Algoritmi e strutture dati

Gabriele Fioco

a.a. 2024-2025

Indice

1 Algoritmi e programmi

1.1 Definizione

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

- Il concetto di passo è relativo sia all'ambito in cui si lavora sia al livello di astrazione.
- Eseguibile vuol dire che l'esecutore può fare i passi.
- Non ambigui: non vengono lasciati gradi di libertà all'esecutore.
- Procedimento che termina: il procedimento deve terminare in un numero finito di passi e dare una risposta. Eccezioni: può non terminare, alcuni algoritmi lavorano in continuazione; algoritmi randomizzati quindi l'esecutore sceglie qualcosa a caso.

Programma Un programma è un insieme ordinato e finito di istruzioni scritte secondo le regole di uno specifico linguaggio di programmazione.

La differenza tra algoritmi e programmi sta nel fatto che l'algoritmo è un'astrazione del procedimento; un programma è l'espressione concreta in un linguaggio.

Sintesi di algoritmi Dato un problema progettare l'algoritmo che lo risolve.
Struttura dati Modo per organizzare i dati.

Un algoritmo dunque è un procedimento per la soluzione di un problema. Le strutture dati sono modi per organizzare dati usati degli algoritmi.

Algoritmica L'algoritmica è il settore dell'informatica che si occupa dello studio di algoritmi. In particolare si occupa di:

- Progettazione degli algoritmi
- Studio delle strutture dati utilizzate
- Analisi dell'efficienza
- Studio delle limitazioni e della complessità dei problemi
- Definizione di nuovi modelli di calcolo

Lo studio degli algoritmi è importante sia per un aspetto pratico - risovlere problemi reali - sia per aspetti teorici. Studiare metodologie utili per capire meglio il problema e formalizzare mediante algoritmi porta a una comprensione maggiore del problema stesso.

1.2 Analisi di algoritmi

Dato un algoritmo $\mathcal{A}$ che risolve un problema $\mathcal{P}$ siamo interessati a valutarne:

- Correttezza: dimostrare che $\mathcal{A}$ risolve $\mathcal{P}$.
- Efficienza: calcolare la quantità di risorse (tempo e spazio) utilizzate da $\mathcal{A}$.

L'analisi può avvenire a posteriori eseguendo il programma e valutandone le prestazioni, oppure a priori durante la fase di progetto. La valutazione a posteriori è meno efficiente; la valutazione a priori permette invece un'analisi in fase di progetto con strumenti matematici, dimostrandone la correttezza e calcolando i tempi di esecuzione.

Indipendentemente da come è scritto l'algoritmo, se la strategia è uguale, cambia il tempo ma la crescita asintotica è la stessa. Quindi nel confronto tra algoritmi ci interessa in che ordine crescono tempo e spazio, a questo scopo si usano le notazioni asintotiche.

2 Modelli di calcolo

2.1 Macchina RAM

Macchina RAM La macchina RAM è un modello di calcolo astratto

La macchina RAM rappresenta le operazioni che possono fare i computer convenzionali. Ha una memoria ad accesso diretto in cui ogni cella può contenere un intero di lunghezza arbitraria.

Studiando gli algoritmi assumeremo di avere a disposizione una macchina ad accesso diretto (RAM) che esegue le seguenti operazioni elementari:

- Accesso a memoria: `load`, `store`
- Operazioni aritmetiche su interi: `+`, `*`, `/`, `-`, …
- Operazioni di confronto: `<`, `>`, $\leq$, …
- Salto

Ognuna di queste operazioni elementari impiega un tempo costante.

2.2 Complessità di algoritmi

Siano:

- $\mathcal{A}$ un algoritmo
- I un'istanza di $\mathcal{A}$
- tempo(I) il tempo impiegato da $\mathcal{A}$ su istanza I

Si studia la crescita di tempo(I) rispetto al crescere della lunghezza dell'input:

- Tempo in funzione della lunghezza dell'input $T : \mathbb{N} \to \mathbb{N}$: si considera il tempo peggiore ,dunque $T(n) = \max\{\text{tempo}(I) : |I| = n\}$
- Tempo medio: media dei tempi utilizzati su input di lunghezza n, pesata rispetto alle probabilità che l'istanza appaia, dunque $T_{\text{avg}}(n) = \sum_{|I|=n} \text{Prob}(I) \cdot \text{tempo}(I)$

2.2.1 Tempi polinomiali ed esponenziali

Un algoritmo si dice:

- Algoritmo polinomiale: lavora in un tempo limitato da un polinomio. Sono algoritmi considerati ragionevoli ed effettivamente utili.
- Algoritmo esponenziale: lavora in un tempo esponenziale. Sono algoritmi considerati impratcabili.

Per alcuni problemi si conoscono soltanto algoritmi polinomiali.

2.3 Criteri di costo

2.3.1 Criterio di costo uniforme

Il criterio di costo uniforme calcola:

- Tempo: ogni istruzione elementare utilizza un'unità di tempo indipendentemente all'ampiezza degi operandi.
- Spazio: ogni variabile elementari utilizza un'unità di spazio indipendentemente dal valore contenuto.

Il criterio è ragionevole quando i valori usati dall'algoritmo sono di grandezza limitata. Se si manipolano quantità arbitrariamente grandi va tenuto conto della lunghezza delle loro rappresentazioni e quindi bisogna usare altri criteri.

2.3.2 Criterio di costo logaritmico

Il criterio di costo logaritmico calcola:

- Tempo: il tempo di ogni istruzione è proporzionale alla lunghezza dei valori coinvolti.
- Spazio: dato dalla lunghezza della rappresentazione del dato.

Un intero n viene rappresentato da $\log_2(n)$ bit. Dunque: n occupa $\log_2(n)$ spazio; una somma $x + y$ costa tempo $\log_2(x) + \log_2(y) = \Theta(\log(x) + \log(y))$.

Il criterio è utile quando si opera su valori arbitrariamente grandi.

2.4 Complessità di problemi

Dato un problema $\mathcal{P}$ ci si chiede la complessità del problema in termini di tempo. Si possono fissare:

- Limitazione superiore: per fissare una limitazione superiore bisogna trovare un algoritmo $\mathcal{A}$ che risolve $\mathcal{P}$ in un tempo $T(n)$, allora il tempo $T(n)$ è sufficiente per risolvere $\mathcal{P}$. Quindi $\mathcal{P}$ è risolubile in un tempo $O(T(n))$.
- Limitazione inferiore: per fissare una limitazione inferiore bisogna dimostra che ogni algoritmo che risolve $\mathcal{P}$ utilizza almeno tempo $T'(n)$, allora il tempo $T'(n)$ è necessario per risolvere $\mathcal{P}$. Quindi $\mathcal{P}$ è risolubile in un tempo $\Omega(T'(n))$.

Le stesse considerazioni valgono per lo spazio.

2.5 Teorema fondamentale delle equazioni ricorrenti

Siano $m, a, b', b'', c \in R^+$ con $a > 1$. Allora l'equazione:

$$F(n) = \begin{cases} b' & \text{se } n = 1 \\ mF(\frac{n}{a}) + b''n^c & \text{se } n > 1 \end{cases}$$

Ha soluzioni:

$$F(n) = \begin{cases} \Theta(n^c) & \text{se } m < a^c \\ \Theta(n^c \log(n)) & \text{se } m = a^c \\ \Theta(n^{\log_a(m)}) & \text{se } m > a^c \end{cases}$$

Il teorema risulta utile per risolvere le equazioni ricorrenti usate nell'analisi di algoritmi ricorsivi. In particolare si nota che:

- m: chiamate ricorsive all'interno della funzione fuori dal caso base;
- b': istruzioni nel caso base;
- b'': istruzioni oltre alle chiamate ricorsive fuori dal caso base;
- $\frac{n}{a}$: lunghezza dell'input passato alle chiamate ricorsive.

2.6 Analisi di costo ammortizzato

L'analisi ammortizzata studia le prestazioni di una sequenza di operazioni su una collezione di dati, piuttosto che la singola esecuzione di un algoritmo.

Costo ammortizzato Il costo ammortizzato T_a di un algoritmo su una sequenza di k operazione è definito come:

$$T_a(n, k) = \frac{T(n, k)}{k}$$

con $T(n, k)$ tempo totale dell'algoritmo, nel caso peggiore, per k operazioni su input di lunghezza n.

2.6.1 Esempio: incrementa contatore binario

Si prenda d'esempio l'operazione di incremento di un contatore binario su n bit:

```
ALGORITMO incrementaContatore(Array v, Intero n)
FOR i <- 0 TO n - 1 DO
    v[i] <- NOT v[i]
    IF v[i] = 1 THEN BREAK
```

L'algoritmo si ferma non appena inverte uno 0. Nel caso peggiore il contatore non contiene nessuno 0, pertanto il costo è $O(n)$. Si analizza ora il costo totale di k operazioni su un contatore da n bit: si osserva che il bit in posizione i viene invertito ogni 2^i operazioni, quindi il costo totale per k operazioni è dato dalla somma del numero di volte che ogni bit i viene invertito:

$$T(n,k) = k + \lfloor \frac{k}{2} \rfloor 2 + \lfloor \frac{k}{2} \rfloor 2^2 + \cdots + \lfloor \frac{k}{2} \rfloor 2^{n-1} = \sum_{i=0}^{n-1} \lfloor \frac{k}{2} \rfloor 2^i \leq \sum_{i=0}^{n-1} \frac{k}{2^i} = 2k$$

con costo ammortizzato:

$$T_a(n,k) = \frac{2k}{k} = 2 = O(1)$$

3 Strutture dati e tipi di dato elementari

Tipo di dato Il tipo di dato è un attributo che specifica l'insieme di valori che una variabile può assumere e le relative operazioni.

Struttura dati Una struttura dati specifica l'organizzazione delle informazioni che permette di realizzaare e implementare un tipo di dati.

La differenza tra tipo e struttura è che il tipo di una variabile specifica cosa un tipo rappresenti, la struttura specifica l'organizzazione dei dati in modo da implementare in modo efficiente le operazioni del tipo. Per lo stesso tipo di dato esistono diverse strutture dati, alcune delle quali sono vantaggiose rispetto alle altre in funzione di cosa bisogna fare.

3.1 Collezioni

Esistono due grandi famigie di strutture dati per rappresentare collezioni di dati: le strutture indicizzate (array), chiamate strutture statiche; le strutture collegate, chiamate strutture dinamiche.

3.1.1 Strutture indicizzate (array)

Array Collezione di elementi dello stesso tipo, ciascuno dei quali accessibile in base alla posizione.

Un array presenta queste caratteristiche:

- Memorizzato in una porizione di memoria continua.
- Accesso con indice.
- Tempo di accesso indipendente dalla posizione del dato (grazie alle prime due caratteristiche).

Si tenga in considerazione che siccome c'è bisogno di una porzione contigua l'array è statico. Per aggiungere un elemento bisogna riallocare l'array, impiegando tempo $\Omega(n)$. Una possibile soluzione, rinunciando all'ordinamento dell'array, è la tecnica di raddoppiamento-dimezzamento che ammortizza i tempi di inserimento/cancellazione a $O(1)$:

Le variabili array sono puntatori all'array.

3.1.2 Strutture collegate

In una struttura collegata i dati non sono rappresentanti in maniera contigua ma in porizioni differenti di memoria. Gli elementi sono collegati tra loro e il passaggio da un elemento all'altro avviene attraverso questi collegamenti. Prendono il nome di strutture dinamiche

3.2 Liste lineari

Lista lineare Una lista lineare è un insieme ordinato di nodi collegati linearmente uno dopo l'altro. Ogni nodo contiene il dato della collezione e l'informazione per accedere al nodo successivo, cioè il puntatore.

3.3 Pile

Il tipo pila è una struttura di tipo LIFO, così specificata:

```
tipo Pila<E>
    isEmpty() -> boolean // restituisce true se la pila è vuote; false altrimenti
    push(E)              // inserisce un nuovo elemento in cima alla pila
    pop() -> E           // rimuove e restituisce l'elemento in cima alla pila
    top() -> E           // restituisce l'elemento in cima alla pila
```

3.3.1 Implementazione tramite array

L'implementazione tramite array presenta due problemi:

- top() e pop() su pila vuota, è giusto che restituiscano errore
- push() potrei avere raggiunto la fine dell'array ma questo è un dettaglio implementativo e non va bene restituire un array. La pila ha una capacità che non dovrebbe avere a causa dell'array

3.3.2 Implementazione tramite strutture collegate

puntatore top al primo elemento della lista. Pila vuota: top -> null

```
FUNZIONE isEmpty() -> boolean
IF top = null
    RETURN true
ELSE
    RETURN false
```

```
FUNZIONE top() -> E
RETURN top.dato
```

```
FUNZIONE pop() -> E
x <- top.dato
top <- top.pros
RETURN x
```

```
PROCEDURA push(E x)
r <- alloca_nodo()
r.dato <- x
r.pros <- top
top <- r
```

Tutte le operazioni sopra implementate sono eseguite con tempo $O(1)$.

3.4 Code

Il tipo coda è una struttura di tipo FIFO, così specificata:

```
tipo Coda<E>
    isEmpty() -> boolean  // restituisce true se la coda è vuota; false altrimenti
    enqueue(E)           // aggiunge un nuovo elemento in fondo alla coda
    dequeue() -> E       // rimuove e restituisce il primo elemento della coda
    first() -> E             // restituisce il primo elemento della coda
```

3.4.1 Implementazione tramite array

L'idea è avere un indice per il primo elemento e un indice per l'ultimo. Se si aggiunge allora si sposta l'ultimo indice in avanti; se si preleva allora si sposta il primo indice in avanti.

Il problema è capacità dell'array: inserendo e prelevando, infatti, la coda si sposta verso destra e quindi si spreca memoria. Di conseguenza bisogna spostare tutto a sinistra, quindi inserire e prelevare ha costo $O(n)$. Si potrebbe trattare l'array come array circolare, ma resterebbe il problema della capacità.

3.4.2 Implementazione tramite strutture collegate

Si tiene puntatore al primo elemento e all'ultimo. coda vuota: primo = ultimo = null

```
FUNZIONE isEmpty() -> boolean
IF primo = null
    RETURN true
ELSE
    RETURN false
```

```
FUNZIONE first() -> E
RETURN primo.dato
```

```
FUNZIONE dequeue() -> E
x <- primo.dato
primo <- primo.pros
IF primo = null THEN // la coda potrebbe essersi svuotata, ma ultimo punta ancora
```

```
    ultimo = null
RETURN x
```

```
PROCEDURA enqueue(E x)
r <- alloca_nodo()
r.dato <- x
r.pros <- null
IF primo = null THEN // coda vuota
    primo <- r
ELSE                     // coda non vuota
    ultimo.pros <- r
ultimo <- r
```

Questa implementazione esegue tutte le operazioni in tempo costante.

4 Alberi

Albero Un albero è una collezione non vuota di:

- Nodi con nome.
- Lati che collegano tra loro due nodi.

Cammino Un cammino è una sequenza di nodi collegati tra loro. In un albero vi è esattamente un cammino tra due nodi (altrimenti sarebbe un grafo).

4.1 Alberi con radice

Albero con radice Un albero con radice si definisce induttivamente come:

- Base: una struttura vuota.
- Passo: una radice a cui sono associati $k \geq 0$ nodi.

Un albero con radice è caratterizzato dai seguenti elementi:

- Radice: l'inizio dell'albero.
- Foglie: nodi senza figli.
- Nodi interni: nodi con almeno un figlio.

Profondità Si definisce profondità di un nodo, induttivamente:

- Radice: profondità 0
- Figli di un nodo di profondità k: $k + 1$.

Altezza Si definisce altezza di un albero la massima profondità raggiunta dalle sue foglie.

Negli alberi con radice ogni nodo ha un numero arbitrario di figli, da definizione vi è una radice e un certo numero di sottoalberi.

Grado di un nodo Si definiscei grado di un nodo il numero di figli.
Grado dell'albero Si definisce grado dell'albero il max grado dei nodi.

Figura 4.1: Un albero generico di grado 3

4.1.1 Rappresentazione

Si analizzano diversi modi per rappresentare alberi con radice.

Vettori dei padri Si tiene un array con i valori dei nodi e uno con la posizione del padre di ciascun nodo.

Vettore dei figli Si tiene array con i valori dei nodi e k array contenenti la posizione dei figli, con k grado dell'albero.

Puntatori ai figli Analogamente agli alberi binari, dato il grado d di un albero, ogni nodo contiene d puntatori ai figli. Si osserva che bisogna conoscere il grado fin dall'inizio e che alcuni nodi potrebbero non avere d figli, occupando memoria inutilmente.

Figura 4.2: Albero in figura 4.1 rappresentato con puntatori ai figli

Lista dei fratelli Ogni nodo presenta due puntatori, quello sinistra punta al figlio, quello a destra a una lista dei fratelli. Quindi è come un albero binario con a sinistra i figli e a destra i fratelli.

Figura 4.3: Albero in figura 4.1 rappresentato con lista dei fratelli

4.2 Alberi binari

Alberi binari Un albero binario è un albero con radice a cui ad ogni nodo possono esere associati al più due successori detti figlio destro e figlio sinistro.

Albero binario completo Un albero binario è detto completo se e solo se ogni nodo ha entrambi i figli.

Albero binario quasi completo Un albero binario di altezza h è detto quasi completo se e solo se ogni nodo di profondità $h-1$ possiede entrambi i figli. Ovvero se è completo almeno fino al penultimo livello.

Il modo più semplice per rappresentare un albero binario è tramite liste collegate ma con due puntatori.

Figura 4.4: Un albero binario con radice 8

4.3 Attraversamento di alberi binari

Si danno ora algoritmi per visitare alberi binari. Lo schema generico funziona come segue: si procede creando un insieme di nodi `S` partendo dalla radice. Successivamente l'algoritmo preleva un nodo `v` da `S`, lo visita, e mette dentro `S` tutti i figli di `v`.

```
// r è il nodo radice
ALGORITMO visitiGenerica(AlberoBinario r)
Insieme S
S <- {r}
WHILE S != empty DO
```

```
    preleva un nodo v da S
    visita v
    S <- S unione {figli di v}
```

4.3.1 Visita in ampiezza (BFS)

La visita in ampiezza (o BFS) utilizza una coda per tenere i nodi. In quanto struttura FIFO la visita dell'albero avviene in orizzontale.

```
// r è il puntatore alla radice
ALGORITMO visitaAmpiezza(AlberoBinario r)
Coda c
c.enqueue(r)
WHILE NOT (c.isEmpty()) DO
    n <- c.dequeue()
    IF n != null THEN
        visita n
        c.enqueue(n.dx)
        c.enqueue(n.sx)
```

Dati n nodi il tempo dell'algoritmo è $O(n)$.

La visita dei nodi nell'albero in figura 4.4 è, in ordine: 8 14 1 22 12 5 1 3

4.3.2 Visita in profondità (DFS)

La visita in profondità (o DFS) scende il più possibile verso un ramo, poi ritorna indietro a passi. Si utilizza utilizza una pila.

```
ALGORITMO visitaProfondita(AlberoBinario r)
Pila p
p.push(r)
WHILE NOT (p.isEmpty()) DO
    n <- c.pop()
    IF n != null THEN
        visita n
        c.push(n.sx)
        c.push(n.dx)
```

Si osserva che un albero binario è una struttura ricorsiva: o è vuoto o è un nodo con due sottoalberi. Quindi si visita in profondità ricorsivamente. Vi sono tre versioni ricorsive. Negli esempi si fa riferimento all'albero in figura 4.4:

- Visita in ordine anticipato: visita prima la radice, poi sottoalbero sinistra e destro. Nell'esempio, in ordine: $8, 14, 22, 12, 1, 3, 1, 5$.
- Visita simmetrica: visita prima il sottoalbero sinistro, poi la radice e infine il sottoalbero destro. Nell'esempio, in ordine: $22, 14, 1, 12, 3, 8, 1, 5$.
- Visita in ordine posticipato: visita prima il sottoalbero sinistro, poi quello destro e per ultima la radice. Nell'esempio, in ordine: $22, 1, 3, 12, 14, 5, 1, 8$.

```
ALGORITMO visitaAnticipata(AlberoBinario r)
IF r != null THEN
    visita r
    visitaAnticipata(r.sx)
    visitaAnticipata(r.dx)

ALGORITMO visitaSimmetrica(AlberoBinario r)
IF r != null THEN
    visitaSimmetrica(r.sx)
    visita r
    visitaSimmetrica(r.dx)

ALGORITMO visitaPosticipata(AlberoBinario r)
IF r != null THEN
    visitaPosticipata(r.sx)
    visitaPosticipata(r.dx)
    visita r
```

TIP: In genere con delle strutture ricorsive è bnene utilizzare algoritmi ricorsivi. Qua vi è uno stack ricorsivo ma nell'algoritmo iterativo vi è uno stack esplicito.

4.4 Proprietà numeriche degli alberi binari

Si da un albero binario di altezza h con n nodi, ci si chiede la relazione tra h e n:

- Numero minimo di nodi m: si ottengono quando ogni nodo ha un solo figlio, ovvero l'albero è una lista, quindi $m = h - 1$.
- Numero massimo di nodi M: si ottengono quando ogni nodo ha entrambi i figli, cioè quando i è un albero binario completo. Si osserva che in questo caso a un generico livello d sono presenti 2^d nodi, quindi con altezza h si ottiene:

$$M = \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$$

Quindi si conclude:

$$\begin{aligned} & m \leq n < M \\ & h - 1 < n < 2^{h+1} - 1 \\ & h + 1 < n < 2^{h+1} \\ & \log_2(n) < h < n \end{aligned}$$

Negli alberi binari quasi completi, per calcolare il numero minimo di nodi, si osserva che fino al livello $h - 1$ vi sono tutti i nodi, mentre vi è almeno un nodo di altezza h, quindi:

$$m = \sum_{i=0}^{h-1} 2^h + 1 = 2^h - 1 + 1 = 2^h$$

Negli alberi binari quasi completi, quindi vale $2^h \leq n < 2^{h+1}$ quindi $h = \lfloor \log_2 n \rfloor$

Da questi calcoli si ottiene una proprietà molto importante degli alberi binari quasi completa: la loro altezza è logaritmica rispetto al numero di nodi.

5 Heap

Heap Uno heap (o max-heap) è un albero binario quasi completo in cui la chiave contenuta in ciascun nodo è maggiore o uguale alle chiavi dei figli.

Figura 5.1: Un max heap

Dalla definizione segue che la la radice di uno heap contiene il valore più grande.

Per semplicità si considerano solo heap con foglie dell'ultimo livello più a sinistra possibile.

> ***FUN FACT:*** "Heap" è tradotto come mucchio che richiama il fatto che un mucchio diventa sempre più piccolo salendo (per quanto i max-heap hanno la caratteristica contraria).

5.1 Risistemare uno heap

Dato uno heap, è possibile costruire un array ordinato prelevando la radice. Successivamente verrebbe naturale far salire il figlio più pesante della radice al posto della stessa, ma la struttura ottenuta non necessariamente è un albero quasi completo e dunque uno heap.

La soluzione è, una volta prelevata la radice, sostituirla con la foglia più a destra, ottenendo uno heap sbagliato con la sola radice da risistemare. La procedura `risistema(Albero)` sistema uno heap con la sola radice sbagliata. La strategia consiste nel far scendere la radice verso il basso, scambiandola man mano col figlio più grosso.

```
PROCEDURA risistema(Heap H)
v <- H
x <- v.chiave
y <- v.campi
da_collocare <- true
WHILE da_collocare DO
    IF v è una foglia THEN
        da_collocare <- false
    ELSE
        u <- figlio di v di valore massimo
        IF u.chiave > x THEN
            v.chiave <- u.chiave
            v.campi <- u.campi
            v <- u
        ELSE
            da_collocare <- false
v.chiave <- x
v.campi <- y
```

Nel caso peggiore la radice scende all'ultimo livello. Poichè si fanno 2 confronti per figlio, in un albero alto h il numero di confronti è $\Theta(h)$. Ricordando che in un albero binario quasi completo con n nodi l'altezza è massimo $h = \lfloor \log_2(h) \rfloor$, si conclude che il numero di confronti è $\Theta(\log(n))$ con n numero di nodi.

5.2 Costruzione di uno heap

Dato un albero binario quasi completo si vogliono riordinare i nodi in modo tale che questo rispetti la definizione di heap. Seguono due strategie.

5.2.1 Soluzione con tecnica divi-et-impera

La ricorsione è data da due casi:

- Se l'albero è vuoto allora è già uno heap.
- Se l'albero non è vuoto allora si trasformano ricorsivamente i due sottoalberi in heap; a questo punto l'unico elemento fuori posto è la radice quindi si usa risistema.

```
PROCEDURA creaHeap(Albero T)
IF T != albero vuoto THEN
    creaHeap(T.sx)
    creapHeap(T.dx)
    risistema(T)
```

Senza ulteriori analisi si lascia perdere perchè usa lo stack e quindi memoria aggiuntiva.

5.2.2 Soluzione iterativa

Si applica `risistema()` ad ogni nodo visitato dal basso, partendo dalla foglia più a destra. Poichè si parte dal basso, quando si visita un certo nodo di profondità `p` siamo sicuri che solo il suo sottoalbero sia sbagliata in quanto i sottoalberi inferiori sono già stati sistemati.

```
PROCEDURA creaHeap(Albero T)
h <- altezza di T
FOR p <- h DOWNTO 0 DO
    FOREACH nodo x di profondità p DO
        risistema(x)
```

Analisi Per calcolare il caso peggiore si considera un albero binario completo di altezza h. Prima di tutto si osserva che un sottoalbero con radice di profondità p ha altezza $h - p$, quindi la procedura risistema effettua al più $h - p$ confronti. Successivamente si osserva che a profondità p sono presenti 2^p nodi, dunque per il livelo p l'algoritmo effettua $2^p(h - p)$ confronti. Ricordando che $h = \lfloor \log_2(n) \rfloor$, il numero di confronti totali è così calcolato:

$$C(h) = \sum_{p=0}^{h}(h-p)2^p =$$

$$\begin{aligned}
\sum_{p=0}^{h} h2^p - \sum_{p=0}^{h} p2^p &= \\
h\sum_{p=0}^{h} 2^p - \sum_{p=0}^{h} p2^p &= \\
h(2^{h+1}-1) - (h-1)2^{h+1} - 2 &= \\
h2^{h+1} - h - h^{2h+1} + 2^{h+1} - 2 = \Theta(n)
\end{aligned}$$

5.3 Operazioni su heap

Si descrivono una serie di operazioni sugli heap.

Trovare l'elemento di chiave massima Trovare l'elemento di chiave massima è semplice, questo si trova infatti sulla radice. Il tempo è costante.

Cancellare l'elemento di chiave massima Si pone l'ultima foglia a destra come radice radice e si chiama `risistema()` sullo heap. Il tempo è $\Theta(\log n)$.

Inserimento Si inserisce l'elemento come foglia nella posizione più a sinistra disponibile nell'ultimo livello (o su un nuovo livello se l'ultimo è pieno). A questo punto si risistema lo heap dal basso con una procedura `risistemaFoglia()` che similmente a `risistema()` mette a posto lo heap supponendo l'ultima foglia come nodo errato. L'algoritmo `risistemaFoglia()` controlla il padre della foglia e si scambia se è minore, così via fino a che la foglia non sarà nella posizione corretta. Nel caso peggiore si sale fino alla radice, quindi si ha tempo $\Theta(\log(n))$.

Cancellare un elemento data la sua chiave Si sostituisce l'elemento da cancellare con l'ultima foglia a destra. Siano f la chiave della foglia e x la chiave dell'elemento da cancellare, allora possono verificarsi due casi:

- $f < x$: f potrebbe essere minore della chiave dei figli di x, quindi si ha un sottoalbero heap con radice sbagliata e si chiama `risistema` su f
- $f > x$: f potrebbe essere maggiore della chiave del padre, quindi si ha un sottoalbero heap con foglia sbagliata e si chiama chiama `risistemaFoglia` su f.

In entrambi i casi il tempo è $\Theta(\log n)$.

Modificare la chiave di un elemento data la sua chiave attuale Sia f la nuova chiave e x la chiave attuale, possono verificarsi due casi analoghi alla cancellazione:

- $f < x$: si chiama `risistema` su f.
- $f > x$: si chiama`risistemaFoglia` su f.

In entrambi i casi il tempo è $\Theta(\log n)$.

6 Code con priorità

Le code con priorità sono code FIFO i cui elementi hanno assegnata una chiave indicante la priorità, e vengono prelevati secondo uesta priorità. Solitamente chiavi inferiori indicano

priorità maggiori.

Il tipo con le relative operazioni è così descritto:

```
Tipo Coda_con_priorità<E>

Operazioni:

findMin()                       // restituisce l'elemento minimo della coda senza rimuoverlo
deleteMin() -> E                        // restituisce l'elemento minimo della coda e lo rimuove
insert(E e, Chiave k)           // inserisce nella coda l'elemento 'e' con associata priorità 'k
delete(E e)                     // cancella l'elemento 'e'
changeKey(E e, Chiave d)            // imposta la priorità dell'elemento 'e' come 'd'
```

Le code con priorità possono essere implementate con min-heap. Utilizzando le operazioni sugli heap i tempi sono i seguenti:

- `findMin()`: $O(1)$
- `deleteMin()` $\Theta(\log n)$
- `insert()` $\Theta(\log n)$
- `delete()` $\Theta(\log n)$ supponendo di sapere la posizione dell'elemento
- `changeKey()` $\Theta(\log n)$ supponendo di sapere la posizione dell'elemento

Per evitare di cercare la posizione degli elementi si potrebbe tenere una struttura ausiliaria che dato un elemento ne restituisce la posizione nello heap.

Esistono implementazioni più efficienti basati su heap detti heap di Fibonacci.

Si osserva che vi sono due gradi di astrazione: quello delle code implementate con min-heap; quello del min-heap.

> ***NOTA:*** guai a chi dice "coda **di** priorità". Se vai in un supermercato in coda in cassa diventi un numero o ti viene assegnato???

7 Tecniche algoritmiche

7.1 dividi-et-impera

Siano $\mathcal{P}$ un problema e $\mathcal{I}$ un'istanza di $\mathcal{P}$. Con la tecnica dividi-et-impera si vuole dividere $\mathcal{I}$ in istanze di minore lunghezza, risolverle separatamente e combinarle.

Si da lo schema generale:

```
ALGORITMO risolviP(Istanza I) -> Soluzione
IF |I| <= C THEN    // caso base in cui si risolve P direttamente
    sol <- risolvi P su I direttamente
    RETURN sol
ELSE            // si divide in problemi più piccoli e si risolvono
    dividi I in I_1, ..., I_M t.c. |I_j| < |I| i=1..n
    sol_1 <- I_1
    ...
    sol_M <- I_M
    RETURN combina(sol_1, ..., sol_M)
```

Il tempo è cosi calcolato:

$$T(I) = \begin{cases} \text{costante} & \text{se } |I| \leq C \\ T_{\text{dividi}} + T(I_1) + ... + T(I_M) + T_{\text{combina}}(\text{sol}_1, ... \text{sol}_M) & \text{altrimenti} \end{cases}$$

7.1.1 Esempio: minimo e massimo di un array

L'algoritmo banale per trovare il minimo e il massimo di un array è quello sequenziale:

```
ALGORITMO minMax(Array A[0..n-1]) -> (Elemento, Elemento)
min <- A[0]
max <- A[0]
FOR i <- 1 TO n - 1
    IF A[i] < min THEN min <- A[i]
    IF A[i] > max THEN max <- A[i]
RETURN (min, max)
```

Questo algoritmo esegue un numero di confronti pari a $2n - 2$

Si da ora una versione basata su tecnica divid et impera:

```
FUNZIONE minMax(Array A[0..n-1], Indice i, Indice f) -> (Elemento, Elemento)
IF f - i = 1 THEN
    RETURN (A[i], A[i])
ELSE IF f - i = 2 THEN
    IF A[i] < A[f] THEN
        RETURN (A[i], A[f])
    ELSE
        RETURN (A[f], A[i])
ELSE
    m <- (i + f) / 2
    (min1, max1) <- minMax(A, i, m)
    (min2, max2) <- minMax(A, m, f)

    IF min1 < min2 THEN
        min <- min1
    ELSE
        min <- min2

    IF max1 > max2 THEN
        max <- max1
    ELSE
        max <- max2

    RETURN (min, max)
```

Il numero di confronti è dato da:

$$C(n) = \begin{cases} 0 & \text{se } n = 1 \\ 1 & \text{se } n = 2 \\ C(\lfloor \frac{n}{2} \rfloor) + C(\lfloor \frac{n}{2} \rfloor) + 2 & \text{altrimenti} \end{cases}$$

Calcolando si ottiene, per n potenza di 2: $C(n) = \frac{3}{2}n - 2$.

7.2 greedy

Problema di ottimizzazione Tra tutte le soluzioni possibili, ovvero che soddisfano dei vincoli, si vuole determinare una soluzione ottima secondo un certo criterio (solitamente minimizzando o massimizzando).

La strategia greedy permette di risolvere problemi di ottimizzazione, definiti dei vincoli che rendono la soluzione ammissibile e dei vincoli che la rendono ottimale. Dato un insieme `S` di candidati selezionati, questo inizialmente è vuoto. Si procede iterando su `C`: ad ogni iterazione si seleziona il miglior candidato `x` da `C` e lo si aggiunge all'insieme `S`; se l'isieme `S` non è ammissibile allora l'elemento `x` è scartato; viceversa si mette definitivamente `x` in `S`. Si continua finchè una soluzione ottima non è stata trovata o tutti i candidati sono stati esaminati.

```
ALGORITMO strategiaGreedy(Candidati C) -> Soluzione
Insieme S
WHILE NOT C.isEmpty() DO
    // preleva elemento migliore da C
    x <- seleziona(C)
    C <- C - {x}
    IF S + {x} è ammissibile THEN
        S <- S + {x}
    IF ottimo(S) THEN RETURN S
RETURN S
```

7.2.1 Esempio: zaino monodimensionale

Dato uno zaino alto h e k contenitori $c_1, \cdots, c_k$ di altezza $h_1, \cdots, h_k$ si vuole scegliere quali contenitori inserire in modo da riempire lo zaino il più possibile

Una soluzione è:

- Ammissibile: $S \subseteq \{c_1, \cdots, c_k\}$ è ammissibile se $f(S) \leq h$ con $f(S) = \sum_{c_i \in S} h_i$
- Ottimale: $S* \subseteq \{c_1, \cdots, c_k\}$ ammissibile è ottimale se $f(S*) \geq f(S)$ per ogni S ammissibile.

Applicando la strategia greedy, i candidati sono $C = \{c_1, \dots, c_k\}$. Si ispezionano i contenitori in ordine di altezza dal più alto al più basso e se stanno nello zaino si mettono, altrimenti si scartano. Non trova sempre una soluzione ottima.

7.2.2 Esempio: zaino con valori

Dato un sacco di peso massimo P e k oggetti $C = \{c_1, \dots, c_k\}$ di peso $p_1, \dots, p_k$ e valore $v_1, \dots, v_k$, scegliere gli oggetti da collocare nel sacco in modo da massimizzare il valore totale

senza eccedere il peso.

Una soluzione è:

- Ammissibile: $S \subset C$ è ammissibile se peso$(S) \leq P$
- Ottimale: $S^\star \subseteq C$ è ottimale se $S^\star$ è ammissibile e $\forall S \subseteq$ ammissibile allora val$(S^\star) \geq$ val(S).

La strategia greedy esamina gli oggetti in ordine di valore e li aggiunge al sacco se e solo il peso totale, aggiungendoli, non eccede il peso P. Non trova sempre una soluzione ottima.

7.3 Programmazione dinamica

La programmazione dinamica è una tecnica che risolve un problema partendo dalle istanze più semplici e risolvendo i problemi più grossi. In generale, i passi sono:

1. Si individuano sottoproblemi del problema dato, i cui risultati vanno salvati in una tabella
2. Si definiscono i valori iniziali della tabella
3. Al generico passo si risolve il problema consultando le soluzioni dei sottoproblemi salvate nella tabella
4. La soluzione finale si ottiene dalla tabella stessa oppure componendo i risultati.

Principio di ottimalità La programmazione dinamica utilizza il principio di ottimalità, che si enuncia: una soluzione ottima è data dalle combinazioni delle sottosoluzioni ottime dei problemi più piccoli.

La tecnica divid-et-impera risulta disastrosa in termini di tempo perchè alcuni valori potrebbero essere ripetutamenta calcolati.

> ***NOTA:*** si chiama programmazione dinamica perchè riempire le tabella veniva chiamato "riempire le tabelle", il riempimento avviene dinamicamente alle soluzioni.

7.3.1 Esempio: fibonacci

```
ALGORITMO Fibonacci(Intero n) -> Intero
Array Fib[0..n+1]
Fib[0] <- 1
Fib[1] <- 1
FOR i <- 2 TO n DO
    Fib[i] <- Fib[i - 1] + Fib[i - 2]
RETURN Fib[n]
```

7.3.2 Esempio: distanza tra stringhe

Siano x e y due stringhe allora la distanza $d(x, y)$ tra le due è il numero minimo di operazioni che permettono di passare da x a y. Le operazioni permesse, a livello di carattere, sono: sostituzione; inserimento; cancellazione.

Il caso facile è quando una delle due stringhe è vuota, in questo caso la distanza sarà $n = |x|$ o $n = |y|$ con n inserimenti o n cancellazioni.

Si ipotizzi di usare un tecnica dividi et impera. Allora si passa da casi difficili a casi facili, ovvero si riconduce il problema del calcolo di $d(x,y)$ al calcolo di distanze più corte. Date due stringhe $x = x_1...x_m$ e $y = y_1...y_n$ allora un algoritmo ricorsivo calcolerebbe $d(x,y)$ come segue:

$$d(x,y) = \begin{cases} m & n = 0 \\ n & m = 0 \\ d(x_1...x_m, y_1...y_n) & n \neq 0, m \neq 0, x_m = y_n \\ 1 + \min\{d(x_1...x_{m-1}, y_1...y_{n-1}), d(x_1...x_{m-1}, y_1...y_n\}, d(x_1...x_m, y_1...y_{n-1}) \end{cases}$$

Nel caso peggiore $T(n,m) \geq 3^n$ quindi la soluzione non è ottimale. Il problema principale è dato dal fatto che molti calcoli vengono eeguiti più volte, come fibonacci.

Si usa ora una soluzione con programmazione dinamica. Sia D una matrice $(m+1) \times (n+1)$, si definisce $D[i,j] = d(x_1...x_i, y_1...y_j)$. Allora:

$$d[i,j] = \begin{cases} i & j = 0 \\ j & i = 0 \\ d[i-1, j-1] & x_i = y_j \\ 1 + \min\{d[i-1,j-1], d[i-1,j], d[i,j-1]\} & x_i \neq y_j \end{cases}$$

L'algoritmo è così dato:

```
ALGORITMO distanza(Stringa x[0..m], Stringa y[0..n]) -> Intero
Matrice_interi D[0..m, 0..n]
FOR i <- 0 TO m - 1 DO D[i, 0] <- i
FOR j <- 0 TO n - 1 DO D[0, j] <- j

FOR i <- 1 TO m - 1 DO
    FOR j <- 1 TO m - 1 DO
        IF x[i] = y[j] THEN
            D[i, j] = D[i - 1, j - 1]
        ELSE
            t <- D[i - 1, j - 1]
            IF D[i - 1, j] < t THEN t <- D[i - 1, j]
            IF D[i, j - 1] < t THEN t <- D[i, j - 1]
            D[i ,j] <- t
RETURN D[m, n]
```

Il tempo quindi è $\Theta(n \cdot m)$.

8 Algoritmi di ricerca

Ricerca in un array:

- Input: Array `A`, elemento `x`
- Output: indice `i` se esiste `i` t.c. `A[i] = x`, `-1` se non esiste `i` t.c. `A[i] = x`

8.1 Ricerca sequenziale

L'unico modo per cercare un elemento in un array è la ricerca sequenziale.

```
ALGORITMO ricercaSequenziale(Array A[0..n-1], elemento x) -> indice
i <- 0
WHILE i < n AND A[i] != x DO
    i <- i + 1
IF i = n THEN RETURN -1
ELSE RETURN i
```

Osservazione: ricercando dal fondo si può evitare la selezione finale

```
ALGORITMO ricercaSequenziale(Array A[0..n-1], elemento x) -> indice
i <- n - 1
WHILE i >= 0 AND A[i] != x DO
    i <- i - 1
RETURN i
```

8.2 Ricerca binaria

Se l'array è ordinato si può applicare una ricerca binaria (o dicotomica).

8.2.1 Algoritmo ricorsivo

Si da l'algoritmo ricorsivo:

```
ALGORITMO ricercaDicotomicaRicorsiva(Array A[0..n-1], elemento x) -> indice
    return ricercaRicorsiva(A, 0, n, x)

FUNZIONE ricercaRicorsiva(Array A, indice sx, indice dx, elemento x) -> indice
IF dx <= sx THEN return -1
ELSE
    m <- (sx + dx) / 2
    IF x = A[m] THEN
        RETURN m
    ELSE IF x < A[m] THEN
        RETURN ricercaRicorsiva(A[0..n-1], sx, m, x)
    ELSE
        RETURN ricercaRicorsiva(A[0..n-1], m+1, dx, x)
```

Per calcolare le prestazioni si osserva che il caso base si ottiene per 1 o 0 elementi. Alla `i`-esima chiamata considero $\frac{n}{2^{n-i}}$ elementi, per arrivare a un solo elemento si risolve $\frac{n}{2^{n-i}} = 1 \iff i = 1 + \log_2(n)$. Dunque per arrivare al caso base si fanno $i = 2 + \log_2(n)$ chiamate da cui consegue che l'altezza della pila è $\Theta(\log(n))$. Ogni chiamata ha tempo costante e spazio costante, quindi si ottengono le seguenti prestazioni:

- Tempo $\Theta(\log(n))$
- Spazio $\Theta(\log(n))$

8.2.2 Algoritmo iterativo

La ricorsione implementata è detta ricorsione in coda (o terminale) e quindi la trasformazione in iterazione è banale. Si da l'algoritmo iterativo:

```
ALGORITMO ricercaBinariaIterativa(Array A[0..n-1], elemento x) -> indice
    sx <- 0
    dx <- n
    pos <- -1
    WHILE sx < dx AND pos = -1 DO
        m <- (sx + dx) / 2
        IF A[m] = x THEN
            pos <- m
        ELSE IF x < A[m] THEN
            dx <- m
        ELSE
            sx <- m + 1
    RETURN pos
```

L'algoritmo iterativo ha prestazioni:

- Tempo $\Theta(\log(n))$
- Spazio $O(1)$

9 Algoritmi di ordinamento

Il problema dell'ordinamento è così formalizzato:

- Input: n elementi $x_1, \cdots, x_n$ provenienti da un dominio su cui è definita una relazione di ordine totale denotata con $\leq$
- Output: sequenza $x_{j_1}, \cdot x_{j_n}$ con $(j_1, .., j_n)$ permutazione di $(1, ..., n)$ tale che $x_{j_1} \leq \cdots \leq x_{j_n}$

Stabilità Un algoritmo di ordinamento si dice stabile quando $x_{j'_k} = x_{j''_k}, j_{k'} \leq j_{k''} \implies k' \leq k''$. Ovvero date due strutture con la medesima chiave queste preservano il loro ordine relativo all'interno del vettore.

Si distinguono due tipi di ordinamento:

- Ordinamento interno: i dati da ordinare si trovano in memoria centrale. L'accesso al dato è costante.
- Ordinamento esterno: i dati da ordinare in memoria di massa. L'accesso al dato non è costante, tuttavia non si estrae un solo elemento ma dei blocchi.

In questo capitolo si studiano algoritmi di ordinamento interno.

Le risorse sono calcolate come segue:

- Spazio: calcolato come la memoria aggiuntiva utilizzata oltre all'array da ordinare, includendo quella per la ricorsione sullo stack.
- Tempo: calcolato come il numero di confronti tra chiavi moltiplicato per il tempo utilizzato per ciascun confronto. Questo perchè il confronto è l'operazione più costosa.

Gli algoritmi basati sui confronti studiati sono i seguenti, classificati:

- Algoritmi elementari: sono semplici da implementare ma non hanno prestazioni elevate. Lavorano in numero di confronti limitato da $\Theta(n^2)$, sempre raggiunto nel caso peggiore.
  - `selectionSort`
  - `insertionSort`
  - `bubbleSort`
- Algoritmi avanzati: usano strategie più complicate ma hanno numero di confronti sempre pari a $\Theta(n \log n)$:
  - `mergeSort`
  - `quickSort`: unica eccezione nelle prestazioni, nel caso peggiore raggiunge un tempo $\Theta(n^2)$
  - `heapSort`

Gli algoritmi considerati sono tutti eseguiti in loco, ovvero non necessitano strutture ausiliare eccetto mergeSort e quicksort. Non sono tutti stabili.

9.1 Dimostrazione limite inferiore per algoritmi basati sui confronti

Si dimostra che ogni algoritmo di ordinamento basato sui confronti utilizza almeno $n \log(n)$ confronti.

Si costruisce un albero di decisione dell'algoritmo così composto:

- Nodi interni: ci si chiede se un elemento è minore o uguale all'altro.
- Foglie: l'array ordinato.

L'algoritmo di ordinamento fa domande basate su quest'albero di decisione.

Si osserva che il numero di foglie f è pari al numero di permutazioni possibili quindi $n!$. Nel caso peggiore un algoritmo effettua un numero di confronti pari all'altezza h dell'albero. Si conclude quindi che ogni algoritmo utilizza sempre un numero di confronti pari all'altezza h minima dell'albero binario con $n!$ foglie.

Sappiamo che $h \geq \log_2(\text{numero\_nodi}) \geq log_2(numero\_foglie)$ e quindi $C(n) \geq log_2(n!)$.

Dalla formula di Starling si sa che $n! \approx \sqrt{2\pi n}(\frac{n}{e})^n$, quindi si calcola:

$$
\begin{aligned}
&C(n) \geq log_2(n!) \approx log_2(\sqrt{2\pi n}(\frac{n}{e})^n) = log_2(\sqrt{2\pi n}) + \log_2(\frac{n}{e})^n \\
&= \frac{1}{2}\log_2(2\pi) + \frac{1}{2}\log_2(n) + n\log_2(n) - n\log_2 e \\
&= \Theta(n\log(n))
\end{aligned}
$$

Segue quindi $C(n) = \Omega(n \log n)$.

9.2 `selectionSort`

9.2.1 Strategia

Si scorre l'array e si seleziona l'elemento minimo, che si sposta nella parte ordinata dell'array. Al passo `k` dell'algoritmo si trova l'elemento minore e lo si mette nella posizione `k`-esima corretta, scambiandolo con l'elemento nella posizione `k`.

9.2.2 Implementazione

```
ALGORITMO selectionSort(Array A[0..n-1])
FOR k <- 0 TO n - 1 DO
    min <- k
    FOR j <- k + 1 TO n - 1 DO
        IF A[j] < A[min] THEN
            min <- j
    A[m], A[k] <- A[k], A[m]
```

Il ciclo esterno può arrivare a `n - 2`:

```
ALGORITMO selectionSort(Array A[0..n-1])
FOR k <- 0 TO n - 2 DO
    min <- k
    FOR j <- k + 1 TO n - 1 DO
        IF A[j] < A[min] THEN
            min <- j
    A[m], A[k] <- A[k], A[m]
```

Questa implementazione non è stabile.

9.2.3 Analisi

Alla k-esima iterazione si effettuano $n-k-1$ confronti. Il numero di confronti totali è pari a:

$$\begin{aligned} C(n) &= \sum_{k=0}^{n-2}(n-k-1) = \sum_{i=1}^{n-1} i \\ &= \frac{(n-1)\cdot n}{2} \\ &= \Theta(n^2) \end{aligned}$$

sostituendo $i = (n-k-1)$

9.3 `insertionSort`

9.3.1 Strategia

Dato l'array `A[0..n-1]`, alla `k`-esima iterazione si inserisce l'elemento `A[k]` nella sua posizione corretta. Questo avviene tenendo da parte `A[k]` e spostando tutti gli elementi a destra fino a quando si trova un numero minore; a quel punto si insersice `A[k]`.

Visualizzazione grafica di `insertionSort`: `https://it.wikipedia.org/wiki/File:Insertion-sort-example`

9.3.2 Implementazione

```
ALGORITMO insertionSort(Array A[0..n-1]) -> Array
FOR k <- 1 TO n - 1 DO
    x <- A[k] // elemento da inserire
```

```
    // ricerca da destra la posizione incui inserire x
    // spostando in avanti gli elementi maggiori
    i <- k - 1
    WHILE i >= 0 AND A[i] > x DO
        A[i + 1] <- A[i]
        i <- i - 1
    A[i + 1] <- x
```

L'algoritmo è stabile.

9.3.3 Analisi

Nel caso peggiore all'iterazione k si effettuano k confronti, quindi il numero di confronti totali nel caso peggiore è:

$$\sum_{k=1}^{n-1} k = \frac{(n-1)\cdot n}{2} = \Theta(n^2)$$

.

Nel caso migliore è:

$$\sum_{k=1}^{n-1} 1 = n-1$$

9.4 bubbleSort

9.4.1 Strategia

Ad ogni passata si porta l'elemento più grande alla fine.

> ***FUN FACT:***: Il nome "bubble" richiama le bolle nell'aria pesanti che vanno verso l'alto.

9.4.2 Implementazione

```
ALGORITMO bubbleSort(Array A[0..n-1])
DO
    scambiato <- false
    FOR j <- 1 TO n - 1 DO
        IF A[j] < A[j - 1] THEN
            A[j], A[j - 1] <- A[j - 1], A[j]
            scambiato <- true
WHILE scambiato AND i < n
```

Alla posizione `i` si trova già il numero più grande alla fine, quindi si può evitare di scorrere da `i` in poi:

```
ALGORITMO bubbleSort(Array A[0..n-1])
i <- 1
DO
    scambiato <- false
    FOR j <- 1 TO n - i DO
        IF A[j] < A[j - 1] THEN
            A[j], A[j - 1] <- A[j - 1], A[j]
            scambiato <- true
    i <- i + 1
WHILE scambiato AND i < n
```

9.4.3 Analisi

Nel caso migliore, se l'array è già ordinato, si effettuano $n-1$ confronti. Nel caso peggiore il numero di confronti è:

$$C(n) = \sum_{i=1}^{n-1}(n-i) = \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2} = \Theta(n^2)$$

sostituendo $k = (n-i)$.

9.5 Algoritmo di fusione

Dati due array `B` e `C` ordinati si vuole ottenere l'array ordinato `X` fondendo i primi due.

9.5.1 Strategia

Si ispezionano `B` e `C` da sinistra, salvando per ognuno rispettivamente gli indici `i_1` e `i_2`. Ad ogni passo si mette in `X` il mino tra `B[i_1]` e `C[i_2]` incrementando il rispettivo indice. Quando si finisce di ispezionare uno dei due array si procede inserendo tutti i rimanenti elementi dell'array non finito di ispezionare.

9.5.2 Implementazione

L'implementazione della strategia sopra descritta è la seguente:

```
ALGORITMO merge(Array B[0..l_b], Array C[0..l_c]) -> Array
Array X[0..l_b + l_c - 1]
i_1 <- 0
i_2 <- 0
k <- 0
WHILE i_1 < l_b AND i_2 < l_c // finchè nessuno dei due array è finito
    IF B[i_1] <= C[i_2] THEN
        X[k] <- B[i_1]
        i_1 <- i_1 + 1
    ELSE
        X[k] <- C[i_2]
        i_2 <- i_2 + 1
```

```
    k <- k + 1

IF i_1 < l_b THEN // B non è finito
    FOR j <- i_1 TO l_b - 1 DO
        X[k] <- B[j]
        k <- k + 1
ELSE             // C non è finito
    FOR j <- i_2 TO l_c - 1 DO
        X[k] <- C[j]
        k <- k + 1

RETURN X
```

Si da ora un'implementazione che esegue la fusione di due porzioni dello stesso array, delimitate da indici, in particolare fonde `A[i..m]` con `A[m..f]`. `X` è un'array usiliario utile nel caso l'algoritmo venga chiamato molte volte.

```
PROCEDURA merge(Array A, Indice i, Indice m, Indice f, Array x)
i_1 <- i
i_2 <- m
k <- 0
WHILE i_1 < m AND i_2 < f DO
    IF A[i_1] < A[i_2] THEN
        X[k] <- A[i_1]
        i_1 <- i_1 + 1
    ELSE
        X[k] <- A[i_2]
        i_2 <- i_2 + 1
    k <- k + 1

IF i_1 < m THEN
    FOR j <- i_1 TO m - 1 DO
        X[k] <- A[j]
        k <- k + 1
ELSE
    FOR j <- i_2 TO f - 1 DO
        X[k] <- A[j]
        k <- k + 1

FOR k <- 0 TO f - i - 1 DO // copia X in A
    A[i + k] <- X[k]
```

9.5.3 Analisi

9.5.3.1 Tempo Si pone $n = l_b + l_c$, nel caso peggiore il numero di confronti è $C(n) = n - 1$.

9.5.3.2 Spazio Si utilizza un array aggiuntivo per il risultato e tre variabili quindi è costante.

9.6 mergeSort

9.6.1 Strategia

Il `mergeSort` si basa sulla tecnica dividi et impera. Nel caso base l'array contiene un solo elemento e quindi è già ordinato. Se l'array `A` contiene $n > 1$ elementi allora si divide `A` a metà, si ordinano le due metà separatamente e infine si fondono con un algoritmo di fusione. Lo schema di base è il seguente:

```
ALGORITMO mergeSort(Array A[0..n-1])
IF n > 1 THEN
    m <- n / 2
    B <- A[0..m-1]
    C <- A[m-1..n-1]
    mergeSort(B)
    mergeSort(C)
    A <- merge(B, C)
// se A è lungo 1 non faccio nullaaaaeeiooou daje forza Roma
```

9.6.2 Implementazione

L'implementazione diretta richiede l'uso di spazio per gli array `B` e `C` e di tempo per la copiatura da `A` in essi. Si da una soluzione che utilizza un solo array usiliario `X`, passato alla procedura, in modo tale da non aggiungerne uno nello stack ad ogni chiamata:

```
PROCEDURA mergeSort(Array A, Indice i, Indice j, Array X)
IF f - i > 1 THEN
    m <- (i + f) / 2
    mergeSort(A, i, m, X)
    mergeSort(A, m, f, X)
    merge(A, i, m, f, X)

ALGORITMO mergeSort(Array A[0..n-1])
X <- [0..n-1]Array
mergeSort(A, 0, n, X)
```

L'algoritmo è stabile

9.6.3 Analisi

9.6.3.1 Confronti Il numero di confronti è dato da:

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

Per n pari e ricordando che $C_{\text{merge}}(n) = n - 1$:

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

Per sostituzione, con n potenza di 2, si calcola: $C(n) = n\log_2(n) - n + 1 = \Theta(n\log(n))$. Ricordando che esiste un numero N potenza di 2 tale che $n \leq N \leq 2n$ si conclude $C(n) = \Theta(n\log(n))$ anche nel caso n non potenza di 2.

9.6.3.2 Spazio

L'altezza della pila è:

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

Per n pari:

$$H(n) = \begin{cases} 1 & \text{se } n = 1 \\ 1 + H(\frac{n}{2}) & \text{se n} > 1 \end{cases}$$

Per n potenza di 2: $H(n) = 1 + \log_2(n)$, quindi $H(n) = \Theta(\log(n))$.

L'algoritmo utilizza un array ausiliario che occupa $\Theta(n)$ che è maggiore dello stack, quindi lo spazio è $\Theta(n)$.

9.7 quickSort

9.7.1 Strategia

Il `quickSort` utilizza una tecnica dividi et impera, ma a differenza di `mergeSort` la parte di dividi è complessa, mentre è banale la parte di impera. La strategia è la seguente:

- Dividi: si sceglie un elemento x dell'array detto pivot, si partiziona in due array contenenti uno gli elementi minori o uguali di x e l'altro gli elementi maggiori di x e si ordinano.
- Impera: si concatenano i due array ottenuti dalla parte di dividi.

Si da uno schema ad alto livello:

```
ALGORITMO quickSort(Array A)
IF lunghezza di A > 1 THEN
    pivot <- scegli elemento di A
    B <- {y in A t.c. y <= pivot}
    C <- {y in A t.c. y > pivot}

    quickSort(B)
    quickSort(C)

    RETURN concatenazione di B e C
```

9.7.2 Implementazione

Si scrive ora l'algoritmo di partizionamento. La soluzione naive sarebbe creare due nuovi array, scorrere linearmente l'originale e inserire gli elementi di conseguenza. Si vuole però lavorare in loco. L'algoritmo scorre parallelamente l'array da sinistra a destra e da destra a sinistra fino a quando non trova un elemento fuori posto, quindi scambia i due elementi fuori posto e continua a scorrere. In questo modo posiziona gli elementi minori del perno a destra e maggiori a sinistra. Si ferma non appena gli indici di sinistra e di destra si incrociano.

```
ALGORITMO partiziona(Array A, Indice i, Indice f) -> Indice
perno <- A[i]
dx <- f
sx <- i
WHILE sx < dx DO
    DO dx <- dx - 1 WHILE A[dx] > perno // se continua ad andare a sinistra prima o poi arriva al
    DO sx <- sx + 1 WHILE sx < dx AND A[sx] <= perno
    IF sx < dx THEN
        A[sx], A[dx] <- A[dx], A[sx]
A[i], A[dx] <- A[dx], A[i]
RETURN dx
```

Con l'algoritmo di partizionamento il perno è già nella sua posizione ordinata, pertanto l'algoritmo quickSort può limitarsi a riordinare la parte destra e quella sinistra.

```
PROCEDURA quickSort(Array A, Indice i, Indice f)
IF f - i > 1 THEN
    m <- partiziona(A, i, f)
    quickSort(A, i, m)
    quickSort(A, m + 1, f)

ALGORITMO quickSort(Array A[0..n-1])
quickSort(A, 0, n)
```

9.7.3 Analisi

9.7.3.1 Numero di confronti Il numero di confronti varia in funzione del perno. Dato un perno in posizione finale k dopo il partizionamento, l'equazione di ricorrenza è:

$$C(n) = \begin{cases} C_{\text{partiziona}}(n) + C(k) + C(n-k-1) & \text{se } n > 1 \\ 0 & \text{altrimenti} \end{cases}$$

Dove $C(k)$ è prima chiamata ricorsiva; $C(n-k-1)$ è la seconda chiamata ricorsiva.

Si osserva che nell'algoritmo di partizione ogni elemento deve essere confrontato con il perno, eccetto il perno stesso. In alcuni casi l'indice destro viene confrontato di nuovo col perno. Quindi $C_{\text{partiziona}} = n - 1$.

Si calcola il caso peggiore $C_w(n)$:

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

Il massimo si ottiene quando la partizione è completamento sbilanciata, ovvero se il perno è l'elemento minore o l'elemento massimo, ovvero se $k = 0$ oppure $k = n - 1$. Sostituendo:

$$C_w(n) = n + C_w(n-1) = \sum_{i=2}^{n} i = \Theta(n^2)$$

Si calcola il caso migliore $C_b(n)$:

$$C_b(n) = \begin{cases} n + \min(C_w(k), C_w(n-k-1) : 0 < k \leq n-1) & \text{se n} > 1 \\ 0 & \text{altrimenti} \end{cases}$$

Il caso migliore si ottiene quando il perno è in mezzo, quindi si può approsimare con:

$$C_b(n) = n + 2C_b(\frac{n}{2})$$

Che ha soluzione $C_b(n) = n \log_2(n) = \Theta(n \log(n))$

Si calcola ora il caso medio.

Per calcolare il caso medio si fa la media considerando tutti i valori di k. Se $n > 1$ si ottiene:

$$C(n) = \frac{1}{n}\sum_{k=0}^{n-1} n + \frac{1}{n}\sum_{k=0}^{n-1} C(k) + \frac{1}{n}\sum_{k=0}^{n-1} C(n-k-1) = n + \frac{2}{n}\sum_{i=2}^{n-1} C(i)$$

dove $C(k)$ è la parte sinistra della partizione e $C(n-k-1)$ è la parte destra.

Quindi:

$$C(n) = \begin{cases} n + \frac{2}{n}\sum_{i=2}^{n-1} C(i) & \text{se n} > 1 \\ 0 & \text{altrimenti} \end{cases}$$

Ora si dimostra per induzione che $C(n) \leq 2n \ln(n)$ per $n >= 1$:

- Base $n = 1$: $0 = C(1) \leq 2 \cdot 1 \ln(1) = 0$
- Passo, si assume che la formula valga per $n - 1$:

$$C(n) = n + \frac{2}{n}\sum_{i=2}^{n-1} C(i) \leq n + \frac{2}{n}\sum_{i=2}^{n-1} i2\ln(i) = n + \frac{4}{n}\sum_{i=2}^{n-1} i\ln(i) \leq n + \frac{4}{n}(\frac{n^2}{2}\ln(n) - \frac{n^2}{4}) = n + 2n\ln(n) - n = 2n\ln(n)$$

Quindi il caso medio è $O(n \log(n))$.

Poichè il caso medio è molto vicino a quelo migliore, capita molto spesso. Per ottenere il caso migliore si possono fare scambi a caso per creare confusione.

9.7.3.2 Altezza della pila Innanzitutto si osserva che ogni chiamata ricorsiva usa memoria costante, pertanto la memoria usata è proporzionale all'altezza massima della pila.

Il caso migliore si ottiene per partizioni bilanciate, ovvero quando l'algoritmo viene chiamato lo stesso numero di volte per entrambe le parti. L'altezza in questo caso è $\log n$.

Il caso peggiore si ottiene partizioni sbilanciate, ovvero quando il perno è sempre il maggiore. Le prima chiamata ricorsiva cresce a sinistra fino ad n, mentre quella a destra rimane bassa.

Si può ottimizzare il codice eliminando l'ultima chiamata ricorsiva in coda e ordinando per prima la parte più piccola, lasciando alla prossima iterazione la parte più grande. Così facendo, una chiamata ordina un array lungo al più metà rispetto all'array ordinato dal chiamante, e quindi l'altezza della pila è $\log n$.

```
ALGORITMO quickSort(Array A, Indice i, Indice f)
WHILE i - f > 1 DO
    m <- partiziona(A, i, f)
    IF m - i < f - m  THEN
        quickSort(A, i, m)
        i <- m + 1
    ELSE
        quickSort(A, m + 1, f)
        f <- m
```

9.8 heapSort

9.8.1 Strategia

L'algoritmo heapSort si basa sulla struttura dati heap. Dapprima si costruisce un albero binario quasi completo a partire dall'array, poi lo si trasforma in uno heap. A questo punto si svuota l'albero rimuovendo la radice e risistemando fin quando non è vuoto. Lo schema è il seguente:

```
ALGORITMO heapSort(Array A) -> Lista
H <- costruisci albero binario quasi completo da A
creaHeap(H)
Lista X
WHILE H != albero vuoto DO
    rimuovi la radice di H e aggiungilo nella lista
    rimuovi la foglia più a destra dell'ultimo livello e ponila nella radice
    risistema(H)
RETURN X
```

9.8.2 Implementazione

Si osserva che in un albero quasi completo rappresentato con un array, se un nodo è in posizione i allora i figli sono in posizione $2i + 1$ e $2i + 2$. Quindi lo heap lo si rappresenta con l'harray. A questo punto si implementano le operazioni `risistema()` e `creaHeap()`con array.

```
/*
* Risistema l'heap rappresentato dall'array A i cui primi l elementi
* formano un heap con radice r sbagliata.
*/
PROCEDURA risistema(Array A[0..n-1], Intero l, Intero r)
v <- r
x <- A[v]
da_collocare <- true
WHILE da_collocare DO
    IF 2 * v + 1 >= l THEN
        da_collocare <- false
    ELSE
        u <- 2 * v + 1
        IF u + 1 < l AND A[u + 1] > A[u] THEN
            u <- u + 1
        IF A[u] > x THEN
            A[v] <- A[u]
            v <- u
        ELSE
            da_collocare <- false
A[v] <- x

// poichè le foglie sono già heap, si può evitare
// di risistemarle e quindi si inizia con `i = n / 2`.
PROCEDURA creaHeap(Array A[0..n-1])
FOR i <- n / 2 DOWNTO 0 DO
    risistema(A, i, n)
```

L'implementazione di heapSort è banale:

```
PROCEDURA heapSort(Array A[0..n-1])
creaHeap(A)
FOR l <- n - 1 DOWNTO 1 DO // l'indice l tiene conto dell'ultima foglia dello heap
    A[0], A[l] <- A[l], A[0]
    risistema(A, 0, l)
```

L'implementazione non è stabile.

9.8.3 Analisi

`creapHeap()` effettua $\Theta(n)$ confronti, `risistema()` effettua $O(\log(n))$. La parte iterativa chiama `risistema()` n volte. Quindi l'algoritmo effettua $\Theta(n) + O(n \log n) = O(n \log n)$ confronti. Dimostrato che un algoritmo basato sui confronti effettua almeno $n \log(n)$ confronti si conclude che il numero di confronti per heapSort è $\Theta(n \log(n))$.

L'algoritmo è in loco e quindi non usa spazio.

Riepilogo algoritmi di ordinamento basati sui confronti

Figura 9.1: Riepilogo algoritmi di ordinamento basati sui confronti

9.9 Riepilogo algoritmi di ordinamento basati sui confronti

9.10 integerSort

Si analizzeranno ora diversi algoritmi non basati sui confronti.

Si supponga di voler ordinare un array A composto da n interi nell'intervallo $[0, k-1]$. `integerSort` risolve questo problema usando un array ausiliario Y di dimensione k. Un primo ciclo scandisce A e per ogni chiave v incontrata incrementa $Y[v]$. In questo modo, $Y[i]$ contiene il numero di elementi i in A. A questo punto si scorre Y e si posizionano correttamente gli elementi in A.

```
ALGORITMO integerSort(Array A[0..n-1], Intero k)
Array Y[0..k]
FOR i <- 0 TO k DO Y[i] <-0
FOR i <- 0 TO n - 1 DO Y[X[i]] <- Y[X[i]] + 1 // Y[i] contiene il numero di volte che si ripete i


j <- 0
FOR i <- 0 TO n DO
    WHILE Y[i] > 0
        X[j] <- Y[i]
        Y[i] <- Y[i] - 1
        j <- j + 1
```

`integerSort` non utilizza nessun confronto. Per analizzarlo si fa un'analisi generale. Il numero di passi è $\Theta(n+k)$, dunque:

- Se $k = O(n)$ allora $T(n) = O(n)$
- Se $k = O(n^2)$ allora $T(n) = O(n^2)$.

9.11 bucketSort

`bucketSort` generalizza integerSort ordinando n record con chiavi intere nell'intervallo $[0, k-1]$. Si utilizza sempre un array ausiliario Y di lunghezza k, ma questa volta i suoi elementi non sono contatori interi ma liste di record. Gli elementi $Y[i]$ sono "bucket" che contengono una lista dei record con chiave intera i.

```
ALGORITMO bucketSort(Array A[o..n-1], Intero b)
Array Y[0..b-1]
FOR i <- 0 TO b - 1 DO
    Y[i] <- coda_vuota
FOR i <- 0 TO n - 1 DO
    x <- A[i].chiave
    Y[x].enqueue(A[i])
j <- 0ù
FOR i <- 0 TO b - 1 DO
    WHILE NOT Y[i].isEmpty() DO
```

```
        A[j] <- Y[i].dequeue()
        j <- j + 1
```

Le liste sono implementate come coda, garantendo così stabilità. Il tempo di esecuzione è $T(n,k) = O(n+k)$. Dunque, di nuovo:

- Se $k = O(n)$ allora $T(n) = O(n)$
- Se $k = O(n^2)$ allora $T(n) = \Theta(n^2)$

L'algoritmo non è in loco, ma si osserva che lo è se si usa per ordinare liste concatenate, per cui risulta molto utile. Basta infatti collocare ciascun nodo di A nella coda corrispondente alla chiave e infine concatenare le liste una dopo l'altra.

9.12 radixSort

Si considerino n numeri in base b in un intervallo $[0, k-1]$. `bucketSort` non conviene per valori di k molto grossi. Tuttavia se si accede alle singole cifre, allora $k = b$. `radixSort` applica `bucketSort` utilizzando come chiave le cifre da destra a sinistra.

```
/** Ordina gli interi di base b in A in base alla t-esima cifra */
PROCEDURA bucketSort(Array A[0..n-1], Intero b, Intero t)
Array Y[0..n-1]
// predisposizione bucket
FOR i <- 0 TO n - 1 DO
    Y[i] <- Coda vuota
// riempimento bucket
FOR i <- 0 TO n - 1 DO
    x <- (A[i].chiave \ b^t) % b // x contiene la t-esima cifra della chiave di A[i]
    Y[x].enqueue(A[i])

i <- 0
FOR i <- 0 TO b - 1 DO
    WHILE NOT Y[i].isEmpty() DO
        A[j] <- Y[i].dequeue()
        j <- j + 1

ALGORITMO radixSort(Array A[0..n-1], Intero b)
t <- 0
WHILE (Esiste chiave K in A t.c. (K / b^t) != 0) DO
    bucketSort(A, b, t)
    t <- t + 1
```

`radixSort` richiede $\log_b k$ passate di `bucketSort`, ognuna delle quali richiede tempo $O(n)$. Quindi il tempo totale è $O(n \log_b k)$.

10 Problema Union-Find

Il problema Union-Find consiste nel mantenere una collezione di insiemi disgiunti su cui effettuare le operazioni:

- `union(A,B)`: unisce gli insiemi `A` e `B` in un unico insieme di nome `A`.
- `find(x)`: restituisce il nome dell'insieme a cui appartiene `x`.
- `makeset(x)`: crea un insieme di nome `x` a cui appartiene solo `x`.

Si osserva che su n elementi, creati con n `makeset`, il numero di `union` che si possono chiamare è limitato superiormente da $n-1$, dopodichè rimarrà un unico insieme contenente tutti gli elementi.

Vi sono soluzioni elementari e non. In tutte le soluzioni presentate ogni insieme è rappresentato da un albero con radice: i nodi sono gli elementi dell'insieme; la radice è il nome dell'insieme; la foresta è l'insieme degli insiemi disgiunti, ognuno dei quali è un albero come qua descritto. A loro volta gli alberi sono rappresentati con puntatori verso l'alto.

10.1 `quickFind`

Negli algoritmi di tipo `quickFind` si adoperano alberi di altezza 1 nei quali la radice rappresenta il nome dell'insieme e gli elementi dell'insieme sono le foglie.

Figura 10.1: Da sinistra a destra: albero che rappresenta l'insieme $2 = \{2, 5, 7\}$ e albero che rappresenta l'insieme $9 = \{1, 9\}$

Si analizzano ora le strategie e i costi per implementare le operazioni con tale rappresentazione, con n numero di elementi dell'insieme:

- `makeset(x)`: crea un albero composto dalla radice `x` e dal figlio `x`. $T(n) = O(1)$
- `find(x)`: basta salire verso il padre. $T(n) = O(1)$
- `union(A, B)`: sostituisce i puntatori delle foglie di `B` con puntatori verso la radice di `A` e cancella `B`. Nel caso peggiore l'insieme `B` contiene $n-1$ elementi, mentre `A` contiene un solo elemento. $T(n) = O(n)$

10.2 `quickFind` con bilanciamento

È possibile ottimizzare gli algoritmi di tipo `quickFind`, ottimizzando l'operazione di `union(A, B)`. Precedentemente l'implementazione prevedeva di collegare tutti i nodi di `B` verso `A`, tuttavia se la cardinalità di `B` è maggiore di `A` è meglio collegare i le foglie di `A` verso `B`; si tenga conto che così facendo bisogna cambiare il nome della radice di `B` con quello della radice di `A`. Si memorizza la cardinalità dell'insieme nella radice. Si osserva che vengono spostati al massimo $\frac{n}{2}$ elementi.

Con un analisi di costo ammortizzato si osserva che se capitano operazioni di unione costose allora quelle dopo non lo saranno. Si può dimostrare che dopo n di operazioni `makeset` e $O(n)$ do operazioni `union` e `find` il tempo totale è $O(\log(n))$.

10.3 `quickUnion`

Negli algoritmi di tipo `quickUnion` si utilizzano alberi con radice di varie altezze nei quali i nodi sono gli elementi dell'insieme e la radice è l'elemento che da il nome all'insieme.

Si analizzano ora le strategie e i costi per implementare le operazioni con tale rappresentazione, con n numero di elementi dell'insieme:

Figura 10.2: Da sinistra a destra: albero che rappresenta l'insieme $1 = \{1, 2, 3\}$, albero che rappresenta l'insieme $6 = \{4, 5, 10, 7, 8\}$ e albero che rappresenta l'insieme $9 = \{9\}$

- `makeSet(x)`: crea un nuovo albero con solo il nodo `x`. $T(n) = O(1)$.
- `find(x)`: accede al nodo `x` e segue i puntatori ai padri fino ad arrivare alla radice, contenente il nome dell'insieme. Il tempo dipende dalla distanza di `x` alla radice; il caso peggiore si ottiene con un albero di grado 1 con tutti i n elementi. $T(n) = O(n)$
- `union(A, B)`: rende la radice di `B` figlia di `A`, collegando il puntatore della radice di `B` alla radice di `A`. $T(n) = O(1)$.

10.4 `quickUnion` con bilanciamento in altezza

Si indica con `rank(X)` l'altezza di un albero `X`. Per evitare che l'altezza dell'albero cresca velocemente, nella `union(A, B)`non si attacca `B` sotto ad `A` ma si attacca l'albero più basso sotto l'albero più alto. Se `rank(B)` > `rank(A)` allora bisogna cambiare il nome della radice

A questo punto l'operazione `find(x)` ha tempo $T(n) = O(h)$, bisogna calcolare h.

Lemma Un albero quickUnion bilanciato in altezza ha almeno $\geq 2^{\text{rank}(x)}$ nodi.
Si dimostra per induzione sul numero k di operazioni di `union` con cui l'albero è stato costruito.

- Base: $k = 0$ allora l'albero ha solo la radice, quindi $n = 1$ e $h = 0$, vale $n \geq 2^0$.
- Passo: si suppone valga per un numero minore di $k > 0$ `union`. Nell'ultima operazione `union(A, B) A` e `B` sono stati costruiti con meno di k `union`. Si dimostrano le tre casistiche, la notazione n_i indica il numero di nodi dell'albero con radice i:
  - `rank(A)` > `rank(B)`: $n_x = n_a + n_b \geq 2^{\text{rank(A)}} + 2^{\text{rank(B)}} > 2^{\text{rank(A)}} > 2^{\text{rank(x)}}$
  - `rank(A)` < `rank(B)`: dimostrazione analoga al caso `rank(A)` > `rank(B)`
  - `rank(A)` = `rank(B)`: $n_x = n_a + n_b \geq 2^{\text{rank(A)}} + 2^{\text{rank(B)}} = 2^{\text{rank(A)}+1} = \text{rank(x)}$

Corollario $h \leq \log_2(n)$

Quindi la find usa tempo $O(\log(n))$.

10.5 `quickUnion` bilanciata in altezza con compressione di cammino

Per migliorare i tempi di esecuzione bisogna diminuire l'altezza degli alberi. Si applica un'euristica di compressione del cammino.

Euristica di compressione di cammino Siano $u_0, \cdots, u_{l-1}$ i nodi che visita un'operazione `find(x)` con $u_0 = x$ e u_{l-1} radice dell'albero. Per $l \geq 3$, l'euristica di compressione del cammino rende i nodi u_i con $0 \leq i \leq l-3$ figli della radice u_{l-1}.

Un albero quickUnion con compressione di cammino, dopo n makeset e $O(n)$ di `union` e `find` ha tempo totale $O(n \log^* n)$, quindi il costo ammortizzato di find è $O(\log^* n)$.

Riepilogo degli algoritmi di Union-Find

Figura 10.3: Riepilogo degli algoritmi di Union-Find

10.6 Riepilogo

11 Grafi

Grafo Un grafo è una coppia $G = (V, E)$ con:

- V insieme finito di nodi (o vertici)
- $E \subseteq V \times V$ insieme di archi (o lati, o spigoli)

Un grafo rappresenta una relazione tra oggetti. V è l'insieme degli oggetti, E è l'insieme delle relazioni. Cosa rappresenti ung rafo dipende dal problema che si vuole modellare.

Grafi orientati e non orientati Vi sono due tipi di grafi:

- Grafi non orientati: E è una relazione simmetrica
- Grafi orientati (o diretti): E è una relazione non simmetrica diretta

Si indicherà con n il numero di vertici $|V|$; con m il numero di archi $|E|$.

Figura 11.1: Un grafo non orientato

Figura 11.2: Un grafo orientato

11.1 Glossario dei termini

Si danno una serie di definizioni sui grafi. Alcune variano per i grafi orientati, se non specificato comunque si fa riferimento a entrambi i tipi di grafo.

Archi incidenti e adiacenti Un arco (x, y) si dice incidente sui vertici x e y, inoltre:

- Nei grafi orientati si dice che l'arco (x, y) esce dal vertice x ed entra nel vertice y e che y è adiacente a x.
- Nei grafi non orientati si dice che x e y sono adiacenti, ovvero x è adiacente a y e y è adiacente a x.

Grado di un vertice Il grado di un vertice v è il numero di arci incidenti su v. Si denota con $\delta(v)$. Vale:

$$\sum_{v \in V} = 2m \text{con } m \text{ numero di vertici}$$

In un grafo orientato si dice grado in ingresso $\delta_{in}(v)$ il numero di vertici che entrano in v e grado in uscita $\delta_{out}(v)$ il numero di vertici che escono da v. Si osserva che $\delta(v) = \delta_{in}(v) + \delta_{out}(v)$. Inoltre:

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

quindi anche nei grafi orientati si ottiene:

$$\sum_{v \in V} \delta(v) = \sum_{v \in V} \delta_{in}(v) + \sum_{v \in V} \delta_{out}(v) = 2m$$

Cammini e cicli Un cammino da un vertice x a un vertice y è una sequenza di vertici $v_0, v_1, \cdots, v_k$ t.c: $v_0 = x, v_k = y$ e $(v_{i-1}, v_i) \in E$ per $i = 1, \cdots, k$. Si dice che il cammino ha lunghezza k e che contiene (passa per) i vertici $v_0, v_1, \cdots, v_k$ e gli archi $(v_0, v_1), (v_1, v_2), \cdots, (v_{k-1}, v_k)$. Nei grafi orientati un cammino da x a y si denota con $x\ y$

Un cammino si dice semplice se non contiene vertici ripetuti.

Se esiste un cammino da x a y si dice che y è raggiungibile da x, ovvero che x è un antenato di y e che y è un discendente di x.

Un cammino che inizia da un vertice e ritorna allo stesso vertice si dice ciclo.

Un ciclo si dice semplice se solo il vertice è ripetuto.

Catene Un catena da x a y è una sequenza di vertici $v_0, v_1, \cdots, v_k$ t.c: $v_0 = x, v_k = y$ e $((v_{i-1}, v_i) \in E \vee (v_i, v_{i-1} \in E)$ per $i = 1, \cdots, k$. La catena si dice circuito se $x = y$.

Grafi connessi e fortemente connessi Un grafo si dice connesso se esiste una catena tra ogni coppia di vertici. Un grado si dice fortemente connesso se esiste un cammino tra ogni coppia di vertici. Due vertici x e y si dicono fortemente connessi se esiste un cammino da x a y e un cammino da y a x. Due vertici fortemente connessi si denotano con $x \Leftrightarrow y$.

Sottografo Si dice che $G' = (V', E')$ è un sottogrado di $G = (V, E)$ se $V' \subseteq V$ e $E' \subseteq E$. G' si dice sottografo di G indotto da V' se vale anche $E' = \{(x, y) \in E : x, y \in V'\} = E \cap (V' \times V')$.

Componente fortemente connessa Una componenete fortemente connessa è un sottografo indotto fortemente connesso massimale. Con massimale si intende che, dato un sottografo fortemente connesso, si possono aggiungere altri vertici tale per cui la componente rimanga fortemente conenssa. Le componenti fortemente connesse di un grafo formano una partizione, in particolare le componenti fortemente connesse sono le classi di equivalenza rispetto alla relazione "fortemente connesso a".

Grafi pesati Un grafo pesato $G = (V, E)$ è un grafo a cui è associata una funzione peso $w : E \to \mathbb{R}$ che assegna ad ogni vertice un peso.

Cricca Una cricca è un grafo non orientato completo. Una cricca in un grafo è un sottografo completo.

11.2 Circuiti

Circuito Hamiltoniano Un circuito Hamiltoniano è un circuito che passa per ogni vertice del grafo una e una sola volta.

Figura 11.3: Circuito hamiltoniano

Circuito Euleriano Un circuito Euleriano è un circuito che passa per ogni arco del grafo una e una sola volta.

Figura 11.4: Circuito Euleriano

Teorema sui circuiti Euleriano Esiste un circuito euleriano se e solo se $\forall v \delta(v)$ è pari.

11.3 Alberi come grafi

Si analizzano ora gli alberi utilizzando la teoria dei grafi, in particolare si definierà un albero come un grafo e si dimostreranno diverse proprietà.

Albero Un albero è un grafo non orientato, connesso e privo di cicli.
Foresta Una foresta è un insieme di alberi.
Albero ricoprente (o di supporto) Sia $G = (V, E)$ un grafo, si definisce albero ricoprente l'albero $G' = (V', E')$ con $V' = V$ e $E' \subseteq E$

11.3.1 Proprietà 1

Enunciato Un grafo non orientato $G = (V, E)$ è un albero se e solo se tra ogni coppia di vertici esiste uno e un solo cammino.

11.3.2 Proprietà 2

Enunciato Sia $G = (V, E)$ un albero, allora $m = n - 1$.
Dimostrazione (TODO) Si dimostra per induzione su n.

- Base $n = 1$: vi è un solo vertice quindi 0 archi $m = n - 1$, $0 = 1 - 1 = 0$.
- Passo $n > 1 \implies n$: si prende un nodo x qualunque, rimuovendo x e i relativi k archi si ottengono k foreste $G_i = (V_i, E_i) \forall i = 1, \cdots, k$. Vale $v_i < n$ quindi per l'induzione $m_i = n_i - 1$. Inoltre $n = n_1 + \cdots + n_k + 1 = m_1 + 1 + \cdots + m_k + 1 + 1$ e $m = k + m_1 + \cdots + m_k$. Combinando $n = k + m_1 + \cdots + m_k + 1 = m + 1$ dunque $m = n - 1$.

11.3.3 Proprietà 3

Enunciato Sia $G = (V, E)$ un grafo non orientato e connesso, se $m = n - 1$ allora G è un albero.
Dimostrazione Si procede per assurdo. Si assume che G non orientato e connesso non è un albero, ma allora vi sono dei cicli, ma per ipotesi $m = n - 1$, quindi è assurdo e quindi G è un albero.

11.3.4 Teorema

Enunciato Un grafo $G = (V, E)$ non orientato e connesso è un albero se e solo se $m = n - 1$.
Corollario Sia G una foresta di k alberi, allora $m = n - k$.

11.4 Rappresentazione

11.4.1 Lista di archi

Il modo più semplice per rappresentare un grafo è tramite una lista di archi.

Lo spazio occupato è $\Theta(n + m)$.

Fissati n vertici si osserva che: $0 \leq m \leq n^2$

Se G connesso: il minimo numero di lati si ottiene quando è un albero "dritto" quindi $|V| - 1 \leq |E|$

Utile se non vi è ordine con cui bisogna visitare gli archi

11.4.2 Lista di adiacenza

La lista di adiacenza prevede un array di vertici a cui è collegata una lista dei vertici adiacenti.

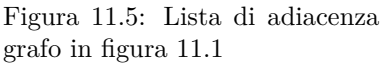

Figura 11.5: Lista di adiacenza del grafo in figura 11.1

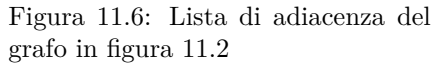

Figura 11.6: Lista di adiacenza del grafo in figura 11.2

Nei grafi orientati lo spazio totale delle liste è $2m$, nei grafi non orientati è m. Quindi lo spazio è $\Theta(n+m)$, per i grafi non orientati vi è una ripetizione nelle liste di vertici adiacenti.

Questa rappresentazione è utile per visitare un grafo esaminando gli archi incidenti sui suoi vertici.

11.4.3 Lista di incidenza

La lista di incidenza prevede una lista degli archi e una lista adiacenza, quest'ultima contiene i riferimenti alla lista di archi.

Lo spazio occupato è $\Theta(n + m)$.

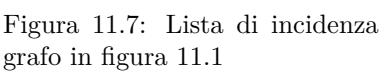

Figura 11.7: Lista di incidenza del grafo in figura 11.1

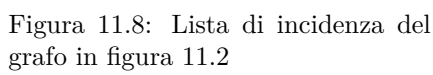

Figura 11.8: Lista di incidenza del grafo in figura 11.2

11.4.4 Matrice di adiacenza

La matrice di adiacenza è una matrice quadrata $n \times n$ contenente valori $0/1$, definita come segue:

$$M[u,v] = \begin{cases} 1 \text{ se } (u,v) \text{ è un arco del grafo} \\ 0 \text{ altrimenti} \end{cases}$$

Lo spazio occupato è $\Theta(n^2)$. Nei grafi sparsi con $n \leq m^2$ lo spazio è $O(n)$.

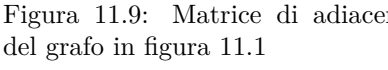

Figura 11.9: Matrice di adiacenza del grafo in figura 11.1

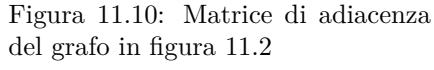

Figura 11.10: Matrice di adiacenza del grafo in figura 11.2

La matrice di adiacenza è utile per grafi piccoli. Inoltre è di interesse per operazioni algebriche, ad esempio M^k rappresenta cammini di lunghezza k.

11.4.5 Matrice di incidenza

La matrice di incidenza è una matrice $n \times m$ contenente sulle righe i vertici, sulle colonne gli archi. $M[v, l]$ contiene 1 se il lato l incide sul vertice v, 0 viceversa; nei grafi orientati -1 se l è incidente su v, ma v è il vertice destinazione.

Figura 11.11: Matrice di incidenza del grafo in figura 11.1

Figura 11.12: Matrice di incidenza del grafo in figura 11.2

11.4.6 Lista di adiacenza con pesi

Una lista di adiacenza con pesi è simile alla matrice di adiacenza. Data la lista in $d[v]$, ogni elemento contiene due campi: il nodo u adiacente a v e il peso dell'arco (v, u).

11.4.7 Matrice dei pesi

Una matrice dei pesi D che rappresenta un grafo pesato $G = (V, E, \omega)$ formato dai vertici $v_1, \ldots, v_k$ è una matrice $k \times k$ definita come segue:

$$D[i, j] = \begin{cases} \omega(v_i, v_j) & \text{se } (v_i, v_j) \in E \\ \infty & \text{altrimenti} \end{cases}$$

11.5 Attraversamento di grafi

11.5.1 Visita in ampiezza

La visita in ampiezza inizia visitando un vertice `S`, successivamente si visitano i vertici adiacenti a `S` e così ricorsivamente

```
ALGORITMO visitaInAmpiezza(Grafo G=(V, E), Vertice S) -> Albero
Coda C
Albero T
marca S come raggiunto
C.enqueue(S)
WHILE NOT C.isEmpty() DO
    u <- C.dequeue()
    FOR EACH (u, v) IN E DO
        IF v non è marcato THEN
            aggiungi v e (u, v) a T
            marca v come raggiunto
            C.enqueue(V)
RETURN T
```

L'algoritmo dato esegue $\Theta(n \cdot m)$ passi.

11.5.2 Visita in profondità

La visità in profondità parte da vertice di partenza `S` e percorre un cammino terminando su un vertice che ha già tutti i vicini come raggiunti. Ripete dal passo precedente con un

vertice ancora non raggiunto.

```
ALGORITMO visitaInProfondita(Grafo G, Vertice S) -> Albero
Albero T
visitaRicorsiva(G, S, T)
RETURN T

PROCEDURA visitaRicorsiva(Grafo G, Vertice S, Albero T) -> Albero
marca S come raggiunto
FOR EACH (u, v) IN E DO
    IF NOT v è marcato come visitato THEN
        aggiungi v e (u, v) a T
        visitaRicorsiva(G, v, T)
```

12 Albero ricoprente minimo

Dato $G = (V, E)$ non orientato connesso con una funzione peso $w : E \to \mathbb{R}$, si vuole trovare un albero ricoprente $T = (V, E_T)$ di peso minimo.

12.1 Algoritmo di Kruskal

12.1.1 Strategia

L'algoritmo di Kruskal applica una strategia greedy. Inizialmente si costruisce un grafo formato da tutti i vertici di G, successivamente si analizza ogni arco (x, y) in maniera crescente e si aggiunge al grafo se e solo se x e y non sono connessi (ltrimenti l'aggiunta dell'arco formerebbe un ciclo). Lo schema generale è:

```
ALGORITMO Kruskal(Grafo G = (V, E)) -> Albero
ordina E in maniera crescente rispetto ai pesi
T <- (V, emptySet)
FOR EACH (x, y) IN E DO
    IF x e y non sono connessi in T THEN // se fossero connessi, aggiungere l'arco formerebbe un
        aggiungi a T l'arco (x, y)
RETURN T
```

Si può dimostrare che l'algoritmo di Kruskal trova sempre la soluzione ottima per qualsiasi input. Lavora correttamente anche nel caso di pesi negativi.

12.1.2 Implementazione

Ai fini dell'ordinamento, il grafo è rappresentato come lista di archi. Nell'algoritmo di ordinamento utilizzato la chiave è il peso dell'arco.

Per capire se due vertici x e y sono connessi si utilizza una struttura union-find che rappresenta una partizione dell'insieme V. Ogni partizione rappresenta una componente connessa di T, ovvero un albero della foresta T. Due elementi sono connessi se e solo se appartengono alla stessa partizione.

```
ALGORITMO Kruskal(Grafo G = (V, E)) -> Albero
ordina E in maniera non decrescente in base ai pesi
T <- (V, emptySet)
FOR EACH v IN V DO makeset(v)
FOR EACH (x, y) IN E DO
    Tx <- find(x)
    ty <- find(y)
    IF Tx != Ty THEN
        union(Tx, Ty)
        aggiungi a T l'arco (x, y)
RETURN T
```

12.1.3 Analisi

Si analizza tempo e spazio dell'algoritmo in funzione di n e m.

Come struttura per rappresentare le partizioni si sceglie una quickUnion con bilanciamento in altezza, di cui si ricordano i costi: `makeset`$O(1)$, `find` $O(\log(n))$, `union` $O(1)$.

Le operazioni effettuate sono:

1. Ordinamento utilizzando heapSort sugli m archi: $O(m \log m)$
2. Una `makeSet` per ognuno degli n vertici: $nO(1) = O(n)$
3. m iterazioni sugli archi:
    1. per ogni iterazione si effettuano due operazioni `find`: $2mO(\log n) = O(m \log n)$
    2. le `union` avvengono per un numero pari a $n-1$ (si ricorda un albero di n vertici contiene esattamente $n-1$ archi): $(n-1)O(1) = O(n)$

Ricordando che se G è connesso allora $n - 1 \leq m \leq n^2$, il tempo complessivo è:

$$\begin{aligned} T(n,m) &= O(m \log m) + O(n) + O(m \log n) + O(n) \\ &\leq O(m \log n^2) + O(m \log n) = O(2m \log n) + O(m \log n) \\ &= O(m \log n) \end{aligned}$$

Utilizzando radixSort per ordinare e quickUnion bilanciata in altezza con compressione del cammino per rappresentare le partizioni, si può arrivare a un tempo $T(n,m) = O(m \log^* n)$.

12.2 Algoritmo di Prim

12.2.1 Strategia

Si analizza ora un'altra strategia greedy.

L'algoritmo di Prim inizia con un albero T contenente un vertice qualunque. Procede espandendo T inserendo per ogni passo l'arco di peso minimo tra quelli che hanno un vertice in T l'altro non in T, fino ad esaurire i vertici del grafo. Lo schema generale è dato da:

```
ALGORITMO Prim(G = (V, E)) -> Abero
Albero T
s <- vertice qualunque in V
```

```
T.vertici <- {s}
WHILE T ha meno di n vertici DO
    (x, y) <- vertice di peso minore con x in T e y non in T
    T.vertici <- T.vertici + {y}
    T.archi <- T.archi + {(x, y)}
RETURN T
```

Si può dimostrare che l'algoritmo di Prim trova sempre una soluzione ottima.

12.2.2 Implementazione

Strategia per scegliere l'arco di peso minimo con un vertice in T e uno non in T
Per ogni passo bisogna scegliere l'arco di peso minimo che ha un vertice in T e uno non in T. A questo scopo è utile associare ad ogni vertice v non in T due informazioni:

- $d[v]$: minimo peso di un arco tra un vertice dell'albero T e v, ovvero $d[v] = \min(\{\omega(u,v) : u \in V_T\} \cup \{\infty\})$. $d[v] = \infty$ se non vi è nessun arco tra un vertice in T e v.
- vicino$[v]$: vertice u in T tale che $d[v] = \omega(u,v)$.

Inizialmente l'albero è vuoto, quindi tutti i vertici sono candidati e hanno $d[v] = \infty$. Ad ogni passo:

1. Tra i candidati si sceglie il vertice y con $d[y]$ minimo.
2. Si sceglie l'arco (x,y) con vicino$[y] = x$.
3. Il vertice x e l'arco (x,y) vengono aggiunti in T.
4. y viene rimosso dai candidati.
5. Si aggiorna il valori di d e vicino: tenendo conto che sia stato aggiunto un vertice y a T, per ogni arco (y,z) se z non è in T e $d[z]$ è maggiore di $\omega(y,z)$ si assegna $d[z] = \omega(y,z)$ e vicino$(z) = y$.

Si procede fino a quando l'insieme dei candidati è vuoto. Lo schema è così descritto:

```
ALGORITMO Prim(Grafo G=(V, E)) -> Albero
Array vicino indicizzato con v in V
Array d indicizzato con v in V

// inizialmente, per ogni v, d[v] = infty
FOREACH v IN V DO
    d[v] <- infty
Albero T

DO
    // seleziona il vertice y con d minimo
    y <- candidato con d minimo

    // si aggiunge a T il vertice y e l'arco (x, y)
    V_T <- V_T + y
    // al primo passo l'albero è vuoto, d[y] è infinito e non va aggiunto nessun arco
    IF d[y] != infty THEN
        x <- vicino[y]
```

```
        E_T <- E_T + {(x, y)}

    // per ogni vertice z adiacente a y non in T, se l'arco (y, z)
    // ha peso minore di d[z], si aggiorna d[z] e vicino[z]
    FOREACH (y, z) IN E DO
        IF (NOT z IN T.vertici) AND (peso(y, z) < d[z]) THEN
            d[z] <- peso(y, z)
            vicino[z] <- y
WHILE non ci sono più candidati

RETURN T
```

Strutture dati per l'implementazione Per scegiere il vertice candidato con d minimo, si potrebbe scandire il vettore e trovare quello con valore minimo. Quest'operazione però ha un costo n e verrebbe eseguita n volte, per un costo totale di $\Theta(n^2)$. Pertanto si utilizza una coda con priorità C contenente i vertici candidati cui è associata la priorità d. Per trovare l'elemento minima basta chiamare l'operazione `findMin`, successivamente si cancella con l'operazione `deleteMin`.

Il codice è:

```
ALGORITMO Prim(Grafo G=(V, E)) -> Albero
Coda_priorità C
Array vicino
Array d

FOREACH v IN V DO
    d[v] <- infty
    C.insert(v, \infty)
Albero T

DO
    y <- C.deleteMin()
    V_T <- V_T + {y}

    IF d[y] != null DO
        x <- vicino[y]
        E_T <- E_T + {(x, y)}

    FOREACH (y, z) IN E DO
        IF (NOT z IN V_T) AND (peso(y, z) < d[z]) THEN
            d[z] <- peso(y, z)
            vicino[z] <- y
            C.changeKey(z, peso(y, z))
WHILE NOT C.isEmpty()

RETURN T
```

NOTA: la coda con priorità è una struttura di supporto, non una

rappresentazione del grafo.

12.2.3 Analisi

Si assume che il grafo sia rappresentato con liste di incidenza o di adiacenza. I costi delle operazioni sono:

1. Ciclo `for-each` sugli n vertici per riempire d e la coda `C`: la coda è rappresentata come vettore posizionale e quindi il costo è $O(n)$
2. Ciclo `do-while` sugli n vertici:
   1. Trova l'elemento con minore d nella coda: $n \cdot O(\log n) = O(n \log n)$
   2. Ciclo `for-each` sugli archi adiacenti a y, ovvero un numero di iterazioni pari a $\delta(y)$, per un totale di $2m$ iterazioni, due per ogni arco. Il costo di `changeKey` è $O(\log n)$: $m \cdot O(\log n) = O(m \log n)$.

Da cui si conclude:

$$T(n, m) = O(n) + O(n \log n) + O(m \log n) = O(m \log n)$$

ricordando che $n - 1 \leq m \leq n^2$

13 Cammini minimi

13.1 Definizioni e proprietà di base

Sia $G = (V, E)$ un grafo con funzione peso $\omega : E \to \mathbb{R}$. Si danno definizioni e proprietà relative al problema del cammino minimo nel grafo.

Costo di un cammino Il costo di un cammino è la somma dei costi dei suoi archi. Formalmente il costo $\omega(\pi)$ del cammino $\pi = \{v_0, v_1, \cdots, v_k\}$ è:

$$\omega(\pi) = \sum_{i=1} k\omega(v_{i-1}, v_i)$$

il cammino formato da un solo vertice ha costo 0.

Cammino minimo $\pi^\star$ è detto cammino di costo minimo tra due vertici connessi x a y se e solo se:

- $\pi^\star$ è un cammino da x a y
- per ogni cammino π da x a y allora $\omega(\pi^\star) < \omega(pi)$.

Proprietà dei cammini minimi Valgono le seguenti proprietà:

- Sottostrutture ottima: Ogni sottocammino minimo in G è un cammino minimo.
- Se non ci sono cicli negativi, allora tra ogni coppia di vertici esiste sempre un cammino minimo semplice. Trovare un cammino minimo semplice in grafi con cicli negativi è un problema NP-completo.
- Se tutti i pesi sono positivi, allora ogni cammino minimo è semplice.

13.2 Distanza tra vertici

Si definisce distanza minima fra du vertici x e y il costo del camminimo minimo che li connette, o $+\infty$ se non esiste un cammino tra x e y. Si denota con d_{xy}.

Disuguagliana triangolare Siano $x, y, z \in V$ tre vertici, allora vale la diseguaglianza triangolare:

$$d_{xz} \leq d_{xy} + d_{yz}$$

dalla diseguaglianza triangolare si deriva la condizione di Bellman:

Condizione di Bellman Per ogni arco $(u, v) \in E$ e per ogni vertice $s \in V$ le distanze tra vertici soddisfano la diseguaglianza:

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

Appartenenza a un cammino minimo Un arco $(u, v) \in E$ appartiene al cammino minimo tra i vertici s e v se e solo se vale l'ugaglianza nella condizione di Bellman, ovvero se u è raggiungibile da s e $d_{su} + \omega(u, v) = d_{sv}$

Tecnica del rilassamento Per calcolare un cammino minimo d_{xy} tra x e y si può applicare la tecnica del rilassamento. Si parte stimando una distanza per eccesso D_{xy} e a si aggiorna progressivamente fino a quando $D_{xy} = d_{xy}$. L'aggiornamento di D_{xy} avviene considerando un vertice v, il cammino π_{vy} e applicando il passo di rilassamento:

$$\text{IF } D_{xv} + \omega(\pi_{vy}) \leq D_{xy} \text{ THEN } D_{xy} \leftarrow D_{xv} + \omega(\pi_{vy})$$

13.3 Problemi sui cammini minimi

I problemi sui cammini minimi sono tre:

- Problema del cammino minimo fra una singola coppia: trovare il cammino minimo tra due vertici
- Problema del cammino minimo a sorgente singola: trovare i cammini minimi tra un vertice di partenza e tutti gli altri
- Problema del cammino minimo fra tutte le coppie: trovare i cammini minimi tra ogni coppia di vertici

13.4 Algoritmo di Floyd e Warshall

L'algoritmo di Floyd e Warshall risolve il problema di cammino minimo fra tutte le coppie di vertici

13.4.1 Strategia

Si utilizza una tecnica di programmazione dinamica.

Siano $V = \{v_1, v_2, \cdots, v_n\}$ gli n vertici del grafo. Si dice distanza k-vincolata e si scrive $d_{ij}^{(k)}$ il peso del cammino minimo che si può percorrere da v_i a v_j passando per i vertici v_i t.c:

$i \leq k$, o ∞ se non esiste un cammino. Si denota d_{ij} il cammino minimo tra v_i e v_j. Si osserva $d_{ij} = d_{ij}^{(n)}$. Quindi i problemi da risolvere sono:

- Il problema $\mathcal{P}$ è determinare d_{ij}.
- Il problema $\mathcal{P}(k)$ è determinare $d_{ij}^{(k)}$

allora $\mathcal{P} = \mathcal{P}(n)$.

Per risolve $\mathcal{P}(k)$:

per $k = 0$:

$$d_{ij}^{(0)} = \begin{cases} 0 & \text{se } i = j \\ \infty & \text{altrimenti} \end{cases}$$

per $k > 0$:

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

Lo schema è così dato:

```
ALGORITIMO FloydWarshall(Grafo G=(V, E, w)) -> Matrice
// La matrice D_i contiene la soluzione del problema P(i)
Matrici D_0[1..n, 1..n], ..., D_n[1..n, 1..n]

// soluzione k = 0
FOR i <- 1 TO n DO
    FOR j <- 1 TO n DO
        IF i = j THEN D_0[i, j] <- 0
        ELSE IF (V_i, V_j) IN E THEN D_0[i, j] <- w(i, j)
        ELSE D_0[i, j] <- infty

// soluzione k > 0
FOR k <- 1 TO n DO
    km <- k - 1
    FOR i <- 1 TO n DO
        FOR j <- 1 To n DO
            D_k[i, j] = min(d_km[i, j], d_km[i, k] + d_km[k, j])
RETURN D_n
```

L'algoritmo è corretto anche in caso di pesi negativi, senza cicli negativi. Si ricorda che in quest'ultimo caso il problema è mal posto.

13.4.2 Implementazione

```
ALGORITIMO FloydWarshall(Grafo G=(V, E, w)) -> Matrice
Matrice D[1..n, 1..n]

// soluzione k = 0
```

```
FOR i <- 1 TO n DO
    FOR j <- 1 TO n DO
        IF i = j THEN D[i, j] <- 0
        ELSE IF (V_i, V_j) in E THEN D[i, j] <- w(i, j)
        ELSE D[i, j] <- infty

FOR k <- 1 TO n DO
    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]
RETURN D
```

Per ricavare i vertici attraversati del cammino tra V_i e V_j si utilizza una matrice ausiliaria P, il cui contenuto in $P[i,j]$ è 0 se $i = j$ o non c'è alcun cammino tra V_i e V_j, l'indice del penultimo vertice sul cammino da V_i a V_j altrimenti. Ripercorrendo a ritroso si ottiene il cammino. L'algoritmo viene così modificato:

```
ALGORITIMO FloydWarshall(Grafo G=(V, E, w)) -> Matrice
Matrice D[1..n, 1..n]

// soluzione k = 0
FOR i <- 1 TO n DO
    FOR j <- 1 TO n DO
        P[i, j] <- 0
        IF i = j THEN
            D[i, j] <- 0
        ELSE IF (V_i, V_j) in E THEN
            D[i, j] <- w(i, j)
            P[i, j] <- i
        ELSE
            D[i, j] <- infty

FOR k <- 1 TO n DO
    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]
                P[i, j] <- P[k, j]
RETURN D
```

13.4.3 Analisi

Il tempo è $T(n) = \Theta(n^3)$, lo spazio $\Theta(n^2)$

13.5 Algoritmo di Bellman e Ford

L'algoritmo di Bellman e Ford permette di trovare il cammino minimo tra un vertice e tutti gli altri vertici.

Sia G un grafo privo di cicli negativi, allora un arco $(u, v) \in E$ appartiene al cammino minimo tra s e v se e solo se u è raggiungibile da s e $d_{su} + \omega(u, v) < d_{sv}$. Applicando questa proprietà:

```
ALGORITMO BellmanFord(Grafo G, Vertice S) -> Vettore
Vettore d // d[v] = peso del cammino minimo da S a v sinora trovato
d[S] <- 0
FOR EACH v IN V-{S} DO d[v] <- infty

FOR k <- 0 TO n - 1 DO
    FOR EACH (u, v) IN E DO
        IF d[u] + w(u, v) < d[v] THEN
            d[v] <- d[u] + w(u, v)

RETURN d
```

Il tempo è $\Theta(n \cdot m)$.

13.6 Algoritmo di Dijkstra

L'algoritmo di Dijkstra permette di calcolare il cammino minimo tra un vertice `s` e tutti gli altri. Risolve quindi lo stesso problema dell'algoritmo di Bellman e Ford, ma con complessità algoritmica inferiore.

13.6.1 Strategia

Si definisce un vettore `d[v]` che contiene la distanza tra `s` e `v`. Procedendo con strategia greedy si considera l'insieme dei candidati come i vertici del grafo. Ad ogni passo si preleva il vertice `u` con `d[u]` minimo, a questo punto `d[u]` diventa definitivo e si aggiorna `d[v]` per ogni vertice `v` adiacente a `u`.

```
ALGORITMO Dijkstra(Grafo G, Vertice S) -> Vettore
Vettore d
d[s] <- 0
FOREACH v IN V-{s} DO d[v] <- infty
C <- V
WHILE C non è vuoto DO
    u <- elemento in C con d[u] minore
    C <- C-{u}
    FOREACH (u, v) IN E DO
        IF d[u] + w(u, v) < d[v] THEN
            d[v] <- d[u] + w(u, v)
RETURN d
```

L'algoritmo restituisce i cammini minimi se e solo se non contiene archi con peso negativo.

13.6.2 Implementazione

Il grafo è rappresentato come una lista di adiacenza o incidenza.

L'insieme dei candidati è rappresentato con una coda con priorità a cui i vertici `v` è assegnata la priorità `d[v]`. Il grafo si rappresenta con una lista di adiacenza o di incidenza.

Per ricavare i percorsi, si costruisce un albero dei cammini minimi col vettore `pred[v]`.

```
ALGORITMO Dijkstra(Grafo G, Vertice S) -> Vettore
Vettore d, pred
d[s] <- 0
FOREACH v IN V-{s} DO d[v] <- infty
Coda_con_priorità C
FOREACH v IN V DO C.Insert(v, d[v])
WHILE NOT C.IsEmpty() DO
    u <- C.DeleteMin()
    FOREACH (u, v) IN E DO
        IF d[u] + w(u, v) < d[v] THEN
            d[v] <- d[u] + w(u, v)
            pred[v] <- u
            C.ChangeKey(v, d[v])
RETURN d
```

13.6.3 Analisi

L'inizializzazione del vettore `d` e della coda con priorità `C` richiedono $O(n)$ passi. Il ciclo while esegue n iterazioni:

- Estrarre `u` ha costo logaritmico, per n volte $O(n \log n)$
- Successivamente si scorrono tutti gli archi che partono da u, di conseguenza ogni arco viene preso solo una volta poichè `u` viene poi eliminato. Di conseguenza il tempo è $O(m)$
- Per ogni arco si potrebbe eseguire l'operazione di modifica sulla coda con priorità cche ha costo logaritmico, per m volte quindi $O(m \log n)$.

Il tempo totale è:

$$O(n) + O(n) + O(n \log n) + O(m) + O(m \log n) = O(n \log n) + O(m \log n) = O(m \log n)$$

ricordando che se G è connesso allora $m \geq n - 1$.

14 Alberi di ricerca

Si osservano ora altre strutture per implementare i dizionari.

14.1 Alberi binari di ricerca

Un albero binario di ricerca è un albero binario tale che, per ogni nodo n:

- Il valore di ogni chiave contenuta nel sottoalbero sinistro di n è minore della chiave contenuta in n

- Il valore di ogni chiave contenuta nel sottoalbero destro di n è maggiore o uguale della chiave contenuta in n.

14.1.1 Operazioni sugli alberi binari di ricerca

14.1.1.1 Ricerca

Soluzione ricorsiva:

```
FUNZIONE trova(AlberoRicerca r, Chiave k) -> Nodo
IF r = null THEN
    RETURN null
ELSE
    IF r.chiave < k THEN
        RETURN trova(r.sx, k)
    ELSE IF r.chiave > k THEN
        RETURN trova(r.dx, k)
    ELSE // r.chiave = k
        RETURN r
```

Soluzione iterativa:

```
FUNZIONA trova(AlberoRicerca r, Chiave k) -> Nodo
n <- r
WHILE n != nill AND n.chiave != k
    IF k < .chiave THEN
        n <- n.sx
    ELSE
        n <- s.dx
RETURN n
```

14.1.1.2 Inserimento

Soluzione ricorsiva:

```
FUNZIONE inserisci(RicercaBinaria r, Elemento d) -> AlberoRicerca
k <- d.chiave
IF r = null THEN
    r <- nuovoNodo()
    r.campi <- d.campi
    r.chiave <- d.chiave
    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
```

Soluzione iterativa:

```
FUNZIONE inserisci(AlberoRicerca r, Elemento d) -> AlberoRicerca
k <- d.chiave
t <- nuovoNodo()
t.chiave <- k
```

```
t.campi <- d.campi
t.sx <- null
t.dx <- null
padre <- null
n <- r
WHILE n != null DO
    padre <- n
    // trova posizione
    IF k < n.chiave THEN
        n <- n.sx
    ELSE
        n <- n.dx
    IF padre = null THEN
        r <- t
    ELSE IF k < padre.chiave THEN
        padre.sx <- t
    ELSE
        padre.dx <- t
RETURN t
```

14.1.1.3 Cancellazione

14.1.2 Prestazioni

Le operazioni di `search`, `delete` e `insert` sono effettuate in tempo $O(h)$. Si ricorda che negli alberi binari con n nodi vale $log_2 n \leq h < n$ quindi le operazioni vengono eseguite in tempo $O(n)$. Se l'albero fosse però bilanciato, allora le operazioni si potrebbero eseguire in tempo $O(\log n)$.

14.2 Alberi binari perfettamente bilanciati

Albero perfettamente bilanciato Un albero binario è detto perfettamente bilanciato se e solo se per ogni nodo la differenza in valore assoluto tra i numeri di nodi presenti nei suoi sottoalbero sinistro e destro è al massimo 1.

14.2.1 Proprietà numeriche degli alberi perfettamente bilanciati

In un albero binario di altezza h il numero massimo di nodi è dato dall'albero completo dunque $2^{h+1} - 1$; il numero minimo di nodi è 2^h.

Dimostrazione limite inferiore Induzione su h

- Base $h = 0$: l'albero di altezza 0 è quello con solo la radice, per cui $1 \geq 2^0$
- Passo $h - 1 \implies h$: sia T un albero perfettamente bilanciato con due sottoalberi T' e T''. Sia $h_{T'} = h - 1$ quindi $T`$ ha almeno 2^{h-1} nodi. Per ipotesi l'albero è perfettamente bilanciato, quindi T'' può differire da T' di al massimo un nodo, quindi ha almeno $2^{h-1} - 1$ nodi. In totale vi sono almeno $1 + 2^{h-1} + 2^{h-1} - 1 = 2 \cdot 2^{h-1} = 2^h$.

Poichè vale $2 \leq n < 2^{h+1}$ allora $h = \lfloor \log_2 n \rfloor$.

14.3 Alberi binari bilanciati in altezza (alberi AVL)

Passare da un albero binario di ricerca a un albero perfettamente bilanciato costa molto. Questi hanno senso solo se non si fanno inserimenti e si hanno già tutte le chiavi.

Albero binario bilanciato in altezza (o AVL) Un albero binario è detto bilanciato in altezza (o AVL) se e solo se per ogni nodo la differenza in valore assoluto tra le altezze dei suoi sottoalberi sinistro e destro è al massimo 1.

Se un albero è perfettamente bilanciato allora è bilanciato.

Fattore di bilanciamento Il fattore di bilanciamento di un nodo v è la differenza di altezza tra il sottoalbero sinistro e quello destro.

14.3.1 Proprietà numeriche e prestazioni

Fissata un'altezza h, in un albero AVL:

- il numero massimo di nodi è dato dall'albero completo, dunque $n2^{h+1} - 1$.
- il numero minimo di nodi è dato dall'albero di Fibonacci

Albero di Fibonacci Un albero di Fibonacci è un albero AVL con minimo numero di nodi.

Per ricavare n_h:

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

Si dimostra per induzione che in un albero di fibonacci alto h il numero di nodi è $n_h = F_{h+3} - 1$:

- Base $h = 0$: $n_0 = F_3 - 1 = 2 - 1 = 1$
- Base $h = 1$: $n_1 = F_4 - 1 = 3 - 1 = 2$
- Passo $h - 1 \implies h$: $n_h = n_{h-1} + n_{h-2} + 1 = F_{h+2} - 1 + F_{h+1} - 1 + 1 = F_{h+3} - 1$

Ricordando che:

$$F_n \approx \frac{\phi^n}{\sqrt{5}}$$

allora:

$$\begin{aligned} n_h &\approx \frac{\phi^{h+3}}{\sqrt{5}} - 1 \\ \sqrt{5} n_h &\approx phi^{h+3} \\ h + 3 &\approx \log_\phi(\sqrt{5} n) \\ h &\approx \log_\phi \sqrt{5} \log_\phi n - 3 \end{aligned}$$

si può dimostrare che $h = 1.44 \log_2 n$ quindi $h = \Theta(\log n)$. Si conclude che le operazioni sul dizionario rappresentato con albero AVL avvengono in tempo logaritmico.

14.3.2 Bilanciamento di alberi AVL

Sia v un nodo con fattore di bilanciamento ± 2, allora esiste un sottoalbero T di v con altezza tale da sbilanciare il nodo. A seconda della posizione di T, possono esserci quattro casi:

- Sinistra - snistra (SS): T è il sottoalbero sinistro del sottoalbero sinistro di v
- Destra - destra (DD): T è il sottoalbero destro del sottoalbero destro di v
- Sinistra - destra (SD): T è il sottoalbero destro del figlio sinistro di v
- Destra - sinistra (DS): T è il sottoalbero sinistro del figlio destro di v

Si possono applicare delle operazioni dette rotazioni, che sono simmetrice tra SS e DD e tra SD e DS, pertanto si vedono solo due casi per ribilanciare v:

- Rotazione SS: per ribilanciare v basta applicare una rotazione semplice verso destra con perno in v.
- Rotazione SD: richiede una rotazione doppia. Sia z il figlio sinistro di v, la rotazione SD applica una prima rotazione verso sinitra con perno in z e successivamente una rotazione a destra con perno in v.

Una rotazione SS, DD, SD o DS applicata a un nodo v con fattore di bilanciamento ± 2 fa decrescere di 1 l'altezza del sottoalbero radicato in v prima della rotazione.

14.4 Alberi 2-3

Gli alberi 2-3 permettono di mantenere il bilanciamento senza utilizzare le rotazioni.

Albero 2-3 Un albero 2-3 è un albero in cui:

- Ogni nodo interno ha 2 o 3 figli
- Tutte le foglie si trovano allo stesso livello

Alberi 2-3 di ricerca Gli alberi 2-3 sono impiegati come alberi di ricerca inserendo le chiavi nelle foglie in maniera crescente da sinistra a destra, i nodi interni contengono informazioni di instradamento:

- se il nodo ha 2 figli contiene la chiave pià grande del sottoalbero sinistro;
- se il nodo ha 3 figli contiene la chiave più grande del sottoalbero sinistro e la chiave più grande del sottoalbero centrale.

Per inserimenti e cancellazioni è utile mantenere anche un puntatore al padre.

14.4.1 Proprietà numeriche e prestazioni degli alberi 2-3

Si da un albero 2-3 con n nodi, altezza h e f foglie, allora valgono le seguenti relazioni:

- Numero di nodi $2^{h+1} - 1 \leq n \leq (3^{h+1} - 1)/2$
- Numero di foglie f: $2^h \leq f \leq 3^h$

quindi $\log_3 f \leq h \leq log_2 f$ quindi l'altezza è logaritmica rispetto al numero di foglie: $h = \Theta(\log f)$.

14.4.2 Bilanciamento di alberi 2-3

Durante l'inserimento bisogna mantenere l'albero bilanciato. Si supponga di inserire un elemento e di chiave k, allora si crea un nuovo nodo u e si cerca la posizione in cui inserirlo cercando la chiave k nell'albero. Si trova così un nodo v al penultimo livello a cui andrebbe attaccato u. A questo punto possono accadere due cose:

- v ha due figli: u può essere correttamente inserito come figlio sinistro, destro o centrale.
- v ha tre figli: u non può essere correttamente inserito come figlio di v, quindi si esegue un'operazione chiamata split. Si divide v in due, creando un nuovo nodo w. Considerando anche u, si attaccano a v i due nodi di chiave minima e a w i due nodi di chiave massima. Infine, w diventa figlio del padre di v. Se il padre di v aveva due figli l'operazione è finita, altrimenti bisogna procedere con altri split verso l'alto.

Poichè ciascuno split richiede tempo costante e durante un inserimento si possono effettuare un numero di split massimo pari all'altezza dell'albero, allora il tempo dell'inserimento è $O(\log f)$.

14.5 B-Alberi

I B-alberi permettono di accedere a dizionari in memoria secondaria. Tenendo conto che:

- L'accesso ai dati e più lento che in memoria centrale.
- L'accesso è a blocchi, quindi bisogna garantire località nell'accesso ai dati.

Dunque l'accesso al singolo dato è più lento, ma in un unico accesso si ottengono molti dati. Diminuendo l'altezza dell'albero e aumentando la quantità di elementi per ogni nodo, allora si hanno pochi accessi a memoria e si sfrutta l'intera dimensione dei blocchi.

Definizione Un B-albero di ordine t (grado minimo t) è un albero T tale che T è vuoto oppure è strutturato come segue:

- Tutte le foglie hanno la stessa profondità
- Ogni nodo v diverso dalla radice mantiene k chiavi ordinate con $t-1 \leq k \leq 2t-1$; la radice contiene k chiavi ordinate con $1 \leq k \leq 2t-1$
- Ogni nodo interno ha $k+1$ figli
- Le chiavi k_i di v separano gli intervalli di chiavi memorizzate in ciascun sottoalbero T_i di v: sia c_i una chiave qualsiasi nel sottoalbero T_i allora $c_1 \leq k_1 \leq c_2 \leq k_2 \leq \cdots \leq c_i \leq k_i \leq c_{i+1}$

dunque, mentre negli alberi 2-3 i nodi interni contengono solamente informazioni per l'instradamento, nei B-alberi servono sia a memorizzare i dati sia per l'instradamento. Solitamente si sceglie t in modo che sia contenuto interamente in un nodo.

Figura 14.1: Un B-albero di ordine $t = 3$

14.5.1 Proprietà numeriche

Si trova il numero minimo di nodi in funzione dell'altezza. Dalla definizione la radice ha almeno 1 chiave quindi 2 figli; un nodo interno ha almeno $t-1$ chiavi quindi t figli. Quindi

ad altezza h vi sono $2t^{h-1}$ nodi. Ogni nodo ha $(t-1)$ chiavi, quindi data l'altezza h ci sono un numero minimo di nodi n:

$$n \geq 1 + \sum_{i=0}^{h-1} 2t^i(t-1) = 1 + 2(t-1)\sum_{i=0}^{h-1} t^i = 1 + 2(t-1)\frac{t^h - 1}{t-1} = 2t^h - 1$$

ovvero:

$$t^h \leq \frac{n+1}{2}$$
$$h \leq \log_t \frac{n+1}{2}$$

quindi se ci sono n chiavi l'altezza è massimo $\log_t \frac{n+1}{2}$.

14.5.2 Ricerca

La ricerca è simile a quella binaria, solo che invece di andare a destra o a sinistra bisogna procedere controllando le $t-1 \leq k$ $leq 2t-1$ chiavi, quindi per scendere di un livello si eseguono $\Theta(t)$ passi, che diventano $\Theta(\log t)$ passi. Per scendere fino all'ultima foglia quindi i passi sono $\Theta(h \log t)$. Poichè l'altezza è logaritmica rispetto al numero di nodi, allora i passi sono $\Theta(\log n \log t) = \Theta(\log n)$ indipendetemente da t.

14.5.3 Inserimento

Per l'inserimento va ricercata la foglia in cui l'elemento va inserito. Vi sono poi diversi casi:

- Caso 1: C'è spazio nelle foglie e quindi si inserisce l'elemento nella posizione corretta.
- Caso 2: Non c'è spazio nella foglia ovvero la foglia ha $2t-1$ chiavi. Si esegue uno split in due foglie da t e $t-1$ chiavi e la nuova chiave viene inserita nel padre.

Inserimento quindi può richiedere un tempo $\Theta(t \log n)$.

15 Tabelle hash

Una tabella di hash è una struttura dati che permette di implementare un dizionario. L'idea generale è utilizzare una tabella indicizzata e applicare una funzione alla chiave per ottenere il suo indice. La funzione è detta funzione di hash $h : U \to \{0, \cdot, m-1\}$ ed associa ad ogni elemento dell'universo delle chavi uno degli m indici.

Fattore di carico Il fattore di carico di una tabella di hash è il rapporto $\alpha = n/m$ tra gli n elementi memorizzati e la sua dimensione m.

15.1 Tabelle ad accesso diretto

La tabella di hashing più semplice è la tabella ad accesso diretto. Si consideri il caso in cui le chiavi degli n elementi siano interi nell'intervallo $[0, m)$ con $n \leq m$, allora se c'è un elemento elem di chiave k si accede indicizzando un array $v[k] = \text{elem}$. Nel caso in cui m sia molto più grande di m vi è però un grande spreco di memoria cioè un fattore di carico basso.

15.2 Definire funzioni di hash

Funzione di hash perfetta Una funziona di hash h è detta perfetta se è iniettiva, cioè se $k_0 \neq k_1 \implies h(k_0)! = h(k_1) \forall k_0, k_1 \in U$.

Spesso l'universo delle chiavi è molto grande, di conseguenza costruire funzioni di hash perfette richiede molta memoria. Una funziona di hash non perfetta potrebbe associare la stessa posizione ad elementi diversi, dunque bisogna prevedere tecniche di gestione delle collissioni. Un modo per uniformare le collisioni è sparpagliare gli indici, ovvero uniformare la funzione.

Uniformità Sia h una funzione di hash, sia $\mathcal{P}(x)$ la probabilità che si scelga una chiave $x \in U$, allora h si dice uniforme se la probabilità $Q(i)$ che un elemento sia associato alla posizione i è:

$$Q(i) = \frac{1}{m}$$

ovvero se distribuisce uniformemente le chiavi nello spazio degli indici.

Una buona funzione ha poche funzioni di hashing ed è quindi uniforme. Deve inoltre essere veloce da calcolare. Bisognerebbe sapere la probabilità con cui ogni chiave potrebbe apparire, ma di solito non è così. Si vedono ora metodi generali per costruire funzioni di hash con una buona uniformità. Prima si osserva che si può sempre ricondurre una chiave non intera a una intera, guardando la sua rappresentazione: sia y una chiave non intera rappresentata dai bit $y_0, \cdots y_l$ con $y_i \in \{0, 1\}$, allora si associa ad y il numero:

$$b(y) = \sum_{i=0}^{l} y_i \cdots 2^i$$

Metodo della divisione Il metodo della divisione associa alla chiave l'indice dato dal resto della divisione tra chiave e la dimensione m della tabella.
In genere una tabella di hash ha come dimensione una potenza di 2. Di conseguenza, il risultato dipende solo da una parte dei bit. La funzione quindi è facile da calcolare ma non uniforme. Un'alternativa è scegliere come m un numero primo molto vicino a una potenza di 2.

Metodo del ripiegamento Il metodo del ripiegamento suddivide i bit della rappresentazione della chiave in parti $k_1, \cdots, k_l$ e applica ai k_i una funzione $f : U^l \to \{0, \cdots, m\}$, ovvero:

$$h(k) = f(k_1, \cdots, k_l)$$

ad esempio, si potrebbe applicare l'operazione di XOR tra i k_i oppure applicare il metodo della divisione ad ogni k_i e concatenare i risultati.

15.3 Gestione delle collissioni

Le tecniche di gestione delle collissioni possono essere:

- Esterne: utilizzano altre tabelle.
- Interne: utilizzano la tabella stessa.

15.3.1 Liste di collissione (concatenamento esterno)

Invece che associare ad ogni cella della tabella un solo elemento, si associa una lista che contiene le chiavi che collidono. Solitamente si utilizza una lista concatenata non ordinata.

La lunghezza media di una lista di collissione è pari al fattore di carico, pertanto assumendo che la funzione di hash sia uniforme le operazioni di ricerca ed eliminazione hanno tempo $T_{\text{avg}}(n, m) = O(1 + n/m)$. L'inserimento ha sempre tempo costante.

Le liste di collissione permettono fattori di carico maggiori di uno e che una chiave può occorrere più volte.

Si osserva che per $m = 1$ allora tutte le chiavi sono in una sola lista, allora la struttura dati diventa una lista con ricerca sequenziale e tempo $T(n, m) = O(n\ m = O(n)$; viceversa per $m = |U|$ allora si può usare una funzione di hash perfetta con $T(n, m) = O(1)$.

15.3.2 Indirizzamento aperto

L'indirizzamento aperto permette a un elemento con una certa chiave di occupare l'indice di un'altra chiave.

Durante l'inserimento di una chiave k, se l'indice $h(k)$ è già occupato allora si cerca un'altra posizione libera. La sequenza delle posizioni da cercare è data dalla funzione $c(k, i)$, in particolare la sequenza di elementi analizzata è data da $c(k, 0) = h(k), c(k, 1), \cdots, c(k, m-1)$. La funzione c deve rispettare due requisiti:

- $c(k, 0) = h(k)$
- $\{c(k, 0), c(k, 1), \cdots, c(k, m-1)\} = \{0, 1, \cdots, m-1\}$ ovvero deve garantire che tutti gli indici siano analizzati così che se vi è almeno un posto libero l'elemento viene inserito

Scansione lineare Un metodo semplice di generare la sequenza si ottiene usando una scansione lineare, ovvero una funzione:

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

Questo metodo porta a un fenomeno di agglomerazione primaria: le chiavi k che devono posizionarsi nello stesso posto tenderanno a formare un gruppo che si allungherà di 1.

Scansione quadratica Un altro metodo è la scansione quadratica, data dalla funzione:

$$c(k, i) = \lfloor (h(k) + c_1 i + c_2 i^2) \mod m \rfloor$$

con c_1, c_2 ed m opportuni. Ad esempio per $c_1 = c_2 = 1/2$ e m potenza di 2 si può dimostrare che vengono scansionati tutti gli indici.
La scansione quadratica distribuisce le chiavi in maniera da evitare l'agglomerazione primaria, ma si ha una fenomeno di agglomerazione secondaria: due chiavi k_1 e k_2 tali che $h(k_1) = h(k_2)$ porteranno a sequenze identiche.

Hashing doppio Per evitare l'agglomerazione secondaria si può usare una seconda funzione di hash in modo tale che, nel caso ideale in cui $h(k_1) \neq h(k_2) \implies h'(k_1) \neq h'(k_2)\ \forall k_1, k_2 \in U$ allora la sequenza sia diversa:

$$c(k,i) = (h(k) + ih'(k)) \mod m$$

con $h' : U \to \{0, \cdots, m-1\}$
Poichè l'hashing doppio funzioni m e $h'(k)$ devono essere primi tra di loro.

15.4 Analisi delle prestazioni

I costi delle operazioni sul dizionario dipendono dal costo di scansione, ovvero dalle posizioni da controllare.

Nel caso peggiore si arriva a $O(n)$ passi. Se la funzione di hash è uniforme allora il numero di passi medio è:

15.5 Re-hashing

Quando la tabella di hash supera un certo fattore di carico bisogna crearne una più grossa (solitamente il doppio) in cui ricollocare gli elementi. I problemi principali di questa operazione sono la definizione della nuova funzione di hash e i costi della stessa.

La funzione di hash si può definire come $h(k) = f(k) \mod m$, se f è uniforme allora lo è anche h.

Si fa ora un'analisi di costo ammortizzato. Si da una tabella di hash di dimensione iniziale m con fattore di carico massimo constenito $\alpha = 1/2$, dunque l'operazione di rehashing avviene ogni qualvolta la tabella sia piena al 50%. Si indica con T_i la tabella con $i = 0, 1, \cdots, k$ operazioni di rehashing allora: T_0 ha dimensione m e fattore di carico massimo $m/2$, T_1 ha dimensione $2m$ e fattore di carico massimo m; T_2 ha dimensione $4m$ e fattore di carico massimo $2m$; allra T_k ha dimensione $2^k m$ e fattore di carico massimo $2^{k-1}m$. Dunque per N inseriento occorrono k rehashing calcolati come:

$$\begin{aligned}
&2^{k-2}m < N \leq 2^{k-1}m \\
&2^{k-2} < \frac{N}{m} \leq 2^{k-1} \\
&k - 2 < \log_2 \frac{N}{m} \leq k - 1 \\
&k = 1 + \lceil \log_2 \frac{N}{m} \rceil
\end{aligned}$$

Si osserva che le chiavi da trasferire da T_i a T_{i+1} sono $2^{i-1}m$, quindi da T_0 a T_k:

$$\sum_{i=0}^{k-1} 2^{i-1}m =$$
$$m\sum_{i=0}^{k-1} 2^{i-1} =$$
$$m\sum_{i=0}^{k-1} 2 \cdot 2^{i} =$$
$$\frac{m}{2}\sum_{i=0}^{k-1} =$$
$$\frac{m}{2}(2^k - 1) =$$
$$\frac{m}{2}(2^{1+\lceil \log_2 \frac{N}{m} \rceil} - 1) =$$
$$\frac{m}{2}(2^2\frac{N}{m} - 1) \leq \frac{m}{2}2^2\frac{N}{m} =$$
$$2N$$

dunque: per N inserimenti vi sono $< 2N$ inserimenti dovuti a rehashing per un totale di $3N$ inserimenti, ricordando che un inserimento è costante allora il tempo totale per gli inserimenti è $O(N)$ che applicando il costo ammortizzato si riduce a:

$$T_{\text{a}}(N) = \frac{O(N)}{N} = O(1)$$

16 Teoria della NP-Completezza

16.1 Caratterizzazione e classificazione dei problemi

Si definisce problema una relazione $\pi \subseteq I \times S$ che associa all'universo delle possibili istanze I (input) l'universo delle soluzioni S. Quindi $(x, s) \in \pi$ se e solo se s è una soluzione del problema π con input x. In alternativa si può vedere π come un predicato P che data un'istanza $x \in I$ restituisce 1 se e solo se $(x, s) \in P$, 0 altrimenti.

Si da una classificazione dei problemi:

- Ricerca: data un'istanza $x \in I$ trovare $s \in S$ t.c: $(x, s) \in \pi$. Esempi di problemi di ricerca sono trovare un albero ricoprente o un cammino tra due nodi (es. albero ricoprente).
- Ottimizzazione: data un'istanza $x \in I$, trovare $s^* \in S$ t.c. sia migliore, secondo un criterio definito, di ogni altra soluzione $s \in S$. Un esempio di problema di ottimizzazione è il cammino minimo tra due nodi, dove il criterio per definire la bontà di una soluzione è la somma dei pesi di ogni arco attraversato (es. albero ricoprente minimo).
- Decisione: data un'istanza $x \in I$ richiede una risposta binaria, quindi $S = \{0, 1\}$. In particolare si chiede se x soddisfa una certa proprietà. Si può creare una partizione di π composta da $(x, 1) \in \pi$ dette istanze positive e $(x, 0) \in \pi$ dette istanze negative.

Non vi possono essere contraddizzioni: o $\pi(x) = 0$ o $\pi(x) = 1$. Esempi di problemi di decisione sono decidere se un numero è primo o se un grafo è connesso (es. esiste per un grafo l'albero ricoprente di peso massimo k?).

Ci interessano i problemi di decisione perchè per questi non bisogna spendere tempo a restituire la risposta, pertanto nei prossimi capitoli si farà spesso riferimento a questi. (es. il problema di trovare una cricca è un problema di ottimizzazione, stabilire se un grafo contiene una cricca di k vertici è un problema di decisione).

16.2 Classi di complessità

Complessità computazionale Lo studio della complessità computazionale si occupa di classificare i problemi in base alle risorse utilizzate per la loro soluzione.

Classi di complessità Si definisce classe di complessità l'insieme di problemi che possono essere risolti utilizzando la stessa quantità di una determinata risorsa.

Problemi di decisione e classi di complessità Sia $f(n)$ una funzione, si chiamano TIME($f(n)$) e SPACE($f(n)$) i problemi di decisione che possono essere risolti rispettivamente in tempo e spazio $f(n)$.

Si introducono ora diverse classi di problemi, dette classi unione, ottenute dall'unione dei problemi in TIME e SPACE:

- P è la classe dei problemi di decisione risolubili in tempo polinomiale:

$$\mathrm{P} = \bigcup_{c=0}^{\infty} \mathrm{TIME}(n^c)$$

- PSPACE è la classe dei problemi di decisione risolubili con spazio polinomiale:

$$\mathrm{PSACE} = \bigcup_{c=0}^{\infty} \mathrm{SPACE}(n^c)$$

- EXPTIME è la classe dei problemi di decisione risolubili in tempo esponenziale:

$$\mathrm{EXPTIME} = \bigcup_{c=0}^{\infty} \mathrm{TIME}((2^c)^n)$$

Relazioni spazio/tempo Vale:

- P $\subseteq$ PSPACE: si osserva che un algoritmo polinomiale può al più accedere a uno spazio polinomiale.
- PSPACE $\subseteq$ EXPTIME: si assume che le locazioni di memoria siano binarie, di conseguenza n^c locazioni di memoria posso trovarsi al più in 2^{n^c} stati.

Si conclude che P $\subseteq$ PSPACE $\subseteq$ EXPTIME. Si è dimostrato che P $\neq$ EXPTIME quindi o P $\neq$ PSPACE o PSPACE $\neq$ EXPTIME, ma ancora non si sa.

16.3 Algoritmi non deterministici e classe NP

Problema di siddisfacibilità Il problema di soddisfacibilità (SODD) ha come istanza una formula booleana ϕ in forma normale congiunta con insieme di variabili V, e ci si chiede

se esiste un'assegnazione alle variabili in V che rendono vera ϕ. Quindi è un problema di decisione.

L'unico modo per risolvere SODD è tentare tutte le combinazioni. Data ϕ con n variabili allora vi sono 2^n combinazioni quindi il tempo è $O(2^n)$. Quindi SODD $\in$ EXPTIME.

Algoritmi non deterministici Un algoritmo non deterministico ammette un'istruzione indovina $\in \{0, 1\}$ che può rispondere sì/no.

SODD può essere così risolto in tempo polinomiale.

```
ALGORITMO sodd(Formula ph(x_1...x_n)) -> Boolean
FOR i <- 1 TO n DO
    z_i <- indovina valore in {0, 1}
r <- ph(z_1...z_n)
RETURN r
```

Classe NP La classe NP è la classe dei problemi di decisione risolubili in tempo polinomiale da algoritmi non deterministici e verificabili in tempo polinomiale da algoritmi deterministici.

Si osserva che un problema risolubile in tempo polinomiale si può vedere come risolubile da un algoritmo non deterministico che non usa l'istruzione indovina. Quindi P $\subseteq$ NP. Non si sa se P = NP.

Si osserva che SODD $\in$ NP.

NOTA: In NP, la N sta per "Non deterministico", mentre la P sta per "Polinomiale".

Classe NPSPACE La classe NPSPACE è la classe dei problemi di decisione risolubili in spazio polinomiale da algoritmi non deterministici.

Similmente NP $\subseteq$ NPSPACE, di conseguenza P $\subseteq$ NP $\subseteq$ NPSPACE. Si è dimostrato che PSPACE = NPSPACE, quindi:

$$\text{P} \subseteq \text{NP} \subseteq \text{PSPACE} \subseteq \text{NPSPACE}$$

16.4 NP-Completezza

Riducibilità di problemi Siano $\pi_1 : I_1 \to \{0,1\}$ e $\pi_2 : I_2 \to \{0,1\}$ due problemi di decisione, si dice che π_1 è riducibile in π_2 se e solo se esiste una funzione $f : I_1 \to I_2$ t.c. $\forall x \in I_1\ \pi_1(x) = \pi_2(f(x))$. Quindi se esiste f e un algoritmo a_2 per risolvere π_2, si può applicare f a I_1 e poi a_2 con input $f(I_1)$.

Riducibilità polinomiale Si dice che un problema π_1 è riducibile polinomialmente in un altro problema π_2 se e solo se esiste una riduzione $f : I_1 \to I_2$ calcolabile da un algoritmo in tempo polinomiale. Si denota con $\pi_1 \leq_P \pi_2$. Valgono le seguenti proprietà:

- $\pi_1 \leq_P \pi_2 \wedge \pi_2 \in \text{P} \implies \pi_1 \in \text{P}$.
- $\pi_1 \leq_P \pi_2 \wedge \pi_2 \leq_P \pi_3\ implies \pi_1 \leq_P \pi_3$.

Esempio di riduzione: da SODD a Clique Sia ϕ una forma normale congiuntiva, allora si trasforma in un grafo $G = (V, E)$ con $v = \{\text{occorrenza letterali in } \phi\}$ e l'insieme degli archi definito come $(u, v) \in E$ se e solo se u e v sono in clausole differenti e non sono il negato dell'altro..

Problemi NP-Difficili Un problema π si dice NP-Difficile se e solo se ogni $\pi' \in$ NP è riducibile polinialmente a π. Cioè $\forall \pi' \in \text{NP } \pi' \leq_P \pi$.

Problemi NP-Completi Un problema π si dice NP-Completo se e solo se:

- $\pi \in NP$
- π è NP-Difficile

Teorema sui problemi NP-Completi Sia π un problema NP-Completo, allora:

- $\pi \notin \text{P} \implies \text{P} \neq \text{NP}$. Intuitivamente perchè $\pi \in \text{NP} \setminus \text{P}$.
- $\pi \in \text{P} \implies \text{P} = \text{NP}$. Per ipotesi π è NP-Completo e quindi NP-Difficile, quindi $\forall \pi' \in \text{NP } \pi' \leq_P \pi$ quindi se $\pi \in \text{P}$ allora $\pi' \in \text{P}$.

Teorema di Cook SODD è NP-Completo

Altri problemi NP-Completi sono: clique; cammino hamiltoniano; cammino più lungo; 3-SODD.

A Accenni matematica

A.1 Notazioni asintotiche

Siano $f, g : \mathbb{N} \to \mathbb{R}^+$ due funzioni, ci chiediamo quale cresce di più verso infinito. Si utilizzano le notazioni asintotiche:

- Limitazione superiore: $f(n)$ è O-grande di $g(n)$ e si scrive $f(n) = O(g(n)) \iff \exists c > 0, n_0 \in \mathbb{N} : \forall n > n_0 : f(n) \leq c \cdot g(n)$. Ovvero f cresce al massimo come g.
- Limitazione inferiore: $f(n)$ è Ω-grande di $g(n)$ e si scrive $f(n) = \Omega(g(n)) \iff \exists c > 0, n_0 \in \mathbb{N} : \forall n > n_0 : f(n) \geq c \cdot g(n)$. Ovvero f cresce almeno come g.
- Stesso ordine di grandezza: $f(n)$ è Θ-grande di $g(n)$ e si scrive $f(n) = \Theta(g(n)) \iff \exists c, d > 0, n_0 \in \mathbb{N} : \forall n > n_0 : c \cdot g(n) \leq f(n) \leq d \cdot g(n)$. Ovvero f cresce come g.

Per O e Θ valgono:

- $f(n) = O(g(n)) \implies \forall k > 0 k \cdot f(n) = O(g(n))$
- $f_1(n) = O(g_1(n)) \wedge f_2(n) = O(g_2(n)) \implies f_1(n) + f_2(n) = O(g_1(n) + g_2(n)) \wedge f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$

A.2 Partizione di un insieme

Sita A un insieme, si definisce partizione di A una famiglia di sottoinsiemi $a_1, \cdots, a_k \subseteq A$ t.c:

- $A_i \neq \emptyset$ per $i = 1, \cdots, k$
- $a_i \cap a_j = \emptyset \forall i \neq j$
- $a_1 \cup \cdots \cup a_k = A$

B Formule utili

Somma dei primi n naturali $\sum_{k=1}^{n} k = \frac{(n+1)\cdot n}{2}$

Logaritmi e notazioni asintotiche $\log_a(n) = \frac{\log_c(n)}{\log_c(a)}$ quindi $\log_a(10) \cdot \log_{10}(n) = O(\lg(n))$

Somma parziale di 2^n: $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$

Limitazione superiore per $\sum_{i=2}^{n-1} i\ln(i)$ $\sum_{i=2}^{n-1} i\ln(i) \leq \int_2^n i\ln(i)dx = \frac{n^2}{2}\ln(n) - \frac{n^2}{4}$

Dato n, trovare prossimo N potenza di 2 più vicino Si converte n in binario, si impostano tutti i bit a 0 e si inserisce un 1 a sinistra. Si osserva che $N \leq 2n$

Somma parziale di $n \cdot 2^n$: $\sum_{i=0}^{n} i2^i = (n-1)2^{n+1} + 2$. Si verifica per induzione.

Logaritmo iterato Si definisce $\log^{(i)}(n) = \log n \log n \cdots \log n$ per i volte. Si definisce poi $\log^*(n) = \min(\{i : \log^{(i)} n \leq 1\})$ ovvero il minimo numero di applicazioni del logaritmo per ottenere un numero minore o uguale a 1. Si osserva che $\log^* n$ cresce molto lentamente.

C Note sullo speudocodice

Indici degli array Il range degli indici dell'array viene indicato negli argomenti degli algoritmi solo se rilevante per l'algoritmo stesso. Altrimenti viene omesso.

Operatori logici Si assume venga usata la valutazione cortocircuitata.

Passaggio di parametri Per i tipi semplici il passaggio avviene per valore; per i tipi strutturati il passaggio avviene per riferimento.