ **SPACE**(n^c) spazio polinomiale
- **EXPTIME** = $\bigcup_{c=0}^{\infty}$ **TIME**(2^{n^c}) tempo esponenziale

Esistono alcune relazioni tra lo spazio utilizzato ed il tempo utilizzato da un algoritmo:

- tempo polinomiale \Rightarrow spazio polinomiale, quindi **P** \subseteq **PSPACE**
- spazio polinomiale \Rightarrow tempo esponenziale, quindi **PSPACE** \subseteq **EXPTIME**

Da queste due considerazioni deduco che **P** \subseteq **PSPACE** \subseteq **EXPTIME**

25.2 Problemi NP-completi

Sono i problemi più difficili nella classe **NP** (problem non deterministici in tempo polinomiale). Vediamo alcuni esempi:

Problema delle partizioni

Dato un insieme finito di oggetti, trovare due sottoinsiemi tali che la somma degli elementi dei due sottoinsiemi sia uguale. Il tempo necessario per il calcolo delle partizioni è circa 2^n quindi ricade in **EXPTIME**

Problema delle cricche

Dato un grafo non orientato e un intero k , stabilire se esiste un sottografo completo con k vertici. La verifica avviene in tempo polinomiale, ma il costo della ricerca effettiva è $\binom{n}{k}$ che può essere anche esponenziale.

25.2.1 Problema soddisfacibilità (SODD)

Istanza: Formula booleana Φ in forma normale congiuntiva con insieme di variabili V .

Questione: Esiste un assegnamento alle variabili in V che rende vera Φ , cioè tale che $\Phi(f) = 1$.

Per decidere se Φ è soddisfacibile proviamo tutti i possibili assegnamenti di valori alle variabili. Quando si dice che la questione è vera e la clausola è soddisfacibile, bisogna portare il certificato, cioè la soluzione che risolve la formula. Anche in questo caso verificare il problema è piuttosto semplice ma trovare la soluzione richiede tempo esponenziale. Introduciamo il termine *non deterministico*, ovvero un concetto utilizzato negli automi per definire la scelta di soluzione "indovinata".

La verifica è polinomiale, ma cosa possiamo dire della computazione non deterministica di *indovina*? Introduciamo la classe **NP** in cui abbiamo certificati verificabili in tempo polinomiale (ricordiamo che stiamo sempre parlando di problemi di decisione).

Chiamiamo **NTIME** di $f(n)$ la classe dei problemi che possono essere risolti da algoritmi non

Algorithm 25: Problema di soddisfacibilità

```

Algoritmo sodd( $\Phi(x_1 \dots x_n)$ )
  for  $i \leftarrow 1$  to  $n$  do
     $z \leftarrow$  indovina un valore in  $\{0, 1\}$ 
     $r \leftarrow$  valore della formula  $\Phi(z_0 \dots z_n)$  /* verifica */
  return  $r$ 
```

deterministici in tempo $V(f(n))$.

(**NP** sta per "Non Deterministic P" e **NON** "Non Polinomiale"!)

NP = $U_{c=0}^{\infty}$ **NTIME**(n^c)

25.3 Relazioni tra classi di complessità

Anche i problemi della partizione e della cricca sono risolvibili con un algoritmo **NP** in tempo $O(n)$ e $O(n^2)$, hanno le stesse caratteristiche di SODD e quindi appartengono alla classe **NP**.

Lo stesso discorso fatto per **NSPACE** potremmo riprenderlo per **NPSPACE**.

Ogni algoritmo deterministico è un caso particolare di un algoritmo non deterministico

Possiamo quindi concludere che

$$\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE} \subseteq \mathbf{EXPTIME}$$

Vogliamo ora dimostrare che $\mathbf{P} \neq \mathbf{NP}$. Esiste qualcosa che sta in **NP** ma non in **P**? Questi problemi sono detti NP-completi.

Supponiamo di avere due problemi $\pi_1 : I_1 \rightarrow \{0, 1\}$ e $\pi_2 : I_2 \rightarrow \{0, 1\}$.

π_1 è *riconducibile* a π_2 se esiste $f : I_1 \rightarrow I_2$ tale che:

- Per ogni $x \in I_1$ $\pi_1(x) = 1$ se e solo se $\pi_2(f(x)) = 1$
- F è calcolabile in tempo polinomiale (nella lunghezza dell'input) da un algoritmo deterministico

La funzione f è detta *riduzione polinomiale*.

La riduzione trasforma un problema in un problema di un altro tipo, purchè se la risposta a π_1 è 1 anche la risposta a π_2 è 1, in tempo di soluzione polinomiale.

Proprietà fondamentale:

Se $\pi_1 \leq_p \pi_2$ (è riduzione polinomiale) e $\pi_2 \in P$ allora $\pi_1 \in P$.

Ora possiamo definire formalmente i problemi NP-completi:

1. π problema di decisione è **NP-HARD** se per ogni $\pi' \in NP \rightarrow \pi' \leq_p \pi$
2. π è NP-completo se è **NP-HARD** e $\in NP$

Qualcuno è riuscito a dimostrare che SODD non è NP-completo.

Alcuni esempi di problemi NP-completi sono:

- Cammino Hamiltoniano
- SODD3 (clausole grandi solo 3)

- Partizione
- Commesso viaggiatore
- Cammino massimo