Algoritmi e Strutture Dati

Indice

Capitolo 1

Introduzione

1.1 Algoritmi

Un **algoritmo** è una procedura di calcolo ben definita che prende un valore, o un insieme di valori, come **input** e genera un valore, o un insieme di valori, come **output**. Un algoritmo è quindi una sequenza di passi computazionali che trasforma l'input in output e che risolve un **problema computazionale** ben definito per ogni possibile istanza di input.

La descrizione di un problema computazionale specifica in termini generali la **relazione di ingresso/uscita** desiderata. L'algoritmo, quindi, descrive un modo per ottenere tale relazione.

Un algoritmo si dice **corretto** se, per ogni istanza di input, termina con l'output corretto. Si dice che un algoritmo corretto **risolve** un problema computazionale dato.

1.2 Strutture dati

Una **struttura dati** è un modo per memorizzare e organizzare i dati e semplificarne l'accesso e la modifica. Dunque consente un' organizzazione in memoria dei dati ed è caratterizzata da un insieme di operatori utilizzabili per:

- Manipolare la struttura. Quindi consentire operazioni come: aggiungere, rimuovere, modificare elementi e modificare la struttura stessa.
- Accedere alla struttura per esempio per leggere o cercare elementi.

Tra le strutture dati elementari ci sono: il **vettore di memoria** (memorizza elementi dello stesso tipo), gli **array** (vettori su più dimensioni) e i **record** (elementi di tipo diverso con nome proprio di cui non importa la posizione).
Una possibile implementazione degli array è la memorizzazione standard per righe attraverso la quale, data la matrice $A[R][C]$, il generico elemento $A[i][j]$ è memorizzato in $i \cdot \mathrm{C} + j$. Se invece molti elementi sono nulli è più conveniente utilizzare le matrici sparse in cui si memorizzano le triplette (i, j, val) per i soli elementi $A[i][j] \neq 0$. Nel primo caso, cioè se una matrice viene rappresentata per righe (o colonne), un matrice di n $\times$ n interi da 4 byte occupa $4 \cdot n^2$ byte; se invece viene memorizzata come matrice sparsa contenente v valori diversi da

zero occupa $v \cdot 3 \cdot 4$ byte. In realtà se i e j sono grandi serviranno più di 4 byte. Le triplette vengono memorizzate con il dizionario.

Vi sono poi altre strutture dati più avanzate come gli **insiemi**, la **pila (stack)** e la **coda**, il **dizionario** (implementabile come un array ordinato o un array non ordinato mantenuto con la tecnica del raddoppiamento-dimezzamento, consente operazioni quali inserimento, cancellazione e verifica dell'appartenenza di un elemento a un certo insieme) e il **grafo**.

Negli **insiemi** ogni elemento è rappresentato da un oggetto i cui attributi possono essere esaminati e manipolati e può contenere **attributi** e **dati satelliti**. Per alcuni tipi di insiemi dinamici si suppone che uno degli attributi dell'oggetto sia una **chiave** di identificazione. In informatica, a differenza dell'insieme matematico che è immutabile, un insieme manipolato da un algoritmo è una entità **dinamica** che cresce, si riduce, cambia nel tempo e che può avere elementi uguali tra loro (non a caso viene detto **insieme dinamico**). Si distinguono insiemi ordinati (sequenza) e insiemi dinamici. Possono essere realizzati con vettori booleani, con liste non ordinate o con liste ordinate. I vettori booleani sono semplici ed efficienti per le operazioni di unione, intersezione e per verificare se un elemento appartiene all'insieme, tuttavia hanno lo svantaggio di avere molte operazioni inefficienti e un'occupazione di memoria indipendente dalla dimensione dell'insieme. Le liste non ordinate hanno il vantaggio di un'occupazione di memoria proporzionale alla dimensione dell'insieme e di avere efficienti operazioni di inserimento sia in testa che in coda (questo è dovuto al fatto che la lista non è ordinata). Gli svantaggi sono invece legati alle operazioni di ricerca e cancellazione (perché si deve scorrere la lista per trovare ed eliminare un elemento) nonché alle operazioni insiemistiche di unione, intersezione e differenza. Infine le liste ordinate hanno come vantaggi l'occupazione di memoria proporzionale alla dimensione dell'insieme (come per le liste non ordinate) e le operazioni di unione, intersezione e differenza mentre hanno come svantaggi la ricerca, l'inserimento e la cancellazione.

Una **stringa** è un tipo particolare di insieme ordinato contenente caratteri alfanumerici sulla quale possono essere eseguite operazioni specifiche e che può avere diverse implementazioni

- Terminare con NULL, come in C
  Ma in questo caso non si potrebbe rappresentare NULL stesso
- Indicando prima la lunghezza, come in Pascal
  Questa implementazione ha la limitazione nel numero che codifica la lunghezza della stringa
- Con lunghezza fissa
  In questo caso si ha il vantaggio di poter eseguire un accesso diretto

Lo **stack** e le **code** sono insiemi dinamici dove l'elemento che viene rimosso dall'operazione DELETE è predeterminato; questo perché lo stack segue una politica LIFO (l'elemento cancellato è quello inserito per ultimo) e la coda una politica FIFO (l'elemento cancellato è quello inserito per primo). Possono essere implementate come liste bidirezionali, vettori o array circolari.

Un **dizionario (Map)** è una relazione univoca che associa ad ogni elemento di un insieme A un solo elemento di un insieme B ma non viceversa. In particolare, è un insieme S di coppie ($chiave$, $valore$) caratterizzato da:

- Una relazione $R : D \rightarrow C$
- Un dominio D rappresentante le chiavi
- Un codominio C rappresentante i valori

I **grafi** e **alberi**, infine, sono una rappresentazione grafica di una interrelazione tra oggetti. Un grafo è un insieme di nodi e archi che connettono i nodi. Può essere rappresentato attraverso la lista degli archi, le liste di adiacenza o incidenza, le matrici di adiacenza o incidenza ecc. Su tale struttura dati è possibile eseguire molteplici operazioni tra cui la visita. Un albero ordinato è un caso particolare di grafo (grafo non orientato connesso e aciclico) formato da un insieme finito di elementi detti **nodi** tra i quali uno è designato come **radice** mentre i rimanenti sono partizionati in insiemi ordinati e disgiunti, anch'essi alberi ordinati.

1.3 Dati

Un **dato** è un valore che una variabile può assumere.
Un **tipo di dato** è invece un modello matematico costituito da:

- Un insieme di valori
- Un insieme di operazioni ammesse su tali valori

La specifica nasconde dettagli implementativi.

Capitolo 2

Insertion sort

2.1 Il problema dell'ordinamento

Il problema dell'ordinamento è un particolare problema che riceve in **input** una sequenza di n numeri $< a_1, a_2, a_3, ..., a_n >$ e restituisce come **output** una permutazione (riarrangiamento) $< a'_1, a'_2, a'_3, ..., a'_n >$ della sequenza di input tale che: $a'_1 \leq a'_2 \leq ... \leq a'_n$.
La sequenza di input è detta **istanza** del problema dell'ordinamento; mentre i numeri da ordinare sono detti **chiavi**.
In generale, l'**istanza di un problema** è formata dall'input (che soddisfa tutti i vincoli imposti nella definizione del problema) richiesto per calcolare una soluzione del problema.

La scelta dell'algoritmo più appropriato a una data applicazione dipende dal numero di elementi da ordinare, dal livello di ordinamento iniziale degli elementi, da eventuali vincoli sui valori degli elementi, dall'architettura del calcolatore e così via.

2.2 L'algoritmo Insertion sort

INSERTION-SORT risolve il problema dell'ordinamento ed è un algoritmo efficiente per ordinare pochi elementi.

Opera nella stesso modo in cui si ordinano le carte da gioco: partendo con una mano vuota si prende via via una carta per volta dal tavolo e la si inserisce nella posizione corretta. Per trovare la posizione corretta di una carta la si confronta con le singole carte già ordinate in mano, da destra verso sinistra. In qualsiasi momento le carte tenute in mano sono ordinate; originariamente queste carte erano le prime della pila di carte da cui sono state estratte.

INSERTION-SORT prende come parametro in input un array $A[1 ... n]$ contenente una sequenza di lunghezza n che deve essere ordinata (nel codice il numero n di elementi di A è indicato con $A.length$). L'algoritmo ordina i numeri di input **sul posto**: i numeri sono risistemati all'interno dell'array A avendo, in ogni istante, al più un numero costante di essi memorizzati all'esterno dell'array. Quando la procedura INSERTION-SORT è completata, l'array di input A contiene la sequenza di output ordinata. Tramite un indice j si indicherà l'elemento corrente che viene ordinato.

```
INSERTION-SORT(A)
1   for j ← 2 to A.length
2       key ← A[j]
3       // Inserisce A[j] nella sequenza ordinata A[1 ... j - 1]
4       i ← j - 1
5       while i > 0 and A[i] > key
6           A[i + 1] ← A[i]
7           i ← i - 1
8       A[i + 1] ← key
```

Commenti:

1 Cicla su j per scegliere la chiave da ordinare. Si parte dal secondo elemento supponendo che il 1^o sia già ordinato (cioè $A[1]$ ordinato). All'inizio di ogni iterazione del ciclo **for**, il cui indice è j, il sottoarray che è formato dagli elementi $A[1 \ldots j - 1]$ costituisce il sottoinsieme di elementi già ordinati e gli elementi $A[j + 1 \ldots n]$ corrispondono invece a tutti quegli elementi che devono ancora essere ordinati. Inoltre gli elementi $A[1 \ldots j - 1]$ sono quelli che originariamente occupavano le posizioni da 1 a $j - 1$, ma adesso sono ordinati

2 All'inizio $A[2] = key$

4 $j - 1$ è l'ultimo elemento ordinato

5 - 7 Cicla su i per trovare la posizione giusta, sposta l'elemento j-esimo nel posto giusto scambiando $A[i + 1]$ e $A[i]$ se quello a sinistra è più grande. Sposta gli elementi più grandi di quello valutato nello stesso ordine in cui sono collocati

2.3 Correttezza

Un algoritmo si dice **corretto** se, per ogni istanza di input, termina con l'output corretto.

Per algoritmi **iterativi** si usa spesso un' **invariante di ciclo**, ovvero una proposizione che vale prima, durante e alla fine di un ciclo. Nel caso di INSERTION-SORT l'invariante di ciclo è la seguente:

> *All'inizio di ogni iterazione del ciclo for (righe 1 - 8) il sottoarray* $A[1 \ldots j - 1]$ *è ordinato ed è formato dagli stessi elementi che erano originariamente in* $A[1 \ldots j - 1]$ *ma ordinati*

Per verificare la correttezza di un algoritmo iterativo sfruttando l'invariante di ciclo bisogna dimostrare che tale invariante valga in tre casi:

Inizializzazione È vera prima della prima iterazione del ciclo

Conservazione Se è vera prima di un'iterazione del ciclo, rimane vera prima della successiva iterazione

Conclusione Quando il ciclo termina, l'invariante fornisce una proprietà utile per mostrare che l'algoritmo è corretto

Quando le prime due proprietà sono valide, l'invariante di ciclo è vera prima di ogni iterazione del ciclo. Si nota un'analogia con l'induzione matematica dove, per verificare che una proprietà è valida, si prova un caso base e un passo induttivo. Caso base e passo induttivo corrispondono ad inizializzazione e conservazione, la conclusione è invece diversa perché nell'induzione matematica il passo induttivo è usato all'infinito mentre nell'invariante di ciclo si termina l'induzione quando il ciclo in esame termina.

Analizzando INSERTION-SORT si verificano i tre casi dell'invariante di ciclo:

Inizializzazione Prima della prima iterazione del ciclo, quando $j = 2$, il sottoarray $A[1 \ldots j - 1]$ è formato dal solo elemento $A[1]$, che infatti è l'elemento originale in $A[1]$. Inoltre, questo sottoarray è ordinato (ovviamente) e ciò dimostra che l'invariante di ciclo è vera prima della prima iterazione del ciclo

Conservazione Il ciclo **for** esterno opera spostando $A[j - 1]$, $A[j - 2]$, $A[j - 3]$ e così via di una posizione verso destra, finché non troverà la posizione appropriata per $A[j]$ (righe 4 - 7), dove inserirà il valore di $A[j]$ (riga 8). Il sottoarray $A[1 \ldots j]$ quindi è ordinato ed è formato dagli stessi elementi che originariamente erano in $A[1 \ldots j]$. Dunque l'incremento di j per la successiva iterazione del ciclo for preserva l'invariante di ciclo

Conclusione La condizione che determina la conclusione del ciclo for è $j > A.length = n$. Poiché ogni iterazione del ciclo aumenta j di 1, alla fine del ciclo si avrà $j = n + 1$. Sostituendo j con $n + 1$ nella formulazione dell'invariante di ciclo, si ottiene che il sottoarray $A[1 \ldots n]$ è formato dagli elementi **ordinati** che si trovavano originariamente in $A[1 \ldots n]$. Ma il sottoarray $A[1 \ldots n]$ è l'intero array e dunque tutto l'array è ordinato.

Pertanto l'algoritmo è **corretto**.

2.4 Analisi degli algoritmi

Analizzare un algoritmo significa prevedere le risorse che l'algoritmo richiede. Raramente sono di primaria importanza risorse come la memoria, la larghezza di banda nelle comunicazioni o l'hardware nei computer, mentre più frequentemente è più importante misurare il **tempo di elaborazione**. Prima di analizzare un algoritmo, bisogna avere un modello della tecnologia di implementazione che sarà utilizzata, incluso un modello per le risorse di tale tecnologia e dei loro costi. Nella maggior parte dei casi si considera come tecnologia di implementazione un generico modello di calcolo a un processore che viene chiamato **random-access machine (RAM)**. In tale modello le istruzioni sono eseguite una dopo l'altra, senza operazioni contemporanee. Il modello RAM dispone di istruzioni comuni eseguite in tempo costante (no sort), ha una quantità infinita di celle di memoria di dimensione finita, consente un accesso in memoria in tempo costante e rappresenta i dati con parole di dimensioni limitate.

In generale, il tempo richiesto da un algoritmo cresce con la dimensione dell'input, quindi è tradizione descrivere il tempo di esecuzione di un programma come una funzione della dimensione del suo input.

La definizione migliore della **dimensione dell'input** dipende dal problema che si sta studiando. Per la maggior parte dei problemi, come l'ordinamento, la misura più naturale è il numero di elementi dell'input.

Il **tempo di esecuzione** di un algoritmo per un particolare input è il numero di operazioni primitive (o passi) che vengono eseguite. Per eseguire una riga dello pseudocodice occorre una quantità costante di tempo. Una riga può richiedere una quantità di tempo diversa da un'altra riga, tuttavia si supporrà che ogni esecuzione dell'i-esima riga richieda un tempo c_i, dove c_i è una costante.

2.4.1 Analisi di Insertion sort

Il tempo richiesto dalla procedura INSERTION-SORT dipende dall'input, in particolare dalla sua dimensione. Inoltre può richiedere quantità di tempo differenti per ordinare due sequenze di input della stessa dimensione a seconda di come gli elementi siano già ordinati.

INSERTION-SORT(A)

		Costo	Numero di volte
1	**for** $j \leftarrow 2$ **to** $A.length$	c_1	n
2	$key \leftarrow A[j]$	c_2	n - 1
3	// Inserisce $A[j]$ nella sequenza ordinata $A[1 \dots j$ - 1]	0	n - 1
4	$i \leftarrow j$ - 1	c_4	n - 1
5	**while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6	$A[i+1] \leftarrow$ A[i]	c_6	$\sum_{j=2}^{n}(t_j - 1)$
7	$i \leftarrow i$ - 1	c_7	$\sum_{j=2}^{n}(t_j - 1)$
8	$A[i+1] \leftarrow key$	c_8	n - 1

Commenti:

2-4 Eseguite n - 1 volte perché non si considera $A[1]$

5 t_j indica il numero di volte che si entra nel ciclo while ovvero il numero di volte che viene eseguito il test del while

5-7 Si utilizza la sommatoria perché il ciclo dipende dai dati (dai valori in input)

Il tempo di esecuzione dell'algoritmo è la somma dei tempi di esecuzione per ogni istruzione eseguita; un'istruzione che richiede c_i passi e viene eseguita n volte contribuirà con $c_i \cdot n$ al tempo di esecuzione totale.

$$costo = \sum_{istruzioni} c_i \cdot (volte\, che\, viene\, eseguita)$$

Per calcolare $T(n)$, il tempo di esecuzione di INSERTION-SORT con un input di n valori, si sommano i prodotti delle colonne costo e numero di volte, ottenendo:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7 \sum_{j=2}^{n}(t_j - 1) + c_8(n-1)$$

$T(n)$ dipende da t_j che dipende dai valori in input.

In INSERTION-SORT il caso migliore si verifica quando l'array è già ordinato. Per ogni $j = 2, 3, \dots, n$, si verifica che nella riga 5 $A[i] \leq key$, quando i ha il suo valore iniziale j - 1. Quindi $t_j = 1$ per $j = 2, 3, \dots, n$ e il tempo di esecuzione nel caso migliore è (non si entra mai nel while quindi $c_6 = c_7 = 0$):

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \end{aligned}$$

Dunque il tempo di esecuzione nel caso migliore di INSERTION-SORT può essere espresso come $T(n) = an + b$, con le costanti a e b che dipendono dai costi c_i delle istruzioni; quindi è una funzione lineare di n con b che all'infinito diventa irrilevante.

Se l'array è ordinato in senso inverso - cioè in ordine decrescente, per cui si verifica sempre $A[i] > key$ - allora si verifica il caso peggiore. Si dovrà confrontare $A[j]$, cioè *key*, con ogni elemento dell'intero sottoarray ordinato $A[1 \dots j\text{-}1]$ ovvero con tutti i j - 1 elementi alla sinistra di j e quindi $t_j = j$ per $j = 2, 3, \dots, n$. Poiché

$$\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} j = \left(\sum_{j=1}^{n} j \right) - 1 = \frac{n(n+1)}{2} - 1$$

$$\sum_{j=2}^{n} (t_j - 1) = \sum_{j=2}^{n} (j-1) = \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2}$$

Avendo posto $k = j-1$. Si ottiene che il tempo di esecuzione di INSERTION-SORT nel caso peggiore è:

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\Big(\frac{n(n+1)}{2} - 1\Big) + c_6\Big(\frac{n(n-1)}{2}\Big) + c_7\Big(\frac{n(n-1)}{2}\Big) + c_8(n-1) \\ &= c_1 n + (c_2 + c_4 + c_8)(n-1) + c_5\Big(\frac{n(n+1)}{2} - 1\Big) + (c_6 + c_7)\Big(\frac{n(n-1)}{2}\Big) \\ &= \Big(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\Big)n^2 + \Big(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\Big)n - (c_2 + c_4 + c_5 + c_8) \end{aligned}$$

Questo tempo di esecuzione può essere espresso come $T(n) = an^2 + bn + c$, con le costanti a, b e c che dipendono dai costi c_i delle istruzioni; quindi è una funzione quadratica di n.

In relazione al caso peggiore si ha il tempo di esecuzione più lungo per qualsiasi input di dimensione n. Non a caso di solito si utilizza proprio il caso peggiore per eseguire l'analisi di un algoritmo, questo essenzialmente per tre motivi:

- Il tempo di esecuzione nel caso peggiore di un algoritmo è un **limite superiore** al tempo di esecuzione per qualsiasi input. Conoscendo questo tempo si ha la garanzia che l'algoritmo non potrà impiegare di più.
- Per alcuni algoritmi il caso peggiore si verifica molto spesso, per esempio la ricerca di un elemento assente.
- Il caso medio spesso è altrettanto cattivo del peggiore. Per esempio dati in input n numeri a caso su cui applicare INSERTION-SORT si avrà che in media metà degli elementi di $A[1 \dots j\text{-}1]$ sono più piccoli di $A[j]$, mentre gli altri elementi sono più grandi. In media quindi, si verificherà solo metà

del sottoarray $A[1 \dots j\text{-}1]$, pertanto t_j vale circa $\frac{j}{2}$ cioè circa la metà del caso peggiore ma sempre una funzione quadratica di n.

In alcuni casi particolari sarà più importante determinare il tempo di esecuzione nel **caso medio** di un algoritmo.

Nell'ambito dell'analisi degli algoritmi un'astrazione semplificativa molto utilizzata è il **tasso di crescita** che rappresenta la velocità con cui cresce il tempo di esecuzione di un algoritmo e che viene definita solo in relazione al **termine principale** di una formula, cioè in relazione al termine di grado maggiore ignorando le costanti (i termini di grado inferiore sono insignificanti per grandi valori di n). È proprio il tasso di crescita il termine di paragone tra i vari algoritmi. Per Insertion-Sort si avrà dunque:

$$T(n) = an^2 + bn + c \Rightarrow an^2 \Rightarrow n^2$$

Il tempo di esecuzione **cresce come** n^2, non è uguale a n^2. Il tasso di crescita è invece: $n^2 \Rightarrow \Theta(n^2)$.

Capitolo 3

Merge sort

3.1 Progettare gli algoritmi

Ci sono varie tecniche per progettare gli algoritmi. Per INSERTION-SORT, algoritmo **iterativo**, si utilizza un approccio **incrementale**: dopo aver ordinato il sottoarray $A[1 \dots j\text{-}1]$, si inserisce il singolo elemento $A[j]$ nella posizione appropriata, ottenendo il sottoarray ordinato $A[1 \dots j]$.
Nel caso degli algoritmi **ricorsivi**, invece, si utilizza un altro approccio detto **divide et impera**.

3.1.1 Il metodo divide et impera

Gli algoritmi ricorsivi, per risolvere un determinato problema, chiamano sé stessi in modo ricorsivo, una o più volte, per trattare sottoproblemi dello stesso tipo. Attraverso il metodo divide et impera suddividono il problema in vari sottoproblemi, che sono simili al problema di partenza, ma di dimensioni più piccole, risolvono i sottoproblemi in modo ricorsivo e, poi, combinano le soluzioni per costruire una soluzione del problema originale. Tale paradigma prevede tre passi ad ogni livello di ricorsione:

Divide Il problema viene diviso in un certo numero di sottoproblemi, che sono istanze più piccole dello stesso problema

Impera I sottoproblemi vengono risolti in modo ricorsivo. Comunque, quando i sottoproblemi hanno una dimensione sufficientemente piccola, essi vengono risolti direttamente

Combina Le soluzioni dei sottoproblemi vengono combinate per generare la soluzione del problema originale

Un esempio:

Fattoriale

L'algoritmo per il calcolo del fattoriale di un numero n può essere implementato sia in forma iterativa che ricorsiva. In entrambi i casi la sua correttezza può essere verificata mediante l'invariante di ciclo.

Iterativo

```
FATT(n)
1  if n = 1
2      return 1
3  Fat ← 1
4  for i ← 1 to n
5      Fat ← Fat · i
6  return Fat
```

Ricorsivo

```
FATT(n)
1  if n = 1
2      return 1
3  else return n · FATT(n − 1)
```

La versione ricorsiva rappresenta un algoritmo ricorsivo **in coda** nel senso che la ricorsione è in fondo (potrebbe essere trasformato in un algoritmo iterativo) e come si nota si passerà come argomento di FATT un valore più piccolo di quello considerato.

Asintoticamente le due versioni costano uguale ma a livello di programma costa di più quello ricorsivo perché deve richiamare sé stesso e allocare nello stack.

Facendo un'analisi delle due versioni si dimostra che per entrambe il tempo di esecuzione cresce come n:

$$T(n) = cn \Rightarrow n$$

Nel caso iterativo si cicla su n valori eseguendo operazioni costanti quindi ottenendo un costo cn. Analogamente nel caso ricorsivo si ottiene cn.

Per quanto riguarda la correttezza, l'invariante di ciclo è la seguente:

> *Ad ogni iterazione del ciclo **for** la variabile Fat contiene il prodotto dei primi i valori*

3.2 L'algoritmo Merge sort

L'algoritmo MERGE-SORT è conforme al paradigma divide et impera e opera nel modo seguente:

Divide Divide la sequenza degli n elementi da ordinare in due sottosequenze di $n/2$ elementi ciascuna

Impera Ordina le due sottosequenze in modo ricorsivo utilizzando l'algoritmo MERGE-SORT

Combina Fonde le due sottosequenze ordinate per generare la sequenza completa ordinata

La ricorsione *tocca il fondo* quando la sequenza da ordinare ha grandezza 1, nel qual caso non c'è più nulla da fare, in quanto ogni sequenza di lunghezza 1 è già ordinata.

L'operazione chiave dell'algoritmo MERGE-SORT è la fusione di due sottosequenze ordinate nella passo *combina*. Per effettuare la fusione si utilizza la funzione MERGE(A, p, q, r), dove A è un array e p, q, r sono indici dell'array tali che $p \leq q < r$. La funzione assume che i sottoarray $A[p \,..\, q]$ e $A[q+1 \,..\, r]$ siano ordinati; li **fonde** per formare un unico sottoarray ordinato che sostituisce il sottoarray corrente $A[p \,..\, r]$.

La procedura MERGE impiega un tempo $\Theta(n)$, dove $n = r - p + 1$ è il numero totale di elementi da fondere.

L'algoritmo MERGE-SORT ordina il vettore in input $A[p \,..\, r]$. Se $p \geq r$, l'array ha al massimo un elemento e quindi è già ordinato; altrimenti, il passo *divide* calcola un indice q che separa $A[p \,..\, r]$ in due sottoarray: $A[p \,..\, q]$ che contiene $\lceil n/2 \rceil$ elementi, e $A[q + 1 \,..\, r]$ che contiene $\lfloor n/2 \rfloor$ elementi. Riassumendo:

Divide Divide $A[p \,..\, r]$ in $A[p \,..\, q]$ e $A[q + 1 \,..\, r]$, q è il punto di mezzo di $A[p \,..\, r]$

Impera Ordina ricorsivamente $A[p \,..\, q]$ e $A[q + 1 \,..\, r]$

Combina Fonde i due sottoarray ordinati $A[p \,..\, q]$ e $A[q + 1 \,..\, r]$ in un singolo array ordinato $A[p \,..\, r]$

```
MERGE-SORT(A, p, r)
1   if p < r
2       q ← ⌊(p + r)/2⌋
3       MERGE-SORT(A, p, q)
4       MERGE-SORT(A, q + 1, r)
5       MERGE(A, p, q, r)
```

La riga 1 controlla il caso base (se il sottoarray ha dimensione 1), la riga 2 corrisponde a *divide*, le righe 3 - 4 a *impera* e infine la riga 5 a *combina*.

```
MERGE(A, p, q, r)
1   n1 ← q − p + 1
2   n2 ← r − q
3   // Crea array L[1 .. n1 + 1] e R[1 .. n2 + 1]
4   for i ← 1 to n1
5       L[i] ← A[p + i − 1]
6   for j ← 1 to n2
7       R[j] ← A[q + j]
8   L[n1 + 1] ← ∞
9   R[n2 + 1] ← ∞
10  i ← 1
11  j ← 1
12  for k ← p to r
13      if L[i] ≤ R[j]
14          A[k] ← L[i]
15          i ← i + 1
16      else A[k] ← R[j]
17          j ← j + 1
```

Commenti:

1-2 n_1 e n_2 sono le dimensioni dei due sottoarray $A[p \,..\, q]$ e $A[q + 1 \,..\, r]$

3 Sia in L che in R c'è $+1$ per poter mettere le due **sentinelle** (permettono di evitare di mettere due **if** che peserebbero di più)

4-5 Copia in L la parte sinistra di A, cioè $A[p \,..\, q]$

6-7 Copia in R la parte destra di A, cioè $A[q+1 \,..\, r]$

8-9 Si definiscono le due sentinelle poste alla fine degli array L e R. Vengono poste uguali a ∞ per essere sicuri che abbiano un valore più grande di tutti gli elementi presenti nell'array. Si usano le sentinelle nel caso in cui i due sottoarray non hanno la stessa dimensione e dunque avanzano valori che dovranno essere copiati

10-11 Si inizializzano gli indici per scorrere i due sottoarray

12 k scorre sulle posizioni finali (verranno eseguiti $n = r - p + 1$ passi)

13-17 Trova il minore tra i due valori puntati e incrementa il puntatore di una delle due parti

3.3 Correttezza

L'ultimo ciclo **for** di MERGE segue la seguente invariante di ciclo:

> *All'inizio di ogni iterazione del ciclo* ***for*** *(righe 12 - 17), il sottoarray* $A[p \,..\, k-1]$ *contiene ordinati i* k *-* p *elementi più piccoli di* $L[1 \,..\, n_1+1]$ *e* $R[1 \,...\, n_2+1]$. *Inoltre* $L[i]$ *e* $R[j]$ *sono i più piccoli elementi dei loro array che non sono stati copiati in* A

Si dimostra ora la sua validità:

Inizializzazione Prima della prima iterazione del ciclo, si ha $k = p$ quindi il sottoarray $A[p \,...\, k\text{ - }1]$ è vuoto. Questo sottoarray vuoto contiene dunque $k - p = 0$ elementi più piccoli di L e R; poiché $i = j = 1$, $L[i]$ e $R[j]$ sono i più piccoli elementi, nei rispettivi array, tra quelli che non sono ancora stati copiati in A

Conservazione Se $L[i] \leq R[j]$ (quindi $L[i]$ è l'elemento più piccolo che non è stato ancora copiato in A), poiché $A[p \,...\, k\text{ - }1]$ contiene i $k - p$ elementi più piccoli, dopo la riga 14 ha copiato $L[i]$ in $A[k]$, il sottoarray $A[p \,...\, k]$ conterrà i $k-p+1$ elementi più piccoli. Incrementando k (aggiornamento nel ciclo **for**) e i (riga 15), si ristabilisce l'invariante di ciclo per la successiva iterazione. Se invece $L[i] > R[j]$, allora le righe 16 - 17 svolgono l'azione appropriata per conservare l'invariante di ciclo

Conclusione Alla fine del ciclo $k = r+1$. Per l'invariante di ciclo, il sottoarray $A[p \,...\, k\text{ - }1]$, che è $A[p \,...\, r]$, contiene $k - p = r - p + 1$ elementi ordinati che sono i più piccoli di $L[1 \,...\, n_1 + 1]$ e $R[1 \,...\, n_2 + 1]$. Assieme, gli array L e R contengono $n_1 + n_2 + 2 = r - p + 3$ elementi. Tutti gli elementi, tranne i due più grandi, sono stati copiati in A; questi due elementi sono le sentinelle

Per verificare che MERGE viene eseguita nel tempo $\Theta(n)$, con $n = r - p + 1$, si nota che ciascuna delle righe 1 - 3 e 8 - 11 impiega un tempo costante, i cicli **for** (righe 4 - 7) impiegano un tempo $\Theta(n_1 + n_2) = \Theta(n)$, e ci sono n iterazioni del ciclo **for** (righe 12 - 17), ciascuna delle quali impiega un tempo costante. Dunque il tempo totale necessario ad eseguire MERGE è $\Theta(n)$. Il ciclo **for** (righe 12 - 17) non dipende dai dati, se non ci fossero le sentinelle si dovrebbero usare degli **if**

andando così ad aumentare il costo ed inoltre in tal caso il ciclo dipenderebbe dai dati per cui si avrebbe che il caso migliore è quando il valore massimo di un sottoarray è maggiore del valore minimo dell'altro sottoarray.

3.4 Analisi degli algoritmi divide et impera

Quando un algoritmo contiene una chiamata ricorsiva a sé stesso, il suo tempo di esecuzione spesso può essere descritto con una **equazione di ricorrenza** o **ricorrenza**, che esprime il tempo di esecuzione totale di un problema di dimensione n in funzione del tempo di esecuzione per input più piccoli. Una ricorrenza per il tempo di esecuzione di un algoritmo divide et impera si basa sui tre passi del paradigma di base. Si suppone che $T(n)$ sia il tempo di esecuzione di un problema di dimensione n:

$$T(n) = \textit{tempo di esecuzione per il problema di dimensione } n$$

Se la dimensione del problema è sufficientemente piccola: $n \leq c$ per qualche costante c, ci si trova nel caso base che richiede un tempo costante indicato con $\Theta(1)$. Altrimenti, si suppone che la suddivisione dei problema generi a sotto-problemi e che la dimensione di ciascuno di essi sia $1/b$ volte la dimensione del problema originale. Serve un tempo $T(n/b)$ per risolvere un sottoproblema di dimensione n/b e quindi, per risolverne a, serve tempo $aT(n/b)$. Se si impiega un tempo $D(n)$ per dividere il problema in sottoproblemi e un tempo $C(n)$ per combinare le soluzioni dei sottoproblemi nella soluzione del problema originale, si ottiene la ricorrenza:

$$T(n) = \begin{cases} \Theta(1) & \text{se } n \leq c \\ aT(n/b) + D(n) + C(n) & \text{altrimenti} \end{cases}$$

3.4.1 Analisi di Merge sort

Si suppone che la dimensione n del problema sia un potenza del 2. Ogni passo *divide* genera due sottosequenze di dimensione esattamente pari a $n/2$ (questa ipotesi non influisce sul tasso di crescita della soluzione della ricorrenza).

Per trovare la ricorrenza per $T(n)$, il tempo di esecuzione nel caso peggiore di MERGE-SORT con n numeri, si può fare il seguente ragionamento. L'algoritmo MERGE-SORT applicato a un solo elemento (caso base $n = 1$) impiega un tempo costante. Se invece si hanno $n \geq 2$ elementi, si suddivide il tempo di esecuzione secondo i passi divide et impera:

Divide Calcola q, ciò richiede un tempo costante, quindi $D(n) = \Theta(1)$.

Impera Risolve in modo ricorsivo i due sottoproblemi di dimensione $n/2$, ciò contribuisce con $2T(n/2)$.

Combina Fonde un array con n elementi, come visto MERGE richiede un tempo $\Theta(n)$, quindi $C(n) = \Theta(n)$.

Sommando le funzioni $D(n)$ e $C(n)$ si ottiene:

$$D(n) + C(n) = \Theta(1) + \Theta(n) = \Theta(n)$$

Sommando al termine $2T(n/2)$ del passo *impera* si ottiene la ricorrenza per il tempo di esecuzione $T(n)$ nel caso peggiore di MERGE-SORT:

$$T(n) = \begin{cases} c & \text{se } n = 1 \\ 2\,T(n/2) + cn & \text{se } n > 1 \end{cases} \tag{3.1}$$

La costante c rappresenta sia il tempo richiesto per risolvere i problemi di dimensione 1 sia il tempo per elemento dell'array dei passi *divide* e *combina*.

Un possibile modo per risolvere la ricorrenza è attraverso la costruzione di un **albero di ricorsione** che permette di visualizzare le espansioni successive: il termine cn è la radice (il costo sostenuto al primo livello di ricorsione) e i due sottoalberi della radice sono le due ricorrenze più piccole $T(n/2)$. Il costo sostenuto per ciascuno dei due sottonodi al secondo livello di ricorsione è $cn/2$. Continuando ad espandere i nodi dell'albero suddividendolo nelle sue componenti come stabilisce la ricorrenza si arriva al caso base in cui le dimensioni dei problemi si riducono a 1 e ciascuno di essi ha costo c.

A questo punto si sommano i costi per ogni livello dell'albero. Il primo livello in alto ha un costo totale cn, il secondo livello ha un costo totale $c(n/2) + c(n/2) = cn$, il terzo livello ha un costo totale $c(n/4)+c(n/4)+c(n/4)+c(n/4) = cn$ e così via (ad ogni livello raddoppiano i sottoproblemi ma si dimezza il costo per ognuno di essi). In generale, il livello i ha 2^i nodi, ciascuno dei quali ha un costo $2^i c(n/2^i) = cn$ (al livello i la dimensione è pari a $\frac{n}{2^i}$). All'ultimo livello in basso ci sono n nodi, ciascuno con un costo c, per un costo totale di cn.

Il numero totale di livelli dell'albero di ricorsione è $lg\,n + 1$, dove n è il numero di foglie che è anche uguale alla dimensione dell'input. Per calcolare il costo totale dalla ricorrenza (3.1) basta sommare i costi di tutti i livelli. Ci sono $lg\,n + 1$ livelli, ciascuno di costo cn, per un costo totale di $cn\,(lg\,n + 1) = cn\,lg\,n + cn$. Ignorando il termine di ordine inferiore e la costante c, si ottiene il risultato $\Theta(n\,lg\,n)$.

Capitolo 4

Crescita delle funzioni

Le notazioni che si utilizzano per descrivere il tempo di esecuzione asintotico di un algoritmo sono definite in termini di funzioni il cui dominio è l'insieme dei numeri naturali $\mathbb{N} = \{0, 1, 2, ...\}$. Tali notazioni sono comode per descrivere la funzione $T(n)$, tempo di esecuzione nel caso peggiore, che di solito è definita soltanto con dimensioni intere dell'input. A volte, però, è possibile abusare di tale notazione asintotica ed estenderla al dominio dei numeri reali $\mathbb{R}$ o limitata a un sottoinsieme dei numeri naturali.

4.1 Notazione O

Viene utilizzata quando si ha un **limite asintotico superiore**. Per una data funzione $g(n)$, si indica con $O(g(n))$ l'insieme delle funzioni:

$$O(g(n)) = \{\, f(n) : \exists\, c \in \mathbb{R}^+,\, n_0 \in \mathbb{N}\ t.c.\ 0 \leq f(n) \leq c \cdot g(n)\ \forall\ n \geq n_0 \,\}$$

$g(n)$ è un **limite asintotico superiore** per $f(n)$.
La notazione O si usa per assegnare un limite superiore ad una funzione, a meno di un fattore costante. Per qualsiasi valore n a destra di n_0, il valore della funzione $f(n)$ coincide o sta sotto $cg(n)$. Si scrive $f(n) = O(g(n))$ per indicare che una funzione $f(n)$ è un membro dell'insieme $O(g(n))$, ovvero che $f(n) \in O(g(n))$.

Esempi:

- $2n^2 = O(n^3),\ con\ c = 1\ e\ n_0 = 2$
- Funzioni in $O(n^2)$
  - n
  - $n^{1.99}$
  - n^2
  - $n^2 + n$
  - $1000n^2 + 100n$

4.2 Notazione Ω

Così come O fornisce un limite asintotico superiore, la notazione Ω fornisce un **limite asintotico inferiore**. Per una data funzione $g(n)$, si indica con $\Omega(g(n))$ l'insieme delle funzioni:

$$\Omega(g(n)) = \{\, f(n) : \exists\, c \in \mathbb{R},\, n_0 \in \mathbb{N}\ t.c.\ 0 \leq c \cdot g(n) \leq f(n)\ \forall\ n \geq n_0 \,\}$$

$g(n)$ è un **limite asintotico inferiore** per $f(n)$.
Per tutti i valori di n a destra di n_0, il valore di $f(n)$ coincide o sta sopra $cg(n)$.

Esempi:

- $\sqrt{n} = \Omega(lg\, n)$
- Funzioni in $\Omega(n^2)$
  - n^2
  - $n^2 \pm n$
  - $1000n^2 \pm 100n$
  - n^3
  - $n^{2.001}$
  - $n^2\ lg\, lg\, lg\, n$

4.3 Notazione Θ

Per una data funzione $g(n)$, si indica con $\Theta(g(n))$ l'insieme delle funzioni:

$$\Theta(g(n)) = \{\, f(n) : \exists\, c_1, c_2, n_0 \in \mathbb{N}\ t.c.\ 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\ \forall\ n \geq n_0 \,\}$$

$g(n)$ è un **limite asintoticamente stretto** per $f(n)$.
Una funzione $f(n)$ appartiene all'insieme $\Theta(g(n))$ se esistono delle costanti positive c_1 e c_2 tali che essa possa essere racchiusa fra $c_1\, g(n)$ e $c_2\, g(n)$, per valori sufficientemente grandi di n. Per tutti i valori di n a destra di n_0, il valore di $f(n)$ coincide o sta sopra $c_1\, g(n)$ e coincide o sta sotto $c_2\, g(n)$. In altre parole, per ogni $n \geq n_0$, la funzione $f(n)$ è uguale a $g(n)$ a meno di un fattore costante.

La definizione di $\Theta(g(n))$ richiede che ogni membro di $f(n) \in \Theta(g(n))$ sia **asintoticamente non negativo**, ovvero che $f(n)$ sia non negativa quando n è sufficientemente grande. Di conseguenza, la funzione $g(n)$ stessa deve essere asintoticamente non negativa, altrimenti l'insieme $\Theta(g(n))$ è vuoto.

Teorema 4.1. *Per ogni coppia di funzioni $f(n)$ e $g(n)$, si ha $f(n) = \Theta(g(n))$ se e soltanto se $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$.*

4.4 Notazione o

Si utilizza la notazione o per denotare un limite superiore che **non** è asintoticamente stretto.

$$o(g(n)) = \{\, f(n) : \forall\, c > 0 \in \mathbb{R}\ \exists\, n_0 > 0 \in \mathbb{N}\ t.c.\ 0 \leq f(n) < c \cdot g(n)\ \forall\ n \geq n_0 \,\}$$

Nella notazione o la funzione $f(n)$ diventa insignificante rispetto a $g(n)$ quando n tende all'infinito; ovvero:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

Esempi:

- $2n = o(n^2)$
- $2n^2 \neq o(n^2)$
- Funzioni in $o(n^2)$
  - $n^{1.999}$
  - $\frac{n^2}{lg\, n}$

4.5 Notazione ω

Si utilizza la notazione ω per indicare un limite inferiore che **non** è asintoticamente stretto.

$$o(g(n)) = \{\, f(n) : \forall\, c > 0 \in \mathbb{R}\ \exists\, n_0 > 0 \in \mathbb{N}\ t.c.\ 0 \leq c \cdot g(n) < f(n)\ \forall\ n \geq n_0 \,\}$$

La relazione $f(n) = \omega(g(n))$ implica che:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

se il limite esiste; cioè $f(n)$ diventa arbitrariamente grande rispetto a $g(n)$ quando n tende all'infinito.

Esempi:

- $\frac{n^2}{2} = \omega(n)$
- $n^2 \neq \omega(n^2)$
- Funzioni in $\omega(n^2)$
  - $n^{2.0001}$
  - $n^2\, lg\, n$

4.6 Confronto di funzioni

Date due funzioni $f(n)$ e $g(n)$ assunte **asintoticamente positive** si definiscono le seguenti proprietà:

Transitività

$$f(n) = O(g(n)) \text{ e } g(n) = O(h(n)) \ \Rightarrow\ f(n) = O(h(n))$$
$$f(n) = \Omega(g(n)) \text{ e } g(n) = \Omega(h(n)) \ \Rightarrow\ f(n) = \Omega(h(n))$$
$$f(n) = \Theta(g(n)) \text{ e } g(n) = \Theta(h(n)) \ \Rightarrow\ f(n) = \Theta(h(n))$$

$f(n) = o(g(n))$ e $g(n) = o(h(n)) \Rightarrow f(n) = o(h(n))$
$f(n) = \omega(g(n))$ e $g(n) = \omega(h(n)) \Rightarrow f(n) = \omega(h(n))$

Riflessività

$f(n) = O(g(n))$
$f(n) = \Omega(g(n))$
$f(n) = \Theta(g(n))$

Simmetria

$f(n) = \Theta(g(n)) \Leftrightarrow g(n) = \Theta(f(n))$

Simmetria Trasposta

$f(n) = O(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$
$f(n) = o(g(n)) \Leftrightarrow g(n) = \omega(f(n))$

Intuitivamente:

$f(n) = O(g(n))$ equivale a $a \leq b$

$f(n) = \Omega(g(n))$ equivale a $a \geq b$

$f(n) = \Theta(g(n))$ equivale a $a = b$

$f(n) = o(g(n))$ equivale a $a < b$

$f(n) = \omega(g(n))$ equivale a $a > b$

Per il confronto tra funzioni **non** vale la proprietà di **tricotomia**: se a e b sono due numeri reali qualsiasi allora $a < b$ o $a > b$ o $a = b$.

$f(n)$ è **asintoticamente più piccola** di $g(n)$ se $f(n) = o(g(n))$.
$f(n)$ è **asintoticamente più grande** di $g(n)$ se $f(n) = \omega(g(n))$.

4.6.1 Logaritmi

Per i logaritmi si segue la seguente notazione:

$lg\, n = \log_2 n$

$ln\, n = \log_e n$

$lg^k\, n = (lg\, n)^k$

Per tutti i numeri reali $a > 0$, $b > 0$, $c > 0$ e $n \neq 1$:

a $= b^{log_b\, a}$

$log_c(ab) = log_c\, a + log_c\, b$

$log_b(a^n) = n\, log_b\, a$

$log_b\, a = \frac{log_c\, a}{log_c\, b}$

$log_b\, \frac{1}{a} = -\, log_b\, a$

$log_b\, a = \frac{1}{log_a\, b}$

$a^{log_b\, c} = c^{log_b\, a}$

Capitolo 5

Ricorrenze

Come già visto, con il metodo divide et impera un problema viene risolto in modo ricorsivo, applicando tre passi a ogni livello di ricorsione:

Divide Il problema viene diviso in un certo numero di sottoproblemi, che sono istanze più piccole dello stesso problema

Impera I sottoproblemi vengono risolti in modo ricorsivo. Comunque, quando i sottoproblemi hanno una dimensione sufficientemente piccola, essi vengono risolti direttamente

Combina Le soluzioni dei sottoproblemi vengono combinate per generare la soluzione del problema originale

Quando i sottoproblemi sono abbastanza grandi da essere risolti ricorsivamente, si ha il cosiddetto **caso ricorsivo**. Una volta che i sottoproblemi diventano sufficientemente piccoli da non richiedere ricorsione, si dirà che la ricorsione ha *toccato il fondo* e che si è raggiunto il **caso base**. A volte, oltre ai sottoproblemi che sono istanze più piccole dello stesso problema, si devono risolvere dei sottoproblemi che non sono uguali al problema originale. La risoluzione di tali problemi farà parte del passo *combina*.

5.1 Cos'è una ricorrenza

Le ricorrenze offrono un modo naturale per caratterizzare i tempi di esecuzione degli algoritmi divide et impera. Una **ricorrenza** è un'equazione o disequazione che descrive una funzione in termini del suo valore con input più piccoli. In genere in termini di uno o più casi base e di sé stessa, con argomenti più piccoli.

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ aT(n/b) + f(n) & \text{se } n > 1 \end{cases} \tag{5.1}$$

Le ricorrenze possono assumere varie forme. I sottoproblemi non devono necessariamente essere una frazione costante della dimensione del problema originale.

Alcune ricorrenze non sono uguaglianze, ma disuguaglianze, per esempio:

$$T(n) \leq 2\,T(n/2) + \Theta(n)$$

In tal caso, poiché tali ricorrenze stabiliscono soltanto un limite superiore su $T(n)$, si esprimerà la soluzione utilizzando la notazione O anziché la notazione Θ. Analogamente, se la disuguaglianza è del tipo:

$$T(n) \geq 2\,T(n/2) + \Theta(n)$$

poiché la ricorrenza fornisce soltanto un limite inferiore su $T(n)$, si utilizzerà la notazione Ω nella sua soluzione.

Esistono dei metodi per risolvere le ricorrenze - cioè per ottenere dei limiti asintotici Θ, Ω o O per la soluzione:

Metodo di sostituzione Si ipotizza un limite e poi si utilizza l'induzione matematica per dimostrare che l'ipotesi è corretta

Metodo dell'albero di ricorsione Converte la ricorrenza in un albero i cui nodi rappresentano i costi ai vari livelli della ricorsione; per risolvere la ricorrenza si adotteranno delle tecniche che limitano le sommatorie

Metodo dell'esperto Fornisce i limiti per ricorrenze della forma

$$T(n) = a\,T(n/b) + f(n) \tag{5.2}$$

dove $a \geq 1$, $b > 1$ e $f(n)$ è una funzione data. Queste ricorrenze si presentano frequentemente. Una ricorrenza della forma (5.2) caratterizza un algoritmo divide et impera che crea a sottoproblemi, ciascuno dei quali ha un dimensione pari a $1/b$ quella del problema originale e in cui i passi *divide* e *combina* insieme richiedono un tempo $f(n)$

Le **condizioni al contorno** rappresentano una classe di dettagli che tipicamente vengono trascurati. Poiché il tempo di esecuzione di un algoritmo con un input di dimensione costante è una costante, le ricorrenze che derivano dai tempi di esecuzione degli algoritmi, generalmente, hanno $T(n) = \Theta(1)$ per valori sufficientemente piccoli di n. Per comodità, quindi, di solito si omettono le definizioni delle condizioni al contorno delle ricorrenze e si assumerà che $T(n)$ sia costante per n piccolo. Così facendo la ricorrenza (5.1) viene definita come:

$$T(n) = a\,T(n/b) + \Theta(n)$$

5.2 Metodo di sostituzione

Il **metodo di sostituzione** per risolvere le ricorrenze richiede due passi:

- Ipotizzare la forma della soluzione
- Usare l'induzione matematica per trovare le costanti e dimostrare che la soluzione funziona

È un metodo potente ma ovviamente può essere applicato soltanto nei casi in cui sia facile immaginare la forma della soluzione. Può essere usato per determinare il limite inferiore o superiore di una ricorrenza. Si distinguono due soluzioni:

Soluzione Esatta Se nella definizione della ricorrenza compare una funzione esatta, la soluzione della ricorrenza sarà esatta

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

1. **Ipotesi** Si ipotizza la soluzione $T(n) = n\,lg\,n + n$.
2. **Induzione** si dimostra la correttezza dell'ipotesi:
   - **Caso base**: $n = 1 \Rightarrow n\,lg\,n + n = 1 = T(n)$
   - **Passo induttivo**: ipotesi induttiva $T(k) = k\,lg\,k + k\ \forall k < n$
     Usa l'ipotesi induttiva per $k = (n/2)$

$$\begin{aligned} T(n) &= 2\,T\Big(\frac{n}{2}\Big) + n = 2\Big(\frac{n}{2}\,lg\,\frac{n}{2} + \frac{n}{2}\Big) + n \\ &= n\,lg\,\frac{n}{2} + n + n = n(lg\,n - lg\,2) + n + n \\ &= n\,lg\,n - n + n + n = n\,lg\,n + n \end{aligned}$$

Soluzione Asintotica È la soluzione usata più spesso

$$T(n) = \begin{cases} O(1) & \text{se } n \text{ sufficientemente piccolo} \\ 2\,T(n\,/\,2) + \Theta(n) & \text{altrimenti} \end{cases}$$

Viene data una soluzione con notazione asintotica: $T(n) = \Theta(n\,lg\,n)$. Non si dimostra in dettaglio il caso base

- $T(n)$ è costante $\forall\, n$ costante
- Interessa la soluzione asintotica quindi si può sempre scegliere un caso base $O(1)$.

Esempio 1

Data $T(n) = 2\,T(n/2) + \Theta(n)$ si dovrà dare un **nome** alla costante per $\Theta(n)$ e mostrare il limite superiore (O) e inferiore (Ω) separatamente. Si suppone $T(n) = \Theta(n\,lg\,n)$.

La limitazione superiore di $T(n) = 2\,T(n/2) + O(n)$ è data dalla disuguaglianza $T(n) \leq 2\,T(n/2) + cn$ per qualche costante positiva c (definizione di $O(n)$). Si ipotizza la soluzione $T(n) = O(n\,lg\,n)$ cioè $T(n) \leq d \cdot n \cdot lg\,n$ per $d > 0$ costante. Se si trova d costante tale che la ricorrenza è verificata, allora si verifica che $T(n) = O(n\,lg\,n)$. c è una costante che dipende da molteplici fattori ed è data, d invece è un valore che soddisfa la definizione di O. Si procede con la sostituzione:

$$\begin{aligned} T(n) &\leq 2\,T\Big(\frac{n}{2}\Big) + cn \\ &\leq 2\Big(d\,\frac{n}{2}\,lg\,\frac{n}{2}\Big) + cn \\ &= d\,n\,lg\,\frac{n}{2} + cn \\ &= d\,n\,lg\,n - d\,n + cn \\ &\leq d\,n\,lg\,n \end{aligned}$$

$\Rightarrow T(n) = O(n\,lg\,n)$ (valido se $-dn + cn \leq 0$ cioè $d \geq c$).

La limitazione inferiore è data dalla disuguaglianza $T(n) \geq 2\,T(n/2) + cn$ per

qualche costante positiva c (definizione di $\Omega(n)$). Si ipotizza $T(n) \geq d \cdot n \cdot lg\, n$ per qualche $d > 0$ costante. Si procede con la sostituzione:

$$\begin{aligned} T(n) &\geq 2\,T\Big(\frac{n}{2}\Big) + cn \\ &\geq 2\Big(d\,\frac{n}{2}\,lg\,\frac{n}{2}\Big) + cn \\ &= d\,n\,lg\,\frac{n}{2} + cn \\ &= d\,n\,lg\,n - d\,n + cn \\ &\geq d\,n\,lg\,n \end{aligned}$$

$\Rightarrow T(n) = \Omega(n\,lg\,n)$ (valido se $-dn + cn \geq 0$ cioè $d \leq c$).

Avendo verificato che $T(n) = O(n\,lg\,n)$ nel limite superiore e $T(n) = \Omega(n\,lg\,n)$ nel limite inferiore, si ottiene che $T(n) = \Theta(n\,lg\,n)$ come si aveva supposto.

Ci sono casi in cui è possibile ipotizzare correttamente un limite asintotico per la soluzione di una ricorrenza, ma in qualche modo sembra che i calcoli matematici non tornino nell'induzione. Di solito, il problema è che l'ipotesi induttiva non è abbastanza forte per dimostrare il limite esatto. Quando ci si imbatte in simili ostacoli, spesso basta correggere l'ipotesi sottraendo un termine di ordine inferiore per far tornare i conti.

Esempio 2

Data $T(n) = 4\,T(n/2) + n$, tale ricorrenza indica che ogni nodo avrà 4 figli e che la somma dei costi delle operazioni *divide* e *combina* è n. Tramite una **stima conservativa** si suppone $T(n) = O(n^3)$ quindi si dovrà verificare che $T(n) \leq c\,n^3$. Si procede con la sostituzione supponendo $T(k) \leq c\,k^3$ per $k < n$:

$$\begin{aligned} T(n) &= 4\,T\Big(\frac{n}{2}\Big) + n \leq 4\,c\Big(\frac{n}{2}\Big)^3 + n \\ &= \frac{c}{2}n^3 + n \\ &= \underbrace{cn^3}_{desiderato} - \underbrace{\Big(\Big(\frac{c}{2}\Big)n^3 - n\Big)}_{residuo} \\ &\leq cn^3 \end{aligned}$$

$\Rightarrow T(n) = O(n^3)$ (valido se residuo $(\frac{c}{2})n^3 - n \geq 0$).
Provando ora il limite più stretto $T(n) = O(n^2)$, si suppone $T(k) \leq c\,k^2$ per $k < n$:

$$\begin{aligned} T(n) &= 4\,T\Big(\frac{n}{2}\Big) + n \leq 4\,c\Big(\frac{n}{2}\Big)^2 + n \\ &= cn^2 + n = O(n^2)\ \textbf{FALSO!!} \end{aligned}$$

L'uguaglianza è vera perché si trascurano gli ordini inferiori ma si voleva arrivare a dimostrare **esattamente** la stessa forma dell'ipotesi induttiva: $T(n) \leq cn^2$. Ciò significa che $\nexists\, c$ tale che $cn^2 + n \leq cn^2$. c dovrebbe dipendere da n dunque non sarebbe costante. È quindi opportuno irrobustire l'ipotesi induttiva andando a sottrarre un termine di ordine inferiore. Partendo dall'ipotesi

$T(k) \leq c_1 k^2 + c_2 k$ per $k < n$ si ottiene:

$$\begin{aligned} T(n) &= 4\,T\Big(\frac{n}{2}\Big) + n \\ &\leq 4\left(c_1\Big(\frac{n}{2}\Big)^2 - c_2\Big(\frac{n}{2}\Big)\right) + n \\ &= c_1 n^2 - 2\,c_2 n + n \\ &= c_1 n^2 - c_2 n - (c_2 n - n) \\ &\leq c_1 n^2 - c_2 n \end{aligned}$$

$\Rightarrow T(n) = O(n^2)$ (valido se $c_2 \geq 1$).
Analogamente si dimostra per il limite inferiore arrivando a verificare che per la ricorrenza considerata $T(n) = \Theta(n^2)$.

5.3 Metodo dell'albero di ricorsione

In un **albero di ricorsione** ogni nodo rappresenta il costo di un singolo sottoproblema da qualche parte nell'insieme delle chiamate ricorsive di funzione. Sommando i costi all'interno di ogni livello dell'albero si ottiene un insieme di costi per livello; sommando poi tutti i costi per livello si determina il costo totale di tutti i livelli della ricorsione.

Un albero di ricorsione è un ottimo metodo per ottenere una buona ipotesi, che poi viene verificata con il metodo di sostituzione. Quando si usa un albero di ricorsione per generare una buona ipotesi, spesso si tollera una certa dose di *approssimazione*, in quanto l'ipotesi sarà verificata in un secondo momento.

Il metodo dell'albero di ricorsione dà solo l'intuizione della soluzione, non dà la soluzione esatta.

Esempio 1

Data $T(n) = T(n/4) + T(n/2) + n^2$, i primi due termini corrispondono all'operazione di *impera* mentre n^2 alle operazioni di *divide* e *combina*. Questa ricorrenza richiama sé stessa due volte con argomenti diversi. La radice dell'albero di ricorsione ha costo n^2 che, come detto, corrisponde alle operazioni di *divide* e *combina*. La radice ha due figli: uno associato al sottoproblema avente dimensione $n/4$ volte quella del problema originale, l'altro associato al sottoproblema avente dimensione $n/2$ volte quella del problema originale. A tale livello corrisponderà un costo dato da $(n/4)^2 + (n/2)^2 = \frac{5}{16}n^2$. Passando al livello successivo, il nodo $(n/4)^2$ avrà due figli: $(n/16)^2$ e $(n/8)^2$, invece il nodo $(n/2)^2$ avrà figli: $(n/8)^2$ e $(n/4)^2$. A tale livello corrisponderà quindi un costo pari a $(n/16)^2 + (n/8)^2 + (n/8)^2 + (n/4)^2 = \frac{25}{256}n^2$. Continuando a sviluppare l'albero di ricorsione fino ad arrivare ad avere foglie che rappresentano il caso base ($\Theta(1)$) si verifica che l'albero non è bilanciato, cioè l'altezza dal nodo principale alle foglie non è uguale per tutte le foglie, in particolare si nota che il lato sinistro dell'albero ha altezza $(1/4)^i$ mentre il lato destro ha altezza $(1/2)^i$ risultando dunque più profondo del sinistro. Andando ora a sommare i costi per livello si ottiene:

$$Totale = n^2\Big(1 + \frac{5}{16} + \Big(\frac{5}{16}\Big)^2 + \Big(\frac{5}{16}\Big)^3 + \Big(\frac{5}{16}\Big)^4 + ...\Big) \tag{5.3}$$

dove l'argomento della parentesi rappresenta una serie geometrica che converge a $\frac{1}{1-\frac{5}{16}} \approx 1.42$. Tuttavia l'uguaglianza (5.3) è vera solo al livello in cui termina il lato sinistro dell'albero che, come detto, è più corto che lato destro. Questo significa che il valore ottenuto da tale equazione **non** è il risultato della ricorrenza ma solo una stima inferiore oltre la quale non si può fare di meglio, corrisponde quindi ad un Ω. Il lato destro, che terminerà invece ad un livello avente costo unitario (caso base), dà una stima superiore corrispondente quindi ad un O. Il caso peggiore di questa ricorrenza corrisponde ad un triangolo pieno che avrebbe altezza $lg\, n$.

Esempio 2

Data $T(n) = T(n/3) + T(2n/3) + \Theta(n)$, il limite superiore ($O$) è definito dalla disuguaglianza $T(n) \leq T(n/3) + T(2n/3) + cn$, quello inferiore ($\Omega$), invece, da $T(n) \geq T(n/3) + T(2n/3) + cn$. La radice ha costo cn e ha due figli: $c(n/3)$ e $c(2n/3)$. Il costo per livello è pari a $c(n/3) + c(2n/3) = cn$. A loro volta $c(n/3)$ avrà figli $c(n/9)$ e $c(2n/9)$, mentre $c(2n/3)$ avrà figli $c(2n/9)$ e $c(4n/9)$. Ogni livello ha costo cn e l'albero ha $log_3\, n$ livelli pieni, questo perché l'albero non è bilanciato, il lato sinistro è più corto e ha altezza $3^x = n \Rightarrow x = log_3\, n$. Il lato destro, e quindi l'albero, ha altezza $log_{3/2}\, n$. Si ottiene quindi:

- Idea per **limite superiore**: $\leq d\, n\, log_{3/2}\, n = O(n\, lg\, n)$
- Idea per **limite inferiore**: $\geq d\, n\, log_3\, n = \Omega(n\, lg\, n)$

Nel caso del limite superiore si ipotizza $T(n) \leq d\, n\, lg\, n$:

$$\begin{aligned}
T(n) &\leq T\Big(\frac{n}{3}\Big)+T\Big(\frac{2n}{3}\Big)+cn \leq d\Big(\frac{n}{3}\Big)lg\frac{n}{3} + d\Big(\frac{2n}{3}\Big)lg\frac{2n}{3} + cn \\
&= \Big(d\Big(\frac{n}{3}\Big)lg\, n - d\Big(\frac{n}{3}\Big)lg\, 3\Big)+\Big(d\Big(\frac{2n}{3}\Big)lg\, n - d\Big(\frac{2n}{3}\Big)lg\frac{3}{2}\Big)+cn \\
&= d\, n\, lg\, n - d\Big(\Big(\frac{n}{3}\Big)lg\, 3+\Big(\frac{2n}{3}\Big)lg\frac{3}{2}\Big)+cn \\
&= d\, n\, lg\, n - d\Big(\Big(\frac{n}{3}\Big)lg\, 3+\Big(\Big(\frac{2n}{3}\Big)lg\, 3 - \Big(\frac{2n}{3}\Big)lg\, 2\Big)+cn \\
&= d\, n\, lg\, n - dn\Big(lg\, 3 - \frac{2}{3}\Big) + cn \\
&\leq d\, n\, lg\, n
\end{aligned}$$

$\Rightarrow T(n) = O(n\, lg\, n)$ (valido se $-dn(lg\, 3 - 2/3) + cn \leq 0$, cioè per $d \geq \frac{c}{lg\, 3-2/3}$).
Nel caso del limite inferiore si ipotizza $T(n) \geq d\, n\, lg\, n$:

$$\begin{aligned}
T(n) &\geq T\Big(\frac{n}{3}\Big)+T\Big(\frac{2n}{3}\Big)+cn \leq d\Big(\frac{n}{3}\Big)lg\frac{n}{3} + d\Big(\frac{2n}{3}\Big)lg\frac{2n}{3} + cn \\
&= \Big(d\Big(\frac{n}{3}\Big)lg\, n - d\Big(\frac{n}{3}\Big)lg\, 3\Big)+\Big(d\Big(\frac{2n}{3}\Big)lg\, n - d\Big(\frac{2n}{3}\Big)lg\frac{3}{2}\Big)+cn \\
&= d\, n\, lg\, n - d\Big(\Big(\frac{n}{3}\Big)lg\, 3+\Big(\frac{2n}{3}\Big)lg\frac{3}{2}\Big)+cn \\
&= d\, n\, lg\, n - d\Big(\Big(\frac{n}{3}\Big)lg\, 3+\Big(\Big(\frac{2n}{3}\Big)lg\, 3 - \Big(\frac{2n}{3}\Big)lg\, 2\Big)+cn \\
&= d\, n\, lg\, n - dn\Big(lg\, 3 - \frac{2}{3}\Big) + cn \\
&\geq d\, n\, lg\, n
\end{aligned}$$

$\Rightarrow T(n) = \Omega(n \lg n)$ (valido per $0 < d \leq \frac{c}{lg\, 3-2/3}$).
Poiché si è verificato che $T(n) = O(n \lg n)$ e $T(n) = \Omega(n \lg n)$, si è dunque dimostrato che $T(n) = \Theta(n \lg n)$.

5.4 Metodo dell'esperto

Il **metodo dell'esperto** risolve le ricorrenze della forma

$$T(n) = a\, T(n/b) + f(n)$$

dove $a \geq 1$ e $b > 1$ sono costanti e $f(n)$ è una funzione asintoticamente positiva ($f(n) > 0$ per $n > n_0$). La ricorrenza considerata descrive il tempo di esecuzione di un algoritmo che divide un problema di dimensione n in a sottoproblemi, ciascuno di dimensione n/b, dove a e b sono costanti positive. I sottoproblemi vengono risolti in modo ricorsivo, ciascuno nel tempo $T(n/b)$. La funzione $f(n)$ comprende il costo per dividere il problema e combinare i risultati dei sottoproblemi. $T(n/b)$ indica $\lfloor n/b \rfloor$ o $\lceil n/b \rceil$.

Con il metodo dell'esperto $T(n)$ può essere asintoticamente limitata distinguendo tre casi:

1. Se $f(n) = O(n^{log_b a - \epsilon})$ per qualche costante $\epsilon > 0$, allora $T(n) = \Theta(n^{log_b a})$. In questo caso il costo è dominato dal numero delle foglie, cresce dalla radice alle foglie ed ϵ indica che la soluzione sta abbastanza sotto $n^{log_b a}$.

2. Se $f(n) = \Theta(n^{log_b a})$, allora $T(n) = \Theta(n^{log_b a} \, lg\, n)$. In questo caso il costo è all'incirca lo stesso in ognuno dei $log_b\, n$ livelli.

3. Se $f(n) = \Omega(n^{log_b a + \epsilon})$ per qualche costante $\epsilon > 0$ e se $f(n)$ è tale che $af(n/b) \leq cf(n)$ per qualche costante $c < 1$ e $\forall\, n \geq n_0$ (**condizione di regolarità**), allora $T(n) = \Theta(f(n))$. In questo caso il costo è dominato dalla radice, decresce dalla radice alle foglie (numero delle foglie decresce velocemente) ed ϵ indica che la soluzione sta abbastanza sopra $n^{log_b a}$.

Si può ottenere un'intuizione del teorema dell'esperto a partire dall'albero di ricorsione relativo a

$$T(n) = a\, T(n/b) + f(n)$$

La radice dell'albero ha costo $f(n)$ mentre il costo per livello è dato da $a^i f(n/b^i)$ fino ad arrivare alle foglie aventi costo $T(1)$ alle quali è associato un costo per livello pari a $n^{log_b a} T(1)$. Si arriva alle foglie quando $n/b^n = 1 \Rightarrow n = b^h \Rightarrow h = log_b n$, quindi l'altezza dell'albero è $h = log_b n$ e il numero di foglie è $a^h = a^{log_b n} = n^{log_b a}$.

In ciascuno dei tre casi, si confronta $n^{log_b a}$ con $f(n)$ e la soluzione dipende dalla **più grande** delle due. Se, come nel caso 1, la funzione $n^{log_b a}$ è la più grande, allora la soluzione è $T(n) = \Theta(n^{log_b a})$. Se, come nel caso 3, la funzione $f(n)$ è la più grande, allora la soluzione è $T(n) = \Theta(f(n))$. Se, come nel caso 2, le due funzioni hanno la stessa dimensione, si moltiplica per un fattore logaritmico e la soluzione è $T(n) = \Theta(n^{log_b a} \, lg\, n) = \Theta(f(n) \, lg\, n)$.

Nel primo caso, $f(n)$ non solo deve essere più piccola di $n^{log_b a}$, ma deve essere polinomialmente più piccola; ovvero $f(n)$ deve essere asintoticamente più piccola di $n^{log_b a}$ per un fattore n^ϵ per qualche costante $\epsilon > 0$. Nel terzo caso,

$f(n)$ non solo deve essere più grande di $n^{log_b a}$, ma deve essere polinomialmente più grande e soddisfare anche la condizione di regolarità $af(n/b) \leq cf(n)$. Questa condizione è soddisfatta dalla maggior parte delle funzioni polinomialmente limitate.

I tre casi **non coprono** tutte le funzioni possibili. C'è un intervallo fra i casi 1 e 2 in cui $f(n)$ è minore di $n^{log_b a}$ ma non in modo polinomiale. Analogamente, c'è un intervallo fra i casi 2 e 3 in cui $f(n)$ è maggiore di $n^{log_b a}$ ma non in modo polinomiale. Se la funzione $f(n)$ ricade in uno di questi intervalli o se la condizione di regolarità nel caso 3 non è soddisfatta, il metodo dell'esperto non può essere usato per risolvere la ricorrenza.

Teorema 5.1. *La **condizione di regolarità** che compare nel terzo caso vale sempre se $f(n) = n^k$ e $f(n) = \Omega(n^{log_b a+\epsilon})$ per qualche costante $\epsilon > 0$. Quindi è soddisfatta se $f(n)$ è un polinomio.*

Dimostrazione. Si considerano $f(n) = \Omega(n^{log_b a+\epsilon})$ e $f(n) = n^k$

$\Rightarrow n^k > n^{log_b a+\epsilon} = n^{log_b a}\, n^\epsilon > n^{log_b a}$

$\Rightarrow k > log_b a$

- I due lati come esponenti con base b:

  $b^k > b^{log_b a} = a \Rightarrow a/b^k < 1$

- a, b, k costanti $\Rightarrow$ si sceglie $c = a/b^k \Rightarrow c < 1$ costante

- $af(n/b) = a(n/b)^k = (a/b^k)n^k = cf(n)$

  $\Rightarrow$ condizione di regolarità soddisfatta

□

Esempio 1

$T(n) = 4\,T(\frac{n}{2}) + n$
Questa ricorrenza ha $a = 4$ e $b = 2 \Rightarrow n^{log_b a} = n^2$ e $f(n) = n$.
Caso 1: $f(n) = O(n^{2-\epsilon})$ con $\epsilon = 1 \Rightarrow T(n) = \Theta(n^2)$.

Esempio 2

$T(n) = 4\,T(\frac{n}{2}) + n^2$
Questa ricorrenza ha $a = 4$ e $b = 2 \Rightarrow n^{log_b a} = n^2$ e $f(n) = n^2$.
Caso 2: $f(n) = \Theta(n^2) \Rightarrow T(n) = \Theta(n^2\, lg\, n)$.

Esempio 3

$T(n) = 4\,T(\frac{n}{2}) + n^3$
Questa ricorrenza ha $a = 4$ e $b = 2 \Rightarrow n^{log_b a} = n^2$ e $f(n) = n^3$.
Caso 3: $f(n) = \Omega(n^{2+\epsilon})$ con $\epsilon = 1$ e condizione di regolarità $4(\frac{n}{2})^3 \leq cn^3$ con $c = \frac{1}{2} \Rightarrow T(n) = \Theta(n^3)$.

Esempio 4

$T(n) = 4\,T(\frac{n}{2}) + \frac{n^2}{lg\,n}$
Questa ricorrenza ha $a = 4$ e $b = 2 \Rightarrow n^{log_b a} = n^2$ e $f(n) = \frac{n^2}{lg\,n} \Rightarrow n^2$ vs. $\frac{n^2}{lg\,n}$.

- Sicuramente $\frac{n^2}{lg\,n} \neq \Omega(n^2)$ quindi **non** può essere **caso 2** e **caso 3**.
- Potrebbe essere **caso 1** ?

  Dovrebbe essere: $f(n) = O(n^{2-\epsilon}) \Rightarrow \frac{n^2}{lg\,n} < \frac{n^2}{n^\epsilon} \Rightarrow n^\epsilon < lg\,n$ ma $\forall\,\epsilon > 0$ si ha $n^\epsilon = \omega(lg\,n) \Rightarrow$ non si applica il metodo dell'esperto!

5.4.1 Ricerca binaria

Trovare un elemento in un array ordinato:

Divide Verificare l'elemento di mezzo

Impera Cercare ricorsivamente in un sottoarray

La **ricorrenza** per la ricerca binaria è la seguente:

$$T(n) = 1\,T(n/2) + \Theta(1)$$

1 è il numero di foglie e ogni livello ha costo costante $\Theta(1)$.
Dal teorema dell'esperto: $n^{log_b a} = n^{log_2 1} = n^0 = 1 \Rightarrow$ **Caso 2:** $T(n) = \Theta(lg\,n)$.

5.4.2 Potenza intera

Il problema consiste nel calcolare a^n, dove $n \in \mathbb{N}$.
L'algoritmo intuitivo ha costo $\Theta(n)$, invece l'algoritmo divide et impera è definito come:

$$a^n = \begin{cases} a^{n/2} \cdot a^{n/2} & \text{se } n \text{ pari} \\ a^{n/2} \cdot a^{n/2} \cdot a & \text{se } n \text{ dispari} \end{cases}$$

$T(n) = T(n/2) + \Theta(1) \Rightarrow T(n) = \Theta(lg\,n)$.
Non si può applicare il metodo dell'esperto perché $T(n) = T(n-1) + c$. È sufficiente calcolare $a^{n/2}$ una sola volta.

5.4.3 Moltiplicazione di matrici

Date due matrici A e B, di dimensione $n \times n$, aventi elementi a_{ij} e b_{ij} la matrice $C = A \cdot B$ si può calcolare per $i, j = 1, 2, \ldots, n$ con:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

In totale si calcolano n^2 elementi ciascuno dei quali è la somma di n valori. Questa modalità ha costo $\Theta(n^3)$.

L'algoritmo SQUARE-MATRIX-MULTIPLY(A, B) calcola il prodotto tra le matrici A e B richiedendo un tempo $\Theta(n^3)$. L'algoritmo di Strassen moltiplica due matrici richiedendo un tempo $o(n^3)$(in realtà $\Theta(n^{lg\,7})$). $O(n^{2.81})$ è asintoticamente migliore di SQUARE-MATRIX-MULTIPLY. Il motivo per cui SQUARE-MATRIX-MULTIPLY costa $\Theta(n^3)$ è dovuto al fatto che nei tre cicli **for** non ci

sono condizioni di uscita, quindi ciascuno di essi esegue esattamente n iterazioni e ciascuna esecuzione della riga 7 richiede un tempo costante.

```
SQUARE-MATRIX-MULTIPLY(A, B)
1   n ← A.rows
2   Sia C una nuova matrice n × n
3   for i ← 1 to n
4       for j ← 1 to n
5           c_ij ← 0
6           for k ← 1 to n
7               c_ij ← c_ij + a_ik · b_kj
8   return C
```

Per l'algoritmo divide et impera per moltiplicare matrici si suppone che n sia un **potenza esatta di 2** (finché $n \geq 2$ si ha $n/2$ intero). Si dividono le matrici A, B e C in quattro matrici $n/2 \times n/2$.

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \; B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \; C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

L'equazione $C = A \cdot B$ può essere scritta come:

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

Che corrisponde alle quattro equazioni:

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}$$
$$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}$$
$$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}$$
$$C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}$$

Ciascuna di queste quattro equazioni specifica due prodotti di matrici $n/2 \times n/2$ e la somma dei loro prodotti $n/2 \times n/2$. È possibile utilizzare queste equazioni per creare un algoritmo ricorsivo divide et impera:

```
SQUARE-MATRIX-MULTIPLY-RECURSIVE(A, B)
1   n ← A.rows
2   Sia C una nuova matrice n × n
3   if n = 1
4       c_11 = a_11 · b_11
5   else suddividi A, B e C
6       C_11 = SMMR(A_11, B_11) + SMMR(A_12, B_21)
7       C_12 = SMMR(A_11, B_12) + SMMR(A_12, B_22)
8       C_21 = SMMR(A_21, B_11) + SMMR(A_22, B_21)
9       C_22 = SMMR(A_21, B_12) + SMMR(A_22, B_22)
10  return C
```

Nella riga 5 la suddivisione delle matrici avviene utilizzando il calcolo degli indici: si andrà ad identificare una sottomatrice mediante un intervallo di indici di riga e un intervallo di indici di colonna della matrice originale. Alla fine si

otterrà ogni sottomatrice specificando solo un intervallo di indici ed è proprio questa modalità che consente di eseguire la riga 5 impiegando un tempo non $\Theta(n^2)$ ma $\Theta(1)$. Nelle righe 6 - 9 intervengono somme tra matrici $n/2 \times n/2$ che costano molto.

Si indica con $T(n)$ il tempo per moltiplicare due matrici $n \times n$ utilizzando la procedura SQUARE-MATRIX-MULTIPLY-RECURSIVE. Nel caso base, quando $n = 1$, si esegue l'unica moltiplicazione scalare nella riga 4, e quindi $T(1) = \Theta(1)$. Il caso ricorsivo si ha per $n > 1$. La divisione delle matrici nella riga 5 richiede un tempo $\Theta(1)$ se si utilizza il calcolo degli indici. Le righe 6 - 9 chiamano ricorsivamente SQUARE-MATRIX-MULTIPLY-RECURSIVE otto volte in totale. Poiché ogni chiamata ricorsiva moltiplica due matrici $n/2 \times n/2$, apportando un contributo di $T(n/2)$ al tempo di esecuzione totale, il tempo richiesto da tutte e otto le chiamate ricorsive è $8\,T(n/2)$. Bisogna anche considerare le quattro somme di matrici nelle righe 6 - 9. Ciascuna di queste matrici contiene $n^2/4$ elementi; quindi ciascuna delle quattro somme di matrici richiede un tempo $\Theta(n^2)$. Poiché il numero di somme di matrici è una costante, il tempo totale impiegato per sommare le matrici nelle righe 6 - 9 è $\Theta(n^2)$. Il tempo totale per il caso ricorsivo, quindi, è la somma del tempo per dividere le matrici, del tempo per eseguire tutte le chiamate ricorsive e del tempo per sommare le matrici risultanti dalle chiamate ricorsive:

$$\begin{aligned} T(n) &= \Theta(1) + 8\,T(n/2) + \Theta(n^2) \\ &= 8\,T(n/2) + \Theta(n^2) \end{aligned}$$

La ricorrenza per il tempo di esecuzione di SQUARE-MATRIX-MULTIPLY-RECURSIVE è:

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ 8\,T(n/2) + \Theta(n^2) & \text{se } n > 1 \end{cases}$$

che ha soluzione $T(n) = \Theta(n^3)$. Quindi SQUARE-MATRIX-MULTIPLY-RECURSIVE non è più veloce della procedura SQUARE-MATRIX-MULTIPLY.

Metodo di Strassen

Rende **meno ramificato** l'albero di ricorsione; ovvero, anziché eseguire otto moltiplicazioni ricorsive di matrici $n/2 \times n/2$, ne saranno eseguite soltanto sette. Il metodo di Strassen è composto da quattro passi:

1. Divide le matrici A, B e C in sottomatrici $n/2 \times n/2$ richiedendo un tempo $\Theta(1)$.

2. Crea dieci matrici S_1, S_2, S_3, ... S_{10}, ciascuna delle quali ha dimensione $n/2 \times n/2$ ed è la somma o differenza di due matrici create nel passo 1. Per creare tutte e dieci le matrici impiega un tempo $\Theta(n^2)$.

3. Utilizzando le sottomatrici create nel passo 1 e le dieci matrici create nel passo 2, calcola ricorsivamente sette matrici prodotto P_1, P_2, P_3, ... P_7 ciascuna avente dimensione $n/2 \times n/2$.

4. Calcola le sottomatrici $C_{1\,1}$, $C_{1\,2}$, $C_{2\,1}$, $C_{2\,2}$ della matrice C sommando e/o sottraendo varie combinazioni delle matrici P_i. Richiede un tempo $\Theta(n^2)$.

Si suppone che, quando la dimensione della matrice n si riduce a 1, si esegua un semplice prodotto scalare, come nella riga 4 della procedura SQUARE-MATRIX-MULTIPLY-RECURSIVE. Quando $n > 1$, i passi 1 2 e 4 richiedono un tempo totale di $\Theta(n^2)$, e il passo 3 richiede di eseguire sette moltiplicazioni di matrici $n/2 \times n/2$. Quindi si ottiene la seguente ricorrenza per il tempo di esecuzione $T(n)$ dell'algoritmo di Strassen:

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ 7\,T(n/2) + \Theta(n^2) & \text{se } n > 1 \end{cases}$$

Si è scambiato una moltiplicazione di matrici con un numero costante di somme di matrici. Questo scambio porta a un tempo di esecuzione asintotico più basso. Per il metodo dell'esperto la ricorrenza ottenuta ha soluzione $T(n) = \Theta(n^{lg\,7})$.

Passo 2
$S_1 \leftarrow B_{1\,2} - B_{2\,2}$
$S_2 \leftarrow A_{1\,1} + A_{1\,2}$
$S_3 \leftarrow A_{2\,1} + A_{2\,2}$
$S_4 \leftarrow B_{2\,1} - B_{1\,1}$
$S_5 \leftarrow A_{1\,1} + A_{2\,2}$
$S_6 \leftarrow B_{1\,1} + B_{2\,2}$
$S_7 \leftarrow A_{1\,2} - A_{2\,2}$
$S_8 \leftarrow B_{2\,1} + B_{2\,2}$
$S_9 \leftarrow A_{1\,1} - A_{2\,1}$
$S_{10} \leftarrow B_{1\,1} + B_{1\,2}$

Passo 3
$P_1 \leftarrow A_{1\,1} \cdot S_1 = A_{1\,1} \cdot B_{1\,2} - A_{1\,1} \cdot B_{2\,2}$
$P_2 \leftarrow S_2 \cdot B_{2\,2} = A_{1\,1} \cdot B_{2\,2} + A_{1\,2} \cdot B_{2\,2}$
$P_3 \leftarrow S_3 \cdot B_{1\,1} = A_{2\,1} \cdot B_{1\,1} + A_{2\,2} \cdot B_{1\,1}$
$P_4 \leftarrow A_{2\,2} \cdot S_4 = A_{2\,2} \cdot B_{2\,1} - A_{2\,2} \cdot B_{1\,1}$
$P_5 \leftarrow S_5 \cdot S_6 = A_{1\,1} \cdot B_{1\,1} + A_{1\,1} \cdot B_{2\,2} + A_{2\,2} \cdot B_{1\,1} + A_{2\,2} \cdot B_{2\,2}$
$P_6 \leftarrow S_7 \cdot S_8 = A_{1\,2} \cdot B_{2\,1} + A_{1\,2} \cdot B_{2\,2} - A_{2\,2} \cdot B_{2\,1} - A_{2\,2} \cdot B_{2\,2}$
$P_7 \leftarrow S_9 \cdot S_{10} = A_{1\,1} \cdot B_{1\,1} + A_{1\,1} \cdot B_{1\,2} - A_{2\,1} \cdot B_{1\,1} - A_{2\,1} \cdot B_{1\,2}$

Passo 4
$C_1 \leftarrow P_5 + P_4 - P_2 + P_6$
$C_2 \leftarrow P_1 + P_2$
$C_3 \leftarrow P_3 + P_4$
$C_4 \leftarrow P_5 + P_1 - P_3 - P_7$

Capitolo 6

Analisi probabilistica e algoritmi randomizzati

6.1 Il problema delle assunzioni

Si suppone di dover assumere un nuovo impiegato rivolgendosi a un'agenzia di selezione del personale che invia un candidato al giorno. Dopo il colloquio si decide subito se assumere o meno il candidato e, in caso positivo, si licenzierà l'attuale impiegato. Si deve pagare all'agenzia un compenso per avere un colloquio con il candidato, tale costo per candidato viene indicato con c_c. L'assunzione effettiva di un candidato costa di più perché si dovrà licenziare l'attuale impiegato e pagare un consistente compenso all'agenzia; tale costo di assunzione viene indicato con c_a. Si suppone inoltre $c_a > c_c$ e che si vada ad assumere sempre il miglior candidato visto. L'obiettivo è quello di andare a determinare il costo di questa procedura che viene chiamata HIRE-ASSISTANT.

I candidati sono numerati da 1 a n. La procedura suppone che, dopo avere avuto un colloquio con il candidato i, si è subito in grado di determinare se questo candidato è il migliore fra quelli intervistati fino a quel momento. All'inizio, la procedura crea un candidato fittizio (con numero 0), che è sempre assunto.

```
HIRE-ASSISTANT(n)
1   best ← 0
2   for i ← 1 to n
3       Colloquio con il candidato i
4       if (il candidato i è migliore del candidato best)
5           best ← i
6           Assumi candidato i
```

Nel raggiungimento dell'obiettivo non si è interessati al tempo di esecuzione di HIRE-ASSISTANT, ma bensì ai costi richiesti per il colloquio e l'assunzione. Le tecniche analitiche adottate sono identiche sia quando si valutano i costi sia quando si valuta il tempo di esecuzione. In entrambi i casi si conta il numero di volte che vengono eseguite determinate operazioni elementari.

Se, su n candidati, se ne assumono m, il costo totale associato all'algoritmo è $O(n\,c_c + m\,c_a)$. Indipendentemente dal numero di persone assunte, si dovrà

sempre avere un colloquio con n candidati e quindi si avrà sempre il costo $n\,c_c$ associato ai colloqui. Quindi si concentrerà l'attenzione sul costo di assunzione $m\,c_a$. Questa quantità varia ogni volta che viene eseguito l'algoritmo, ovvero, dipende dall'**ordine** dei candidati.

Nel caso peggiore, si assume ogni candidato con il quale si ha il colloquio. Questa situazione si verifica se i candidati si presentano in ordine strettamente crescente di qualità, nel qual caso si effettuano n assunzioni, con un costo totale per le assunzioni pari a $O(n\,c_a)$.

Nel caso migliore, invece, il primo assunto è il migliore, quindi si farà una solo assunzione.

6.2 Analisi probabilistica

L'**analisi probabilistica** è l'uso della probabilità nell'analisi dei problemi. Tipicamente, si utilizza l'analisi probabilistica per analizzare il tempo di esecuzione di un algoritmo. A volte, la si utilizza per analizzare altre grandezze. Per svolgere un'analisi probabilistica si deve conoscere la distribuzione degli input o almeno fare delle ipotesi su tale distribuzione. Si analizza quindi l'algoritmo calcolando un tempo di esecuzione nel caso medio, dove la media è fatta sulla distribuzione degli input possibili. Quindi si sta mediando il tempo di esecuzione su tutti gli input possibili. Questo tempo di esecuzione è detto **tempo di esecuzione nel caso medio**.

Per il problema delle assunzioni è possibile supporre che i candidati arrivino in ordine casuale. Si suppone di poter confrontare due candidati qualsiasi e decidere quale dei due abbia i requisiti migliori; ovvero c'è un ordine totale nei candidati. Di conseguenza, è possibile classificare ogni candidato con un numero d'ordine unico da 1 a n, utilizzando **rango(i)** per indicare il **numero d'ordine (rango)** del candidato i, e adottare la convenzione che a un rango più alto corrisponda un candidato più qualificato. La lista ordinata $\langle rango(1), rango(2), \ldots, rango(n)\rangle$ è una permutazione della lista $\langle 1, 2, \ldots, n\rangle$. Dire che i candidati si presentano in ordine casuale equivale a dire che questa lista di ranghi ha la stessa probabilità di essere una qualsiasi delle $n!$ permutazioni dei numeri da 1 a n. In alternativa, si può dire che i ranghi formano una **permutazione casuale uniforme**, ovvero che ciascuna delle $n!$ possibili permutazioni si presenta con uguale probabilità.

6.3 Variabili casuali indicatrici

Dato uno spazio dei campioni S e un evento A, si definisce la variabile casuale indicatrice:

$$I\{A\} = \begin{cases} 1 & \text{se si verifica } A \\ 0 & \text{se non si verifica } A \end{cases}$$

Queste variabili offrono un metodo comodo per la conversione tra probabilità e valori attesi. Il valore atteso di una variabile casuale indicatrice associata a un evento A è uguale alla probabilità che si verifichi A.

Lemma 6.1. *Se S è lo spazio dei campioni e A è un evento nello spazio dei campioni S, ponendo $X_A = I\{A\}$, si ha $E[X_A] = Pr\{A\}$.*

Dimostrazione. È possibile dimostrare tale lemma considerando le definizioni di valore atteso e della variabile casuale indicatrice ed indicando con $\overline{A} = S - A$ il complementare di A:

$$\begin{aligned} E[X_A] &= E[I\{A\}] \\ &= 1 \cdot Pr\{A\} + 0 \cdot Pr\{\overline{A}\} \\ &= Pr\{A\} \end{aligned}$$

□

Si fa una media ponderata dei possibili valori di $I\{A\}$. Queste variabili sono utili per analizzare situazioni in cui si effettuano ripetutamente delle prove casuali.

Numero atteso di teste lanciando una moneta una volta

Si considera una moneta imparziale.
Lo spazio dei campioni è definito da $S = \{T, C\}$ mentre la probabilità che esca testa, uguale alla probabilità che esca croce, è $Pr\{T\} = Pr\{C\} = 1/2$. Si definisce la variabile casuale indicatrice $X_T = I\{T\}$ dove X_T conta il numero di teste in un lancio. Dal lemma si ottiene:
$E[X_T] = Pr\{T\} = 1/2$.

Numero atteso di teste in n lanci

Questo numero NON è una probabilità!
Si definisce X una variabile casuale per il numero di teste in n lanci a cui è associato un valore atteso:

$$E[X] = \sum_{k=0}^{n} k \cdot Pr\{X = k\}$$

La variabile casuale indicatrice $X_I = I\{$ i-mo lancio è evento T $\}$.

$$X = \sum_{i=1}^{n} X_i$$

Il numero atteso di teste (T) è:

$$E[X] = E[\sum_{i=1}^{n} X_i] = \sum_{i=1}^{n} E[X_i] = \sum_{i=1}^{n} Pr\{T\} = \sum_{i=1}^{n} 1/2 = n/2$$

Per la linearità dei valori attesi, il valore atteso della somma è uguale alla somma dei valori attesi.

6.3.1 Analisi del problema delle assunzioni

Si vuole calcolare il numero previsto di volte che si assume un nuovo impiegato. Per applicare l'analisi probabilistica si suppone che i candidati arrivino in ordine **casuale**. Sia X la variabile casuale il cui valore è uguale al numero di volte che

si assume un nuovo impiegato. Applicando la definizione di valore atteso si ottiene:

$$E[X] = \sum_{x=1}^{n} x \cdot Pr\{X = x\}$$

Il calcolo di questa espressione non è semplice, per questo motivo su utilizzano le variabili casuali indicatrici per semplificarlo notevolmente.

Per utilizzare le variabili casuali indicatrici, anziché calcolare $E[X]$ definendo una variabile associata al numero di volte che si assume un nuovo impiegato, si definiscono n variabili correlate al fatto che il candidato venga assunto oppure no. In particolare, si indica con X_i la variabile casuale indicatrice associata all'evento in cui l'i-esimo candidato sia assunto, ovvero:

$$X_i = I\{\text{ il candidato } i \text{ è assunto }\} = \begin{cases} 1 & \text{se il candidato } i \text{ è assunto} \\ 0 & \text{se il candidato } i \text{ non è assunto} \end{cases}$$

E

$$X = X_1 + X_2 + X_3 + ... + X_n$$

Per il lemma precedentemente dimostrato si ottiene che:

$$E[X_i] = Pr\{\text{ il candidato } i \text{ è assunto }\}$$

Quindi si deve calcolare la probabilità che le righe 5 - 6 di HIRE-ASSISTANT siano eseguite. Ovvero, si deve calcolare $Pr\{$ il candidato i è assunto $\}$.

Il candidato i è assunto se e solo se è migliore di tutti i precedenti candidati da 1 a $i-1$. Poiché si è ipotizzato che i candidati arrivino in ordine casuale, i primi i candidati si sono presentati in ordine casuale. Uno qualsiasi dei primi i candidati ha la stessa probabilità di essere classificato come il migliore di tutti. Il candidato i ha la probabilità $1/i$ di essere qualificato migliore dei candidati da 1 a $i-1$ e, quindi, ha la probabilità $1/i$ di essere assunto (prima arriva più ha una alta probabilità di essere assunto). Per il lemma si conclude che:

$$E[X_i] = \frac{1}{i}$$

Adesso è possibile calcolare $E[X]$:

$$\begin{aligned} E[X] &= E\left[\sum_{i=1}^{n} X_i\right] \\ &= \sum_{i=1}^{n} E[X_i] \\ &= \sum_{i=1}^{n} \frac{1}{i} \\ &= ln\, n + O(1) \end{aligned}$$

Questo perchè $\sum_{k=1}^{n} \frac{1}{k}$ è la serie armonica.
Nonostante vengano intervistati n candidati, ne verranno assunti soltanto approssimativamente $\lg n$, in media. Quindi, supponendo che i candidati si presentino in ordine casuale, l'algoritmo HIRE-ASSISTANT ha un costo totale per le assunzioni pari a $O(c_a\, lg\, n)$. Il costo per le assunzioni nel caso medio è un significativo miglioramento rispetto al costo $O(n\, c_a)$ per le assunzioni nel caso peggiore.

6.4 Algoritmi randomizzati

Per poter utilizzare l'analisi probabilistica, si deve sapere qualcosa sulla distribuzione degli input. In molti casi però si hanno scarse (se non nulle) informazioni su tale distribuzione. Anche quando si hanno delle informazioni sulla distribuzione degli input, si potrebbe non essere in grado di modellare computazionalmente questa conoscenza. È però spesso possibile utilizzare la probabilità e la causalità come strumento per progettare e analizzare gli algoritmi, rendendo casuale il comportamento di parte dell'algoritmo.

In generale si dice che un algoritmo è **randomizzato** se il suo comportamento è determinato non soltanto dal suo input, ma anche dai valori prodotti da un **generatore di numeri casuali**. Si suppone di avere a disposizione un generatore di numeri casuali, chiamato RANDOM. Una chiamata di RANDOM(a, b) restituisce un numero intero compreso tra a e b estremi inclusi; ciascuno di questi numeri interi ha la stessa probabilità di essere generato. Ogni intero generato da RANDOM è indipendente dagli interi generati nelle precedenti chiamate.

Quando si analizza il tempo di esecuzione di un algoritmo randomizzato, si considera il valore atteso del tempo di esecuzione rispetto alla distribuzione dei valori restituiti dal generatore di numeri casuali. Si distinguono questi algoritmi da quelli in cui l'input è casuale, chiamando il tempo di esecuzione di un algoritmo randomizzato **tempo di esecuzione atteso**. In generale, si considera il tempo di esecuzione nel caso medio quando la distribuzione della probabilità riguarda gli input dell'algoritmo, mentre si considera il tempo di esecuzione atteso quando l'algoritmo stesso effettua delle scelte casuali.

6.4.1 Problema delle assunzioni randomizzato

Nel problema delle assunzioni, sembra che i candidati si presentino in ordine casuale, tuttavia non si ha modo di sapere se ciò sia vero o no. Quindi, per sviluppare un algoritmo randomizzato per il problema delle assunzioni, si deve avere un controllo maggiore sull'ordine in cui si svolgono i colloqui con i candidati; per questo motivo, si modificherà leggermente il modello. Si suppone, infatti, che l'agenzia di selezione del personale abbia n candidati e che invii in anticipo una lista dei candidati. Sebbene non si abbiano informazioni sui candidati, si ha un significativo cambiamento. Anziché fare affidamento sull'ipotesi che i candidati si presentino in ordine casuale, si ha ora il controllo del processo e si può quindi imporre un ordine casuale.

Anziché ipotizzare una distribuzione degli input, se ne impone una. In pratica, prima di eseguire l'algoritmo, si permutano casualmente i candidati per imporre la proprietà che ogni permutazione sia egualmente probabile. Benché l'algoritmo sia stato modificato, ci si aspetta ancora di assumere un nuovo impiegato approssimativamente $lg\, n$ volte. Ma ora ci si aspetta che questo accada per ogni input e non soltanto per quelli estratti da una particolare distribuzione.

Ogni volta che si esegue l'algoritmo, l'esecuzione dipende dalle scelte casuali fatte ed è probabile che sia diversa dalle precedenti esecuzioni. Per questo algoritmo e per molti altri algoritmi randomizzati, nessun input particolare determina il caso peggiore. L'algoritmo randomizzato si comporta male soltanto se il generatore di numeri casuali produce una permutazione sventurata.

Per il problema delle assunzioni, l'unica modifica da apportare al codice è permutare casualmente l'array.

```
RANDOMIZED-HIRE-ASSISTANT(n)
1  Permutare casualmente la lista dei candidati
2  HIRE-ASSISTANT(n)
```

Ovvero

```
RANDOMIZED-HIRE-ASSISTANT(n)
1  Permutare casualmente la lista dei candidati
2  best ← 0
3  for i ← 1 to n
4      Colloquio con il candidato i
5      if (il candidato i è migliore del candidato best)
6          best ← i
7          Assumi candidato i
```

Con questa modifica è stato creato un algoritmo randomizzato le cui prestazioni corrispondono a quelle ottenute supponendo che i candidati si presentino in ordine casuale. Il costo previsto per assumere nuovi impiegati nella procedura RANDOMIZED-HIRE-ASSISTANT è $O(c_a \, ln \, n)$, perché avendo permutato l'array di input si ha una situazione identica all'analisi probabilistica di HIRE-ASSISTANT deterministico.

La permutazione casuale di un array si realizza tramite permutazioni **sul posto**. Si vuole una **permutazione casuale uniforme**:

```
RANDOMIZE-IN-PLACE(n)
1  n ← A.length
2  for i ← 1 to n
3      scambia A[i] ↔ A[RANDOM(i, n)]
```

Richiede un tempo $O(1)$ per ogni iterazione, dunque $O(n)$ totale.

Capitolo 7

Quicksort

QUICKSORT è un algoritmo di ordinamento il cui tempo di esecuzione nel caso peggiore (quando gli elementi sono già ordinati) è $\Theta(n^2)$ con un array di input di n numeri. Nonostante questo tempo di esecuzione nel caso peggiore sia molto alto, QUICKSORT spesso è la soluzione pratica migliore per effettuare un ordinamento, perché mediamente è molto efficiente: il suo tempo di esecuzione atteso è $\Theta(n \, lg \, n)$ e i fattori costanti nascosti nella notazione $\Theta(n \, lg \, n)$ sono molto piccoli. Inoltre ha il vantaggio di ordinare **sul posto** e funziona bene anche in ambienti con memoria virtuale.

7.1 L'algoritmo Quicksort

L'algoritmo QUICKSORT, come MERGE-SORT, è basato sul paradigma divide et impera. Per ordinare $A[p \, ... \, r]$:

Divide Partiziona l'array $A[p \, ... \, r]$ in due sottoarray $A[p \, ... \, q \text{ - } 1]$ e $A[q + 1 \, ... \, r]$, eventualmente vuoti, tali che ogni elemento di $A[p \, ... \, q \text{ - } 1]$ sia minore o uguale del **pivot** $A[q]$ che, a sua volta, è minore o uguale a ogni elemento del sottoarray $A[q + 1 \, ... \, r]$. È in questa fase che viene calcolato q

Impera Ordina i due sottoarray $A[p \, ... \, q \text{ - } 1]$ e $A[q + 1 \, ... \, r]$ chiamando ricorsivamente QUICKSORT

Combina Poiché i sottoarray sono già ordinati, non occorre alcun lavoro per combinarli: l'intero array $A[p \, ... \, r]$ è ordinato

Per ordinare un intero array A, la chiamata iniziale è QUICKSORT(A, 1, $A.length$). La seguente procedura implementa QUICKSORT.

```
QUICKSORT(A, p, r)
1  if p < r
2      q ← PARTITION(A, p, r)
3      QUICKSORT(A, p, q − 1)
4      QUICKSORT(A, q + 1, r)
```

L'elemento chiave dell'algoritmo è la procedura PARTITION che riarrangia il sottoarray $A[p \, ... \, r]$ sul posto.

```
PARTITION(A, p, r)
1   x ← A[r]
2   i ← p − 1
3   for j ← p to r − 1
4       if A[j] ≤ x
5           i ← i + 1
6           A[i] ↔ A[j]
7   A[i + 1] ↔ A[r]
8   return i + 1
```

Commenti:

0 PARTITION è una funzione ausiliaria. Mette a sinistra gli elementi minori del pivot $A[r]$ e a destra quelli maggiori

2 La sottoparte minore del pivot va da p ad i

3 La sottoparte maggiore del pivot va da $i + 1$ a $j - 1$

7 Sposta il pivot nella posizione corretta

PARTITION seleziona sempre un elemento $x = A[r]$ come **pivot** intorno al quale partizionare il sottoarray $A[p \,..\, r]$. Durante l'esecuzione della procedura, l'array viene suddiviso in quattro regioni (eventualmente vuote). All'inizio di ogni iterazione del ciclo **for** (righe 3 - 6) ogni regione soddisfa l'invariante di ciclo:

*All'inizio di ogni iterazione del ciclo **for**, righe 3 - 6, si verifica che:*

1. $A[r] \leftarrow pivot$
2. $\forall\, A[x] \in A[p \,..\, i] : A[x] \leq pivot$
3. $\forall\, A[x] \in A[i + 1 \,..\, j - 1] : A[x] > pivot$
4. $\forall\, A[x] \in A[j \,..\, r - 1] :$ $A[x]$ *non è stato esaminato*

Si dimostra ora la sua validità:

Inizializzazione Prima della prima iterazione del ciclo, $i = p - 1$ e $j = p$. Non ci sono valori fra p ed i né fra $i + 1$ e $j - 1$, quindi la seconda e la terza condizione dell'invariante di ciclo sono soddisfatte. L'assegnazione della riga 1 soddisfa la prima

Conservazione Ci sono due casi da considerare a seconda del risultato del test nella riga 4. Se $A[j] > pivot$, l'unica azione nel ciclo è incrementare j. Dopo l'incremento di j la terza condizione è soddisfatta per $A[j - 1]$ e tutte le altre posizioni non cambiano. Se invece $A[j] \leq pivot$, viene incrementato l'indice i, vengono scambiati $A[i]$ e $A[j]$ e successivamente viene incrementato l'indice j. In seguito allo scambio si ha $A[i] \leq x$ e la seconda condizione è soddisfatta. Analogamente, anche $A[j - 1] > x$, in quanto l'elemento che è stato spostato in $A[j - 1]$ è, per l'invariante di ciclo, più grande di x

Conclusione Alla fine del ciclo $j = r$. Pertanto, ogni posizione dell'array si trova in uno dei tre insiemi descritti dall'invariante: quelli minori o uguali a x, quelli maggiori di x e un insieme di un solo elemento che contiene x

L'output di PARTITION soddisfa le specifiche del passo *divide*. In effetti, esso soddisfa una condizione un po' più severa: dopo la riga 2 di QUICKSORT, $A[q]$ è strettamente minore di $A[q+1 \,..\, r]$.

Il tempo di esecuzione di PARTITION con il sottoarray $A[p \,..\, r]$ è $\Theta(n)$, dove $n = r - p + 1$ è la dimensione dell'array di interesse.

7.2 Prestazioni di Quicksort

Il tempo di esecuzione di QUICKSORT dipende dal fatto che il partizionamento sia bilanciato o sbilanciato e questo, a sua volta, dipende da quali elementi vengono utilizzati per il partizionamento. Se il partizionamento è **bilanciato**, l'algoritmo viene eseguito con la stessa velocità asintotica di MERGE-SORT. Se il partizionamento è **sbilanciato**, l'algoritmo può essere asintoticamente lento quanto INSERTION-SORT. Il caso medio è analogo a MERGE-SORT.

7.2.1 Partizionamento nel caso peggiore

Il comportamento nel caso peggiore di QUICKSORT si verifica quando la routine di partizionamento produce un sottoproblema con $n-1$ elementi e uno con 0 elementi, quindi sottoarray completamente sbilanciati. Si suppone ora che tale sbilanciamento si verifichi in ogni chiamata ricorsiva. Il partizionamento costa $\Theta(n)$ in termini di tempo. Poiché per una chiamata ricorsiva su un array vuoto $T(0) = \Theta(1)$, la ricorrenza per il tempo di esecuzione può essere espressa così:

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) \\ &= \Theta(n^2) \end{aligned}$$

Applicando il metodo di sostituzione alla ricorrenza si arriva al risultato $\Theta(n^2)$.

In definitiva, il tempo di esecuzione nel caso peggiore di QUICKSORT non è migliore di quello di INSERTION-SORT. Inoltre, il tempo di esecuzione $\Theta(n^2)$ si ha quando l'array di input è già completamente ordinato - una situazione tipica in cui INSERTION-SORT è eseguito nel tempo $O(n)$, non a caso è il suo caso migliore.

7.2.2 Partizionamento nel caso migliore

Nel caso di bilanciamento massimo, cioè quando il pivot è il valore di mezzo, PARTITION produce due sottoproblemi, ciascuno di dimensione non maggiore di $n/2$, in quanto uno ha dimensione $\lfloor n/2 \rfloor$ e l'altro ha dimensione $\lceil n/2 \rceil - 1$. In questo caso, QUICKSORT viene eseguito molto più velocemente. La ricorrenza per il tempo di esecuzione è

$$T(n) = 2\,T(n/2) + \Theta(n)$$

Che per il **caso 2** del teorema dell'esperto, ha soluzione $T(n) = \Theta(n \lg n)$. Dunque, il perfetto bilanciamento dei due lati della partizione a ogni livello di ricorsione produce un algoritmo asintoticamente più veloce.

7.2.3 Partizionamento bilanciato

Il tempo di esecuzione nel caso medio di QUICKSORT è molto più vicino al caso migliore che al caso peggiore.

Supponendo, per esempio, che l'algoritmo di partizionamento produca sempre una ripartizione proporzionale 9 a 1, in questo caso si ottiene la ricorrenza

$$T(n) = T(9n/10) + T(n/10) + \Theta(n)$$

Che corrisponde alla disuguaglianza

$$T(n) \leq T(9n/10) + T(n/10) + cn$$

Ogni livello ha un costo cn, finché non viene raggiunta una condizione al contorno alla profondità $log_{10}\, n = \Theta(lg\, n)$, dopo la quale i livelli hanno al massimo un costo cn. Questo perché si hanno $log_{10}\, n$ livelli pieni e $log_{10/9}\, n$ non vuoti.

La ricorsione termina alla profondità $log_{10/9}\, n = \Theta(lg\, n)$. Il costo totale di QUICKSORT è dunque $O(n\, lg\, n)$. Pertanto, con una partizione proporzionale 9 a 1 ad ogni livello di ricorsione, QUICKSORT viene eseguito nel tempo $O(n\, lg\, n)$ - asintoticamente uguale a quello che si ha nel caso di partizione esattamente a metà. Qualsiasi ripartizione con proporzionalità costante produce un albero di ricorsione di profondità $\Theta(lg\, n)$, dove il costo in ogni livello è $O(n)$. Il tempo di esecuzione è quindi $O(n\, lg\, n)$ quando la ripartizione ha proporzionalità costante.

7.2.4 Intuizione sul caso medio

Il comportamento di QUICKSORT è determinato dall'ordinamento relativo dei valori degli elementi dell'array che sono dati come input, non dai particolari valori dell'array. Si suppone di alternare dei tagli buoni (ripartizione bilanciata - caso migliore) e cattivi (ripartizione molto sbilanciata - caso peggiore). Ciò significa che si alternano casi in cui il pivot è il valore di mezzo (bilanciato) e casi in cui il pivot è il valore massimo o minimo (completamente sbilanciato). L'albero di ripartizione ha radice con costo del partizionamento uguale a n e i sottoarray prodotti hanno dimensione 0 e $n-1$, ovvero il caso peggiore. Nel livello successivo il sottoarray di dimensione $n-1$ è ripartito in due sottoarray di dimensioni $\frac{n-1}{2} - 1$ e $\frac{n-1}{2}$, ovvero il caso migliore. Si suppone che il costo della condizione al contorno sia 1 per il sottoarray di dimensione 0.

La combinazione della ripartizione cattiva del caso peggiore, seguita dalla ripartizione buona del caso migliore, produce tre sottoarray di dimensioni pari a 0, $\frac{n-1}{2} - 1$ e $\frac{n-1}{2}$ con un costo di partizionamento complessivo dato da $\Theta(n) + \Theta(n-1) = \Theta(n)$ (ogni nodo a sinistra ha $\Theta(n) + \Theta(n-1) = \Theta(n)$, invece ogni nodo a destra $\Theta(n)$). Intuitivamente, il costo $\Theta(n-1)$ del caso peggiore può essere assorbito nel costo $\Theta(n)$ del caso migliore, quindi la ripartizione risultante è buona. In definitiva, il tempo di esecuzione di QUICKSORT, quando i livelli si alternano tra buone e cattive ripartizioni, è come il tempo di esecuzione nel caso in cui le ripartizioni siano soltanto buone: $O(n\, lg\, n)$ ma con una costante un po' più grande nascosta dalla notazione O.

7.3 Quicksort randomizzato

Nell'analisi del comportamento di QUICKSORT nel caso medio, si è supposto che tutte le permutazioni dei numeri di input fossero ugualmente probabili. Nel-

la pratica però, non è possibile aspettarsi che questo sia sempre vero. come già visto, a volte è possibile aggiungere la randomizzazione a un algoritmo per ottenere una buona prestazione attesa con tutti gli input. Per QUICKSORT si adotta un metodo di randomizzazione chiamato **campionamento casuale** che consente di semplificare l'analisi. Anziché utilizzare sempre $A[r]$ come pivot, si utilizzerà un elemento scelto a caso dal sottoarray $A[p \dots r]$. Per fare questo, si scambierà l'elemento $A[r]$ con un elemento scelto a caso da $A[p \dots r]$. Questa modifica, con la quale si campiona a caso l'intervallo $p \dots r$, assicura che l'elemento pivot $x = A[r]$ avrà la stessa probabilità di essere uno qualsiasi degli $r - p + 1$ elementi del sottoarray. Poiché il pivot viene scelto a caso, ci si aspetta che la ripartizione dell'array di input sia ben bilanciata in media.

Nella nuova procedura di partizionamento, si implementerà semplicemente lo scambio prima dell'effettivo partizionamento:

```
RANDOMIZED-PARTITION(A, p, r)
1  i ← RANDOM(p, r)
2  Scambia A[r] con A[i]
3  return PARTITION(A, p, r)
```

Il nuovo RANDOMIZED-QUICKSORT chiama RANDOMIZED-PARTITION, anziché PARTITION:

```
RANDOMIZED-QUICKSORT(A, p, r)
1  if p < r
2      q ← RANDOMIZED-PARTITION(A, p, r)
3      RANDOMIZED-QUICKSORT(A, p, q − 1)
4      RANDOMIZED-QUICKSORT(A, q + 1, r)
```

La randomizzazione impedisce a specifici input di causare sempre il caso peggiore. Per esempio: l'array ordinato è il caso peggiore per QUICKSORT, non per RANDOMIZED-QUICKSORT.

7.4 Analisi del caso peggiore di Quicksort

Come già visto, il caso peggiore di QUICKSORT, che si verifica quando ,ad ogni livello di ricorsione, si ha il peggior taglio, ha un tempo di esecuzione $\Theta(n^2)$.

Utilizzando il metodo di sostituzione applicato alla ricorrenza per il caso peggiore è possibile dimostrare che il tempo di esecuzione di QUICKSORT è $O(n^2)$ ed è anche $\Omega(n^2)$, quindi $\Theta(n^2)$. Sia $T(n)$ il tempo nel caso peggiore per la procedura QUICKSORT con un input di dimensione n. Si ottiene la ricorrenza

$$T(n) = \max_{0 \le q \le n-1} (T(q) + T(n - q - 1)) + \Theta(n)$$

La ricorrenza dipende da dove si trova il pivot rispetto ai dati. $T(q)$ è l'array sinistro, mentre $T(n - q - 1)$ è l'array destro (-1 indica il pivot). Supponendo $T(n) \le cn^2$ per qualche costante c e sostituendo, si ottiene:

$$\begin{aligned} T(n) &\le \max_{0 \le q \le n-1} (cq^2 + c(n - q - 1)^2) + \Theta(n) \\ &= c \cdot \max_{0 \le q \le n-1} (q^2 + (n - q - 1)^2) + \Theta(n) \end{aligned}$$

L'espressione $q^2 + (n - q - 1)^2$ è una parabola con la concavità verso l'alto che raggiunge il suo massimo nei due estremi dell'intervallo $0 \leq q \leq n - 1$ del parametro q; ovvero con $q = 0$ o $q = n - 1$ (si sceglie $q = 0$ che sbilancia del tutto). Questa osservazione fornisce il limite:

$$\max_{0 \leq q \leq n-1} (q^2 + (n - q - 1)^2) \leq (n - 1)^2 = n^2 - 2n + 1$$

Riprendendo l'espressione di $T(n)$, si ottiene:

$$\begin{aligned} T(n) &\leq cn^2 - c(2n - 1) + \Theta(n) \\ &\leq cn^2 \end{aligned}$$

perché si può assegnare alla costante c un valore sufficientemente grande affinché il termine $O(2n - 1)$ prevalga sul termine $\Theta(n)$; quindi $T(n) = O(n^2)$ è il caso peggiore. Ma il caso peggiore è anche $\Omega(n^2)$, quindi si ottiene $\Theta(n^2)$. Si è così dimostrato che $\Theta(n^2)$ è il caso peggiore di QUICKSORT.

7.5 Tempo di esecuzione atteso e confronti

Il tempo di esecuzione di QUICKSORT è dominato dal tempo impiegato dalla procedura PARTITION (cioè da quanti confronti vengono eseguiti in PARTITION; idealmente meno confronti si fanno, più è veloce, ma questo dipende dalla posizione e dal pivot cioè dalla disposizione). Ogni volta che viene chiamata la procedura PARTITION, viene selezionato un elemento pivot; questo elemento non sarà mai incluso nelle successive chiamate ricorsive di QUICKSORT e PARTITION. Quindi, ci possono essere al massimo n chiamate di PARTITION durante l'intera esecuzione dell'algoritmo QUICKSORT. Una chiamata di PARTITION impiega il tempo $O(1)$ più una quantità di tempo che è proporzionale al numero di iterazioni del ciclo **for** nelle righe 3 - 6. Ogni iterazione di questo ciclo **for** effettua un confronto fra l'elemento pivot e un altro elemento dell'array A (riga 4). Pertanto, se si conta il numero totale di volte che la riga 4 viene eseguita, è possibile limitare il tempo totale impiegato nel ciclo **for** durante l'intera esecuzione di QUICKSORT.

Se X è il numero **totale** di confronti svolti nella riga 4 di PARTITION nell'intera esecuzione di QUICKSORT su un array di n elementi, allora il tempo di esecuzione di QUICKSORT (cioè il suo costo totale) è $O(n + X)$.

L'obiettivo è quindi quello di calcolare un limite al numero totale di confronti X svolti in tutte le chiamate di PARTITION. Per semplificare l'analisi si rinominano $z_1, z_2, z_3, \ldots, z_n$ gli elementi dell'array A, dove $z_i \leq z_j$ se $i < j$ (ciò significa che ci si interessa all'ordine e non al valore). Si definisce anche $Z_{ij} = \{z_i, z_{i+1}, \ldots, z_j\}$ l'insieme degli elementi compresi tra z_i e z_j, estremi inclusi. Gli elementi z_i e z_j sono confrontati al massimo una volta, in particolare, sono confrontati solo col pivot (quindi uno dei due in un momento è il pivot che verrà posizionato) e, dopo che una particolare chiamata di PARTITION finisce, il pivot utilizzato in questa chiamata non viene più confrontato con nessun altro elemento, cioè non è presente in chiamate successive. Alla fine di PARTITION, il pivot è nel posto giusto e lì rimane.

Si definisce ora la variabile casuale indicatrice

$$X_{ij} = I\{z_i \text{ è confrontato con } z_j\}$$

Si stanno considerando i confronti che vengono eseguiti in un istante qualsiasi durante l'esecuzione dell'algoritmo, non soltanto durante un'iterazione o una chiamata di PARTITION. Poiché ogni coppia viene confrontata al massimo una volta, è possibile rappresentare facilmente il numero totale di confronti svolti dall'algoritmo in questo modo:

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$$

Prendendo i valori attesi da entrambi i lati e poi applicando la linearità del valore atteso, si ottiene:

$$\begin{aligned} E[X] &= E\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}] \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} Pr\{z_i \text{ è confrontato con } z_j\} \end{aligned}$$

Resta da calcolare $Pr\{z_i$ è confrontato con $z_j\}$.
Poiché si suppone che i valori degli elementi siano distinti, una volta che viene scelto un pivot x con $z_i < x < z_j$, si sa che z_i e z_j non potranno essere confrontati in un istante successivo. Se, d'altra parte, viene scelto z_i come pivot prima di qualsiasi altro elemento di Z_{ij}, allora z_i sarà confrontato con ogni elemento di Z_{ij}, tranne sé stesso. Analogamente, se viene scelto z_j come pivot prima di qualsiasi altro elemento di Z_{ij}, allora z_j sarà confrontato con ogni elemento di Z_{ij}, tranne sé stesso. Quindi z_i e z_j vengono confrontati se e soltanto se il primo elemento che verrà scelto come pivot in Z_{ij} è z_i o z_j. Schematizzando

- Numeri in partizioni diverse mai confrontati

  Se $z_i < pivot < z_j \Rightarrow z_i$ e z_j mai confrontati perché il pivot li separa e andranno in chiamate diverse

- Se z_i o z_j scelto per primo in Z_{ij} allora z_i o z_j confrontato con tutti gli altri elementi di Z_{ij}

  $\Rightarrow z_i$ e z_j confrontati $\Leftrightarrow$ il **primo** pivot in Z_{ij} è z_i o z_j

- Pivot scelti casualmente e indipendentemente $|Z_{ij}| = j - i + 1$

  $\Rightarrow$ probabilità z_i o z_j scelto per primo in Z_{ij} è $\frac{1}{j-i+1}$

Quindi si ha

$$\begin{aligned} Pr\{z_i \text{ è confrontato con } z_j\} &= Pr\{z_i \text{ o } z_j \text{ primo pivot scelto da } Z_{ij}\} \\ &= Pr\{z_i \text{ primo pivot scelto da } Z_{ij}\} \\ &= Pr\{z_j \text{ primo pivot scelto da } Z_{ij}\} \\ &= \frac{1}{j-i+1} + \frac{1}{j-i+1} \\ &= \frac{2}{j-i+1} \end{aligned}$$

Sostituendo in E$[X]$ si ottiene

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} Pr\{z_i \text{ è confrontato con } z_j\} = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

Si effettua un cambio di variabile $k = j - i$

$$\begin{aligned} E[X] &= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \\ &< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} \\ &= \sum_{i=1}^{n-1} O(lg\, n) \\ &= O(n \, lg\, n) \end{aligned}$$

Quindi il **tempo di esecuzione atteso** di RANDOMIZED-QUICKSORT è $O(n \, lg\, n)$.

Capitolo 8

Ordinamento in tempo lineare

Come visto, alcuni algoritmi possono ordinare n numeri nel tempo $O(n\,lg\,n)$. MERGE-SORT raggiunge questo limite superiore nel caso peggiore; QUICKSORT lo raggiunge nel caso medio. Inoltre, per ciascuno di questi algoritmi, è possibile produrre una sequenza di n numeri di input tale che l'algoritmo venga eseguito nel tempo $\Omega(n\,lg\,n)$.

Questi algoritmi sono detti **ordinamenti per confronti** perché l'ordinamento che effettuano è basato soltanto su confronti fra gli elementi di input. Qualsiasi ordinamento per confronti deve effettuare $\Omega(n\,lg\,n)$ confronti nel caso peggiore per ordinare n elementi. Quindi MERGE-SORT è asintoticamente ottimale e non esiste un ordinamento per confronti che sia più veloce per più di un fattore costante.

Esistono però degli algoritmi che vengono eseguiti in **tempo lineare**. Ovviamente questi algoritmi usano operazioni diverse dai confronti per effettuare l'ordinamento. Di conseguenza, il limite inferiore $\Omega(n\,lg\,n)$ non vale per questi algoritmi. Tra di essi compaiono COUNTING-SORT e RADIX-SORT.

8.1 Il modello dell'albero di decisione

Gli ordinamenti per confronti possono essere visti astrattamente in termini di **alberi di decisione**. Un albero di decisione è un albero binario pieno che rappresenta i confronti fra elementi che vengono effettuati da un particolare algoritmo di ordinamento che opera su un input di una data dimensione. Il controllo, lo spostamento dei dati e tutti gli altri aspetti dell'algoritmo vengono ignorati.

In un albero di decisione, ogni nodo interno è annotato con $i : j$ per qualche i e j nell'intervallo $1 \leq i,\, j \leq n$, dove n è il numero di elementi nella sequenza di input. Ogni foglia è annotata con una permutazione dei dati in input. L'esecuzione dell'algoritmo di ordinamento corrisponde a tracciare un cammino semplice dalla radice dell'albero di decisione fino a una foglia. Ogni nodo interno rappresenta un confronto $a_i \leq a_j$. Il sottoalbero sinistro detta i successivi confronti per $a_i \leq a_j$; il sottoalbero destro detta i successivi confronti per

$a_i > a_j$. Quando raggiunge una foglia, l'algoritmo ha stabilito l'ordinamento corretto dell'input.

Poiché qualsiasi algoritmo di ordinamento corretto deve essere in grado di produrre ogni permutazione del suo input, una condizione necessaria affinché un ordinamento per confronti sia corretto è che ciascuna delle $n!$ permutazioni di n elementi appaia come una delle foglie dell'albero di decisione e che ciascuna di queste foglie sia raggiungibile dalla radice attraverso un percorso che corrisponde ad una effettiva esecuzione dell'ordinamento per confronti (queste foglie sono chiamata raggiungibili).

8.1.1 Limite inferiore per il caso peggiore

La lunghezza del cammino semplice più lungo dalla radice di un albero di decisione a una delle sue foglie raggiungibili rappresenta il numero di confronti che svolge il corrispondente algoritmo di ordinamento nel caso peggiore. Di conseguenza, il numero di confronti nel caso peggiore per un dato algoritmo di ordinamento per confronti è uguale all'altezza del suo albero di decisione. Un limite inferiore sulle altezze di tutti gli alberi di decisione , dove ogni permutazione compare in una foglia raggiungibile, è pertanto un limite inferiore sul tempo di esecuzione di qualsiasi algoritmo di ordinamento per confronti.

Teorema 8.1. *Ogni albero binario di altezza h ha al massimo 2^h foglie. Indicato con h la sua altezza e con l_h il numero di foglie, allora $l_h \leq 2^h$.*

Dimostrazione. Si procede per induzione su h:

- **Base:** Per $h = 0$, l'albero ha un solo nodo $\rightarrow 2^h = 1$
- **Passo induttivo:** Si suppone **vero** per $h - 1$. Ogni foglia dell'albero di altezza $h - 1$ ha al massimo due nuove foglie, per cui

$$l_h \leq 2 \cdot l_{h-1} = 2 \cdot 2^{h-1} = 2^h$$

□

Teorema 8.2. *Qualsiasi algoritmo di ordinamento per confronti richiede $\Omega(n \lg n)$ confronti nel caso peggiore (cioè ogni albero di decisione che ordina n elementi ha altezza $\Omega(n \lg n)$).*

Dimostrazione. È sufficiente determinare l'altezza di un albero di decisione dove ogni permutazione appare come una foglia raggiungibile. Si considera un albero di decisione di altezza h con l foglie raggiungibili che corrisponde ad un algoritmo per confronti di n elementi. Poiché ciascuna delle $n!$ permutazioni dell'input compare in una foglia, si ha $n! \leq l$. Dal momento che un albero binario di altezza h non ha più di 2^h foglie, si ha $n! \leq l \leq 2^h$. Prendendo i logaritmi, questa relazione implica che

$$\begin{aligned} h &\geq lg(n!) \quad \text{(perché la funzione logaritmo è monotona crescente)} \\ &\geq lg\Big(\frac{n}{e}\Big)^n \\ &= n\,lg\Big(\frac{n}{e}\Big) = n \lg n - n \lg e = \Omega(n \lg n) \end{aligned}$$

□

Si usa Ω perché c'è $\geq$. Quindi MERGE-SORT è un **ordinamento per confronto asintoticamente ottimo**, questo perché il limite superiore $O(n\,lg\,n)$ sul tempo di esecuzione di MERGE-SORT corrisponde al limite inferiore $\Omega(n\,lg\,n)$.

8.2 Counting sort

L'algoritmo COUNTING-SORT suppone che ciascuno degli n elementi di input sia un numero naturale compreso nell'intervallo 0 a k, per qualche intero k. Quando $k = O(n)$, l'ordinamento viene effettuato nel tempo $\Theta(n)$.

COUNTING-SORT determina, per ogni elemento di input x, il numero di elementi minori di x. Esso usa questa informazione per inserire l'elemento x direttamente nella sua posizione nell'array di output. Questo schema deve essere leggermente modificato per gestire il caso in cui più elementi hanno lo stesso valore, per evitare che siano inseriti nella stessa posizione.

Nel codice di COUNTING-SORT si suppone che l'input sia un array $A[1 \ldots n]$, quindi che $length[A] = n$. Occorrono altri due array: $B[1 \ldots n]$, che contiene l'output ordinato, e l'array $C[0 \ldots k]$ che fornisce la memoria temporanea di lavoro (conta quante volte c'è ciascun valore). Quindi occupa il doppio dello spazio, al quale si aggiunge l'array C.

```
COUNTING-SORT(A, B, k)
1   sia C[0..k] un nuovo array
2   for i ← 0 to k
3       C[i] ← 0
4   for j ← 1 to A.length
5       C[A[j]] ← C[A[j]] + 1
6   // C[i] ora contiene il numero di elementi uguali a i
7   for i ← 1 to k
8       C[i] ← C[i] + C[i - 1]
9   // C[i] ora contiene il numero di elementi minori o uguali a i
10   for j ← A.length downto 1
11       B[C[A[j]]] ← A[j]
12       C[A[j]] ← C[A[j]] - 1
```

Commenti:

- 0 k è il valore massimo da ordinare
- 2-3 Azzera il vettore contatore
- 4-5 Scorre e conta quante volte c'è $A[j]$
- 7-8 Scorre C dal primo all'ultimo elemento senza toccare $C[0]$ che indica il numero di elementi uguali a 0 nell'array
- 10 Parte dal fondo (per avere stabilità) e lo posiziona
- 12 Più elementi potrebbero avere lo stesso valore

Dopo che il ciclo **for** (righe 2 - 3) inizializza a zero tutti gli elementi dell'array C, ogni elemento di input viene esaminato nelle righe 4 - 5 del ciclo **for**. Se il valore di un elemento di input è i, si incrementa $C[i]$. Quindi, dopo la riga

5, $C[i]$ contiene il numero degli elementi di input uguali a i per ogni intero $i = 0, 1, \ldots, k$. Le righe 7 - 8 determinano, per ogni $i = 0, 1, \ldots, k$, quanti elementi di input sono minori o uguali a i, mantenendo la somma corrente dell'array C. Infine, le righe 10 - 12 del ciclo **for** inseriscono l'elemento $A[j]$ nella corretta posizione ordinata dell'array di output B.

Se tutti gli n elementi sono distinti, quando viene eseguita per la prima volta la riga 10, per ogni $A[j]$, il valore $C[A[j]]$ rappresenta la posizione finale corretta di $A[j]$ nell'array di output, in quanto ci sono $C[A[j]]$ elementi minori o uguali ad $A[j]$. Poiché gli elementi potrebbero non essere distinti, $C[A[j]]$ viene ridotto ogni volta che viene inserito un valore $A[j]$ nell'array B. La riduzione di $C[A[j]]$ fa sì che il successivo elemento di input con un valore uguale ad $A[j]$, se esiste, venga inserito nella posizione immediatamente prima di $A[j]$ nell'array di output.

Il ciclo **for** alle righe 2 - 3 impiega un tempo $\Theta(k)$, il ciclo **for** alle righe 4 - 5 impiega un tempo $\Theta(n)$, il ciclo **for** alle righe 7 - 8 impiega un tempo $\Theta(k)$ e il ciclo **for** alle righe 10 - 12 impiega un tempo $\Theta(n)$. Quindi il tempo totale è $\Theta(k+n)$. Di solito COUNTING-SORT viene utilizzato quando $k = O(n)$, nel qual caso il tempo di esecuzione è $\Theta(n)$, quindi un ordinamento in **tempo lineare**. COUNTING-SORT batte il limite inferiore di $\Omega(n \, lg \, n)$ perché non è un algoritmo di ordinamento per confronti. Infatti, il codice non effettua alcun confronto fra gli elementi di input. Piuttosto, COUNTING-SORT usa i valori effettivi degli elementi come indici di un array.

Un'importante proprietà di COUNTING-SORT è la **stabilità**: le chiavi con lo stesso valore si presentano nell'array di output nello stesso ordine in cui si trovano nell'array in input. Ovvero, l'uguaglianza di due numeri viene risolta applicando la seguente regola: il numero che si presenta per primo nell'array di input sarà inserito per primo nell'array di output. Normalmente, la proprietà di stabilità è importante soltanto quando i dati satellite vengono spostati insieme con le chiavi da ordinare. Per verificare che COUNTING-SORT è un algoritmo stabile è sufficiente guardare come funziona l'ultimo ciclo **for**. La correttezza di COUNTING-SORT può essere verificata tramite invariante di ciclo.

8.3 Radix sort

L'algoritmo RADIX-SORT risolve il problema dell'ordinamento in una maniera contraria all'intuizione, ordinando le cifre **meno significative** per prime. Indicato con d il numero di cifre che costituiscono i valori da ordinare, occorreranno soltanto d passaggi per completare l'ordinamento. È essenziale che gli ordinamenti delle cifre in questo algoritmo siano **stabili**.

RADIX-SORT(A, d)
1 **for** $i \leftarrow 1$ **to** d
2 Usa un ordinamento **stabile** per ordinare l'array A sulla cifra i

Sono algoritmi **stabili** COUNTING-SORT, INSERTION-SORT, MERGE-SORT se in MERGE c'è il $\leq$ e non $<$, QUICKSORT non è detto che lo sia.

8.3.1 Correttezza di Radix sort

La correttezza di RADIX-SORT si dimostra per induzione sulla colonna i da ordinare:

1. Si suppone che le cifre 1, 2, ... , $i-1$ siano ordinate
   - Se due cifre in posizione i sono **diverse**

     $\Rightarrow$ ordinamento su posizione i è corretto

     Le posizioni 1, ... , $i-1$ sono irrilevanti
   - Se 2 cifre in posizione i sono **uguali** (es. 23 e 25)

     $\Rightarrow$ i numeri sono già nell'ordine giusto (ipotesi induttiva)

     Ordinamento **stabile** sulla cifra i li lascia nell'ordine giusto
2. $\Rightarrow$ ordinamento **stabile** sulla cifra i lascia le cifre 1, ... , i ordinate

8.3.2 Analisi di Radix sort

Dati n numeri di d cifre, dove ogni cifra può avere fino a k valori possibili, la procedura RADIX-SORT ordina correttamente i numeri nel tempo $\Theta(d(n+k))$, se l'ordinamento stabile utilizzato nella procedura impiega un tempo $\Theta(n+k)$. Se d è costante e $k = O(n)$ allora RADIX-SORT viene eseguito in tempo lineare, cioè $\Theta(d\,n)$.

L'analisi del tempo di esecuzione dipende dall'ordinamento stabile che viene utilizzato come algoritmo di ordinamento intermedio. Se ogni cifra si trova nell'intervallo da 0 a $k-1$ e k non è troppo grande, COUNTING-SORT è la scelta ovvia da fare.

Date n parole di b bit diverse e un intero positivo $r \leq b$. È possibile suddividere tali parole in pezzi di r bit; ciascuna parola ha $d = \lceil b/r \rceil$ cifre di r bit. È possibile applicare COUNTING-SORT con $k = 2^r - 1$. Per esempio, per parole di 32 bit e cifre da 8 bit, si ha $b = 32$, $r = 8$, $d = \lceil b/r \rceil = \lceil 32/8 \rceil = 4$ e $k = 2^r - 1 = 2^8 - 1 = 255$. Per n *parole* di b bit e un intero positivo $r \leq b$, RADIX-SORT ordina correttamente questi numeri nel tempo $\Theta(\frac{b}{r}(n+2^r))$ se l'algoritmo di ordinamento stabile usato richiede tempo $\Theta(n+k)$ per input nell'intervallo da 0 a k.
Si sceglie r in modo da bilanciare b/r e $n+2^r$:

- Scegliendo $r \approx lg\, n$ si ha una situazione ottimale:

  $$\Theta(\tfrac{b}{r}(n+2^r)) = \Theta(\tfrac{b}{lg\, n}(n+n)) = \Theta(\tfrac{bn}{lg\, n})$$

  Meglio di $\Theta(n)$
- Se fosse $r < lg\, n \Rightarrow \frac{b}{r} > \frac{b}{lg\, n} \Rightarrow n + 2^r$ non migliora e resta a $\Theta(n)$
- Se fosse $r > lg\, n \Rightarrow n + 2^r$ diventa più grande di n per un fattore moltiplicativo

Per esempio, per ordinare 2^{16} numeri a 32 bit, si usa $r = lg\, 2^{16} = 16$ bit, per un totale di $\lceil b/r \rceil = 2$ passi.

8.3.3 Esempio di Radix sort

L'algoritmo RADIX-SORT risolve il problema dell'ordinamento in una maniera contraria all'intuizione, ordinando le cifre **meno significative** per prime. Indicato con d il numero di cifre che costituiscono i valori da ordinare, occorreranno soltanto d passaggi per completare l'ordinamento.

		S		S		S
326		69**0**		**7**0**4**		**3**2**6**
453		75**1**		6**0**8		**4**3**5**
608		45**3**		3**2**6		**4**5**3**
835		70**4**		8**3**5		**6**0**8**
751	$\rightarrow$	83**5**	$\rightarrow$	4**3**5	$\rightarrow$	**6**9**0**
435		43**5**		7**5**1		**7**0**4**
704		36**6**		4**5**3		**7**5**1**
690		60**8**		6**9**0		**8**3**5**

Capitolo 9

Hashing

Una **tabella hash** è una struttura dati efficace per implementare i dizionari. Sebbene la ricerca di un elemento in una tabella hash richieda, nel caso peggiore, lo stesso tempo $\Theta(n)$ richiesto per ricercare un elemento in una lista concatenata, l'hashing, sotto opportune ipotesi, garantisce un tempo medio per ricercare un elemento in una tabella hash di $O(1)$.

Una tabella hash è una generalizzazione dell'indirizzamento diretto che, in un array ordinario, sfrutta la possibilità di esaminare una posizione arbitraria in un array nel tempo $O(1)$. Si può utilizzare vantaggiosamente l'indirizzamento diretto quando è possibile allocare un array che ha una posizione per ogni chiave possibile. Così facendo, dato un universo di chiavi U che va da 0 a $k-1$, dove k è la lunghezza di un array, l'elemento con chiave k si troverà nella posizione k. Quando il numero di chiavi effettivamente memorizzate è piccolo rispetto al numero totale di chiavi possibili, le tabelle hash diventano una valida alternativa all'indirizzamento diretto di un array, in quanto una tabella hash tipicamente usa un array di dimensione proporzionale al numero di chiavi effettivamente memorizzate. Anziché utilizzare una chiave direttamente come un indice dell'array, la chiave viene usata per calcolare l'indice.

9.1 Tabelle a indirizzamento diretto

L'indirizzamento diretto è una tecnica che funziona bene quando l'universo delle chiavi $U = \{0, 1, 2, 3, \ldots, m-1\}$ è ragionevolmente piccolo.

Si considera un'applicazione che ha bisogno di un insieme dinamico in cui ogni elemento ha una chiave estratta dall'universo $U = \{0, 1, 2, 3, \ldots, m-1\}$, dove m non è troppo grande, e si suppone inoltre che due elementi non possano avere la stessa chiave.

Per rappresentare l'insieme dinamico, si utilizza un array o **tabella a indirizzamento diretto**, che verrà indicata con $T = [0, \ldots, m-1]$, dove ogni posizione o **cella** corrisponde a una chiave nell'universo U. In generale, la cella k punterà a un elemento dell'insieme con chiave k. Se l'insieme non contiene l'elemento con chiave k, allora $T[k] = \text{NIL}$. Nel caso di chiavi duplicate si utilizza una lista collegata.

Le operazioni di dizionario sono semplici da implementare:

```
DIRECT-ADDRESS-SEARCH(T, k)
1  return T[k]

DIRECT-ADDRESS-INSERT(T, x)
1  T[x.key] ← x

DIRECT-ADDRESS-DELETE(T, x)
1  T[x.key] ← NIL
```

Ciascuna di queste operazioni richiede un tempo $O(1)$.

Se l'universo delle chiavi U è troppo grande, memorizzare una tabella T di dimensione $|U|$ può essere impraticabile, se non impossibile, considerando la memoria disponibile in un normale calcolatore. Inoltre, l'insieme K delle chiavi effettivamente memorizzate può essere così piccolo rispetto a U che la maggior parte dello spazio allocato per la tabella T sarebbe sprecato.

9.2 Tabelle hash

Quando l'insieme K delle chiavi memorizzate in un dizionario è molto più piccolo dell'universo U di tutte le chiavi possibili, una tabella hash richiede molto meno spazio di una tabella a indirizzamento diretto. Lo spazio richiesto può essere ridotto a $\Theta(|K|)$, senza perdere il vantaggio di ricercare un elemento nella tabella hash nel tempo $O(1)$.

Con l'indirizzamento diretto, un elemento con chiave k è memorizzato nella cella k. Con l'hashing, questo elemento è memorizzato nella cella $h(k)$ (ovvero memorizza k in $T[h(k)]$); cioè si utilizza una **funzione hash** h per calcolare la cella dalla chiave k. Quindi h associa l'universo U delle chiavi alle celle di una **tabella hash** $T = [0, ..., m-1]$:

$$h : U \to \{0, 1, 2, 3, ..., m-1\}$$

Dove la dimensione m della tabella hash è generalmente molto più piccola di $|U|$. Si dirà che un elemento con chiave k viene mappato nella cella $h(k)$ o anche che $h(k)$ è il **valore hash** della chiave k.

9.3 Funzioni hash

La funzione hash, che ha il compito di ridurre l'intervallo degli indici e di conseguenza la dimensione dell'array, è una funzione **non iniettiva**. La funzione $h(k)$ dovrebbe realizzare un **hash uniforme semplice** per cui ogni chiave ha la stessa probabilità di essere mandata in una qualsiasi delle m celle, indipendentemente dalla cella cui viene mandata una qualsiasi altra cella. In pratica questo non è possibile perché non si conosce la distribuzione di probabilità secondo la quale sono estratte le chiavi che, a loro volta, potrebbero non essere prese indipendentemente l'una dall'altra. Per poter creare una buona funzione hash si andranno ad utilizzare delle informazioni qualitative relative al dominio delle chiavi.

Un buon approccio consiste nel derivare il valore hash in modo che sia indipendente da qualsiasi regolarità che possa esistere nei dati. Per esempio, il

metodo della divisione calcola il valore hash come il resto della divisione fra la chiave e un determinato numero primo. Questo metodo spesso fornisce buoni risultati, purché il numero primo sia scelto in modo da non essere correlato a nessuna regolarità nella distribuzione della chiavi.

La maggior parte delle funzioni hash suppone che l'universo delle chiavi sia l'insieme dei numeri naturali $\mathbb{N} = \{0, 1, 2, 3, ...\}$. Quindi, se le chiavi non sono numeri naturali, occorre un metodo per interpretarle come tali. Per esempio, una stringa di caratteri può essere interpretata come un numero intero espresso in una notazione posizionale di base opportuna (vedi codifica ASCII).

9.3.1 Il metodo della divisione

Quando si applica il **metodo della divisione** per creare una funzione hash, una chiave k viene associata a una delle m celle prendendo il resto della divisione fra k e m; cioè la funzione hash è:

$$h(k) = k \bmod m = k - \left\lfloor \frac{k}{m} \right\rfloor \cdot m$$

Il vantaggio di questo metodo è che è veloce, mentre, lo svantaggio, è che quando si utilizza il metodo della divisione, di solito, si devono evitare determinati valori di m. Per esempio, m non dovrebbe essere una potenza di 2, perché se $m = 2^p$, allora $h(k)$ rappresenta proprio i p bit meno significativi di k (critico per rappresentare le stringhe). A meno che non sia noto che tutte le configurazioni dei p bit di ordine inferiore abbiano la stessa probabilità, è meglio rendere la funzione hash dipendente da tutti i bit della chiave. Scegliere $m = 2^p - 1$, quando k è una stringa di caratteri interpretata nella base 2^p, potrebbe essere una cattiva soluzione, perché la permutazione dei caratteri di k non cambia il suo valore hash. Un **numero primo** non troppo vicino a una potenza esatta di 2 è spesso una buona scelta per m.

9.3.2 Il metodo della moltiplicazione

Il **metodo della moltiplicazione** per creare funzioni hash si svolge in due passi. Prima si moltiplica la chiave k per una costante A appartenente all'intervallo $0 < A < 1$ e si estrae la parte frazionaria di kA. Poi si moltiplica tale valore per m e si prende la parte intera inferiore del risultato. In sintesi, la funzione hash è:

$$h(k) = \lfloor m \cdot (k \cdot A \bmod 1) \rfloor$$

Dove $(k \cdot A \bmod 1)$ rappresenta la parte frazionaria di kA, cioè $kA - \lfloor kA \rfloor$.

Lo svantaggio del metodo della moltiplicazione è che è più lento del metodo della divisione. Al contrario, un suo vantaggio è che il valore di m non è critico. Tipicamente, lo si sceglie come una potenza del 2 ($m = 2^p$ per qualche intero p), il che rende semplice implementare la funzione hash nella maggior parte dei calcolatori.

Si suppone che la dimensione della parola della macchina sia w bit e che k entri in una sola parola (k richiede w bit). Come A si prende una funzione della forma $\frac{s}{2^w}$, dove s è un intero nell'intervallo $0 < s < 2^w$ (s richiede w bit). Inizialmente, si moltiplica k per l'intero di w bit, $s = A \cdot 2^w$. Il risultato è un valore di $2w$ bit definito come $r_1 2^w + r_0$, dove r_1 è la parte intera, cioè la

più significativa, di kA ($r_1 = \lfloor kA \rfloor$) e r_0 è la parte frazionaria, cioè la meno significativa, del prodotto kA ($r_0 = kA \bmod 1 = kA - \lfloor kA \rfloor$). Il valore hash desiderato di p bit è formato dai p bit più significativi di r_0.

$\lfloor m \cdot (k \cdot A \bmod 1) \rfloor \Rightarrow$ shift a sinistra r_0 di $p = lg\, m$ bit, si prendono i p bit a sinistra del punto binario. Non serve shiftare, si prendono i p bit più significativi di r_0.

Sebbene questo metodo funzioni con qualsiasi valore della costante A, tuttavia con qualche valore funziona meglio che con altri sulla base della chiave su cui fare l'hash. La scelta ottimale dipende dalle caratteristiche dei dati da sottoporre all'hashing. Knuth propone $A \approx \frac{\sqrt{5}-1}{2} = 0,618033..$ come valore che funziona ragionevolmente bene. Quindi dato w si sceglie s intero tale che $\frac{s}{2^w} \approx \frac{\sqrt{5}-1}{2}$.

9.4 Collisioni e tecniche per risolverle

Può verificarsi che due chiavi possano essere mappate nella stessa cella, ovvero si può verificare che $h(k) = h(k')$ con $k \neq k'$. Questo evento si chiama **collisione**. Può capitare se $|K| \leq m$ e capita sicuramente si $|K| > m$. Esistono però delle tecniche efficaci per risolvere i conflitti creati dalle collisioni. Tra queste: il concatenamento e l'indirizzamento aperto.

9.4.1 Concatenamento

Nel **concatenamento** si pongono tutti gli elementi che sono associati alla stessa cella (cioè che hanno lo stesso hash) in una **lista collegata**. Si potrà quindi verificare che una generica cella j contenga un puntatore alla testa di una lista di tutti gli elementi memorizzati che vengono mappati in j; se non ce ne sono, la cella j contiene la costante NIL. Non è una struttura dati molto dinamica.

CHAINED-HASH-INSERT(T, x)
inserisce x in testa alla lista $T[h(x.key)]$

Il tempo di esecuzione nel caso peggiore per l'inserimento è $O(1)$. La procedura di inserimento è veloce, anche perché si suppone che l'elemento x da inserire non sia presente nella tabella; se necessario, questa ipotesi può essere verificata (con un costo aggiuntivo) ricercando un elemento la cui chiava sia $x.key$, prima di effettuare l'inserimento.

CHAINED-HASH-SEARCH(T, k)
ricerca un elemento con chiave k nella lista $T[h(k)]$

Per la ricerca, il tempo di esecuzione nel caso peggiore è proporzionale alla lunghezza della lista.

CHAINED-HASH-DELETE(T, x)
cancella x dalla lista $T[h(x.key)]$

La cancellazione di un elemento x può essere realizzata nel tempo $O(1)$ se le liste

sono doppiamente concatenate (non considera il tempo per trovare l'elemento da cancellare). Con liste semplicemente collegate, costa quanto la ricerca.

Analisi di hash con concatenamento

Data una tabella hash T con m celle dove sono memorizzati n elementi, si definisce **fattore di carico** α della tabella T il rapporto n/m, ossia il numero medio di elementi memorizzati in una lista (n = numero di elementi memorizzati nella tabella, m = numero di slot nella tabella). Tale fattore α può essere minore, uguale o maggiore di 1.

Il comportamento nel caso peggiore dell'hashing con concatenamento è pessimo: tutte le n chiavi sono associate alla stessa cella, creando una singola lista di lunghezza n. Il tempo di esecuzione della ricerca è quindi $\Theta(1)+\Theta(n) = \Theta(n)$, dove $\Theta(1)$ è il tempo per calcolare la funzione hash. Le tavole hash non sono utilizzate per le loro prestazioni nel caso peggiore.

Le prestazioni nel caso medio dipendono dal modo in cui la funzione hash h distribuisce mediamente l'insieme delle chiavi da memorizzare tra le m celle. Si suppone che qualsiasi elemento abbia la stessa probabilità di essere mandato in una qualsiasi delle m celle, indipendentemente dalle celle in cui sono mandati gli altri elementi. Questa ipotesi è detta **hashing uniforme semplice**.

Per $j = 0, 1, 2, 3, \ldots, m-1$, indicando con n_j la lunghezza della lista $T[j]$, si avrà:

$$n = n_0 + n_1 + n_2 + \ldots + n_{m-1}$$

e il valore atteso di n_j sarà:

$$E[n_j] = \alpha = \frac{n}{m}$$

Si suppone che basti un tempo $O(1)$ per calcolare il valore hash $h(k)$, in modo che il tempo richiesto per cercare un elemento con chiave k dipenda linearmente dalla lunghezza $n_{h(k)}$ della lista $T[h(k)]$. Mettendo assieme in un tempo $O(1)$ il tempo richiesto per calcolare la funzione hash e accedere alla cella $h(k)$, si considera il numero atteso di elementi esaminati dall'algoritmo di ricerca, ovvero il numero di elementi nella lista $T[h(k)]$ che vengono controllati per vedere se le loro chiavi sono uguali a k. Si considerano due casi. Nel primo caso, la ricerca **non ha successo**: nessun elemento nella tabella ha la chiave k. Nel secondo caso, la ricerca **ha successo** e viene trovato un elemento con chiave k.

Teorema 9.1. *In una tabella hash le cui collisioni sono risolte con il concatenamento, una ricerca senza successo richiede un tempo $\Theta(1+\alpha)$ nel caso medio, nell'ipotesi di hashing uniforme semplice.*

Dimostrazione. Nell'ipotesi di hashing uniforme semplice, qualsiasi chiave k non ancora memorizzata nella tabella ha la stessa probabilità di essere associata ad una qualsiasi delle m celle. Il tempo atteso per ricercare senza successo una chiave k è il tempo atteso per svolgere le ricerche fino alla fine della lista $T[h(k)]$, che ha una lunghezza attesa pari a $E[n_{h(k)}] = \alpha$. Quindi, il numero atteso di elementi esaminati in una ricerca senza successo è α e il tempo totale richiesto (incluso quello per calcolare $h(k)$) è $\Theta(1 + \alpha)$. □

Il caso di una ricerca con successo è un po' differente, perché ogni lista non ha la stessa probabilità di essere oggetto delle ricerche. La probabilità che una

lista sia oggetto delle ricerche è proporzionale al numeri di elementi che contiene. Nonostante questo, il tempo atteso è ancora $\Theta(1+\alpha)$.

Teorema 9.2. *In una tabella hash le cui collisioni sono risolte con il concatenamento, una ricerca con successo richiede un tempo* $\Theta(1+\alpha)$ *nel caso medio, nell'ipotesi di hashing uniforme semplice.*

Dimostrazione. Si suppone che l'elemento da ricercare abbia la stessa probabilità di essere uno qualsiasi degli n elementi memorizzati nella tabella. Il numero di elementi esaminati durante una ricerca con successo di un elemento x è uno in più del numero di elementi che si trovano prima di x nella lista di x. Gli elementi che precedono x nella lista, sono stati inseriti tutti dopo di x, perché i nuovi elementi vengono posti all'inizio della lista. Per trovare il numero atteso di elementi esaminati, si prende la media, sugli n elementi x nella tabella, di 1 più il numero atteso di elementi aggiunti alla lista di x dopo che x è stato aggiunto alla lista. Si indica con x_i l'i-esimo elemento inserito nella tabella, per $i = 1, 2, 3, \ldots, n$, e si definisce $k_i = x_i.key$. Per le chiavi k_i e k_j, si definisce la variabile casuale indicatrice

$$X_{ij} = I\{h(k_i) = h(k_j)\}$$

Nell'ipotesi di hashing uniforme semplice, si ha

$$Pr\{h(k_i) = h(k_j)\} = \frac{1}{m}$$

E quindi

$$E[X_{ij}] = \frac{1}{m}$$

Dunque il numero atteso di elementi esaminati in una ricerca con successo è

$$\begin{aligned}
E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}X_{ij}\right)\right] &= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}E[X_{ij}]\right) \\
&= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\frac{1}{m}\right) \\
&= 1+\frac{1}{n\,m}\sum_{i=1}^{n}(n-i) \\
&= 1+\frac{1}{n\,m}\left(\sum_{i=1}^{n}n-\sum_{i=1}^{n}i\right) \\
&= 1+\frac{1}{n\,m}\left(n^2-\frac{n(n+1)}{2}\right) \\
&= 1+\frac{n-1}{2m} \\
&= 1+\frac{\alpha}{2}-\frac{\alpha}{2n}
\end{aligned}$$

In conclusione, il tempo totale richiesto per una ricerca con successo (incluso il tempo per calcolare la funzione hash) è $\Theta(2+\alpha/2-\alpha/2n) = \Theta(1+\alpha)$. □

9.4.2 Indirizzamento aperto

Nell'**indirizzamento aperto**, tutti gli elementi sono memorizzati nella tabella hash stessa; ovvero ogni cella della tabella contiene un elemento dell'insieme dinamico (una chiave) o la costante NIL. Quando si cerca un elemento, si esamina sistematicamente le celle della tabella finché non si trova l'elemento desiderato o finché non ci si rende conto che l'elemento non si trova nella tabella.

Diversamente dal concatenamento, non ci sono liste né elementi memorizzati all'esterno della tabella. Quindi, nell'indirizzamento aperto, la tabella hash può riempirsi al punto tale che non possono essere effettuati altri incrementi: una conseguenza è che il fattore di carico α non supera mai 1.

Per effettuare un inserimento mediante l'indirizzamento aperto, si esaminano in successione le posizioni della tabella hash (**ispezione**), finché non si trova una cella vuota in cui inserire la chiave. Anziché eseguire sempre lo stesso ordine $0, 1, \ldots, m-1$ (che richiede un tempo di ricerca $\Theta(n)$), la sequenza delle posizioni esaminate durante un'ispezione dipende dalla chiave da inserire. Per determinare quali celle esaminare, si estende la funzione hash in modo da includere l'ordine di ispezione (a partire da 0) come secondo input. Quindi, la funzione hash diventa

$$h : U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}$$

Con l'indirizzamento aperto si richiede che, per ogni chiave k, la **sequenza di ispezione**

$$\langle h(k,0), h(k,1), \ldots, h(k,m-1) \rangle$$

sia una permutazione di $\langle 0, 1, \ldots, m-1 \rangle$, in modo che ogni posizione della tabella hash possa essere considerata come possibile cella in cui inserire una nuova chiave mentre la tabella si riempie. Se necessario, si esaminano tutti gli slot, inoltre, nessuno slot verrà esaminato più di una volta.

La procedura HASH-INSERT riceve in ingresso una tabella hash T e una chiave k. Essa ritorna il numero della cella in cui ha memorizzato la chiave k oppure segnala un errore se la tabella era già piena.

```
HASH-INSERT(T, k)
1   i ← 0
2   repeat
3       j ← h(k, i)
4       if T[j] = NIL
5           T[j] ← k
6           return j
7       else i ← i + 1
8   until i = m
9   error "overflow della tabella hash"
```

Commenti:

0 Inserisce il valore nel primo NIL trovato

3 Un esempio di funzione hash è:
$h(k, i) = (h'(k) + i) \bmod m = (k \bmod m + i) \bmod m$

L'algoritmo che ricerca la chiave k esamina la stessa sequenza di celle che ha esaminato l'algoritmo di inserimento quando ha inserito la chiave k. Quindi, la ricerca può terminare (senza successo) quando trova una cella vuota, perché la chiave k sarebbe stata inserita lì e non dopo nella sua sequenza di ispezione. La procedura HASH-SEARCH prende come input una tabella hash T e una chiave k; restituisce j se la cella j contiene la chiave k oppure NIL se la chiave k non si trova nella tabella T.

```
HASH-SEARCH(T, k)
1   i ← 0
2   repeat
3       j ← h(k, i)
4       if T[j] = k
5           return j
6       i ← i + 1
7   until T[j] = NIL or i = m
8   return NIL
```

Commenti:

3 Calcolare la posizione della cella da esaminare attraverso la funzione hash

5 L'elemento è stato trovato e si trova nella cella di indice j

7 Si ripete fintantoché non si trova una posizione vuota o finché non si sono visitate tutte le celle

8 Restituisce NIL se non è stato trovato

La cancellazione da una tabella hash a indirizzamento aperto è un'operazione difficile. Quando si cancella una chiave dalla cella i, non si può semplicemente marcare questa cella come vuota inserendovi la costante NIL. Così facendo, potrebbe essere impossibile ritrovare qualsiasi chiave k nel cui inserimento si abbia esaminato la cella i e la si abbia trovata occupata. Una soluzione consiste nel marcare la cella registrandovi il valore speciale DELETED, anziché NIL. Si dovrà quindi modificare la procedura HASH-INSERT per trattare tale cella come se fosse vuota, in modo da potere inserire una nuova chiave. Nessuna modifica è richiesta per HASH-SEARCH, perché questa procedura ignora i valori DELETED durante la ricerca.

Nell'analisi si fa l'ipotesi di **hashing uniforme**: si suppone che ogni chiave abbia la stessa probabilità di avere come sequenza di ispezione una delle $m!$ permutazioni di $\langle 0, 1, \ldots, m-1 \rangle$. L'hashing uniforme estende il concetto di hashing uniforme semplice, definito precedentemente, al caso in cui la funzione hash produca, non un singolo numero, ma un'intera sequenza di ispezione. Poiché è difficile implementare il vero hashing uniforme, in pratica si usano delle approssimazioni accettabili. Tre tecniche comunemente utilizzate per calcolare la sequenza di ispezione richieste dall'indirizzamento aperto sono: **ispezione lineare**, **ispezione quadratica** e **doppio hashing**. Tutte e tre le tecniche garantiscono che $\langle h(k,0), h(k,1), \ldots, h(k,m-1) \rangle$ sia una permutazione di $\langle 0, 1, \ldots, m-1 \rangle$ per ogni chiave k. Tuttavia, nessuna di queste tecniche soddisfa l'ipotesi di hashing uniforme, in quanto nessuna di esse è in grado di

generare più di m^2 sequenze di ispezione differenti (anziché $m!$ come richiesto dall'hashing uniforme).

Ispezione lineare

Data una funzione hash ordinaria $h' : U \to \{0, 1, 2, 3, \ldots, m-1\}$, che prende il nome di **funzione hash ausiliaria**,il metodo dell'**ispezione lineare** usa la funzione hash

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

per $i = 0, 1, \ldots, m-1$. Data la chiave k, la prima cella esaminata è $T[h'(k)]$, che è la cella data dalla funzione hash ausiliaria; la seconda cella esaminata è $T[h'(k)+1]$ e così via fino alla cella $T[m-1]$. Poi, l'ispezione riprende dalle celle $T[0]$, $T[1]$, ..., $T[h'(k)-1]$. Poiché la prima cella ispezionata determina l'intera sequenza di ispezioni, ci sono soltanto m sequenze di ispezione distinte.

L'ispezione lienare è facile da implementare ma presenta un problema noto come **addensamento primario**: si formano lunghe file di celle occupate, che aumentano il tempo medio di ricerca. Gli addensamenti si formano perché una cella vuota preceduta da i celle piene ha la probabilità $(i + 1)/m$ di essere la prossima a essere occupata. Le lunghe file di celle occupate tendono a diventare sempre più lunghe e il tempo di ricerca medio aumenta.

Ispezione quadratica

L'**ispezione quadratica** usa una funzione hash della forma

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

dove h' è una funzione hash ausiliaria, c_1 e $c_2 \neq 0$ sono costanti ausiliarie e $i = 0, 1, \ldots, m-1$. La posizione iniziale esaminata è $T[h'(k)]$; le posizioni successivamente esaminate sono distanziate da quantità che dipendono in modo quadratico dal numero d'ordine di ispezione i. Questa tecnica funziona molto meglio dell'ispezione lineare, ma per fare pieno uso della tabella hash, i valori c_1, c_2 ed m non si possono scegliere arbitrariamente e devono essere vincolati per assicurare che si abbia una permutazione completa di $\langle 0, 1, \ldots, m-1\rangle$. Inoltre, se due chiavi hanno la stessa posizione iniziale di ispezione, allora le loro sequenze di ispezione sono identiche, perché $h(k_1, 0) = h(k_2, 0)$ implica $h(k_1, i) = h(k_2, i)$. Questa proprietà porta a una forma più lieve di addensamento, che prende il nome di **addensamento secondario**. Come per l'ispezione lineare, la prima posizione determina l'intera sequenza, quindi vengono utilizzate soltanto m sequenze di ispezione distinte.

Doppio hashing

È uno dei metodi migliori disponibili per l'indirizzamento aperto perché le permutazioni prodotte hanno molte delle caratteristiche delle permutazioni scelte a caso. Il **doppio hashing** usa due funzioni hash ausiliarie h_1 per la prima esplorazione e h_2 per le esplorazioni restanti

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

L'ispezione inizia dalla posizione $T[h_1(k)]$; le successive posizioni sono distanziate dalle precedenti posizioni di una quantità $h_2(k)$ modulo m. Quindi, diversamente dal caso dell'ispezione lineare o quadratica, la sequenza di ispezione qui dipende in due modi dalla chiave k, perché possono variare sia la posizione iniziale di ispezione sia la distanza fra due posizioni successive di ispezione.

Il valore $h_2(k)$ deve essere primo relativo con la dimensione m della tabella hash perché venga ispezionata l'intera tabella hash. Un modo pratico per garantire questa condizione è scegliere m potenza del 2 e definire h_2 in modo che produca sempre un numero dispari. Un altro modo è scegliere m primo e definire h_2 in modo che generi sempre un numero intero positivo minore di m. Per esempio, si potrebbe scegliere m primo e porre

$$h_1(k) = k \bmod m$$
$$h_2(k) = 1 + (k \bmod m')$$

dove m' deve essere scelto un po' più piccolo di m (come $m - 1$).

Quando m è primo oppure una potenza di 2, il doppio hashing è migliore delle ispezioni lineari e quadratiche in quanto usa $\Theta(m^2)$ sequenze di ispezione, anziché $\Theta(m)$, perché ogni possibile coppia $(h_1(k), h_2(k))$ produce una distinta sequenza di ispezione. Di conseguenza, per questi valori di m, le prestazioni del doppio hashing risultano molto prossime a quelle dello schema ideale dell'hashing uniforme.

Nel doppio hashing, per la ricerca senza successo e l'inserimento, il numero atteso di esplorazioni è al più $\frac{1}{1-\alpha}$. Per la ricerca con successo, il numero atteso di esplorazione è al più $\frac{1}{\alpha} ln \frac{1}{1-\alpha}$.

Capitolo 10

Alberi binari di ricerca

Gli alberi di ricerca sono strutture dati che supportano molte operazioni sugli insiemi dinamici, fra le quali SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT e DELETE. Quindi, un albero di ricerca può essere utilizzato sia come dizionario sia come una coda di priorità. Le operazioni di base su un albero binario di ricerca richiedono un tempo proporzionale all'altezza dell'albero. Per un albero binario completo con n nodi, tali operazioni sono eseguite nel tempo $\Theta(lg\, n)$ nel caso peggiore. Se invece, l'albero è una catena lineare di n nodi, le stesse operazioni richiedono un tempo $\Theta(n)$ nel caso peggiore.

10.1 Cos'è un albero binario di ricerca

Un albero binario di ricerca è organizzato in un albero binario, che può essere rappresentato da una struttura dati concatenata in cui ogni nodo è un oggetto. Oltre a una chiave *key* e ai dati satellite, ogni nodo dell'albero contiene gli attributi *left, right* e p che puntano ai nodi che corrispondono, rispettivamente, al figlio sinistro, al figlio destro e al padre del nodo. Se manca un figlio o il padre, i corrispondenti attributi contengono il valore NIL. Il nodo radice (*root*) è l'unico nodo nell'albero il cui attributo padre è NIL.

Le chiavi in un albero binario di ricerca sono sempre memorizzate in modo da soddisfare la **proprietà degli alberi binari di ricerca**:

> Sia x un nodo in un albero binario di ricerca. Se y è un nodo nel sottoalbero sinistro di x, allora $y.key \leq x.key$. Se y è un nodo nel sottoalbero destro di x, allora $y.key \geq x.key$.

La proprietà degli alberi binari di ricerca consente di elencare ordinatamente tutte le chiavi di un albero binario di ricerca con un semplice algoritmo ricorsivo di **attraversamento simmetrico di un albero** (inorder). Questo algoritmo è così chiamato perché la chiave della radice di un sottoalbero viene stampata nel mezzo tra la stampa dei valori nel sottoalbero sinistro e la stampa dei valori nel sottoalbero destro. Analogamente, un algoritmo di **attraversamento anticipato di un albero** (preorder) stampa la radice prima dei valori dei suoi sottoalberi e un algoritmo di **attraversamento posticipato di un albero** (postorder) stampa la radice dopo i valori dei suoi sottoalberi. L'attraversamento **preorder** può essere utile per memorizzare in modo univoco un

albero andando a memorizzare ogni nodo come *n(-,-)* dove n indica il nodo e tra parentesi si indicano prima il figlio sinistro, poi quello destro. Con la chiamata INORDER-TREE-WALK($T.root$) la procedura stampa tutti gli elementi di un albero binario di ricerca T.

```
INORDER-TREE-WALK(x)
1  if x ≠ NIL
2      INORDER-TREE-WALK(x.left)
3      stampa x.key
4      INORDER-TREE-WALK(x.right)
```

La correttezza dell'algoritmo si ricava direttamente per induzione dalla proprietà degli alberi binari di ricerca.

Occorre un tempo $\Theta(n)$ per attraversare un albero binario di ricerca di n nodi perché, dopo la chiamata iniziale, la procedura viene chiamata ricorsivamente esattamente due volte per ogni nodi dell'albero - una volta per il figlio sinistro e una per il figlio destro.

Teorema 10.1. *Se x è la radice di un sottoalbero di n nodi, la chiamata* INORDER-TREE-WALK(x) *richiede il tempo* $\Theta(n)$.

Dimostrazione. Sia $T(n)$ il tempo richiesto dalla procedura INORDER-TREE-WALK quando viene chiamata per la radice di un sottoalbero di n nodi. Poiché INORDER-TREE-WALK visita tutti gli n nodi del sottoalbero, si ha $T(n) = \Omega(n)$. Si dimostra ora che $T(n) = O(n)$.

INORDER-TREE-WALK richiede una piccola quantità costante di tempo con un sottoalbero vuoto (per il resto $x \neq$ NIL), quindi $T(0) = c$ per qualche costante positiva c.

Per $n > 0$, si suppone che INORDER-TREE-WALK sia chiamata per un nodo x il cui sottoalbero sinistro ha k nodi e il cui sottoalbero destro ha $n - k - 1$ nodi. Il tempo per eseguire INORDER-TREE-WALK è

$$T(n) \leq T(k) + T(n - k - 1) + d$$

per qualche costante positiva d che rappresenta un limite superiore per il tempo per eseguire INORDER-TREE-WALK, escludendo il tempo impiegato nelle chiamate ricorsive.

Si applica ora il metodo di sostituzione per trovare che $T(n) = \Theta(n)$ dimostrando che $T(n) \leq (c+d)n + c$. Per $n = 0$, si ha $(c+d) \cdot 0 + c = c = T(0)$. Per $n > 0$ si ha

$$\begin{aligned} T(n) &\leq T(k) + T(n - k - 1) + d \\ &= ((c + d)k + c) + ((c + d)(n - k - 1) + c) + d \\ &= (c + d)n + c - (c + d) + c + d \\ &= (c + d)n + c \end{aligned}$$

che completa la dimostrazione. □

10.2 Interrogazione di un albero binario di ricerca

Una tipica operazione eseguita su un albero binario di ricerca è quella di cercare una chiave memorizzata nell'albero. Oltre all'operazione SEARCH, gli alberi binari di ricerca supportano interrogazioni (query) quali MINIMUM, MAXIMUM, PREDECESSOR e SUCCESSOR.

10.2.1 Ricerca

Per cercare un nodo con una data chiave in un albero binario di ricerca si utilizza la procedura TREE-SEARCH. Dato un puntatore alla radice dell'albero e una chiave k, TREE-SEARCH restituisce un puntatore a un nodo con chiave k, se esiste, altrimenti restituisce il valore NIL.

```
TREE-SEARCH(x, k)
1  if x = NIL or k = x.key
2     return x
3  if k < x.key
4     return TREE-SEARCH(x.left, k)
5  else return TREE-SEARCH(x.right, k)
```

La procedura inizia la sua ricerca dalla radice e segue un cammino semplice verso il basso lungo l'albero. Per ogni nodo x che incontra, confronta la chiave k con $x.key$. Se le due chiavi sono uguali, la ricerca termina.

Se $k < x.key$, la ricerca continua nel sottoalbero sinistro di x, in quanto la proprietà degli alberi binari di ricerca implica che k non può essere memorizzato nel sottoalbero destro. Simmetricamente, se $k > x.key$, la ricerca continua nel sottoalbero destro.

I nodi incontrati durante la ricorsione formano un cammino semplice verso il basso dalla radice dell'albero, quindi il tempo di esecuzione di TREE-SEARCH è $O(h)$, dove h è l'altezza dell'albero.

La procedura è ricorsiva in coda e può essere riscritta in forma iterativa (al massimo ci mette il tempo necessario per andare dalla radice alla foglia) che è più efficiente nella maggior parte dei calcolatori.

```
ITERATIVE-TREE-SEARCH(x, k)
1  while x = NIL or k = x.key
2     if k < x.key
3        x ← x.left
4     else x ← x.right
5  return x
```

La correttezza di questa procedura è garantita dalla definizione di albero binario di ricerca. L'invariante di ciclo relativa alla forma iterativa di TREE-SEARCH è la seguente:

> *All'inizio di una generica iterazione, se la chiave è presente, si troverà nel sottoalbero avente radice x; se non è presente allora non si trova nel sottoalbero di radice x.*

10.2.2 Massimo e minimo

Un elemento con chiave minima in un albero binario di ricerca può sempre essere trovato seguendo, a partire dalla radice, i puntatori *left* dei figli a sinistra, fino a quando non viene incontrato un valore NIL. La procedura TREE-MINIMUM restituisce un puntatore all'elemento minimo nel sottoalbero con radice in un nodo x, che si suppone diverso da NIL.

La proprietà degli alberi binari di ricerca garantisce la correttezza di questa procedura. Se un nodo x non ha un sottoalbero sinistro, allora poiché ogni chiave nel sottoalbero destro di x è almeno grande quanto $x.key$, la chiave minima nel sottoalbero con radice in x è $x.key$. Se il nodo x ha un sottoalbero sinistro, allora, poiché nessuna chiave nel sottoalbero destro è minore di $x.key$ e ogni chiave nel sottoalbero sinistro non è maggiore di $x.key$, la chiave minima nel sottoalbero con radice in x può essere trovata nel sottoalbero con radice in $x.left$.

```
TREE-MINIMUM(x)
1  while x.left ≠ NIL
2      x ← x.key
3  return x
```

```
TREE-MAXIMUM(x)
1  while x.right ≠ NIL
2      x ← x.key
3  return x
```

Entrambe queste procedure vengono eseguite nel tempo $O(h)$ in un albero binario di ricerca di altezza h perché, come in TREE-SEARCH, la sequenza dei nodi incontrati forma un cammino semplice che scende dalla radice.

10.2.3 Successore e predecessore

Se tutte le chiavi appartenenti ad un albero binario di ricerca sono distinte, il successore di un nodo x è il nodo con la più piccola chiave che è maggiore di $x.key$. La struttura di un albero binario di ricerca consente di determinare il successore di un nodo senza mai confrontare le chiavi. La procedura TREE-SUCCESSOR restituisce il successore di un nodo x in un albero binario di ricerca, se esiste, oppure NIL se x ha la chiave massima nell'albero.

```
TREE-SUCCESSOR(x)
1  if x.right ≠ NIL
2      return TREE-MINIMUM(x.right)
3  y ← x.p
4  while y ≠ NIL and x = y.right
5      x ← y
6      y ← y.p
7  return y
```

Se il sottoalbero destro del nodo x non è vuoto, allora il successore di x è il minimo nel sottoalbero destro di x. Al contrario, se il sottoalbero destro del nodo x è vuoto e x ha un successore y, allora y è l'antenato più prossimo di x

il cui figlio sinistro è anche antenato di x. In alcuni casi, per trovare y si deve risalire l'albero partendo da x, finché non si incontra un nodo che è figlio sinistro di suo padre; questa operazione è svolta dalle righe 3 - 7 di TREE-SUCCESSOR.

Il tempo di esecuzione di TREE-SUCCESSOR in un albero di aletzza h è $O(h)$, perché si segue un cammino semplice che sale o uno che scende. Anche la procedura TREE-PREDECESSOR, che è simmetrica a TREE-SUCCESSOR, viene eseguita nel tempo $O(h)$.

10.3 Inserimento

Le operazioni di inserimento e cancellazione modificano l'insieme dinamico rappresentato da un albero binario di ricerca. La struttura dati deve essere modificata per riflettere questa modifica, ma in modo tale che la proprietà degli alberi binari di ricerca resti valida.

Per inserire un nuovo valore v in un albero binario di ricerca T, si utilizza la procedura TREE-INSERT. La procedura riceve un nodo z per cui $z.key = v$, $z.left =$ NIL, $z.right =$ NIL; essa modifica T e qualche attributo di z in modo che z sia inserito in una posizione appropriata nell'albero.

TREE-INSERT inizia dalla radice dell'albero e il puntatore x traccia il cammino semplice in discesa cercando un NIL da sostituire con l'elemento di input z. La procedura mantiene anche un puntatore inseguitore y che punta al padre di x.

```
TREE-INSERT(T, z)
1   y ← NIL
2   x ← T.root
3   while x ≠ NIL
4       y ← x
5       if z.key < x.key
6           x ← x.left
7       else x ← x.right
8   z.p ← y
9   if y = NIL
10      T.root ← z
11  elseif z.key < y.key
12      y.left ← z
13  else y.right ← z
```

Commenti:

3 Finché non arriva ad un punto in cui non ci sono figli

3-7 Ricerca il nodo che non ha figli

10 L'albero T era vuoto

11-13 Aggiorna il padre

11 y è l'ultimo nodo con chiave diversa da NIL

12 Se si viene da destra

13 Se si viene da sinistra

Dopo l'inizializzazione, la righe 3 - 7 del ciclo **while** spostano i puntatori x e y verso il basso, andando a sinistra o a destra a seconda dell'esito del confronto fra $z.key$ e $x.key$, finché a x non viene assegnato il valore NIL. Serve un puntatore inseguitore y perché, quando si trova il NIL dove mettere z, la ricerca è andata un passo oltre il nodo che deve essere modificato. Le righe 8 - 13 impostano i puntatori che servono a inserire z.

La procedura TREE-INSERT viene eseguita nel tempo $O(h)$ in un albero di altezza h. Nel caso migliore, ovvero quando l'albero è bilanciato, $h = lg\, n$, nel caso peggiore, ovvero quando l'albero è completamente sbilanciato, $h = n$. Nel caso medio, $h = n\, lg\, n$. In generale, il valore h dipende dall'**ordine** di inserimento.

È possibile utilizzare TREE-INSERT e INORDER-TREE-WALK per ordinare un insieme di numeri: si costruisce un albero binario di ricerca con i numeri dati in input e poi li si stampa con un attraversamento simmetrico. Per fare in modo che tale ordinamento sia stabile bisogna gestire i duplicati; si suppone per esempio che il duplicato venga messo a destra, così facendo però si sbilancia l'albero. Un'altra possibilità è quella di aggiungere una lista concatenata contenente le chiavi duplicate che verranno aggiunte in coda per essere più stabili. In questo caso si modifica la procedura TREE-INSERT per verificare se $z.key = x.key$ nel primo **if**, e $z.key = y.key$ all'interno di **elseif**. Oppure, si può utilizzare un flag booleano, che si aggiorna ad ogni iterazione, per distribuire i duplicati a sinistra o a destra del nodo.

10.4 Cancellazione

La procedura per cancellare un nodo z da un albero binario di ricerca considera tre casi base a seconda del numeri di figli

1. ***z* non ha figli**: si cancella z andando a modificare il puntatore del padre. Si modifica suo padre $p[z]$ per sostituire z con NIL come suo figlio.

2. ***z* ha un figlio**: si cancella z e si aggiorna il padre $z.p$ in modo che punti al figlio di z invece che a z. Ciò significa che si eleva il figlio di z in modo da occupare la posizione di z nell'albero, modificando il padre di z per sostituire z con il figlio di z.

3. ***z* ha due figli**: si trova il successore y di z, cioè il minimo del sottoalbero destro. Tale nodo y può non avere figli o può averne al massimo uno (altrimenti non sarebbe il successore) e in tal caso è solo il figlio destro, infatti, se avesse il figlio sinistro, non sarebbe il minimo. Si cerca $y = \min(z.right)$, si cancella y (considerando il caso 1 o 2) e si sostituisce la chiave di z e dati satellite con quelli di y. Questo significa che, dopo aver trovato il successore y del nodo z, si dovrà fare in modo che y assuma la posizione di z nell'albero. La parte restante del sottoalbero sinistro di z diventa il nuovo sottoalbero sinistro di y. Nel momento in cui si cancella y si devono considerare due sottocasi:

   - y è il figlio di z
   - y non è figlio di z

La procedura per cancellare un dato nodo z da un albero binario di ricerca T richiede come argomenti i puntatori a T e z. La procedura organizza la cancellazione in tre casi:

1. Se z non ha un figlio sinistro, si sostituisce z con il suo figlio destro, che può essere NIL oppure no. Se il figlio destro di z è NIL, si ha il caso in cui z non ha figli. Se il figlio destro di z non è NIL, si ha il caso in cui z ha un solo figlio, che è il suo figlio destro.

2. Se z ha un solo figlio, che è il suo figlio sinistro, si sostituisce z con il suo figlio sinistro.

3. Altrimenti, z ha un figlio sinistro e un figlio destro. Si trova il successore y di z, che si trova nel sottoalbero destro di z e non ha figlio sinistro. Si vuole staccare y dalla sua posizione corrente per metterlo al posto di z nell'albero:

   - Se y è il figlio destro di z, si sostituisce z con y, lasciando a y soltanto il figlio destro.
   - Altrimenti, y si trova nel sottoalbero destro di z, ma non è il figlio destro di z. In questo caso, si sostituisce prima y con il suo figlio destro; poi si sostituisce z con y.

Per poter spostare i sottoalberi all'interno dell'albero binario di ricerca, si definisce la procedura TRANSPLANT, che sostituisce un sottoalbero, come figlio di suo padre, con un altro sottoalbero. Quando TRANSPLANT sostituisce il sottoalbero con radice nel nodo u con il sottoalbero con radice nel nodo v, il padre del nodo u diventa il padre del nodo v, e il padre di u alla fine ha v come figlio.

```
TRANSPLANT(T, u, v)
1   if u.p = NIL
2       T.root ← v
3   elseif u = u.p.left
4       u.p.left ← v
5   else u.p.right ← v
6   if v ≠ NIL
7       v.p ← u.p
```

Le righe 1 - 2 gestiscono il caso in cui u è la radice di T. Altrimenti, u è un figlio sinistro o un figlio destro di suo padre. Le righe 3 - 4 aggiornano $u.p.left$ se u è un figlio sinistro; la riga 5 aggiorna $u.p.right$ se u è un figlio destro. Poiché v può essere NIL, le righe 6 - 7 aggiornano $v.p$ se v non è NIL. TRANSPLANT non aggiorna $v.left$ e $v.right$; fare ciò, o non farlo, è compito della procedura che chiama TRANSPLANT.

Dopo aver definito TRANSPLANT è possibile definire la procedura TREE-DELETE che cancella un nodo z dall'albero binario di ricerca T.

Ogni riga di TREE-DELETE, incluse le chiamate di TRANSPLANT, richiede un tempo costante, tranne la chiamata di TREE-MINIMUM nella riga 5. Quindi, la procedura TREE-DELETE viene eseguita nel tempo $O(h)$ in un albero binario di ricerca di altezza h.

Lo pseudocodice di TREE-DELETE è il seguente:

```
TREE-DELETE(T, z)
1   if z.left = NIL
2       TRANSPLANT(T, z, z.right)
3   elseif z.right = NIL
4       TRANSPLANT(T, z, z.left)
5   else y ← TREE-MINIMUM(z.right)
6       if y.p ≠ z
7           TRANSPLANT(T, y, y.right)
8           y.right ← z.right
9           y.right.p ← y
10      TRANSPLANT(T, z, y)
11      y.left ← z.left
12      y.left.p ← y
```

Le righe 1 - 2 gestiscono il caso in cui il nodo z non ha un figlio sinistro; le righe 3 - 4 gestiscono il caso in cui il nodo z ha un figlio sinistro, ma non ha un figlio destro. Le righe 5 - 12 trattano gli altri due casi in cui z ha due figli. La riga 5 trova il nodo y, che è successore di z. Poiché z ha un sottoalbero destro non vuoto, il suo successore deve essere il nodo di quel sottoalbero con la chiave più piccola; da qui la chiamata alla procedura TREE-MINIMUM. y, come detto, non ha un figlio sinistro. Si stacca y dalla sua posizione corrente e lo si mette al posto di z. Se y è il figlio destro di z, le righe 10 - 12 sostituiscono z, come figlio di suo madre, con y; poi, sostituiscono il figlio sinistro di y con il figlio sinistro di z. Se y non è figlio sinistro di z, le righe 7 - 9 sostituiscono y, come figlio di suo padre, con il figlio destro di y e cambiano il figlio destro di z nel figlio destro di y; le righe 10 - 12 sostituiscono z, come figlio di suo padre, con y e infine, rimpiazzano il figlio sinistro di y con il figlio sinistro di z.

Capitolo 11

Alberi rosso-neri

Gli alberi rosso-neri rappresentano uno dei tanti modi in cui gli alberi di ricerca vengono bilanciati per garantire che le operazioni elementari sugli insiemi dinamici richiedano un tempo $O(lg\, n)$ nel caso peggiore.

11.1 Proprietà degli alberi rosso-neri

Un **albero rosso-nero** è un albero binario di ricerca in cui ogni nodo x ha l'attributo booleano aggiuntivo $x.color$, ovvero il **colore** del nodo che può essere RED o BLACK.

Assegnando dei vincoli al modo in cui i nodi possono essere colorati lungo qualsiasi cammino semplice che va dalla radice a una foglia qualsiasi, gli alberi rosso-neri garantiscono che nessuno di tali cammini sia lungo più del doppio di qualsiasi altro, quindi l'albero è approssimativamente **bilanciato**.

Ogni nodo dell'albero contiene quindi gli attributi *color*, *key*, *left*, *right* e *p*. Se manca un figlio o il padre di un nodo, il corrispondente attributo puntatore del nodo contiene il valore NIL. In particolare, tutti i puntatori a NIL sono sostituiti con puntatori alla sentinella T.NIL. Si utilizza la sentinella T.NIL per tutte le foglie dell'albero rosso-nero T e come padre della radice dell'albero. Per T.NIL il suo attributo *color* è BLACK e gli attributi *key*, *left*, *right* e *p* possono assumere valori arbitrari perché sono irrilevanti.

Un albero rosso-nero è un albero binario di ricerca che gode di cinque proprietà fondamentali:

1. Ogni nodo è **rosso** o **nero**
2. La **radice** è **nera**
3. Ogni **foglia** (T.NIL) è **nera**
4. Se un nodo è **rosso**, allora entrambi i suoi figli sono **neri**. Ovvero, non possono essere presenti due nodi rossi consecutivi in un cammino semplice dalla radice a una foglia.
5. **Tutti** i cammini da ogni nodo alle foglie contengono lo **stesso numero** di nodi **neri** (si hanno tanti cammini quanto il numero di foglie)

Dato un generico nodo x appartenente ad un albero rosso-nero si definiscono due tipologie di altezza: l'**altezza**, indicata con *h(x)*, e l'**altezza nera**, indicata con *bh(x)*.

Altezza $h(x)$: Numero di archi nel cammino più lungo fino ad una foglia

Altezza nera $bh(x)$: Numero di nodi **neri** (incluso $T.\text{NIL}$) nel cammino da x alla foglia (escluso x)

Grazie alla quinta proprietà degli alberi rosso-neri, l'altezza nera di un generico nodo x è ben definita.

Teorema 11.1. *Ogni nodo con altezza h ha altezza nera* $\geq h/2$.

Dimostrazione. La quarta proprietà degli alberi rosso-neri garantisce che se un nodo è rosso entrambi i suoi figli sono neri. Questa proprietà implica che al massimo $h/2$ nodi nel cammino dal nodo alla foglia sono rossi. Dunque, almeno $h/2$ nodi sono neri. □

Teorema 11.2. *Il sottoalbero con radice in un nodo x qualsiasi contiene almeno* $2^{bh(x)} - 1$ *nodi interni.*

Dimostrazione. Si procede per induzione sull'altezza di x. Si pone $h = h(x)$ e $bh = bh(x)$.

Passo base Se l'altezza di x è 0, cioè $h = h(x) = 0$, allora x deve essere una foglia ($bh(x) = 0$) e il sottoalbero con radice in x contiene almeno $2^{bh(x)} - 1 = 2^0 - 1 = 0$ nodi interni.

Passo induttivo Si considera un nodo x che ha un'altezza positiva ed è quindi un nodo interno con due figli. Ogni figlio di x ha altezza $h' = h-1$ e altezza nera bh' pari a $bh(x)$, se il suo colore è rosso, o pari a $bh(x) - 1$ se il suo colore è nero (per bh si valuta il figlio perché nel conteggio di $bh(x)$ si esclude il nodo x). Poiché l'altezza di un figlio di x è minore dell'altezza di x, si può applicare l'ipotesi induttiva per concludere che ogni figlio ha almeno $2^{bh(x)-1} - 1$ nodi interni. Quindi, il sottoalbero con radice in x contiene almeno $(2^{bh(x)-1}-1) + (2^{bh(x)-1}-1) + 1 = 2^{bh(x)}-1$ nodi interni ($2^{bh(x)-1}-1$ compare due volte perché rappresenta sia il sottoalbero destro sia quello sinistro); questo dimostra l'asserzione.

□

Lemma 11.1. *L'altezza massima di un albero rosso-nero con n nodi interni è* $2\,lg(n+1)$.

Dimostrazione.

$$n \geq 2^{bh} - 1 \geq 2^{h/2} - 1 \Rightarrow n + 1 \geq 2^{h/2} \Rightarrow lg(n+1) \geq h/2 \Rightarrow h \leq 2\,lg(n+1)$$

□

Un'immediata conseguenza di questo lemma è che le operazioni sugli insiemi dinamici SEARCH, MINIMUM, MAXIMUM, SUCCESSOR e PREDECESSOR possono essere implementate nel tempo $O(lg\,n)$ negli alberi rosso-neri, perché possono essere eseguite nel tempo $O(h)$ in un albero binario di ricerca di altezza h e qualsiasi albero rosso-nero di n nodi è un albero binario di ricerca di altezza $O(lg\,n)$.

11.2 Rotazioni

Poiché le operazioni TREE-INSERT e TREE-DELETE modificano l'albero su cui si sta operando, il risultato potrebbe violare le proprietà degli alberi rosso-neri. Per ripristinare queste proprietà, si dovrà modificare i colori di qualche nodo dell'albero e anche la struttura dei puntatori. La rotazione consente dunque una **ristrutturazione** dell'albero e viene usata per mantenere gli alberi rosso-neri **bilanciati**.

La struttura dei puntatori viene modificata tramite una **rotazione**: un'operazione locale in un albero di ricerca che mantiene l'ordine delle chiavi e preserva la proprietà degli alberi binari di ricerca. I due tipi di rotazione sono: rotazione sinistra e rotazione destra. Quando si esegue una rotazione sinistra in un nodo x, supponendo che il suo figlio destro y non sia $T.\text{NIL}$; x può essere qualsiasi nodo il cui figlio destro non è $T.\text{NIL}$. La rotazione sinistra fa perno sul collegamento tra x e y; il nodo y diventa la nuova radice del sottoalbero, con x come figlio sinistro di y e il figlio sinistro di y come figlio destro di x.

Lo pseudocodice per LEFT-ROTATE suppone che $x.right \neq T.\text{NIL}$ e che il padre della radice sia $T.\text{NIL}$.

```
LEFT-ROTATE(T, x)
1   y ← x.right
2   x.right ← y.left
3   if y.left ≠ T.NIL
4       y.left.p ← x
5   y.p ← x.p
6   if x.p = T.NIL
7       T.root ← y
8   elseif x = x.p.left
9       x.p.left ← y
10  else x.p.right ← y
11  y.left ← x
12  x.p ← y
```

Commenti:

- 1 Imposta y
- 2 Sposta il sottoalbero sinistro di y nel sottoalbero destro di x
- 3 Controlla che il figlio sinistro di y non sia vuoto
- 5 Collega il padre di x a y
- 6-8 Se x era la sua radice.. altrimenti aggiusta alla riga 8
- 11 Pone x a sinistra di y

Il codice per RIGHT-ROTATE è simmetrico. Sia la procedura LEFT-ROTATE che RIGHT-ROTATE vengono eseguite nel tempo $O(1)$. Soltanto i puntatori vengono modificati da una rotazione; tutti gli altri attributi di un nodo non cambiano. Dopo la rotazione l'albero è più bilanciato.

11.3 Inserimento

L'inserimento di un nodo in un albero rosso-nero di n nodi può essere effettuato nel tempo $O(lg\, n)$. Si utilizza una versione modificata della procedura TREE-INSERT per inserire un nodo z nell'albero T come se fosse un normale albero binario di ricerca e poi si colora z di rosso. Per garantire che le proprietà degli alberi rosso-neri siano preservate, si chiama una procedura ausiliaria RB-INSERT-FIXUP che ricolora i nodi ed effettua delle rotazioni.

```
RB-INSERT(T, z)
1    y ← T.NIL
2    x ← T.root
3    while x ≠ T.NIL
4        y ← x
5        if z.key < x.key
6            x ← x.left
7        else x ← x.right
8    z.p ← y
9    if y = T.NIL
10       T.root ← z
11   elseif z.key < y.key
12       y.left ← z
13   else y.right ← z
14   z.left ← T.NIL
14   z.right ← T.NIL
16   z.color ← RED
17   RB-INSERT-FIXUP(T, z)
```

Commenti:

1-15 Normale inserimento in un albero binario di ricerca

17 Dopo aver colorato il nuovo nodo in rosso si potrebbero violare le proprietà degli alberi rosso-neri il che comporta che l'albero si sta sbilanciando; si chiama dunque la procedura RB-INSERT-FIXUP per aggiustare l'albero

Tutte le istanze di NIL in TREE-INSERT sono sostituite con T.NIL. Si assegna T.NIL a $z.left$ e $z.right$ nelle righe 14 - 15 per mantenere la struttura appropriata dell'albero.

Dopo aver inserito il nuovo nodo z e averlo colorato in rosso si potrebbe violare una proprietà degli alberi rosso-neri:

P1 OK

P3 OK

P5 OK

P2 Violata se z è la nuova radice dell'albero rosso-nero

P4 Violata se $z.p$ è rosso. In tal caso si avrebbero due rossi di fila

Per arrivare a definire completamente la procedura RB-Insert-Fixup si parte dal ciclo **while** più esterno

```
RB-Insert-Fixup(T, z)
1   while z.p.color = RED
⋮
16  T.root.color ← BLACK
```

A questo ciclo **while** è associata la seguente invariante di ciclo composta da tre parti:

> *All'inizio di ogni iterazione del ciclo:*
>
> - *Il nodo z è rosso*
> - *Se z.p è la radice, allora z.p è nero*
> - *Se ci sono violazioni delle proprietà degli alberi rosso-neri, al massimo ce n'è una su tutto l'albero e riguarda la proprietà 2 o la proprietà 4. Se c'è una violazione della seconda proprietà, questa si verifica perché il nodo z è la radice ed è rossa. Se c'è una violazione della quarta proprietà, essa si verifica perché z e z.p sono entrambi rossi*

Si dimostra ora la sua validità:

Inizializzazione Prima della prima iterazione del ciclo, si parte da un albero rosso-nero senza violazioni a cui si è aggiunto un nodo rosso z:

- z è settato rosso
- Se $z.p$ è la radice, allora se era rosso viene violata la quarta proprietà, altrimenti tutte le proprietà sono rispettate
- Il resto dell'albero rimane immutato, è stato aggiunto un nodo che viene colorato in rosso ma questo non varia le altezze nere che sono dunque corrette

Se c'è una violazione della seconda proprietà, allora la radice rossa deve essere il nodo z appena inserito, che in tal caso è l'unico nodo interno nell'albero. Poiché il padre ed entrambi i figli di z sono la sentinella, che è nera, non c'è violazione della quarta proprietà. Quindi, questa violazione della seconda proprietà è l'unica violazione delle proprietà degli alberi rosso-neri nell'intero albero. Se c'è una violazione della quarta proprietà, allora, poiché i figli del nodo z sono sentinelle nere e l'albero non aveva altre violazioni prima dell'inserimento di z, la violazione deve essere attribuita al fatto che z e $z.p$ sono entrambi rossi. Non ci sono altre violazioni delle proprietà degli alberi rosso-neri

Conclusione Il ciclo termina perché $z.p$ è nero. Quindi, non c'è violazione della quarta proprietà alla conclusione del ciclo. Per l'invariante di ciclo, l'unica proprietà che potrebbe essere violata è la seconda. La riga 16 ripristina anche questa proprietà, cosicché quando RB-Insert-Fixup termina, tutte le proprietà degli alberi rosso-neri sono valide

Conservazione Ci sarebbero sei casi da considerare nel ciclo **while** ma tre di essi sono simmetrici agli altri tre, a seconda che il padre $z.p$ di z sia un figlio sinistro o un figlio destro del nonno $z.p.p$ di z; ciò è determinato nella riga 2. Si considera solo il caso in cui $z.p$ è un figlio sinistro. Il nonno $z.p.p$ esiste, in quanto per il secondo punto dell'invariante di ciclo, se $z.p$ è la radice, allora $z.p$ è nero. Poiché si entra in una iterazione del ciclo soltanto se $z.p$ è rosso, si sa che $z.p$ non può essere la radice. Quindi $z.p.p$ esiste. Il caso 1 si distingue dai casi 2 e 3 per il colore del fratello del padre di z (lo zio di z). La riga 3 fa sì che y punti allo zio $z.p.p.right$ di z e la riga 4 effettua un test. Se y è rosso, allora viene applicato il caso 1, altrimenti il controllo passa ai casi 2 e 3. In tutti e tre i casi, il nonno $z.p.p$ di z è nero, in quanto il padre $z.p$ è rosso, quindi la quarta proprietà è violata solo fra z e $z.p$.

Caso 1 : lo zio y di z è rosso Questo caso viene eseguito quando $z.p$ e y sono entrambi rossi. Poiché il nonno $z.p.p$ è nero (sicuro perché prima dell'inserimento era un albero rosso-nero), si può colorare di nero $z.p$ e y, risolvendo così il problema che z e $z.p$ sono entrambi rossi e rispettando così la quarta proprietà; si colora di rosso $z.p.p$ per conservare la quinta proprietà. A questo punto si ripete il ciclo **while** con $z.p.p$ come il nuovo nodo z. Il puntatore z si sposta di due livelli in alto nell'albero.

```
RB-Insert-Fixup(T, z)
1   while z.p.color = RED
2       if z.p = z.p.p.left
3           y ← z.p.p.right
4           if y.color = RED
5               z.p.color ← BLACK
6               y.color ← BLACK
7               z.p.p.color ← RED
8               z ← z.p.p
9           else
⋮
```

Commenti:

2 Figlio sinistro
4 Caso 1 : lo zio è rosso
5-7 Cambia i colori come visto
8 Va al nonno
9 Else... lo zio è nero

Caso 2 : lo zio y di z è nero e z è un figlio destro Questo caso e il successivo si distinguono a seconda che z sia un figlio destro o sinistro di $z.p$. Le righe 10 - 11 costituiscono il caso 2. In questo caso, il nodo z è un figlio destro di suo padre, $z.p.p$ è ancora nero. Si effettua una rotazione sinistra intorno a $z.p$ per trasformare la situazione nel caso 3 in cui il nodo z è un figlio sinistro. Poiché z e $z.p$ sono entrambi rossi, la rotazione non influisce né sull'altezza nera dei nodi né sulla quinta proprietà. Non vengono cambiati i colori.

```
RB-Insert-Fixup(T, z)
⋮
9          else if z = z.p.right
10                 z ← z.p
11                 Left-Rotate(T, z)
⋮
```

Caso 3 : lo zio *y* di *z* è nero e *z* è un figlio sinistro Sia che si entri nel caso 3 direttamente o tramite il caso 2, lo zio y di z è nero, perché altrimenti si avrebbe eseguito il caso 1. In aggiunta, il nodo $z.p.p$ esiste e, dopo aver spostato z di un livello in alto nella riga 10 e poi di un livello in basso nella 11, l'identità di $z.p.p$ resta invariata. In questo caso, si colora $z.p$ di nero, $z.p.p$ di rosso e si effettua una rotazione destra su $z.p.p$ per preservare tutte le proprietà. Il corpo del **while** non viene eseguito un'altra volta, in quanto $z.p$ ora è nero.

```
RB-Insert-Fixup(T, z)
⋮
12              z.p.color ← BLACK
13              z.p.p.color ← RED
14              Right-Rotate(T, z.p.p)
⋮
```

Si arriva quindi alla versione completa della procedura RB-Insert-Fixup:

```
RB-Insert-Fixup(T, z)
1   while z.p.color = RED
2     if z.p = z.p.p.left
3       y ← z.p.p.right
4       if y.color = RED
5         z.p.color ← BLACK
6         y.color ← BLACK
7         z.p.p.color ← RED
8         z ← z.p.p
9       else if z = z.p.right
10              z ← z.p
11              Left-Rotate(T, z)
12        z.p.color ← BLACK
13        z.p.p.color ← RED
14        Right-Rotate(T, z.p.p)
15    else (come righe 3 - 14 scambiando right e left)
16  T.root.color ← BLACK
```

Commenti:

5-8 Risolve il caso 1

10-11 Risolve il caso 2

11 Trasforma il caso 2 nel caso 3

12-14 Risolve il caso 3

Per un albero rosso-nero di n nodi le righe 1 - 16 di RB-Insert richiedono il tempo $O(lg\, n)$. Nella procedura RB-Insert-Fixup il ciclo **while** viene ripetuto soltanto se viene eseguito il caso 1 e in tal caso il puntatore z si sposta di due livelli in alto nell'albero. Il numero totale di volte che può essere eseguito il ciclo **while** è quindi $O(lg\, n)$. Di conseguenza, RB-Insert richiede un tempo totale pari a $O(lg\, n)$. Il ciclo **while** non effettua mai più di due rotazioni, perché termina se viene eseguito il caso 2 o il caso 3.

11.4 Cancellazione

Quando si cancella un nodo bisogna stare attenti a che colore aveva il nodo rimosso:

Rosso In questo caso non ci sono problemi perché, dopo aver rimosso il nodo, l'altezza nera bh non è variata, non si hanno due nodi rossi a fila e non viene violata la seconda proprietà perché se il nodo eliminato è rosso tale nodo non è sicuramente la radice.

Nero Si potrebbe violare la seconda proprietà se il nodo rimosso era la radice e suo figlio (la nuova radice) era rosso, oppure la quarta proprietà (due rossi a fila), oppure ancora la quinta proprietà (cammino nero).

Ci sarebbero nove casi da considerare e risolvere.

Capitolo 12

Strutture dati aumentate

Raramente si andrà a progettare una struttura dati da zero. Più frequentemente, sarà sufficiente prendere una struttura dati nota e aggiungere nuove informazioni; ovvero sarà sufficiente aumentare una struttura dati elementare memorizzando in essa delle informazioni aggiuntive. Oltre l'aggiunta di nuove informazioni sarà anche possibile andare a definire delle nuove operazioni per la struttura dati considerata. Aumentare una struttura dati non è sempre semplice, in quanto le informazioni aggiuntive devono essere aggiornate e gestite dalle ordinarie operazioni sulla struttura dati e devono essere gestite correttamente senza perdita di efficienza.

12.1 Statistiche d'ordine dinamiche

L'i-esima **statistica d'ordine** di un insieme di n elementi, con $i \in \{1, 2, \ldots, n\}$, è l'elemento dell'insieme con l'i-esima chiave più piccola. Qualsiasi statistica d'ordine su di un insieme dinamico può essere ottenuta nel tempo $O(n)$ da un insieme non ordinato. Si definisce invece **rango** di un elemento, la posizione che occupa nella sequenza ordinata degli elementi dell'insieme. A differenza della statistica d'ordine, il rango può essere determinato nel tempo $O(lg\, n)$.

Un **albero per statistiche d'ordine T** è un albero rosso-nero con un'informazione aggiuntiva memorizzata in ogni nodo. In un nodo x di un albero rosso-nero, oltre agli attributi usuali *x.key*, *x.left*, *x.right*, *x.p* e *x.color* compare il nuovo attributo *x.size*. Questo attributo contiene il numero di nodi interni nel sottoalbero con radice in x (incluso lo stesso x ed escluse le foglie - sentinelle), cioè la dimensione del sottoalbero. Se si definisce che la dimensione della sentinella $T.\text{NIL}$ è 0, ovvero si imposta $T.\text{NIL}.size$ a 0, allora si ha l'identità:

$$x.size = x.left.size + x.right.size + 1$$

Il +1 indica il nodo x stesso.

In un albero per statistiche d'ordine non è richiesto che le chiavi siano distinte. In presenza di chiavi uguali, la precedente notazione di rango non è ben definita. Si elimina questa ambiguità definendo il rango di un elemento come la posizione in cui l'elemento sarebbe elencato in un attraversamento simmetrico dell'albero (inorder).

12.1.1 Ricerca di un elemento con un dato rango

La procedura OS-SELECT consente di trovare un elemento con un dato rango. Ritorna il puntatore al nodo che contiene l'i-esima chiave più piccola nel sottoalbero con radice in x. Per trovare il nodo con l'i-esima chiave più piccola in un albero per statistiche d'ordine T, la chiamata iniziale è OS-SELECT($T.root, i$).

```
OS-SELECT(x, i)
1   r ← x.left.size + 1
2   if i = r
3       return x
4   elseif i < r
5       return OS-SELECT(x.left, i)
6   else return OS-SELECT(x.right, i - r)
```

Commenti:

0 Parte dalla radice e via via scende

1 r è il rango del nodo x

5 Va nel sottoalbero sinistro se minore

6 Va nel sottoalbero destro se maggiore

6 i - r sono i valori che seguono r che hanno rango successivo

Nella riga 1 di OS-SELECT si calcola r, il rango del nodo x nel sottoalbero di radice x. Il valore di *x.left.size* è il numero di nodi che precedono x in un attraversamento simmetrico del sottoalbero con radice in x. Quindi, il valore *x.left.size* + 1 è il rango di x all'interno del sottoalbero con radice in x. Se $i = r$, allora il nodo x è l'i-esimo elemento più piccolo, quindi la riga 3 restituisce x. Se $i < r$, allora l'i-esimo elemento più piccolo è nel sottoalbero sinistro di x, quindi la riga 5 effettua una ricorsione su *x.left*. Se $i > r$, allora l'i-esimo elemento più piccolo è nel sottoalbero destro di x. Poiché ci sono r elementi nel sottoalbero con radice in x che precedono il sottoalbero destro di x in un attraversamento simmetrico, l'i-esimo elemento più piccolo nel sottoalbero con radice in x è l'$(i-r)$-esimo elemento più piccolo nel sottoalbero con radice in *x.right*. Questo elemento è determinato in modo ricorsivo nella riga 6.

La procedura OS-SELECT è ricorsiva in coda e può dunque essere vista come una procedura iterativa a cui è associata l'invariante di ciclo:

> *Prima di ogni iterazione l'i-esimo valore si trova nel sottoalbero con radice in x*

Indicato con r il rango di x nel sottoalbero con radice in x:

- Se $i = r$, l'invariante è verificata perché ritorna x
- Se $i < r$, allora l'i-esimo elemento più piccolo è nel sottoalbero sinistro
- Se $i > r$, allora l'i-esimo elemento più piccolo si trova nel sottoalbero destro. Si tolgono gli r elementi nel sottoalbero di x che precedono quelli nel sottoalbero destro di x

Poiché per ogni chiamata ricorsiva si scende di un livello nell'albero per statistiche d'ordine, il tempo totale di OS-SELECT, nel caso peggiore, è proporzionale all'altezza dell'albero. Poiché l'albero è un albero rosso-nero, la sua altezza è $O(lg\, n)$, dove n è il numero di nodi. Quindi, il tempo di esecuzione di OS-SELECT è $O(lg\, n)$ per un insieme dinamico di n elementi.

12.1.2 Determinare il rango di un elemento

Dato un puntatore a un nodo x in un albero per statistiche d'ordine T, la procedura OS-RANK restituisce la posizione di x nell'ordinamento lineare determinato da un attraversamento simmetrico dell'albero T.

```
OS-Rank(T, x)
1   r ← x.left.size + 1
2   y ← x
3   while y ≠ T.root
4       if y = y.p.right
5           r ← r + y.p.left.size + 1
6       y ← y.p
7   return r
```

Commenti:

0 Parte da un nodo e risale verso la radice

3 Finché non arriva alla radice

4-5 Si deve capire se il nodo è un figlio destro o un figlio sinistro del padre. Se è un figlio destro, cambia posizione

6 Se figlio sinistro la posizione non cambia

Il rango di x può essere considerato come il numero di nodi che precedono x in un attraversamento simmetrico dell'albero, più 1 per x stesso. OS-RANK conserva la seguente invariante di ciclo:

> *All'inizio di ogni iterazione del ciclo* ***while*** *(righe 3 - 6), r è il rango di x.key nel sottoalbero con radice nel nodo y. Ovvero si verifica che r = rank(x.key, y)*

Si dimostra ora la sua validità:

Inizializzazione Prima della prima iterazione, la riga 1 assegna a r il rango di *x.key* all'interno del sottoalbero con radice in x. L'assegnazione di x ad y nella riga 2 rende l'invariante vera la prima volta che viene eseguito il test nella riga 3

Conservazione Se, all'inizio del ciclo, r è il rango di x.key nel sottoalbero con radice nel nodo y, allora, alla fine del ciclo, r è il rango di *x.key* nel sottoalbero con radice nel nodo *y.p*. Se y è un figlio sinistro, il sottoalbero del fratello ha nodi che seguono x, dunque r non cambia. Se, invece, y è un figlio destro, tutti i nodi nel sottoalbero del fratello precedono tutti i nodi nel sottoalbero di y, dunque $r \leftarrow r + y.p.left.size + 1$

Conclusione Il ciclo termina quando $y = T.root$, quindi il sottoalbero con radice in y è l'intero albero. Dunque, il valore di r è il rango di $x.key$ nell'intero albero

Poiché ogni iterazione del ciclo **while** impiega il tempo $O(1)$ e y risale di un livello nell'albero a ogni iterazione, il tempo di esecuzione di OS-Rank, nel caso peggiore, è proporzionale all'altezza dell'albero: $O(lg\, n)$ in un albero per statistiche d'ordine di n nodi.

12.1.3 Gestione delle dimensioni dei sottoalberi

Dato l'attributo *size* in ogni nodo, OS-Select e OS-Rank possono calcolare rapidamente le informazioni sulle statistiche d'ordine. Tuttavia, questo lavoro risulterebbe inutile se questi attributi non potessero essere gestiti con efficienza dalle operazioni di base che modificano gli alberi rosso-neri. Si dovranno quindi gestire le dimensioni dei sottoalberi durante le operazioni di inserimento e cancellazione senza influire sul tempo di esecuzione asintotico di ciascuna operazione.

L'inserimento in un albero rosso-nero si svolge in due fasi. Nella prima fase, si discende dalla radice dell'albero, inserendo il nuovo nodo come figlio di un nodo esistente. Nella seconda fase si risale verso la radice, cambiando i colori ed effettuando qualche rotazione per conservare le proprietà degli alberi rosso-neri.

Per gestire le dimensioni dei sottoalberi nella prima fase, si incrementa semplicemente $x.size$ per ogni nodo x nel cammino semplice dalla radice fino alle foglie. Il nuovo nodo che viene aggiunto ha l'attributo *size* pari a 1. Poiché ci sono $O(lg\, n)$ nodi lungo il cammino percorso, il costo addizionale per gestire gli attributi *size* è $O(lg\, n)$.

Nella seconda fase, le uniche modifiche strutturali dell'albero rosso-nero di base sono provocate dalle rotazioni, che sono al massimo due. Inoltre, una rotazione è un'operazione locale: soltanto due nodi hanno gli attributi *size* invalidati, i due nodi uniti dal collegamento attorno ai quali viene effettuata la rotazione. Facendo riferimento al codice della procedura Left-Rotate(T, x) si aggiornano le seguenti righe:

13 $y.size \leftarrow x.size$
14 $x.size \leftarrow x.left.size + x.right.size + 1$

La modifica di Right-Rotate è simmetrica.

Poiché vengono effettuate al massimo due rotazioni durante l'inserimento in un albero rosso-nero, occorre soltanto un tempo addizionale $O(1)$ per aggiornare gli attributi *size* nella seconda fase. Quindi, il tempo totale per completare l'inserimento in un albero per statistiche d'ordine di n nodi è $O(lg\, n)$, che è asintoticamente uguale a quello di un normale albero rosso-nero.

Anche la cancellazione da un albero rosso-nero è formata da due fasi: la prima opera sull'albero di ricerca sottostante; la seconda provoca al massimo tre rotazioni, senza altre modifiche strutturali. La prima fase o rimuove un nodo y oppure lo sposta più in alto nell'albero. Per aggiornare le dimensioni dei sottoalberi, si segue un cammino semplice dal nodo y (partendo dalla sua posizione originale nell'albero) fino alla radice, riducendo il valore dell'attributo *size* per ogni nodo che si incontra. Poiché questo cammino semplice ha una

lunghezza $O(lg\, n)$ in un albero rosso-nero di n nodi, il tempo aggiuntivo che viene impiegato per gestire gli attributi *size* nella prima fase è $O(lg\, n)$. Le $O(1)$ rotazioni nella seconda fase della cancellazione possono essere gestite come è stato fatto nell'inserimento. Quindi, le operazioni di inserimento e cancellazione, inclusa la gestione degli attributi *size*, richiedono un tempo $O(lg\, n)$ su di un albero per statistiche d'ordine di n nodi.

12.2 Aumentare una struttura dati

Il procedimento per aumentare una struttura dati può essere suddiviso in quattro passi:

1. Scegliere una struttura dati di base
2. Determinare le informazioni aggiuntive da gestire nella struttura dati di base
3. Verificare che le informazioni aggiuntive possono essere gestite dalle operazioni elementari sulla struttura dati di base che la modificano
4. Sviluppare nuove operazioni

Non occorre seguire i passi nell'ordine in cui sono elencati. Spesso la progettazione include una fase in cui si procede per tentativi e il progresso nei vari passi, di solito, avviene in parallelo.

Per poter progettare gli alberi per statistiche d'ordine:

1. Si scelgono gli alberi rosso-neri come struttura dati di base. Un segnale sull'idoneità degli alberi rosso-neri proviene dal loro efficiente supporto ad altre operazioni sugli insiemi dinamici con ordinamento totale, come MINIMUM, MAXIMUM, SUCCESSOR e PREDECESSOR
2. Si aggiunge l'attributo *size*. Ogni nodo memorizza la dimensione del sottoalbero con radice in x. In generale, le informazioni aggiuntive rendono le operazioni più efficienti
3. Si verifica che si possono gestire le informazioni aggiuntive per le operazioni esistenti sulla struttura dati. In particolare si è garantito che le operazioni di inserimento e cancellazione possono gestire correttamente gli attributi *size*, continuando ad essere eseguite nel tempo $O(lg\, n)$
4. Si sviluppano le nuove operazioni OS-SELECT e OS-RANK

12.2.1 Aumentare gli alberi rosso-neri

Quando una struttura dati aumentata si basa sugli alberi rosso-neri, si può provare che certi tipi di informazioni aggiuntive possono essere gestiti in maniera efficiente nelle operazioni di inserimento e cancellazione.

Teorema 12.1. *Sia f un attributo che aumenta un albero rosso-nero T di n nodi; si suppone che il valore di f per un nodo x possa essere calcolato utilizzando soltanto le informazioni nei nodi x, $x.left$ e $x.right$, inclusi $x.left.f$ e $x.right.f$ (solo dal nodo stesso, dal figlio destro e sinistro e non da tutto l'albero). Allora,*

è possibile gestire i valori di f in tutti i nodi di T durante l'inserimento e la cancellazione, senza influire asintoticamente sulla prestazione $O(lg\, n)$ di queste operazioni.

Dimostrazione. Il concetto che sta alla base della dimostrazione è che la modifica di un attributo f in un nodo x si propaga soltanto agli antenati di x nell'albero. Ovvero, la modifica di $x.f$ potrebbe richiedere l'aggiornamento di $x.p.f$, ma nient'altro; l'aggiornamento di $x.p.f$ potrebbe richiedere l'aggiornamento di $x.p.p.f$, ma nient'altro; e così via risalendo l'albero. Quando viene aggiornato $T.root.f$, nessun altro nodo dipende da questo nuovo valore, quindi il processo termina. Poiché l'altezza di un albero rosso-nero è $O(lg\, n)$, la modifica di un attributo f in un nodo richiede un tempo $O(lg\, n)$ per aggiornare i nodi che dipendono da tale modifica.

L'inserimento di un nodo x nell'albero T si compone di due fasi. Durante la prima fase, x viene inserito come figlio di un nodo esistente $x.p$. Il valore di $x.f$ può essere calcolato nel tempo $O(1)$ perché, per ipotesi, dipende soltanto dalle informazioni negli altri attributi dello stesso x e dalle informazioni dei figli di x, ma i figli di x sono entrambi la sentinella $T.\text{NIL}$. Una volta calcolato $x.f$, la modifica si propaga verso l'alto nell'albero. Quindi, il tempo totale per la prima fase dell'inserimento è $O(lg\, n)$. Durante la seconda fase, le uniche modifiche strutturali dell'albero derivano dalle rotazioni. Poiché in una rotazione cambiano soltanto due nodi, il tempo totale per aggiornare gli attributi f è $O(lg\, n)$ per rotazione. Dal momento che in un inserimento ci sono al massimo due rotazioni, il tempo totale per l'inserimento è $O(lg\, n)$.

Come l'inserimento, anche la cancellazione si svolge in due fasi. Nella prima fase, la modifiche dell'albero si verificano quando il nodo cancellato viene effettivamente rimosso dall'albero. Se il nodo cancellato aveva due figli il suo successore viene messo al suo posto e quindi il nodo effettivamente rimosso è il successore. La propagazione degli aggiornamenti di f indotti da queste modifiche costa al massimo $O(lg\, n)$, in quanto le modifiche cambiano localmente l'albero. La sistemazione dell'albero rosso-nero durante la seconda fase richiede al massimo tre rotazioni, ciascuna delle quali impiega al massimo il tempo $O(lg\, n)$ per propagare gli aggiornamenti di f. Quindi, come per l'inserimento, il tempo totale per la cancellazione è $O(lg\, n)$. □

In molti casi, come la gestione degli attributi *size* negli alberi per statistiche d'ordine, il costo di aggiornamento dopo una rotazione è $O(1)$, anziché $O(lg\, n)$ come appena dimostrato.

12.3 Alberi di Intervalli

Tramite gli alberi di intervalli è possibile gestire un insieme di intervalli, per esempio gli intervalli temporali. È possibile rappresentare un intervallo $[t_1, t_2]$, con $t_1 \leq t_2$, come un oggetto i, con gli attributi $i.low = t_1$ (**estremo inferiore**) e $i.high = t_2$ (**estremo superiore**). Si dirà che gli intervalli i e j **si sovrappongono** se e solo se $i \cap j \neq \emptyset$, ovvero se $i.low \leq j.high$ e $j.low \leq i.high$. Alternativamente, i e j **non si sovrappongono** se e solo se $i.low > j.high$ o $j.low > i.high$. Due intervalli qualsiasi i e j soddisfano la **tricotomia degli intervalli**; ovvero una sola delle seguenti proprietà può essere vera

- i e j si sovrappongono
- i è a sinistra di j (cioè $i.high < j.low$)
- i è a destra di j (cioè $j.high < i.low$)

Un **albero di intervalli** è un albero rosso-nero che gestisce un insieme dinamico di elementi, in cui ogni elemento x contiene un intervallo $x.int$. Gli alberi di intervalli supportano le seguenti operazioni:

INTERVAL-INSERT(T, x) aggiunge l'elemento x, il cui attributo int si suppone contenga un intervallo, all'albero di intervalli T

INTERVAL-DELETE(T, x) rimuove l'elemento x dall'albero di intervalli T

INTERVAL-SEARCH(T, i) restituisce un puntatore a un elemento x nell'albero di intervalli T tale che $x.int$ si sovrappone all'intervallo i o restituisce un puntatore alla sentinella $T.\text{NIL}$ se non esiste tale elemento nell'insieme

12.3.1 Progetto di un albero di intervalli

Come visto precedentemente, per poter aumentare una struttura dati, si segue un approccio diviso in quattro passi:

1. Scegliere una struttura dati sottostante
2. Determinare le informazioni aggiuntive da gestire
3. Verificare che si possono gestire le informazioni aggiuntive per le operazioni esistenti sulla struttura dati
4. Sviluppare nuove operazioni

Nel caso degli alberi di intervalli:

Passo 1 : Struttura dati di base Si sceglie un albero rosso-nero in cui ogni nodo x contiene un intervallo $x.int$ e la chiave di x è l'estremo inferiore, $x.int.low$, dell'intervallo. Quindi, un attraversamento simmetrico (attraversamento inorder) della struttura dati elenca ordinatamente gli intervalli in funzione dell'estremo inferiore

Passo 2 : Informazioni aggiuntive Oltre agli intervalli, ogni nodo x contiene un valore $x.max$, che è il massimo tra tutti gli estremi destri degli intervalli memorizzati nel sottoalbero con radice in x.

$$x.max = max \begin{cases} x.int.high \\ x.left.max \\ x.right.max \end{cases}$$

Si può verificare che $x.left.max > x.right.max$ perché si ordina rispetto a $x.int.low$, cioè la posizione nell'albero è determinata solo dall'estremo inferiore non da quello superiore.

Passo 3 : Gestione delle informazioni Si deve verificare che le operazioni di inserimento e cancellazione possono essere svolte nel tempo $O(lg\, n)$ in un albero di intervalli di n nodi. Se si conosce l'intervallo *x.int*, quindi *x.int.high*, e i valori *max* dei figli *x.left* e *x.right* del nodo x, è facile determinare *x.max* come visto nel punto precedente. Per il teorema (12.1), precedentemente dimostrato, le operazioni di inserimento e cancellazione vengono eseguite nel tempo $O(lg\, n)$. In realtà, l'aggiornamento degli attributi *max* può essere eseguito nel tempo $O(1)$ per ogni rotazione.

Passo 4 : Sviluppare le nuove operazioni L'unica operazione da implementare è INTERVAL-SEARCH(T, i) che trova un nodo nell'albero T il cui intervallo si sovrappone all'intervallo i. Se non c'è un intervallo che si sovrappone a i nell'albero, viene restituito un puntatore alla sentinella T.NIL.

```
INTERVAL-SEARCH(T, i)
1  x ← T.root
2  while x ≠ T.NIL e i non si sovrappone a x.int
3      if x.left ≠ T.NIL e x.left.max ≥ i.low
4          x ← x.left
5      else x ← x.right
6  return x
```

La ricerca di un intervallo che si sovrappone a i inizia con x nella radice dell'albero e prosegue verso il basso. Termina quando viene trovato un intervallo che si sovrappone a i o quando x punta alla sentinella T.NIL. Poiché ogni iterazione del ciclo di base impiega il tempo $O(1)$ e poiché l'altezza di un albero rosso-nero di n nodi è $O(lg\, n)$, la procedura INTERVAL-SEARCH impiega il tempo $O(lg\, n)$.

12.3.2 Correttezza di Interval search

Per spiegare perché INTERVAL-SEARCH è corretta, si parte dall'idea che sarà sufficiente controllare solo uno dei due figli del nodo.

Teorema 12.2. *La procedura* INTERVAL-SEARCH(T, i) *restituisce un nodo il cui intervallo si sovrappone a i oppure restituisce* T.NIL *se l'albero T non contiene alcun nodo il cui intervallo si sovrappone a i. In altri termini:*

- *Se la ricerca va a destra, allora:*
  - *C'è una sovrapposizione nel sottoalbero destro*

    Oppure
  - *Non c'è sovrapposizione in nessuno dei due sottoalberi*
- *Se la ricerca va a sinistra, allora:*
  - *C'è una sovrapposizione nel sottoalbero sinistro*

    Oppure
  - *Non c'è sovrapposizione in nessuno dei due sottoalberi*

Se non c'è sovrapposizione in uno dei due, allora non c'è in nessuno dei due.

Dimostrazione. Se viene eseguita la riga 5, allora per la condizione di diramazione nella riga 3, la ricerca va a destra e si ha $x.left = T.\text{NIL}$ oppure $x.left.max < i.low$; ciò comporta che il sottoalbero con radice in $x.left$ non contiene un intervallo che si sovrappone a i. Quindi, non si ha sovrapposizione a sinistra. Se c'è una sovrapposizione nel sottoalbero destro non ci sono problemi, altrimenti si deve mostrare che non c'è sovrapposizione a sinistra. Se si suppone $x.left \neq T.\text{NIL}$ e $x.left.max < i.low$, per ogni intervallo j nel sottoalbero sinistra di x si ha:

$$\begin{aligned} j.high &\leq x.left.max \\ &< i.low \end{aligned}$$

Per la tricotomia degli intervalli, i e j non si sovrappongono. Allora il sottoalbero sinistro di x non contiene intervalli che si sovrappongono a i.

Se, invece, viene eseguita la riga 4, allora per la condizione di diramazione nella riga 3, la ricerca va a sinistra e si ha $i.low \leq x.left.max = j.high$ per qualche j nel sottoalbero sinistro. Inoltre, per la definizione dell'attributo max, ci deve essere un intervallo j nel sottoalbero sinistro di x tale che

$$\begin{aligned} j.high &= x.left.max \\ &\geq i.low \end{aligned}$$

Poiché i e j non si sovrappongono e poiché non è vero che $j.high < i.low$, allora per la tricotomia degli intervalli si ha $i.high < j.low$. Gli alberi di intervalli usano come chiavi gli estremi inferiori degli intervalli, quindi la proprietà dell'albero di ricerca implica che, per qualsiasi intervallo k nel sottoalbero destro di x si ha:

$$\underbrace{j.low}_{sinistra} \leq \underbrace{k.low}_{destra}$$

E dunque:

$$\begin{aligned} i.high &< j.low \\ &\leq k.low \end{aligned}$$

Per la tricotomia degli intervalli, i e k non si sovrappongono.
Dunque, si è dimostrato che la procedura INTERVAL-SEARCH funziona correttamente. □

Capitolo 13

Programmazione dinamica

La programmazione dinamica, come il metodo divide et impera, risolve i problemi combinando le soluzioni dei sottoproblemi (con il termine programmazione si fa riferimento all'uso di una tecnica tabulare). Gli algoritmi divide et impera suddividono un problema in sottoproblemi indipendenti, risolvono in modo ricorsivo i sottoproblemi e in seguito combinano le loro soluzioni per risolvere il problema originale. La programmazione dinamica, invece, può essere applicata quando i sottoproblemi non sono indipendenti, ovvero quando i sottoproblemi hanno in comune dei sottosottoproblemi. In questo contesto, un algoritmo divide et impera svolge molto più lavoro del necessario, risolvendo ripetutamente i sottoproblemi comuni. Un algoritmo di programmazione dinamica risolve ciascun sottoproblema una sola volta e salva la sua soluzione in una tabella, evitando così il lavoro di dover ricalcolare la soluzione ogni volta che si presenta lo stesso sottoproblema.

La programmazione dinamica si applica tipicamente ai **problemi di ottimizzazione**, quali problemi di massimizzazione o minimizzazione. Per questi problemi ci possono essere molte soluzioni possibili. Ogni soluzione ha un valore e si vuole trovare una soluzione con il valore ottimo. Si dice *una* soluzione ottima del problema e non *la* soluzione ottima, perché ci possono essere più soluzioni che raggiungono il valore ottimo.

Il processo di sviluppo di un algoritmo di programmazione dinamica può essere suddiviso in una sequenza di quattro fasi:

1. Caratterizzare la struttura di una soluzione ottima
2. Definire ricorsivamente il valore di una soluzione ottima del problema
3. Calcolare il valore di una soluzione ottima, di solito con uno schema bottom-up. Dal basso verso l'altro, ovvero, da sottoproblemi semplici a quelli più difficili, fino al problema originario.
4. Costruire una soluzione ottima dalle informazioni calcolate

Le prime tre fasi formano la base per risolvere un problema applicando la programmazione dinamica. La fase 4 può essere omessa se è richiesto soltanto il valore di una soluzione ottima. Quando si deve eseguire tale fase, a volte si memorizzano delle informazioni aggiuntive durante il calcolo della fase 3 per semplificare la costruzione di una soluzione ottima.

13.1 Taglio delle aste

Il **problema del taglio delle aste** è un esempio di programmazione dinamica. Data un'asta di lunghezza n e una tabella dei prezzi p_i, per $i = 1, 2, 3, \ldots, n$, si vuole determinare il ricavo massimo r_n che si può ottenere tagliando l'asta e vendendone i pezzi. Si noti che, se il prezzo p_n di un'asta di lunghezza n è sufficientemente grande, la soluzione ottima potrebbe essere quella di non effettuare alcun taglio.

lunghezza i	1	2	3	4	5	6	7	8	9	10
prezzo p_i	1	5	8	9	10	17	17	20	24	30

Considerando il caso in cui la lunghezza dell'asta da tagliare è $n = 4$, si hanno otto possibili modi di taglio:

	1	2	3	4	5	6	7	8
modo di taglio	4	1-3	1-1-2	1-1-1-1	2-2	1-2-1	3-1	2-1-1
ricavo	9	9	7	4	10	7	9	7

Si nota che tagliare un'asta di lunghezza 4 in due pezzi di lunghezza 2 fornisce un ricavo di $p_2 + p_2 = 5 + 5 = 10$, che è la soluzione ottima.

Un'asta di lunghezza n può essere tagliata in 2^{n-1} modi differenti, in quanto si ha un'opzione indipendente di tagliare, o non tagliare, alla distanza i dall'estremità sinistra (ovvero si può scegliere se tagliare, o non tagliare al i-esimo punto di taglio), per $i = 1, 2, 3, \ldots, n - 1$. Si denota una decomposizione in pezzi utilizzando la normale notazione additiva, cosicché $7 = 2 + 3 + 3$ indica che un'asta di lunghezza 7 viene tagliata in tre pezzi - due di lunghezza 2 e uno di lunghezza 3. Se una soluzione ottima prevede il taglio dell'asta in k pezzi, per $1 \leq k \leq n$, allora una decomposizione ottima

$$n = i_1 + 1_2 + \ldots + i_k$$

dell'asta in pezzi di lunghezze $i_1, i_2, \ldots, i_k$ fornisce il ricavo massimo corrispondente

$$r_n = p_{i_1} + p_{i_2} + \ldots + p_{i_k}$$

Per il problema in esame, si può determinare per ispezione i ricavi ottimi r_i, per $i = 1, 2, 3, \ldots, 10$ e le corrispondenti decomposizioni ottime. Si avrà dunque che è il ricavo massimo r_n è:

$r_4 = 10$ dalla soluzione $4 = 2 + 2$ perché associato al taglio ottimo [2,2]

Più in generale, si possono esprimere i valori dei ricavi massimi r_n per $r \geq 1$ in funzione dei ricavi ottimi r_i (con $i < n$) delle aste più corte. Per ogni taglio i si ha:

$$r_n = r_i + r_{n-i}$$

Da cui

$$r_n = max\,(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1)$$

Il primo argomento p_n, corrisponde alla vendita dell'asta di lunghezza n senza tagli. Gli altri $n - 1$ argomenti corrispondono al ricavo massimo ottenuto facendo un taglio iniziale dell'asta in due pezzi di dimensione i e n - i, per $i = 1, 2, 3, \ldots, n - 1$, e poi tagliando in modo ottimale gli ulteriori pezzi, ottenendo i ricavi r_i e r_{n-i} da questi pezzi.

Poiché non si conosce a priori quale valore di i ottimizza i ricavi, si devono considerare tutti i possibili valori di i e selezionare quello che massimizza i ricavi. Si potrebbe anche non scegliere alcun valore di i, se si può ottenere il massimo ricavo vendendo le aste senza tagliarle.

Per risolvere il problema originale di dimensione n, si risolvono problemi più piccoli dello stesso tipo, ma di dimensioni inferiori. Una volta effettuato il primo taglio, si possono considerare i due pezzi come istanze indipendenti del problema del taglio delle aste. L'ottimo è la somma dei ricavi ottimi delle due semiaste.

Teorema 13.1. *La soluzione ottima complessiva incorpora le soluzioni ottime dei due sottoproblemi correlati che massimizzano i ricavi di ciascuno dei due pezzi.*

Dimostrazione. Si procede per assurdo: si suppone che r_i (o r_{n-i}) non sia l'ottimo del sottoproblema, si sostituisce a r_i la soluzione ottima $r'_i > r_i$ ottenendo $r'_n > r_n$ ma ciò comporta che r_n non sarebbe un ottimo. □

Si dirà dunque, che il problema del taglio delle aste presenta una **sottostruttura ottima**: la soluzioni ottime di un problema incorporano le soluzioni ottime dei sottoproblemi correlati, che possono essere risolti in modo indipendente.

È possibile definire in modo più semplice una struttura ricorsiva per il problema del taglio delle aste, ovvero definire r_n secondo una formulazione più semplice: si considera la decomposizione formata da una primo pezzo di lunghezza i tagliato dall'estremità sinistra e dal pezzo restante di destra di lunghezza $n - i$. Soltanto il pezzo restante di destra potrà essere ulteriormente tagliato.

È possibile vedere ciascuna decomposizione di un'asta di lunghezza n in questo modo: un primo pezzo seguito da un'eventuale decomposizione del pezzo restante. Così facendo, si può esprimere la soluzione senza alcun taglio dicendo che il primo pezzo ha dimensione $i = n$ e ricavo p_n e che al pezzo rimanente di dimensione $n - i$ è associato il taglio ottimo dell'asta rimanente r_{n-i}, per cui si ottiene:

$$r_n = p_i + r_{n-i}$$

Nella formulazione precedente si richiamava ricorsivamente la funzione due volte calcolando r_i e r_{n-i}, adesso sono una volta per trovare r_{n-i}. Vale anche nel caso in cui non si taglia l'asta:

$$r_n = p_n + r_0$$

ovvero quando il pezzo restante ha dimensione 0, con ricavo $r_0 = 0$. Si ottiene quindi la versione semplificata della precedente equazione che definiva r_n:

$$r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i})$$

Secondo questa formulazione, una soluzione ottima incorpora la soluzione di un solo sottoproblema - il pezzo restante - anziché due come avveniva nella formulazione precedente.

Da questa nuova formulazione si ricava facilmente l'algoritmo ricorsivo top-down chiamato CUT-ROD che riceve in ingresso la tabella dei prezzi p e la lunghezza dell'asta n.

```
CUT-ROD(p, n)
1   if n = 0
2       return 0
3   q ← −∞
4   for i ← 1 to n
5       q ← max(q, p[i] + CUT-ROD(p, n - i))
6   return q
```

Commenti:

3 Sentinella per trovare il massimo

5 Implementa la nuova formulazione $r_n = \max_{1 \leq i \leq n}(p_i + r_{n-i})$

La procedura CUT-ROD riceve come input un array $p[1 \ldots n]$ di prezzi e un intero n, e restituisce il massimo ricavo possibile per un'asta di lunghezza n. Se $n = 0$, nessun ricavo è possibile; quindi CUT-ROD restituisce 0 nella riga 2. La riga 3 inizializza il ricavo massimo q a $-\infty$, in modo che il ciclo **for**, righe 4 - 5, calcola correttamente $q = \max_{1 \leq i \leq n}(p_i + \text{CUT-ROD}(p, n - i))$; la riga 6 poi restituisce questo valore. Tale valore q è uguale alla risposta desiderata r_n.

Ogni volta che si incrementa n di 1, il tempo di esecuzione della procedura quasi raddoppia.

L'inefficienza di CUT-ROD è dovuta al fatto che chiama più e più volte sé stessa in modo ricorsivo con gli stessi valori dei parametri; risolve ripetutamente gli stessi problemi.

Per analizzare il tempo di esecuzione di CUT-ROD, si indica con $T(n)$ il numero totale di chiamate di CUT-ROD quando la chiamata viene effettuata con il secondo parametro uguale a n. Questa espressione è uguale al numero di nodi in un sottoalbero la cui radice ha l'etichetta n nell'albero di ricorsione. Il conteggio include la chiamata iniziale alla radice. Quindi, $T(0) = 1$ e

$$T(n) = 1 + \sum_{j=0}^{n-1} T(j)$$

Il valore iniziale 1 riguarda la chiamata della radice, e il termine $T(j)$ conta il numero di chiamate (incluse le chiamate ricorsive) dovute alla chiamata di CUT-ROD$(p, n - i)$, dove $j = n - i$. Si può vedere che $T(n) = \Theta(2^n)$ e quindi il tempo di esecuzione di CUT-ROD è esponenziale a n.

Questo tempo di esecuzione esponenziale è dovuto al fatto che la procedura CUT-ROD considera esplicitamente tutti i 2^{n-1} modi possibili di tagliare un'asta di lunghezza n. L'albero delle chiamate ricorsive ha 2^{n-1} foglie, una per ogni modo possibile di tagliare l'asta. Le etichette nel cammino semplice dalla radice a una foglia forniscono le dimensioni di ciascun pezzo destro prima di effettuare un taglio; ovvero le etichette forniscono i corrispondenti punti di taglio, misurati dall'estremità destra dell'asta.

13.1.1 Programmazione dinamica per il taglio delle aste

Avendo visto che una semplice soluzione ricorsiva non è efficiente perché risolve ripetutamente gli stessi sottoproblemi, applicando la programmazione dinamica si farà in modo che ogni sottoproblema sia risolto una volta soltanto, salvando

la sua soluzione. Se si avrà bisogno di nuovo della soluzione di questo sottoproblema, si potrà riaverla immediatamente, senza bisogno di ricalcolarla. La programmazione dinamica richiede una memoria aggiuntiva extra per ridurre il tempo di esecuzione; si tratta di un tipico esempio di **compromesso tempo-memoria**. Il risparmio di tempo ottenibile può essere notevole: una soluzione con tempo esponenziale può essere trasformata in una soluzione con tempo polinomiale. Un metodo di programmazione dinamica viene eseguito in tempo polinomiale quando il numero di sottoproblemi distinti richiesti è polinomiale nella dimensione dell'input e ciascun sottoproblema può essere risolto in un tempo polinomiale.

Ci sono due modi equivalenti per implementare la programmazione dinamica:

Metodo top-down con annotazione In questo approccio, si scrive la procedura ricorsiva in modo naturale, modificandola per salvare il risultato di ciascun sottoproblema (di solito in un array o in una tabella). La procedura prima verifica se ha risolto precedentemente questo sottoproblema. In caso affermativo, restituisce il valore salvato, risparmiando gli ulteriori calcoli a quel livello; altrimenti la procedura calcola il valore nel modo usuale. Quindi, risolve il problema di dimensione n, e ricorsivamente risolve i sottoproblemi più piccoli, ma prima di lanciare la ricorsione sul sottoproblema più piccolo, controlla nella tabella se non è già disponibile la soluzione.

Metodo bottom-up La risoluzione di un particolare sottoproblema dipende soltanto dalla risoluzione di sottoproblemi più piccoli. Si ordinano i problemi per dimensione e poi si risolvono ordinatamente a partire dal più piccolo. Quando si risolve un particolare sottoproblema, tutti i sottoproblemi più piccoli da cui dipende la sua soluzione sono già stati risolti, e le loro soluzioni sono state salvate. Ogni sottoproblema viene risolto una sola volta e, quando lo si incontra, sono già stati risolti tutti i suoi sottoproblemi.

Questi due approcci generano algoritmi con lo stesso tempo di esecuzione asintotico. L'approccio top-down a volte non esamina ricorsivamente tutti i possibili sottoproblemi. L'approccio bottom-up spesso ha fattori costanti molto migliori, in quanto ha meno costi per le chiamate di procedura.

La versione **top-down** di CUT-ROD si basa sull'annotazione (**memoization**) delle soluzioni dei sottoproblemi già risolti:

```
MEMOIZED-CUT-ROD(p, n)
1  Sia r[0 ... n] un nuovo array
2  for i ← 0 to n
3      r[i] ← −∞
4  return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

La procedura principale MEMOIZED-CUT-ROD inizializza un nuovo array ausiliario $r[0 \ldots n]$ con il valore $-\infty$, una scelta comoda per indicare i valori incogniti (i valori dei ricavi noti sono sempre non negativi). Poi, alla riga 4, chiama la sua routine ausiliaria MEMOIZED-CUT-ROD-AUX. Tale procedura ausiliaria non è altro che la versione dotata di tabella, nella quale memorizzare le soluzioni dei

sottoproblemi, della precedente procedura CUT-ROD.

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
1   if r[n] ≥ 0
2       return r[n]
3   if n = 0
4       q ← 0
5   else q ← −∞
6       for i ← 1 to n
7           q ← max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n - i, r))
8   r[n] ← q
9   return q
```

Commenti:

0 r indica i ricavi massimi ottenuti fino ad un certo punto

1-2 Se si sta considerando un sottoproblema di cui è già stata calcolata la soluzione, tale soluzione viene restituita direttamente nella riga 2

La riga 1 controlla innanzitutto se il valore desiderato è già noto; in questo caso, la riga 2 restituisce tale valore. Altrimenti, le righe 3 - 7 calcolano il valore desiderato q chiamando ricorsivamente MEMOIZED-CUT-ROD-AUX, la riga 8 lo salva in $r[n]$ e la riga 9 lo restituisce.

La versione **bottom-up** non è ricorsiva e per questo motivo non è più possibile ottenere l'albero di ricorsione che rappresenta il suo andamento. Ha il seguente pseudocodice:

```
BOTTOM-UP-CUT-ROD(p, n)
1   Sia r[0 ... n] un nuovo array
2   r[0] ← 0
3   for j ← 1 to n
4       q ← −∞
5       for i ← 1 to j
6           q ← max(q, p[i] + r[j - i])
7       r[j] ← q
8   return r[n]
```

Commenti:

6 $p[i]$ e $r[j - i]$ sono già stati calcolati precedentemente

Per l'approccio bottom-up della programmazione dinamica, BOTTOM-UP-CUT-ROD segue l'ordine naturale dei sottoproblemi: un problema di dimensione i è più piccolo di un sottoproblema di dimensione j, se $i < j$. Quindi, la procedura risolve i sottoproblemi di dimensioni $j = 0, 1, \ldots, n$, in quest'ordine.

La riga 1 crea un nuovo array $r[0 \ldots n]$ in cui salvare i risultati dei sottoproblemi; la riga 2 inizializza $r[0]$ a 0, in quanto un'asta di lunghezza 0 non genera alcun ricavo. Le righe 3 - 6 risolvono ciascun sottoproblema di dimensione j, per $j = 1, \ldots, n$, nell'ordine delle dimensioni crescenti. La riga 6 fa riferimento direttamente all'elemento $r[j - i]$ dell'array, anziché fare una chiamata ricorsiva

per risolvere il sottoproblema di dimensione $j - i$. La riga 7 salva in $r[j]$ la soluzione del sottoproblema di dimensione j. Infine, la riga 8 restituisce $r[n]$ che è uguale al valore ottimo r_n.

Le versioni bottom-up e top-down hanno lo stesso tempo di esecuzione asintotico. Il tempo di esecuzione della procedura BOTTOM-UP-CUT-ROD è $\Theta(n^2)$, a causa della doppia struttura annidata del ciclo. Il numero di iterazioni del suo ciclo più interno **for**, nelle righe 5 - 6, forma una seria aritmetica. Anche il tempo di esecuzione della versione top-down, MEMOIZED-CUT-ROD, è $\Theta(n^2)$. Poiché una chiamata ricorsiva per risolvere un sottoproblema precedentemente risolto termina immediatamente, MOMIZED-CUT-ROD risolve ciascun sottoproblema una sola volta. La procedura risolve i sottoproblemi di dimensione 0, 1, ..., n. Per risolvere un sottoproblema di dimensione n, il ciclo **for**, righe 6 - 7, effettua n iterazioni. Quindi, il numero totale di iterazioni di questo ciclo **for**, per tutte le chiamate ricorsive di MEMOIZED-CUT-ROD, forma una serie aritmetica, per un totale di $\Theta(n^2)$ iterazioni, come il ciclo interno **for** di BOTTOM-UP-CUT-ROD.

13.1.2 Ricostruire una soluzione

La soluzione ottenuta con la programmazione dinamica del problema del taglio delle aste fornisce il valore di una soluzione ottima (massimo ricavo), non la soluzione effettiva (il modo di taglio per ottenere il massimo ricavo): una lista di dimensioni dei pezzi. È possibile estendere l'approccio della programmazione dinamica per memorizzare non soltanto il valore ottimo calcolato per ciascun sottoproblema, ma anche una scelta che determina il valore ottimo. Con questa informazione, si è in grado di stampare facilmente una soluzione ottima.

Si definisce una versione estesa di BOTTOM-UP-CUT-ROD che calcola, per ogni dimensione j dell'asta, non soltanto il ricavo massimo r_j, ma anche s_j, la dimensione ottima del primo pezzo da tagliare.

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
1   Siano r[0 ... n] e s[0 ... n] due nuovi array
2   r[0] ← 0
3   for j ← 1 to n
4       q ← −∞
5       for i ← 1 to j
6           if q < p[i] + r[j - i]
7               q ← p[i] + r[j - i]
8               s[j] ← i
9       r[j] ← q
10  return (r, s)
```

Questa procedura è simile a BOTTOM-UP-CUT-ROD, con la differenza che crea l'array s nella riga 1, e aggiorna $s[j]$ nella riga 8 per conservare la dimensione ottima i del primo pezzo da tagliare quando viene risolto un sottoproblema di dimensione j.

La procedura PRINT-CUT-ROD-SOLUTION riceve una tabella di prezzi p e una dimensione n dell'asta; poi chiama EXTENDED-BOTTOM-UP-CUT-ROD per calcolare l'array $s[1 ... n]$ delle dimensioni ottime dei primi pezzi e stampa la lista

completa delle dimensioni dei pezzi per una decomposizione ottima di un'asta di lunghezza n.

```
PRINT-CUT-ROD-SOLUTION(p, n)
1   (r, s) ← EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
2   while n > 0
3       print s[n]
4       n ← n - s[n]
```

Commenti:

4 Trova il massimo nel pezzo rimanente

Riprendendo la tabella

lunghezza i	1	2	3	4	5	6	7	8	9	10
prezzo p_i	1	5	8	9	10	17	17	20	24	30

Un esempio di esecuzione dell'algoritmo EXTENDED-BOTTOM-UP-CUT-ROD e della procedura PRINT-CUT-ROD è il seguente:

i	0	1	2	3	4	5	6	7	8	9	10
p_i	0	1	5	8	9	10	17	17	20	24	30
r_i	0	1	5	8	10	13	17	18	22	25	30
s_i	0	1	2	3	2	2	6	1	2	3	10

13.2 Longest Common Subsequence

Nel problema della **più lunga sottosequenza comune** sono date due sequenze $X = \langle x_1, \ldots, x_m \rangle$ e $Y = \langle y_1, \ldots, y_n \rangle$ e si vuole trovare una sottosequenza di lunghezza massima che è comune a X e Y. Non è detto che le sequenze siano delle stringhe.

Una sottosequenza di una data sequenza è la sequenza stessa alla quale sono stati tolti zero o più elementi. Formalmente, sia $X = \langle x_1, \ldots, x_m \rangle$ una sequenza, un'altra sequenza $Z = \langle z_1, z_2, \ldots, z_k \rangle$ è una **sottosequenza** di X se esiste una sequenza strettamente crescente $I = \langle i_1, \ldots, i_k \rangle$ di indici di X tale che $i_j > i_h$ se $j > h$ per ogni $j, h = 1, 2, \ldots, k$ e $x_{i_j} = z_j$ per ogni $j = 1, 2, \ldots, k$. Per esempio, $Z = \langle B, C, D, B \rangle$ è una sottosequenza di $X = \langle A, B, C, B, D, A, B \rangle$ con la corrispondente sequenza di indici $I = \langle 2, 3, 5, 7 \rangle$.

Date due sequenze X e Y, si dirà che una sequenza Z è una **sottosequenza comune** di X e Y se Z è una sottosequenza di entrambe le sequenze X e Y. Per esempio, se $X = \langle A, B, C, B, D, A, B \rangle$ e $Y = \langle B, D, C, A, B, A \rangle$, la sequenza $\langle B, C, A \rangle$ è una sottosequenza comune di X e Y e la sequenza $\langle B, C, B, A \rangle$ è una LCS di X e Y.

Un algoritmo a forza bruta per risolvere il problema della più lunga sottosequenza comune consiste nell'enumerare tutte le sottosequenze di X e controllare le singole sottosequenze per vedere se sono anche sottosequenze di Y, tenendo traccia della più lunga sottosequenza trovata. Ogni sottosequenza di X corrisponde a un sottoinsieme degli indici $\{1, 2, \ldots, m\}$ di X. Ci sono 2^m sottosequenze di X da controllare, il tempo necessario per controllare ogni sottosequenza è $\Theta(m)$ quindi questo approccio richiede un tempo esponenziale $\Theta(n \cdot 2^m)$, il che lo rende inutilizzabile per le sequenze lunghe.

Data una sequenza $X = \langle x_1, \ldots, x_m \rangle$, si definisce $X_i = \langle x_1, \ldots, x_i \rangle$ l'i-esimo **prefisso** di X, per $i = 0, 1, \ldots, m$. Per esempio, se $X = \langle A, B, C, D \rangle$ si ha $X_1 = \langle A \rangle$, $X_3 = \langle A, B, C \rangle$ mentre X_0 è la sequenza vuota.

Teorema 13.2. *Siano $X = \langle x_1, \ldots, x_m \rangle$ e $Y = \langle y_1, \ldots, y_n \rangle$ due sequenze; sia $Z = \langle z_1, \ldots, z_k \rangle$ una qualsiasi LCS di X e Y, allora:*

1. *Se $x_m = y_n$, allora $z_k = x_m = y_n$ e Z_{k-1} è una LCS di X_{m-1} e Y_{n-1}*
2. *Se $x_m \neq y_n$, allora $z_k \neq x_m$ implica che Z è una LCS di X_{m-1} e Y*
3. *Se $x_m \neq y_n$, allora $z_k \neq y_n$ implica che Z é una LCS di X e Y_{n-1}*

Dimostrazione. Si dimostrano i tre punti separatamente:

1. Se $x_m = y_n$, supponendo $z_k \neq x_m$, si costruisce una nuova sequenza $Z' = \langle z_1, z_2, \ldots, z_k, x_m \rangle$ (ottenuta accodando $x_m = y_n$ a Z) che risulta essere una sottosequenza comune di X e Y di lunghezza $k + 1$; ma Z' è una sottosequenza comune più lunga di Z quindi si contraddice l'ipotesi che Z sia una LCS di X e Y. Quindi, deve essere $z_k = x_m = y_n$. Ora, il prefisso Z_{k-1} è sicuramente una sottosequenza comune di X_{m-1} e Y_{n-1}, di lunghezza $k - 1$. Si vuole dimostrare che questo prefisso è una LCS. Si suppone per assurdo che esista una sottosequenza comune W di X_{m-1} e Y_{n-1} che sia più lunga di Z_{k-1}, cioè di lunghezza maggiore di $k - 1$. Allora, si costruisce una nuova sequenza W' accodando $x_m = y_n$ a W, e si ottiene una sottosequenza comune di X e Y la cui lunghezza è maggiore di k; ma questo contraddice il fatto che Z sia una LCS di X e Y.

2. Se $z_k \neq x_m$ allora Z è una sottosequenza comune di X_{m-1} e Y. Se esistesse una sottosequenza comune W di X_{m-1} e Y di lunghezza maggiore di k, allora W sarebbe anche una sottosequenza comune di X_m e Y, contraddicendo l'ipotesi che Z sia una LCS di X e Y.

3. La dimostrazione è simmetrica a quella del punto 2.

□

Questo teorema dimostra che una LCS di due sequenze contiene al suo interno (come prefisso) una LCS di prefissi delle due sequenze. Quindi, il problema della più lunga sottosequenza comune gode della proprietà della **sottostruttura ottima**.

Una soluzione ricorsiva gode anche della proprietà dei **sottoproblemi ripetuti**.

Il teorema implica che ci sono uno o due sottoproblemi da esaminare per trovare una LCS di X e Y.

- Se $x_m = y_n$ si deve risolvere un solo sottoproblema. Si trova una LCS di X_{m-1} e Y_{n-1}. Una volta trovata, accodando $x_m = y_n$ a questa LCS, si ottiene una LCS di X e Y.

- Se $x_m \neq y_n$ si devono risolvere due sottoproblemi. Si deve trovare una LCS di X_{m-1} e Y e una LCS di X e Y_{n-1}. La più lunga di queste due LCS è una LCS di X e Y.

La soluzione ricorsiva del problema della più lunga sottosequenza comune richiede la definizione di una ricorrenza per il valore di una soluzione ottima. Si definisce $c[i, j]$ come la lunghezza di una LCS delle sequenze X_i e Y_j. Se $i = 0$ o $j = 0$, una delle sequenze ha lunghezza 0, quindi la LCS ha lunghezza 0. La sottostruttura ottima del problema della LCS consente di scrivere la formula ricorsiva

$$c[i,\, j] = \begin{cases} 0 & \text{se } i = 0 \text{ o } j = 0 \\ c[i-1,\, j-1] + 1 & \text{se } i,\, j > 0 \text{ e } x_i = y_j \\ \max(c[i,\, j-1],\, c[i-1,\, j]) & \text{se } i,\, j > 0 \text{ e } x_i \neq y_j \end{cases}$$

Secondo questa formulazione ricorsiva, una condizione del problema riduce il numero di sottoproblemi che si possono considerare. Quando $x_i = y_j$, si deve considerare soltanto il sottoproblema di trovare la LCS di X_{i-1} e Y_{j-1}. Altrimenti, quando $x_i \neq y_j$, si devono risolvere solo due sottoproblemi: trovare la LCS di X_i e Y_{j-1} e trovare la LCS di X_{i-1} e Y_j. Per il teorema non è stata esclusa nessuna soluzione.

Utilizzando la formula ricorsiva appena definita è possibile scrivere un algoritmo ricorsivo per calcolare la lunghezza di una LCS di due sequenze. La procedura LCS-Length riceve come input due sequenze X e Y, di dimensioni m e n rispettivamente, e memorizza i valori $c[i, j]$ in una tabella $c[0 \ldots m, 0 \ldots n]$. La tabella $c[i, j]$ contiene la lunghezza ottima di LCS per X_i e Y_j. Si usa anche la tabella $b[1 \ldots m, 1 \ldots n]$ per semplificare la costruzione di una soluzione ottima. L'elemento $b[i, j]$ punta alla posizione della tabella che corrisponde alla soluzione ottima del sottoproblema che è stata scelta per calcolare $c[i, j]$; ovvero, $b[i, j]$ indica la strada seguita per risolvere LCS di X_i e Y_j.

```
LCS-Length(X, Y)
1   m ← X.length
2   n ← Y.length
3   Siano b[1 ... m, 1 ... n] e c[0 ... m, 0 ... n] due nuove tabelle
4   for i ← 1 to m
5       c[i, 0] ← 0
6   for j ← 0 to n
7       c[0, j] ← 0
8   for i ← 1 to m
9       for j ← 1 to n
10          if x_i = y_j
11              c[i, j] ← c[i - 1, j - 1] + 1
12              b[i, j] ← " ↖ "
13          elseif c[i − 1, j] ≥ c[i, j − 1]
14              c[i, j] ← c[i - 1, j]
15              b[i, j] ← " ↑ "
16          else c[i, j] ← c[i, j - 1]
17              b[i, j] ← " ← "
18  return c e b
```

Commenti:

12 Indica dove era il massimo precedente

Il tempo di esecuzione di LCS-Length è $\Theta(m \cdot n)$, perché il calcolo di ogni posizione della tabella richiede un tempo $\Theta(1)$.

Partendo da $b[m, n]$ e attraversando la tabella b seguendo le frecce, è possibile ottenere una LSC di X e Y. Ogni volta che si incontra una freccia "$\nwarrow$" in posizione $b[i, j]$, significa che $x_i = y_j$ è un elemento della LCS. In questo modo gli elementi della LCS si incontrano in ordine inverso. La procedura ricorsiva Print-LSC stampa una LSC di X e Y nell'ordine corretto.

```
Print-LSC(b, X, i, j)
1   if i = 0 or j = 0
2       return
3   if b[i, j] = "↖"
4       Print-LSC(b, X, i - 1, j - 1)
5       print x_i
6   elseif b[i, j] = " ↑ "
7       Print-LSC(b, X, i - 1, j)
8   else Print-LSC(b, X, i, j - 1)
```

Commenti:

3 e 6 Verificano la provenienza del massimo attuale

La chiamata iniziale è Print-LSC(b, X, *X.length*, *Y.length*), la procedura impiega un tempo $O(m+n)$, perché a ogni chiamata ricorsiva decrementa almeno uno dei valori i e j.

Esempio

		a	**m**	**p**	**u**	**t**	**a**	**t**	**i**	**o**	**n**
	0	← 0	← 0	0	0	0	0	0	0	0	0
			↑								
s	0	0	0	0	0	0	0	0	0	0	0
				↖							
p	0	0	0	**1**	← 1	← 1	1	1	1	1	1
							↖				
a	0	1	1	1	1	1	**2**	2	2	2	2
							↑				
n	0	1	1	1	1	1	2	2	2	2	2
							↑				
k	0	1	1	1	1	1	2	← 2	2	2	2
									↖		
i	0	1	1	1	1	1	2	2	**3**	← 3	3
											↖
n	0	1	1	1	1	1	2	2	3	3	**4**
											↑
g	0	1	1	1	1	1	2	2	3	3	4
				p			**a**		**i**		**n**

La più lunga sottosequenza comune delle parole *amputation* e *spanking* è *pain*. Il percorso che individua la LCS deve andare lungo la diagonale.

13.2.1 Miglioramenti

Nell'algoritmo della LCS si potrebbe eliminare completamente la tabella b. $c[i, j]$ dipende soltanto da altre tre posizioni della tabella c: $c[i - 1, j - 1]$, $c[i - 1, j]$ e $c[i, j - 1]$. Dato il valore di $c[i, j]$, si può determinare nel tempo $O(1)$ quale di questi tre valori è stato utilizzato per calcolare $c[i, j]$, senza ispezionare la tabella b. Quindi, si può ricostruire una LCS nel tempo $O(m + n)$ utilizzando una procedura simile a PRINT-LSC. Sebbene queste metodo permetta di risparmiare uno spazio $\Theta(m\,n)$ in memoria, tuttavia lo spazio ausiliario richiesto per calcolare una LCS non diminuisce asintoticamente, perché occorre comunque uno spazio $\Theta(m\,n)$ per la tabella c. È però possibile ridurre il fabbisogno asintotico di memoria per LCS-LENGTH, perché questa procedura usa soltanto due righe alla volta della tabella c: la riga da calcolare e la riga precedente (si potrebbe utilizzare uno spazio soltanto un pò più grande di quello richiesto da una riga di c per calcolare la lunghezza di una LCS). Questo miglioramento funziona se occorre calcolare soltanto la lunghezza di una LCS; se si vuole ricostruire gli elementi di una LCS, la tabella più piccola non può contenere le informazioni necessarie per rifare il percorso inverso nel tempo $O(m + n)$.

13.3 Edit distance

Date le due stringhe X e Y di dimensioni m e n, l'obiettivo è quello di effettuare una serie di trasformazioni che cambiano X in Y. Ci sono sei operazioni elementari di trasformazione:

1. Copia
2. Sostituzione
3. Cancellazione
4. Inserimento
5. Scambio
6. Distruzione

Date due sequenze X e Y e un insieme di costi delle operazioni di tarsformazione, la **distanza di editing** tra X e Y è il costo della sequenza di operazioni più economica che trasforma X in Y.

Ciascuna delle operazioni di trasformazione ha un costo associato. Il costo di un'operazione dipende dalla specifica applicazione ma si suppone che il costo di ciascuna operazione sia una costante nota. Si suppone inoltre che i singoli costi delle operazioni di copia e sostituzione siano minori dei costi combinati delle operazioni di cancellazione e inserimento. Il costo di una data sequenza di operazioni di trasformazione è la somma dei costi delle singole operazioni nella sequenza.

Come nel caso di LCS, date due sequenze $X = \langle x_1, \ldots, x_m \rangle$ e $Y = \langle y_1, \ldots, y_n \rangle$, si definiscono i prefissi $X_i = \langle x_1, \ldots, x_i \rangle$ e $Y_j = \langle y_1, \ldots, y_j \rangle$ con $0 \leq i \leq m$ e $0 \leq j \leq n$. L'obiettivo in ciascun sottoproblema sarà di trovare una sequenza minima di operazioni elementari che converta X_i in Y_j. Si vuole quindi minimizzare il numero di operazioni che rappresenterà poi la distanza tra le due

stringhe. Secondo l'idea generale $c[i,j]$ è il costo di una soluzione ottima al problema $X_i \to Y_j$. Supponendo di conoscere l'ultima operazione usata per $X_i \to Y_j$ si può calcolare $c[i,j]$ in funzione dei valori precedenti.

Se l'ultima operazione era una **copia**: doveva essere $x_i = y_j$. Resta dunque da risolvere il problema X_{i-1} e Y_{j-1}. Per la proprietà di sottostruttura ottima, una soluzione ottima al problema $X_i \to Y_j$ deve includere una soluzione ottima a $X_{i-1} \to Y_{j-1}$. Quindi, supponendo che l'ultima operazione fosse una copia

$$c[i,\, j] = c[i-1,\, j-1] + \text{costo(copia)}$$

Se era una **sostituzione**: doveva essere $x_i \neq y_j$. Come nel caso della copia, per la proprietà di sottostruttura ottima, una soluzione ottima al problema $X_i \to Y_j$ deve includere una soluzione ottima a $X_{i-1} \to Y_{j-1}$. Supponendo che l'ultima operazione fosse una sostituzione

$$c[i,\, j] = c[i-1,\, j-1] + \text{costo(sostituzione)}$$

Se era uno **scambio**: doveva essere $x_i = y_{j-1}$ e $x_{i-1} = y_j$, con $i, j \geq 2$. In questo caso, una soluzione ottima al problema $X_i \to Y_j$ deve includere una soluzione ottima a $X_{i-2} \to Y_{j-2}$. Supponendo che l'ultima operazione fosse uno scambio

$$c[i,\, j] = c[i-2,\, j-2] + \text{costo(scambio)}$$

Se era una **cancellazione** non si hanno restrizioni su X e Y. La cancellazione comporta la rimozione di un carattere da X_i lasciando Y_j invariato. Una soluzione ottima al problema $X_i \to Y_j$ deve includere una soluzione ottima a $X_{i-1} \to Y_j$. Supponendo che l'ultima operazione fosse una cancellazione

$$c[i,\, j] = c[i-1,\, j] + \text{costo(cancellazione)}$$

Se era un **inserimento** non si hanno restrizioni su X e Y. L'inserimento comporta la rimozione di un carattere da Y_j lasciando X_i invariato. Una soluzione ottima al problema $X_i \to Y_j$ deve includere una soluzione ottima a $X_i \to Y_{j-1}$. Supponendo che l'ultima operazione fosse un in inserimento

$$c[i,\, j] = c[i,\, j-1] + \text{costo(inserimento)}$$

Per quanto riguarda i **casi base**, se $i = 0$ o $j = 0$, X_0 e Y_0 sono stringhe vuote. Si converte la stringa vuota in Y_j con j inserimenti

$$c[0,\, j] = j \cdot \text{costo(inserimento)}$$

Si converte la stringa X_i in Y_0 con i cancellazioni

$$c[i,\, 0] = i \cdot \text{costo(cancellazione)}$$

Con $i = j = 0$ si ha $c[0,\, 0] = 0$, non c'è costo per convertire la stringa vuota nella stringa vuota. Si arriva così a definire la formulazione ricorsiva per il calcolo della distanza di editing tra due sequenze X e Y. Applicando le formule precedenti si definirà $c[i,j]$, per $i, j > 0$, come:

$$c[i,\, j] = \begin{cases} c[i-1,\, j-1] + \text{costo(copia)} & \text{se } x[i] = y[j] \\ c[i-1,\, j-1] + \text{costo(sostituzione)} & \text{se } x[i] \neq y[j] \\ c[i,\, j] = c[i-2,\, j-2] + \text{costo(scambio)} & \text{se } i, j \geq 2, \\ & x[i] = y[j-1] \\ & \text{e } x[i-1] = y[j] \\ c[i,\, j] = c[i-1,\, j] + \text{costo(cancellazione)} & \text{sempre} \\ c[i,\, j] = c[i,\, j-1] + \text{costo(inserimento)} & \text{sempre} \end{cases}$$

Ogni volta che trova il minimo popola la tabella. Alla fine, l'ultimo valore nella tabella è la distanza di editing delle due sequenze.

L'algoritmo bottom-up per il calcolo della distanza di editing riempie la tabella per righe (funzionerebbe anche per colonne), $op[i,j]$ memorizza l'operazione usata. Esiste anche una versione top-bottom ma richiede MEMOIZATION. La procedura EDIT-DISTANCE riceve in ingresso le due sequenze X e Y di cui si vuole trovare la distanza di editing.

```
EDIT-DISTANCE(x,y)
1   m ← x.length
2   n ← y.length
3   Siano c[0 ... m, 0 ... n] e op[0 ... m, 0 ... n] due nuove tabelle
4   for i ← 0 to m
5       c[i, 0] ← i · cost(delete)
6       op[i, 0] ← DELETE
7   for j ← 0 to n
8       c[0, j] ← j · cost(insert)
9       op[0, j] ← INSERT
10   for i ← 0 to m
11       for j ← 0 to n
12           c[i, j] ← ∞
13           if x_i = y_j
14               c[i, j] ← c[i - 1, j - 1] + cost(copy)
15               op[i, j] ← COPY
16           if x_i ≠ y_j and c[i − 1, j − 1] + cost(replace) < c[i, j]
17               c[i, j] ← c[i - 1, j - 1] + cost(raplace)
18               op[i, j] ← REPLACE(y_j)
19           if i, j ≥ 2 and c[i - 2, j - 2] + cost(twiddle) < c[i, j]
                         and x_i = y_{j−1} and x_{i−1} = y_j
20               c[i, j] ← c[i - 2, j - 2] + cost(twiddle)
21               op[i, j] ← TWIDDLE
22           if c[i − 1, j] + cost(delete) < c[i, j]
23               c[i, j] ← c[i - 1, j] + cost(delete)
24               op[i, j] ← DELETE
25           if c[i, j − 1] + cost(insert) < c[i, j]
26               c[i, j] ← c[i, j - 1] + cost(insert)
27               op[i, j] ← INSERT(y_j)
28   return c e op
```

La procedura viene eseguita nel tempo $\Theta(m \cdot n)$ e richiede uno spazio $\Theta(m\,n)$. Per risparmiare spazio, basterebbe la riga $i-1$ e nel caso dello scambio anche la riga $i-2$; quindi si potrebbe sovrascrivere queste due righe e mettere come colonne la stringa più corta.

13.3.1 Ricostruire una soluzione

Per poter ricostruire la sequenza ottima di operazioni da eseguire per poter trasformare la sequenza X nella sequenza Y si usa la tabella op restituita dalla procedura EDIT-DISTANCE. La procedura OP-SEQUENCE è un algoritmo ricorsivo che nella prima chiamata riceve in ingresso la tabella op e le lunghezze m

ed n delle due sequenze X e Y. Nelle chiamate successive riceverà gli indici i e j per scorrere la matrice op.

```
OP-SEQUENCE(op, i, j)
1   if i = 0 and j = 0
2      return
3   if op[i, j] = COPY or op[i, j] = REPLACE
4      i' ← i - 1
5      j' ← j - 1
6   elseif op[i, j] = TWIDDLE
7      i' ← i - 2
8      j' ← j - 2
9   elseif op[i, j] = DELETE
10     i' ← i - 1
11     j' ← j
12  else // Deve essere op[i, j] = INSERT
13     i' ← i
14     j' ← j - 1
15  OP-SEQUENCE(op, i', j')
16  PRINT op[i, j]
```

13.3.2 Utilizzi di edit distance

L'edit distance viene principalmente utilizzato per:

Correzione ortografica Usata sia per correggere documenti con parole scritte in modo errato, sia per suggerire query all'utente. Vengono utilizzati due approcci principali:

- **Parole isolate** Si controlla ogni parola indipendentemente dalle altre ma non si trovano errori di parole scambiate
- **Context-sensitive** Si guarda il contesto della frase

Correzione di documenti Necessario per documenti letti con OCR. In questo caso si considerano errori specifici usando una conoscenza specifica del dominio (OCR può confondere O e D). Anche i documenti elettronici e pagine Web presentano errori dovuti, per esempio, all'adiacenza delle lettere nella tastiera QWERTY (per esempio si confondono O e I). Spesso non si modifica il documento ma si ottimizza il match query-documenti memorizzati

Correzione di parole isolate In questo caso si ipotizza che esista un lessico con lo spelling corretto di tutte le parole. Due possibili approcci:

- **Usare lessico Standard** Quindi Websters English Dictionary o un lessico specifico per documenti in domini specifici
- **Usare parole della collezione** Quindi utilizzare tutte le parole nel Web, tutti i nomi, acronimi ecc ma includendo anche gli errori

Dato un lessico L e la stringa Q, si vogliono trovare le parole in L più vicine a Q; si deve però dare una definizione di *più vicino*. Le alternative, oltre alla edit distance già vista, sono la Weighted edit distance e l' intersezione di n-gram.

La **Weighted edit distance** è una distanza di editing dove il costo di una operazione dipende dai caratteri coinvolti e non solo dal tipo di operazione (tiene conto dell'errore). Viene usata per gestire errori di OCR o di battitura in quanto il costo è proporzionale alla distanza sulla tastiera. Per poterla utilizzare serve una matrice di pesi W che dipende dall'applicazione. La procedura WEIGHTED-EDDIT-DISTANCE è uguale a EDIT-DISTANCE con la differenza che se $x_i \neq y_j$ si dovrà leggere la tabella W e considerare il peso.

Confrontare Q con tutte le parole in L è troppo costoso, una possibilità è di elencare tutte le sequenze di caratteri da Q con edit distance inferiore ad una certa soglia, intersecare questo insieme con L e mostrare infine i risultati.

Un modo per ridurre l'insieme dei termini candidati nel dizionario è quello di utilizzare l'intersezione di n-gram. Con l'**intersezione di n-gram** si enumerano tutti gli n-gram (n caratteri consecutivi) in Q e in L. Si usa un indice di n-gram per trovare tutti i termini in L che contengono qualunque n-gram di Q oppure si cercano tutti i termini che contengono abbastanza n-gram di Q. Una sequenza di lunghezza ha l ha $l - n + 1$ n-gram.

Per poter avere una misura di sovrapposizione normalizzata si utilizza il **coefficiente di Jaccard** definito come:

$$JC = \frac{|X \cap Y|}{|X \cup Y|}$$

Dove X e Y sono due insiemi che non devono avere la stessa dimensione. Il coefficiente di Jaccard è definito nell'intervallo $0 \leq JC \leq 1$; vale 1 se X e Y hanno gli stessi elementi e 0 se sono disgiunti. Nel caso in esame X e Y sono insiemi di n-gram. Per trovare i termini si considera una soglia, se JC è maggiore di tale soglia si calcola l'edit distance.

Capitolo 14

Algoritmi golosi

Gli algoritmi per i problemi di ottimizzazione eseguono una sequenza di passi, con una serie di scelte a ogni passo. Per molti problemi di ottimizzazione è uno spreco applicare le tecniche della programmazione dinamica per effettuare le scelte migliori: è preferibile utilizzare algoritmi più semplici ed efficienti. Un **algoritmo goloso** (greedy) fa sempre la scelta che sembra ottima in un determinato momento, ovvero fa una scelta localmente ottima, nella speranza che tale scelta porterà a una soluzione globalmente ottima. Tuttavia, gli algoritmi golosi non sempre riescono a trovare le soluzioni ottime. Esempi di applicazione degli algoritmi golosi sono:

- **Problema della selezione di attività** Si suppone di avere un insieme di n attività $S = \{a_1, a_2, \ldots, a_n\}$ che devono utilizzare la stessa risorsa che può essere utilizzata per svolgere una sola attività alla volta. Ogni attività a_i ha un tempo di inizio s_i e di fine f_i, con $0 \leq s_i < f_i < \infty$. Quando viene selezionata, l'attività a_i si svolge durante l'intervallo semiaperto $[s_i, f_i)$. Le attività a_i e a_j sono compatibili se gli intervalli $[s_i, f_i)$ e $[s_j, f_j)$ non si sovrappongono, ovvero se $s_i \geq f_j$ o $s_j \geq f_i$. Il problema della selezione di attività consiste nel selezionare il sottoinsieme che contiene il maggior numero di attività mutualmente compatibili.

- **I codici di Huffman** Tecnica molto efficiente per comprimere i dati ottenendo risparmi dal 20% al 90% a seconda delle caratteristiche dei dati da comprimere. I dati vengono considerati come una sequenza di caratteri. L'algoritmo goloso di Huffman usa una tabella che contiene quante volte compare ciascun carattere (la sua frequenza) per realizzare un metodo ottimo per rappresentare ciascun carattere con una stringa binaria.

- **Problema di programmazione dei lavori** In particolare si considerano lavori di durata unitaria, ovvero lavori che richiedono una sola unità di tempo per essere completati. Dato un insieme finito S di lavori di durata unitaria, un piano di programmazione per l'insieme S è una permutazione di S che specifica l'ordine in cui questi lavori devono essere eseguiti. Il problema della programmazione dei lavori di durata unitaria con scadenza e penalità per un singolo processore ha tre input: un insieme $S = \{a_1, a_2, \ldots, a_n\}$ di n lavori di durata unitaria, un insieme di n interi che rappresentano le scadenze $d_1, d_2, \ldots, d_n$ per cui la generica scadenza

d_i è tale che $1 \leq d_i \leq n$ (si suppone inoltre che il generico lavoro a_i termini al tempo d_i) e un insieme di n pesi non negativi o penalità w_1, w_2, ..., w_n; si paga la penalità w_i se il lavoro a_i non termina entro il tempo d_i, mentre non c'è alcuna penalità se un lavoro termina entro la sua scadenza. Il problema è trovare un piano di programmazione per S che minimizza le penalità totali da pagare per le scadenze non rispettate.

- **Algoritmo di Dijkstra**
- **Algoritmi per gli alberi di connessione minimi**

Un algoritmo goloso produce una soluzione ottima di una problema effettuando una sequenza di scelte. Ad ogni punto di decisione l'algoritmo fa la scelta che in quel momento sembra la migliore. Questa strategia non sempre produce una soluzione ottima ma a volte ci riesce. Gli algoritmi golosi vengono progettati secondo una sequenza di tre passi

1. Esprimere il problema di ottimizzazione in una forma in cui, fatta una scelta, resta un solo sottoproblema da risolvere
2. Dimostrare che esiste sempre una soluzione ottima del problema originale che fa la scelta golosa, quindi la scelta golosa è sempre sicura
3. Dimostrare la sottostruttura ottima verificando che, dopo aver fatto la scelta golosa, ciò che resta è un sottoproblema con la proprietà che, se si combina una soluzione ottima del sottoproblema con la scelta golosa che è stata fatta, si arriva ad una soluzione ottima del problema originale

Sotto ogni algoritmo goloso c'è quasi sempre una soluzione più onerosa di programmazione dinamica.

Non c'è un modo generale per dire se un algoritmo goloso risolverà un particolare problema di ottimizzazione ma la proprietà della scelta golosa e la sottostruttura ottima sono i due punti chiave. Se si dimostra che un problema soddisfa queste proprietà, allora ci si trova sulla buona strada per sviluppare un algoritmo goloso per questo problema.

14.1 La proprietà della scelta golosa

Uno dei punti chiave nello sviluppo degli algoritmi golosi è la **proprietà della scelta golosa**: una soluzione globalmente ottima può essere ottenuta facendo una scelta localmente ottima. In altre parole, quando si valuta la scelta da fare, si fa la scelta che sembra migliore per il problema corrente, senza considerare le soluzioni di sottoproblemi. È qui che gli algoritmi golosi differiscono dalla programmazione dinamica. Nella programmazione dinamica, si fa una scelta a ogni passo, ma di solito la scelta dipende dalla soluzioni dei sottoproblemi. Di conseguenza, tipicamente si risolvono i problemi di programmazione dinamica secondo uno schema bottom-up, elaborando prima i sottoproblemi più piccoli per arrivare a quelli più grandi. In un algoritmo goloso, si fa la scelta che sembra migliore in un determinato momento e poi si risolve il sottoproblema che deriva da tale scelta. La scelta fatta da un algoritmo goloso può dipendere dalle precedenti scelte, ma non può dipendere dalle scelte future o dalle soluzioni dei sottoproblemi. Quindi, diversamente dalla programmazione dinamica, che

risolve i sottoproblemi prima di fare la prima scelta, un algoritmo goloso fa la sua prima scelta prima di risolvere qualsiasi sottoproblema. Un algoritmo di programmazione dinamica procede dal basso verso l'alto, mentre una strategia golosa di solito precede dall'alto verso il basso, facendo una scelta golosa dopo l'altra e riducendo ogni istanza del problema a una più piccola. Di solito la scelta golosa si effettua in modo più efficiente che se si dovesse prendere in considerazione un numero maggiore di scelte. Spesso, grazie a un'elaborazione preliminare dell'input o all'impiego di una struttura dati appropriata (spesso una coda di priorità), si riesce a fare rapidamente delle scelte golose, ottenendo così un algoritmo efficiente.

14.2 Sottostruttura ottima

Il secondo punto chiave nello sviluppo degli algoritmi golosi è la **sottostruttura ottima**. Un problema ha una sottostruttura ottima se una soluzione ottima del problema contiene al suo interno soluzioni ottime dei sottoproblemi.

Nel confronto tra programmazione dinamica e algoritmi golosi si verifica che, di solito, si utilizza un approccio più diretto alla sottostruttura ottima quando la si applica agli algoritmi golosi. Come detto, si arriva a un sottoproblema facendo la scelta golosa nel problema originale. Tutto ciò che si deve fare è dedurre che una soluzione ottima del sottoproblema, combinata con la scelta golosa già fatta, genera una soluzione ottima del problema originale. Questo schema applica implicitamente l'induzione ai sottoproblemi per dimostrare che, facendo la scelta golosa a ogni passo, si ottiene una soluzione ottima.

Capitolo 15

Analisi ammortizzata

Nell'**analisi ammortizzata** il tempo richiesto per eseguire una sequenza di operazioni su una struttura dati viene calcolato come media dei tempi di tutte le operazioni eseguite. L'analisi ammortizzata può essere utilizzata per dimostrare che il costo medio di un'operazione è piccolo, se si considera la media di una sequenza di operazioni, anche se una singola operazione all'interno della sequenza può essere costosa. L'analisi ammortizzata differisce dall'analisi del caso medio perché non applica la teoria della probabilità; l'analisi ammortizzata valuta le prestazioni medie di ciascuna operazione nel caso peggiore. I tre metodi più utilizzati nell'analisi ammortizzata sono

- **Metodo dell'aggregazione**
- **Metodo degli accantonamenti**
- **Metodo del potenziale**

I costi assegnati durante l'analisi ammortizzata servono esclusivamente all'analisi, nel senso che non devono apparire nel codice. Se, per esempio, viene assegnato un credito a un oggetto x quando viene utilizzato il metodo degli accantonamenti, non occorre assegnare un valore corrispondente a qualche attributo $x.credito$ nel codice.

Le informazioni su una particolare struttura dati che si ottengono dall'analisi ammortizzata possono servire a ottimizzare il progetto degli algoritmi.

15.1 Metodo dell'aggregazione

L'analisi basata sul **metodo dell'aggregazione** dimostra che, per ogni n, una sequenza di n operazioni impiega nel caso peggiore un tempo totale $T(n)$. Nel caso peggiore, il costo medio o **costo ammortizzato** per ogni operazione è quindi $T(n)/n$. Questo costo ammortizzato si applica a ciascuna operazione, anche quando ci sono più tipi di operazioni nella sequenza. Gli altri due metodi potrebbero attribuire costi ammortizzati differenti ai diversi tipi di operazioni.

Come esempio si analizza lo stack che include una nuova operazione. Le due operazioni fondamentali sullo stack sono $\textsc{Push}(S, x)$, che inserisce l'oggetto x nello stack S, e $\textsc{Pop}(S)$ che elimina l'oggetto in cima allo stack S e restituisce

l'oggetto eliminato (se si chiama POP su uno stack vuoto, si genera un errore). Poiché ciascuna di queste operazioni impiega un tempo $O(1)$, si assume che il costo di ciascuna di esse sia 1. Quindi, il costo totale di una sequenza di n operazioni di PUSH e POP è n, e il tempo di esecuzione effettivo di n operazioni è $\Theta(n)$.

Si aggiunge la nuova operazione MULTIPOP(S, k) che elimina i primi k oggetti dalla cima dello stack S o svuota l'intero stack se questo contiene meno di k oggetti. Si assume che k sia positivo; altrimenti l'operazione MULTIPOP lascia lo stack invariato. Nello pseudocodice di MULTIPOP, l'operazione STACK-EMPTY restituisce TRUE se non ci sono oggetti nello stack, altrimenti restituisce FALSE.

```
MULTIPOP(S, k)
1   while not STACK-EMPTY(S) and k > 0
2       POP(S)
3       k ← k - 1
```

Il tempo di esecuzione effettivo è lineare nel numero di operazioni POP effettivamente eseguite; quindi è sufficiente analizzare MULTIPOP in funzione del costo astratto 1 per PUSH e per POP. Il numero di iterazioni del ciclo **while** è il numero $\min(s, k)$ di oggetti eliminati dallo stack. Per ogni iterazione del ciclo viene effettuata una chiamata POP nella riga 2. Quindi, il costo totale di MULTIPOP è $\min(s, k)$ e il tempo di esecuzione effettivo è una funzione lineare di questo costo.

Si analizza ora una sequenza di n operazioni PUSH, POP e MULTIPOP su uno stack inizialmente vuoto. Il costo nel caso peggiore di un'operazione MULTIPOP nella sequenza è $O(n)$, in quanto la dimensione dello stack è al massimo n. Il tempo nel caso peggiore di qualsiasi operazione sullo stack è quindi $O(n)$; pertanto una sequenza di n operazioni costa $O(n^2)$, perché si potrebbero avere $O(n)$ operazioni MULTIPOP che costano $O(n)$ ciascuna. Sebbene questa analisi sia corretta, il risultato $O(n^2)$, che è stato ottenuto considerando il costo nel caso peggiore di ogni operazione, non è stretto.

Applicando il metodo dell'aggregazione si può ottenere un limite superiore più stretto che considera l'intera sequenza di n operazioni. Sebbene una singola operazione MULTIPOP possa essere costosa, qualsiasi sequenza di n operazioni PUSH, POP e MULTIPOP su uno stack inizialmente vuoto può costare al più $O(n)$. Questo perché ogni oggetto può essere eliminato al massimo una volta per ogni volta che viene inserito. Di conseguenza, il numero di volte che l'operazione POP può essere chiamata su uno stack non vuoto, incluse le chiamate all'interno di MULTIPOP, è al massimo il numero di operazioni PUSH, che è al più n. Per ogni n, qualsiasi sequenza di n operazioni PUSH, POP e MULTIPOP impiega complessivamente un tempo $O(n)$. Il costo medio di un'operazione è $O(n)/n = O(1)$. Nel metodo dell'aggregazione si attribuisce il costo medio quale costo ammortizzato di ogni operazione. In questo esempio, quindi, tutte e tre le operazioni sullo stack hanno un costo armonizzato pari a $O(1)$.

15.2 Metodo degli accantonamenti

Nel **metodo degli accantonamenti** dell'analisi ammortizzata vengono assegnati costi variabili a operazioni differenti; qualche operazione potrebbe essere

associata a un costo maggiore o minore del suo costo effettivo. Il costo che viene imputato a un'operazione è detto **costo ammortizzato**. Quando il costo ammortizzato di un'operazione supera il costo effettivo, la differenza viene assegnata a specifici oggetti nella struttura dati sotto forma di **credito**. Il credito potrà essere successivamente utilizzato per contribuire a pagare le operazioni il cui costo ammortizzato è minore del costo effettivo. Quindi, il costo ammortizzato di un'operazione può essere visto come se fosse suddiviso fra il costo effettivo e il credito, che può essere depositato o prelevato.

Indicato con c_i il costo effettivo della i-esima operazione e con $\widehat{c_i}$ il costo ammortizzato della i-esima operazione, deve essere

$$\sum_{i=1}^{n} \widehat{c_i} \geq \sum_{i=1}^{n} c_i$$

per ogni sequenza di n operazioni. Il credito totale memorizzato nella struttura dati è la differenza fra il costo ammortizzato totale e il costo effettivo totale, ovvero

$$\text{credito totale} = \sum_{i=1}^{n} \widehat{c_i} - \sum_{i=1}^{n} c_i$$

Per la precedente disequazione, il credito totale associato alla struttura dati deve essere sempre non negativo. Se il credito totale potesse diventare negativo, allora i costi ammortizzati totali sostenuti fino a quel momento sarebbero inferiori ai costi effettivi totali; per la sequenza delle operazioni eseguite fino a quel momento, il costo ammortizzato totale non potrebbe essere un limite superiore per il costo effettivo totale. Quindi, si deve controllare che il credito totale nella struttura dati non diventi mai negativo.

Si riprende l'esempio dello stack ricordando che i costi effettivi delle operazioni erano

PUSH	1
POP	1
MULTIPOP	$\min(s, k)$

dove k è l'argomento passato a MULTIPOP e s è la dimensione dello stack quando viene chiamata l'operazione MULTIPOP. Si assegnano i seguenti costi ammortizzati

PUSH	2
POP	0
MULTIPOP	0

Si dimostra che qualsiasi sequenza di operazioni su stack può essere pagata utilizzando i costi ammortizzati. Si suppone di utilizzare un conto in dollari \$ per rappresentare ciascuna unità di costo. Si parte da uno stack vuoto. Si considera l'analogia fra uno stack e un pila di piatti di una tavola calda. Quando si aggiunge un piatto alla pila (operazione PUSH), si spende un dollaro per pagare il costo effettivo di questa operazione e resta un credito di un dollaro (dei due dollari del costo ammortizzato), che viene lasciato sul piatto. In qualsiasi momento, ogni piatto della pila ha un dollaro di credito su di esso.

Il dollaro lasciato sul piatto è il credito prepagato per il costo richiesto per toglierlo dalla pila (operazione POP). Quando si esegue un'operazione POP, non

si imputa un costo ammortizzato all'operazione e si paga il suo costo effettivo utilizzando il credito rimasto nella pila. Per togliere un piatto dalla pila, si prende il dollaro di credito che è sul piatto e lo si utilizza per pagare il costo effettivo dell'operazione. Quindi, assegnando un costo ammortizzato un po' alto all'operazione PUSH, non c'è bisogno di attribuire un costo ammortizzato all'operazione POP.

Non occorre attribuire un costo ammortizzato neanche all'operazione MULTIPOP. Per togliere il primo piatto, si prende il dollaro di credito sul piatto e lo si utilizza per pagare il costo effettivo di un'operazione POP. Per togliere il secondo piatto, si ha di nuovo a disposizione un dollaro di credito sul piatto per pagare l'operazione POP e così via. Quindi, si ha sempre un credito sufficiente per pagare le operazioni MULTIPOP.

In altre parole, poiché ogni piatto nella pila ha un dollaro di credito su di esso e la pila ha sempre un numero non negativo di piatti, si ha la garanzia che l'ammontare del credito sarà sempre non negativo. Quindi, per qualsiasi sequenza di n operazioni PUSH, POP e MULTIPOP, il costo ammortizzato totale è un limite superiore per il costo effettivo totale. Poiché il costo ammortizzato totale è $O(n)$, anche il costo effettivo totale sarà $O(n)$.

15.3 Metodo del potenziale

Anziché rappresentare il lavoro pagato come credito memorizzato con specifici oggetti nella struttura dati, il **metodo del potenziale** dell'analisi armonizzata rappresenta il lavoro prepagato come energia potenziale (o semplicemente potenziale) che può essere liberata per pagare operazioni future. Il potenziale è associato alla struttura dati nel suo insieme, anziché a specifici oggetti all'interno della struttura dati.

Il metodo del potenziale parte da una struttura dati iniziale D_0 sulla quale vengono eseguite n operazioni. Per ogni $i = 1, 2, \ldots, n$, si indica con c_i il costo effettivo della i-esima operazione e con D_i la struttura dati che si ottiene dopo aver applicato la i-esima operazione alla struttura dati D_{i-1}. Una **funzione potenziale** Φ associa ciascuna struttura dati D_i a un numero reale $\Phi(D_i)$, che è il **potenziale** della struttura dati D_i. Il **costo ammortizzato** $\widehat{c_i}$ della i-esima operazione rispetto alla funzione potenziale Φ è definito dall'equazione

$$\widehat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

Il costo ammortizzato di ciascuna operazione è quindi il suo costo effettivo più l'incremento di potenziale dovuto all'operazione. Per l'equazione che definisce $\widehat{c_i}$, il costo ammortizzato totale delle n operazioni è

$$\begin{aligned}\sum_{i=1}^{n} \widehat{c_i} &= \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)\end{aligned}$$

Se si definisce una funzione potenziale Φ tale che $\Phi(D_n) \geq \Phi(D_0)$, allora il costo ammortizzato totale $\sum_{i=1}^{n} \widehat{c_i}$ è un limite superiore per il costo effettivo totale $\sum_{i=1}^{n} c_i$. In pratica, non sempre si sa quante operazioni potrebbero essere

eseguite. Quindi, se si impone che $\Phi(D_i) \geq \Phi(D_0)$ per ogni i, allora si ha la garanzia, come nel metodo degli accantonamenti, che si paga in anticipo. Spesso è comodo porre $\Phi(D_0) = 0$ e poi dimostrare che $\Phi(D_i) \geq 0$ per ogni i.

Intuitivamente, se la differenza di potenziale $\Phi(D_i) - \Phi(D_{i-1})$ della i-esima operazione è positiva, allora il costo ammortizzato $\widehat{c_i}$ rappresenta un valore sovrastimato per la i-esima operazione, e il potenziale della struttura dati aumenta. Se la differenza di potenziale è negativa, allora il costo ammortizzato rappresenta un valore sottostimato per la i-esima operazione, e il costo effettivo dell'operazione è pagato dalla riduzione del potenziale. I costi ammortizzati definiti dalle due equazioni precedenti dipendono dalla scelta della funzione potenziale Φ. Differenti funzioni potenziale possono produrre costi ammortizzati differenti, che sono comunque limiti superiori per i costi effettivi. Spesso la scelta della funzione potenziale è il risultato di qualche compromesso; la funzione potenziale migliore da utilizzare dipende dai limiti di tempo desiderati.

Per applicare il metodo del potenziale si riprende l'esempio dello stack con le operazioni PUSH, POP e MULTIPOP. Si definisce la funzione potenziale Φ per uno stack come il numero di oggetti nello stack. Per uno stack vuoto D_0, dal quale si inizia, si ha $\Phi(D_0) = 0$. Poiché il numero di oggetti nello stack non è mai negativo, lo stack D_i che si ottiene dopo la i-esima operazione ha un potenziale non negativo e quindi

$$\begin{aligned} \Phi(D_i) &\geq 0 \\ &= \Phi(D_0) \end{aligned}$$

Il costo ammortizzato totale di n operazioni rispetto a Φ, quindi, rappresenta un limite superiore per il costo effettivo.

Si calcolano adesso i costi ammortizzati delle varie operazioni sullo stack. Se la i-esima operazione su uno stack che contiene s oggetti è un'operazione PUSH, allora la differenza di potenziale è

$$\begin{aligned} \Phi(D_i) - \Phi(D_{i-1}) &= (s+1) - s \\ &= 1 \end{aligned}$$

Per la precedente equazione che definisce $\widehat{c_i}$, il costo ammortizzato di questa operazione PUSH è

$$\begin{aligned} \widehat{c_i} &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= 1 + 1 \\ &= 2 \end{aligned}$$

Supponendo che l'i-esima operazione sullo stack sia MULTIPOP(S, k) e che $k' = min(k, s)$ oggetti vengano eliminati dallo stack, il costo effettivo dell'operazione è k' e la differenza di potenziale è

$$\Phi(D_i) - \Phi(D_{i-1}) = -k'$$

Quindi, il costo ammortizzato dell'operazione MULTIPOP è

$$\begin{aligned} \widehat{c_i} &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= k' - k' \\ &= 0 \end{aligned}$$

Analogamente, il costo ammortizzato di un'ordinaria operazione POP è 0.

Il costo ammortizzato di ciascuna delle tre operazioni è $O(1)$, quindi il costo ammortizzato totale di una sequenza di n operazioni è $O(n)$. Avendo già dimostrato che $\Phi(D_i) \geq \Phi(D_0)$, il costo totale ammortizzato di n operazioni è un limite superiore per il costo totale effettivo. Il costo nel caso peggiore di n operazioni è quindi $O(n)$.

15.4 Tavole dinamiche

In alcune applicazioni non si sa in anticipo quanti oggetti saranno memorizzati in una tavola. Se dopo aver allocato dello spazio in memoria per la tavola, tale spazio non fosse più sufficiente, si dovrebbe allocare nuovamente la tavola con una dimensione maggiore e tutti gli oggetti memorizzati nella tavola originale dovrebbero essere copiati nella nuova tavola più grande. Analogamente, se sono stati cancellati molti elementi dalla tavola, potrebbe essere conveniente riallocare la tavola con una dimensione più piccola. Applicando l'analisi ammortizzata si dimostra che il costo ammortizzato per eseguire un inserimento o una cancellazione in una tavola dinamica è soltanto $O(1)$, anche se il costo effettivo di un'operazione è grande quando viene eseguita un'espansione o una contrazione della tavola. Inoltre, è importante garantire che lo spazio inutilizzato in una tavola dinamica non superi mai una frazione costante dello spazio totale.

Si suppone che la tavola dinamica supporti le operazioni TABLE-INSERT e TABLE-DELETE. TABLE-INSERT inserisce nella tavola un elemento che occupa una singola **cella**, ovvero uno spazio per un elemento. Analogamente, TABLE-DELETE può essere vista come l'eliminazione di un elemento dalla tavola, con conseguente liberazione della corrispondente cella. Nell'analisi ammortizzata è utile considerare il **fattore di carico** $\alpha(T)$ di una tavola non vuota T che definisce il numero di elementi memorizzati nella tavola diviso per la dimensione della tavola. Per definizione, una tavola vuota ha dimensione 0 e il suo fattore di carico è 1. Se il fattore di carico di una tavola dinamica è limitato inferiormente da una costante, lo spazio inutilizzato nella tavola non supera mai una frazione costante della quantità totale di spazio.

15.4.1 Espansione di una tavola

Una tavola si riempie quando tutte le celle sono state utilizzate ovvero quando il suo fattore di carico è 1. Se un elemento viene inserito in una tavola piena, si può **espandere** la tavola allocando una nuova tavola che ha più celle della vecchia tavola. Poiché si vuole che la tavola risieda in spazi di memoria contigui, si deve allocare un nuovo array per la tavola più grande e poi copiare gli elementi della vecchia tavola in quella nuova.

Un tipico metodo consiste nell'allocare una nuova tavola che ha il doppio di celle della vecchia tavola. Se si effettuano soltanto inserimenti, il fattore di carico di una tavola è sempre almeno $1/2$ e quindi, la quantità di spazio sprecato non supera mai la metà dello spazio totale nella tavola.

Nello pseudocodice di TABLE-INSERT si suppone che T sia un oggetto che rappresenta la tavola. L'attributo $T.table$ contiene un puntatore al blocco di memoria che rappresenta la tavola. L'attributo $T.num$ contiene il numero di elementi della tavola; l'attributo $T.size$ è il numero totale di celle nella tavola.

Inizialmente, la tavola è vuota: $T.num = T.size = 0$. Ci sono due procedure di inserimento: la stessa procedura TABLE-INSERT e l'**inserimento elementare** in una tavola nelle righe 6 e 10.

```
TABLE-INSERT(T, x)
1   if T.size = 0
2       alloca T.table con una cella
3       T.size ← 1
4   if T.num = T.size
5       alloca new-table con 2 · T.size celle
6       inserisce tutti gli elementi di T.table in new-table
7       rilascia T.table
8       T.table ← new-table
9       T.size ← 2 · T.size
10  inserisce x in T.table
11  T.num = T.num + 1
```

È possibile analizzare il tempo di esecuzione di TABLE-INSERT in base al numero di inserimenti elementari, assegnando un costo 1 a ogni inserimento elementare. Si suppone che il tempo di esecuzione effettivo di TABLE-INSERT sia lineare nel tempo per inserire i singoli elementi e che quindi il costo aggiuntivo per allocare una tavola iniziale nella riga 2 sia costante e il costo aggiuntivo per allocare e rilasciare lo spazio nelle righe 5 e 7 sia dominato dal costo di trasferimento degli elementi nella riga 6. Si chiama **espansione** l'evento che si verifica quando viene eseguita la clausola **then** nelle righe 5 - 9.

Si determina ora il costo c_i della i-esima operazione in una sequenza di n operazioni TABLE-INSERT su una tavola inizialmente vuota. Se c'è spazio nella tavola corrente, allora $c_i = 1$, perché si deve eseguire l'unico inserimento elementare nella riga 10. Se invece la tavola corrente è piena e si verifica un'espansione, allora $c_i = i$: il costo è 1 per l'inserimento elementare nella riga 10 più $i - 1$ per gli elementi che devono essere copiati dalla vecchia tavola in quella nuova (riga 6). Se vengono eseguite n operazioni, il costo nel caso peggiore di un'operazione è $O(n)$, che determina un limite superiore pari a $O(n^2)$ per il tempo di esecuzione totale di n operazioni. Questo limite non è stretto, perché la tavola viene espansa raramente durante l'esecuzione di n operazioni TABLE-INSERT. Più precisamente, la i-esima operazione provoca un'espansione soltanto se $i - 1$ è una potenza esatta di 2. Infatti, il costo ammortizzato di un'operazione è $O(1)$, come è possibile dimostrare applicando il metodo dell'aggregazione. Il costo della i-esima operazione è

$$c_i = \begin{cases} i & \text{se } i-1 \text{ è una potenza esatta di 2} \\ 1 & \text{negli altri casi} \end{cases}$$

Il costo totale di n operazioni TABLE-INSERT è quindi

$$\begin{aligned} \sum_{i=1}^{n} c_i &\le n + \sum_{j=0}^{\lfloor lg\, n \rfloor} 2^j \\ &< n + 2n \\ &= 3n \end{aligned}$$

perché ci sono al massimo n operazioni che costano 1 e i costi delle restanti operazioni formano una serie geometrica. Poiché il costo totale di n operazioni TABLE-INSERT è 3n, il costo ammortizzato di una singola operazione è 3.

Intuitivamente, applicando il metodo degli accantonamenti, ogni elemento paga per tre inserimenti elementari: inserire sé stesso nella tavola corrente, spostare sé stesso quando la tavola viene espansa e spostare un altro elemento, che è già stato spostato una volta, quando la tavola viene espansa. Supponendo che la dimensione della tavola sia m immediatamente dopo un'espansione. Allora, il numero di elementi nella tavola è $m/2$ e la tavola non contiene alcun credito. Si addebita un costo di 3 dollari per ogni inserimento. L'inserimento elementare che viene effettuato subito costa 1 dollaro. Un altro dollaro viene posto come credito sull'elemento inserito. Il terzo dollaro è posto come credito su uno degli $m/2$ elementi già presenti nella tavola. Il riempimento della tavola richiede altri $m/2 - 1$ inserimenti; pertanto, quando la tavola contiene m elementi ed è piena, ogni elemento ha un dollaro per pagare il suo reinserimento durante l'espansione.

Anche il metodo del potenziale può essere utilizzato per analizzare una sequenza di n operazioni TABLE-INSERT e anche in tal caso si ottiene un costo c_i pari a 3; inoltre verrà anche utilizzato per progettare un'operazione TABLE-DELETE anch'essa con un costo ammortizzato $O(1)$.

15.4.2 Contrazione di una tavola

Per implementare un'operazione TABLE-DELETE, è sufficiente eliminare l'elemento specificato dalla tavola. Tuttavia, spesso è preferibile **contrarre** la tavola quando il fattore di carico della tavola diventa troppo piccolo, in modo che lo spazio sprecato non sia eccessivo. La contrazione della tavola è simile all'espansione: quando il numero di elementi nella tavola diventa troppo piccolo, si alloca una nuova tavola più piccola e poi si copiano gli elementi della vecchia tavola in quella nuova. Lo spazio in memoria per la vecchia tavola può essere poi rilasciato restituendolo al sistema di gestione della memoria. L'ideale sarebbe che fossero preservate due proprietà:

- Il fattore di carico della tavola dinamica è limitato inferiormente da una costante
- Il costo ammortizzato di un'operazione sulla tavola è limitato superiormente da una costante

Si suppone che il costo possa essere misurato in base al numero di inserimenti e cancellazioni elementari. Si potrebbe ritenere che si debba raddoppiare la dimensione della tavola quando un elemento viene inserito in una tavola piena e dimezzare la dimensione quando una cancellazione fa sì che la tavola sia piena per meno della metà. Questa strategia assicura che il fattore di carico della tavola non sia mai maggiore di 1/2, tuttavia può comportare un aumento eccessivo del costo ammortizzato di un'operazione.

Il difetto di questa strategia è dovuto al fatto che dopo un'espansione, non si eseguono un numero sufficiente di cancellazioni per pagare una contrazione. Analogamente, dopo una contrazione, non si eseguono un numero sufficiente di inserimenti per pagare un'espansione.

È possibile migliorare questa strategia consentendo al fattore di carico della tavola di scendere sotto $1/2$. Specificatamente, si continua a raddoppiare la dimensione della tavola quando un elemento viene inserito in una tavola piena, ma si dimezza la dimensione della tavola quando una cancellazione porta la tavola a essere piena per meno di $1/4$, anziché per meno di $1/2$. Il fattore di carico della tavola è quindi limitato inferiormente dalla costante $1/4$.

Si considera come ideale il fattore di carico $1/2$, nel qual caso il potenziale della tavola sarà 0. Quando il fattore di carico si discosta da $1/2$, il potenziale aumenta in modo che, quando la tavola si espande o contrae, essa ha acquisito un potenziale sufficiente per pagare la copia di tutti gli elementi nella tavola appena allocata. Quindi, occorre una funzione potenziale che cresca fino a $T.num$ (numero di elementi nella tavola) quando il fattore di carico aumenta fino a 1 o diminuisce fino a $1/4$. Dopo un'espansione o una contrazione della tabella, il fattore di carico ritorna a $1/2$ e il potenziale della tabella ritorna a 0.

Il codice della procedura TABLE-DELETE è simile a quello di TABLE-INSERT e anch'essa ha un costo ammortizzato $O(1)$. Per l'analisi è utile supporre che, se il numero di elementi nella tavola scende a 0, lo spazio in memoria della tavola venga rilasciato; ovvero, se $T.num = 0$, allora $T.size = 0$.

Per l'analisi del costo di una sequenza di n operazioni TABLE-INSERT e TABLE-DELETE si utilizza il metodo del potenziale e la funzione potenziale

$$\Phi(T) = \begin{cases} 2 \cdot T.num - T.size & \text{se } \alpha(T) \geq 1/2 \\ T.size/2 - T.num & \text{se } \alpha(T) < 1/2 \end{cases}$$

Il potenziale di una tavola vuota è 0 e il potenziale non è mai negativo. Quindi, il costo ammortizzato totale di una sequenza di operazioni rispetto a Φ è un limite superiore per il costo effettivo della sequenza.

Applicando il metodo del potenziale si arriva ad ottenere un costo ammortizzato $\widehat{c_i}$ pari a 0 o 3 per TABLE-INSERT e pari a 1 o 2 per TABLE-DELETE. Quindi, anche il costo ammortizzato è limitato superiormente da una costante (le due proprietà sono quindi preservate).

In sintesi, poiché il costo ammortizzato di ciascuna operazione è limitato superiormente da una costante, il tempo effettivo per qualsiasi sequenza di n operazioni su una tavola dinamica è $O(n)$.

Capitolo 16

Algoritmi elementari per grafi

Un **grafo orientato** (diretto) è una coppia ordinata $G = (V, E)$ di insiemi, dove V è un insieme finito ed E è una relazione binaria in V. L'insieme V è detto **insieme dei vertici** di G e i suoi elementi sono detti vertici. L'insieme E è detto **insieme degli archi** di G e i suoi elementi sono detti archi. I vertici sono rappresentati da cerchi; gli archi sono rappresentati da frecce. La cardinalità dell'insieme E può essere al minimo 0 (nessun arco) e al massimo V^2. Un arco del grafo G è un qualsiasi insieme $\{u, v\}$ tale che $u, v \in V$ e $u \neq v$. Pur essendo un insieme, per convenzione l'arco viene indicato con (u, v). Se (u, v) è un arco in un grafo orientato $G = (V, E)$, si dirà che (u, v) esce dal vertice u ed entra nel vertice v. In un grafo orientato, il grado uscente di un vertice è il numero di archi che escono nel vertice; il grado entrante di un vertice è il numero di archi che entrano nel vertice. Il grado di un vertice in un grafo orientato è la somma del suo grado entrante e del suo grado uscente; rappresenta il numero di archi che incidono nel vertice. Un vertice il cui grado è 0 si dice isolato.

In un **grafo non orientato** (indiretto) $G = (V, E)$, l'insieme degli archi E è composto da coppie di vertici non ordinate, anziché da coppie ordinate. Se (u, v) è un arco in un grafo non orientato $G = (V, E)$, si dirà che (u, v) è incidente nei vertici u e v.

Se (u, v) è un arco di un grafo $G = (V, E)$, si dirà che il vertice v è adiacente al vertice u. Se il grafo non è orientato, la relazione di adiacenza è simmetrica. Se il grafo è orientato, la relazione di adiacenza non è necessariamente simmetrica.

Un **cammino** di lunghezza k da un vertice u a un vertice u' in un grafo $G = (V, E)$ è una sequenza $\langle v_0, v_1, v_2, \ldots, v_k \rangle$ di vertici tali che $u = v_0$, $u' = v_k$ e $(v_{i-1}, v_i) \in E$ per $i = 1, 2, \ldots, k$. La lunghezza del cammino è il numero di archi nel cammino.

In un grafo orientato, un cammino $\langle v_0, v_1, v_2, \ldots, v_k \rangle$ forma un **ciclo** se $v_0 = v_k$ e il cammino contiene almeno un arco. Un grafo senza ciclo è **aciclico** (per esempio gli alberi).

Un grafo orientato è **connesso** se ogni coppia di vertici è collegata attraverso un cammino. Le **componenti connesse** di un grafo sono le classi di equivalenza dei vertici secondo la relazione "è raggiungibile da". Un grafo non orientato è

connesso se ha esattamente una componente connessa, ovvero se ogni vertice è raggiungibile da ogni altro vertice.

Un grafo orientato è **fortemente connesso** se due vertici qualsiasi sono raggiungibili l'uno dall'altro. Le **componenti fortemente connesse** di un grafo orientato sono le classi di equivalenza dei vertici seconda la relazione "sono mutuamente raggiungibili". Un grafo orientato è fortemente connesso se ha una sola componente fortemente connessa.

16.1 Rappresentazione dei grafi

I due modi principali per rappresentare un grafo $G = (V, E)$ sono:

1. Liste di adiacenza
2. Matrice di adiacenza

Entrambi possono essere applicati sia a grafi orientati che a quelli non orientati. Di solito si preferisce la rappresentazione con liste di adiacenza perché permette di rappresentare in modo compatto i grafi **sparsi**, ovvero quei grafi in cui $|E|$ è molto più piccolo di $|V|^2$. Tuttavia, potrebbe essere preferibile una rappresentazione con matrice di adiacenza, quando il grafo è **denso**, ovvero quando $|E|$ è prossimo a $|V|^2$, o quando si deve essere in grado di dire rapidamente se c'è un arco che collega due vertici particolari.

Il tempo di esecuzione richiesto dalle varie operazioni e lo spazio richiesto dalle due rappresentazioni vengono spesso espressi considerando sia $|V|$ che $|E|$. Nella notazione asintotica si omette la cardinalità.

16.1.1 Liste di adiacenza

La **rappresentazione con liste di adiacenza** di un grafo $G = (V, E)$ consiste in un vettore Adj di $|V|$ liste, una per ogni vertice di V. Per ogni nodo $u \in V$, la lista di adiacenza $Adj[u]$ contiene tutti i vertici v tali che esista un arco $(u, v) \in E$. Ovvero $Adj[u]$ include tutti i vertici adiacenti a u in G (per i grafi indiretti si considera sia (u, v) che (v, u)).

Se G è un grafo orientato, la somma delle lunghezze di tutte le liste di adiacenza è $|E|$, perché un arco della forma (u, v) è rappresentato inserendo v in $Adj[u]$. Se G è un grafo non orientato, la somma delle lunghezze di tutte le liste di adiacenza è $2|E|$, perché se (u, v) è un arco non orientato, allora u appare nella lista di adiacenza di v e viceversa.

Per i grafi orientati e non orientati, la rappresentazione con liste di adiacenza richiede la quantità di memoria $\Theta(V + E)$ (questo perché ci sono tutti e soli gli archi del grafo, quindi si considera Θ).

Tramite le liste di adiacenza è possibile rappresentare anche i **grafi pesati**, cioè, i grafi per i quali ogni arco ha un **peso** associato, definito tramite la **funzione peso** $w : E \to \mathbb{R}$. Per esempio, sia $G = (V, E)$ un grafo pesato con la funzione peso w; il peso $w(u, v)$ dell'arco $(u, v) \in E$ viene memorizzato insieme al vertice v nella lista di adiacenza di v come se fosse un attributo. La rappresentazione con liste di adiacenza è molto robusta, nel senso che può essere modificata per supportare molte altre varianti di grafi.

Il tempo necessario per elencare tutti i nodi adiacenti ad un nodo u è proporzionale al numero di nodi adiacenti, ovvero è $\Theta(u.degree)$, dove *degree* indica

il numero di archi uscenti da u (si una Θ perché si scorre la lista fino in fondo). Il tempo per determinare se $(u, v) \in E$ è invece $O(u.degree)$; stavolta si usa O perché ci si potrebbe fermare prima di essere arrivati alla fine della lista.

Uno svantaggio potenziale della rappresentazione con liste di adiacenza è che non c'è un modo più veloce per determinare se un particolare arco (u, v) è presente nel grafo che cercare v nella lista di adiacenza $Adj[u]$. Per porre rimedio a questo svantaggio, si può rappresentare il grafo con una matrice di adiacenza, al costo di usare asintoticamente una maggiore quantità di memoria.

16.1.2 Matrice di adiacenza

Per la **rappresentazione con matrice di adiacenza** di un grafo $G = (V, E)$ si suppone che i vertici siano numerati 1, 2, ..., $|V|$ in modo arbitrario. La rappresentazione con matrice di adiacenza di un grafo G consiste in una matrice $A = (a_{ij})$ di dimensioni $|V| \times |V|$ tale che

$$a_{ij} = \begin{cases} 1 & \text{se } (i, j) \in E \\ 0 & \text{negli altri casi} \end{cases}$$

La matrice di adiacenza di un grafo richiede una memoria $\Theta(V^2)$, indipendentemente dal numero di archi nel grafo. Il tempo necessario per elencare tutti i vertici adiacenti a u è $\Theta(V)$ perché si deve scorrere una riga fino alla fine; il tempo necessario per determinare se $(u, v) \in E$ è $\Theta(1)$.

Nel caso di un grafo non orientato la matrice è simmetrica rispetto alla diagonale principale. Poiché (u, v) e (v, u) rappresentano lo stesso arco in un grafo non orientato, la matrice di adiacenza di A di un grafo non orientato è uguale alla sua trasposta: $A = A^T$.

Come la rappresentazione con liste di adiacenza di un grafo, anche la rappresentazione con matrice di adiacenza può essere utilizzata per i grafi pesati. Il peso di ogni arco (u, v) è memorizzato come l'elemento nella riga u e nella colonna v della matrice di adiacenza.

Sebbene la rappresentazione con liste di adiacenza sia asintoticamente efficiente almeno quanto la rappresentazione con matrice di adiacenza, tuttavia quando i grafi sono abbastanza piccoli potrebbe essere preferita la matrice di adiacenza per la sua semplicità. Inoltre, se il grafo non è pesato, c'è un'ulteriore vantaggio per la rappresentazione con matrice di adiacenza che riguarda la memoria richiesta. Anziché utilizzare una parola della memoria del calcolatore per ogni elemento della matrice di adiacenza, basta utilizzare un solo bit per ogni elemento.

Nelle liste di adiacenza solitamente si utilizzano i puntatori ai vertici che sono quindi più lenti. Per scegliere tra liste di adiacenza e matrice di adiacenza si considerano i byte (pesi) necessari. In genere, le liste di adiacenza hanno costanti più alte. Per le liste di adiacenza si dovrebbe conoscere il peso w, il numero di byte per ciascun puntatore p e la dimensione k della chiave definita come $k = log_2\lceil V \rceil$. Si ottiene così una dimensione pari a $(k+p+w)E+(k+p)V$. Per la matrice di adiacenza, invece, la dimensione è pari a $wV^2 + kV$, dove tuttavia V è probabilmente irrilevante asintoticamente. Si confronta quindi $(k+p+w)E$ con wV^2 e se

$$E > \frac{wV^2}{k+p+w}$$

è più conveniente utilizzare la matrice di adiacenza.

16.2 Visita in ampiezza

Dato un grafo $G = (V, E)$ e un vertice distinto s, detto **sorgente**, la visita in ampiezza ispeziona sistematicamente gli archi di G per scoprire tutti i vertici che sono raggiungibili da s. Per ogni arco $v \in V$ calcola la distanza (numero minimo di archi) $v.d$ da s a v e il vertice $u = v.\pi$, detto **predecessore** di v, tale che (u, v) è l'ultimo arco nel cammino minimo $s \rightsquigarrow v$. Genera anche un albero BF (breadth-first tree) con radice s che contiene tutti i vertici raggiungibili. Per ogni vertice v raggiungibile da s, il cammino semplice nell'albero BF che va da s a v corrisponde ad un cammino minimo da s a v in G. L'algoritmo opera sui grafi orientati e non orientati.

La visita in ampiezza è chiamata così perché espande la frontiera fra i vertici scoperti e quelli da scoprire in maniera uniforme lungo l'ampiezza della frontiera (i vertici vengono scoperti per livelli). Ovvero l'algoritmo scopre tutti i vertici che si trovano a distanza k da s, prima di scoprire i vertici a distanza $k + 1$.

L'algoritmo usa anche una coda Q con schema FIFO su cui esegue le operazioni ENQUEUE e DEQUEUE aventi costo costante. La lista può essere implementata come una lista collegata.

Per tenere traccia del lavoro svolto, la visita in ampiezza colora i vertici di bianco, grigio o di nero. Inizialmente tutti i vertici sono bianchi. Quando un vertice viene incontrato per la prima volta durante la visita si dice che viene scoperto e diventa grigio. Quando sono stati visitati tutti i vertici adiacenti di un vertice, tale vertice diventa nero. Se $(u, v) \in E$ e il vertice u è nero, allora il vertice v è grigio oppure nero; ovvero tutti i vertici adiacenti ai vertici neri sono stati scoperti. I vertici grigi possono avere qualche vertice bianco adiacente; essi rappresentano la frontiera fra i vertici scoperti e quelli da scoprire.

```
BFS(G, s)
 1  for ogni vertice u ∈ G.V − {s}
 2      u.color ← WHITE
 3      u.d ← ∞
 4      u.π ← NIL
 5  s.color ← GRAY
 6  s.d ← 0
 7  s.π ← NIL
 8  Q ← ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11      u ← DEQUEUE(Q)
12      for ogni vertice u ∈ G.Adj[u]
13          if v.color = WHITE
14              v.color ← GRAY
15              v.d ← u.d + 1
16              v.π ← u
17              ENQUEUE(Q, v)
18      u.color ← BLACK
```

Commenti:

2 Ancora non è stato trovato

5 È stato scoperto ma ancora non sono stati trovati tutti i suoi vicini

9 Inizializza la coda FIFO

10 Finché c'è qualcosa nella coda

11 Prende il primo elemento messo nella coda. All'inizio c'è s

12 Guarda tutti i nodi raggiungibili da u

15 È stato scoperto dopo u

16 È stato scoperto tramite u

18 Ha finito di visitare u

La procedura BFS suppone che il grafo di input $G = (V, E)$ sia rappresentato con le liste di adiacenza. Il colore di ogni vertice $u \in V$ è memorizzato nell'attributo $u.color$ e il predecessore di u è memorizzato nell'attributo $u.\pi$.

L'idea è che l'algoritmo manda un'onda da s, prima colpisce tutti i nodi a distanza 1 da s, poi colpisce tutti i nodi a distanza 2 da s e così via. Memorizza il fronte d'onda nella coda Q, $v \in Q$ se e solo se l'onda ha colpito v, ma non è ancora uscita da v. Con l'eccezione del vertice sorgente s, le righe 1 - 4 colorano di bianco tutti i vertici, assegnano all'attributo $u.d$ il valore infinito per ogni vertice u e assegnano al padre di ogni vertice il valore NIL. La riga 5 colora s di grigio, perché questo vertice è considerato scoperto quando inizia la procedura. La riga 6 inizializza $s.d$ a 0; la riga 7 assegna al predecessore della sorgente il valore NIL. Le righe 8 - 9 inizializzano Q con la coda che contiene il solo vertice s. Il ciclo **while** (righe 10 - 18) si ripete finché restano dei vertici grigi, che sono vertici scoperti le cui liste di adiacenza non sono state ancora completamente esaminate. Questo ciclo conserva la seguente invariante di ciclo:

> *Quando viene eseguito il test della riga 10, la coda Q è formata dall'insieme dei vertici grigi*

Per determinare il tempo di esecuzione di BFS si può usare il metodo dell'aggregazione. Le operazioni di inserimento e cancellazione dalla coda richiedono un tempo $O(1)$, quindi il tempo totale dedicato alle operazioni per la coda è $O(V)$ perché ogni nodo è messo nella coda al massimo una volta. Poiché la lista di adiacenza di ciascun vertice viene ispezionata soltanto quando il vertice viene rimosso dalla coda, ogni lista di adiacenza viene ispezionata al più una volta. Poiché la somma delle lunghezze di tutte le liste di adiacenza è $\Theta(E)$, il tempo totale impiegato per ispezionare le liste di adiacenza è $O(E)$ (ogni nodo è tolto dalla coda al massimo una volta e si esamina (u, v) solo quando u è tolto dalla coda, inoltre, ogni arco è analizzato al massimo una volta se diretto e al massimo due volte se non diretto). Il costo aggiuntivo di inizializzazione è $O(V)$, quindi il tempo di esecuzione totale di BFS è $O(V + E)$.

16.3 Visita in profondità

Nella visita in profondità, gli archi vengono ispezionati a partire dall'ultimo vertice scoperto v che ha ancora archi non ispezionati che escono da esso. Quando

tutti gli archi di v sono stati ispezionati, la visita *fa marcia indietro* per ispezionare gli archi che escono dal vertice dal quale v era stato scoperto. Questo processo continua finché non saranno stati scoperti tutti i vertici che sono raggiungibili dal vertice sorgente originale. Se restano dei vertici non scoperti, allora uno di essi viene selezionato come nuovo vertice sorgente e la visita riparte da questa sorgente. L'intero processo viene ripetuto finché non saranno scoperti tutti i vertici del grafo.

Come nella visita in ampiezza, quando un vertice v viene scoperto durante un'ispezione della lista di adiacenza di un vertice u già scoperto, la visita in profondità registra questo evento assegnando u all'attributo $v.\pi$ di v (il predecessore).

Diversamente dalla visita in ampiezza, il cui sottografo dei predecessori forma un albero, il sottografo dei predecessori prodotto da una visita in profondità può essere formato da più alberi, perché la visita può essere ripetuta da più sorgenti. Il **sottografo dei predecessori** di una visita in profondità forma una **foresta DF** (depth-first forest) composta da veri **alberi DF**.

Come nella visita in ampiezza, i vertici vengono colorati durante la visita in profondità per indicare il loro stato. Inizialmente, tutti i vertici sono bianchi. Un vertice diventa grigio quando viene **scoperto** durante la visita; diventa nero quando viene **completato**, ovvero quando la sua lista di adiacenza è stata completamente ispezionata. Questa tecnica garantisce che ogni vertice vada a finire in un solo albero DF, in modo che questi alberi siano disgiunti.

Oltre a creare un foresta DF, la visita in profondità associa anche a ciascun vertice delle informazioni temporali. Ogni vertice v ha due informazioni temporali: la prima $v.d$ registra il momento in cui il vertice v viene scoperto (e colorato di grigio); la seconda $v.f$ registra il momento in cui la visita completa l'ispezione della lista di adiacenza del vertice v (che diventa nero).

La procedura DFS registra nell'attributo $u.d$ il momento in cui scopre il vertice u e nell'attributo $u.f$ il momento in cui completa la visita del vertice u. Queste informazioni temporali sono numeri interi compresi fra 1 e $2|V|$ (ciascuno dei $|V|$ vertici è scoperto una sola volta e la sua visita è completata una sola volta). Per ogni vertice u si ha $u.d < u.f$. Il vertice u è WHITE prima del tempo $u.d$, GRAY fra il tempo $u.d$ e il tempo $u.f$, e BLACK successivamente.

```
DFS(G)
1   for ogni vertice u ∈ G.V
2       u.color ← WHITE
3       u.π ← NIL
4   time ← 0
5   for ogni vertice u ∈ G.V
6       if u.color = WHITE
7           DFS-VISIT(G, u)
```

Commenti:

1-4 Inizializzazione

6-7 Se il nodo u non era già stato visitato, quindi era bianco, lo visita

```
DFS-VISIT(G, u)
 1  time ← time + 1
 2  u.d ← time
 3  u.color ← GRAY
 4  for ogni v ∈ G.Adj[u]
 5      if v.color = WHITE
 6          v.π ← u
 7          DFS-VISIT(G, v)
 8  u.color ← BLACK
 9  time ← time + 1
10  u.f ← time
```

Commenti:

2 Il tempo di scoperta coincide con il tempo attuale

3 Non controlla se è bianco perché il controllo è già stato fatto in DFS

4 Valuta i vicini di u

5 Se il vicino è bianco lo visita

8 Quando ha visto tutti i vicini di u, il vertice diventa nero

La righe 1 - 3 di DFS colorano di bianco tutti i vertici e inizializzano i loro attributi π e NIL. La riga 4 azzera il contatore globale del tempo. Le righe 5 - 7 controllano, uno alla volta, tutti i vertici in V e, quando trovano un vertice bianco, lo visitano utilizzando la procedura DFS-VISIT. Ogni volta che viene chiamata la procedura DFS-VISIT(u) nella riga 7, il vertice u diventa la radice di un nuovo albero della foresta DF. Quando la procedura DFS termina, ad ogni vertice u è stato assegnato un **tempo di scoperta** $u.d$ e un **tempo di completamento** $u.f$.

In ogni chiamata di DFS-VISIT(u), il vertice u è inizialmente bianco. La riga 1 incrementa la variabile globale *time*, la riga 2 registra il nuovo valore di *time* come il tempo di scoperta $u.d$ e la riga 3 colora di grigio u. Le righe 4 - 7 ispezionano ogni vertice v adiacente a u e visitano in modo ricorsivo il vertice v, se è bianco. Dopo che tutti i nodi che escono da u sono stati ispezionati, le righe 8 - 10 colorano di nero u, incrementano *time* e registrano in $u.f$ il tempo di completamento della visita.

I risultati della visita in profondità potrebbero dipendere dall'ordine in cui i vertici sono ispezionati nella riga 5 della procedura DFS e dall'ordine in cui i vicini di un vertice vengono visitati nella riga 4 della procedura DFS-VISIT. In pratica, queste differenze nell'ordine in cui vengono effettuate le visite non causano problemi, in quanto qualsiasi risultato della visita in profondità di solito può essere efficacemente utilizzato, perché i risultati ottenuti sono essenzialmente equivalenti.

I cicli nelle righe 1 - 3 e nelle righe 5 - 7 di DFS impiegano un tempo $\Theta(V)$, escluso il tempo per eseguire le chiamate di DFS-VISIT. Applicando il metodo dell'aggregazione e considerando che la procedura DFS-VISIT è chiamata esattamente una volta per ogni vertice $v \in V$ (viene invocata soltanto se un vertice è bianco e la prima cosa che fa è colorarlo di grigio), si ottiene che il ciclo nelle righe 4 - 7 di DFS-VISIT ha un costo totale di $\Theta(E)$. Il tempo di esecuzione di DFS è dunque $\Theta(V + E)$ (Θ e non solo O perché esplora ogni nodo e arco).

16.3.1 Teoremi sui grafi

La visita in profondità fornisce informazioni preziose sulla struttura di un grafo. Una delle più importanti è che i tempi di scoperta e di completamento hanno una **struttura di parentesi**. Se si rappresenta le scoperta del vertice u con una parentesi aperta (u e il suo completamento con una parentesi chiusa u), allora la storia delle scoperte e dei completamenti produce una sequenza di parentesi opportunamente annidate.

Teorema 16.1 (**Teorema delle parentesi**). *In una visita in profondità di un grafo $G = (V, E)$ (orientato o non orientato), per ogni coppia di vertici u e v, è soddisfatta una sola delle seguenti tre condizioni:*

1. *Gli intervalli $[u.d, u.f]$ e $[v.d, v.f]$ sono completamente disgiunti ovvero $u.d < u.f < v.d < v.f$ oppure $v.d < v.f < u.d < u.f$; inoltre u e v non sono discendenti l'uno dell'altro in un albero DF*
2. *L'intervallo $[u.d, u.f]$ è interamente contenuto nell'intervallo $[v.d, v.f]$ cioè $u.d < v.d < v.f < u.f$; inoltre u è un discendente di v in un albero DF*
3. *L'intervallo $[v.d, v.f]$ è interamente contenuto nell'intervallo $[u.d, u.f]$ cioè $v.d < u.d < u.f < v.f$; inoltre v è un discendente di u in un albero DF*

Non può succedere che $u.d < v.d < u.f < v.f$, non c'è possibilità di intersezione.

Corollario 16.1. *Il vertice v è un discendente del vertice u nella foresta DF per un grafo G (orientato o non orientato) se e soltanto se $u.d < v.d < v.f < u.f$.*

Teorema 16.2 (**Teorema del cammino bianco**). *In una foresta DF di un grafo $G = (V, E)$ (orientato o non orientato), il vertice v è un discendente del vertice u se e soltanto se, al tempo $u.d$ in cui viene scoperto u, il vertice v può essere raggiunto da u lungo un cammino $u \rightsquigarrow v$ che è formato esclusivamente da vertici bianchi (ad eccezione di u che è stato appena colorato di grigio).*

16.3.2 Classificazione degli archi

La visita in profondità permette di classificare gli archi del grafo. Questa classificazione può essere usata per raccogliere informazioni su un grafo. Ci sono quattro tipi di archi in base alla foresta della visita in profondità del grafo G:

Archi d'albero T Gli archi nella foresta DF trovati esplorando l'arco (u, v)

Archi all'indietro B Sono quegli archi (u, v) che collegano un vertice u a un antenato v in un albero DF

Archi in avanti F Sono gli archi (u, v) che collegano un vertice u a un discendente v in un albero DF, ma non sono T

Archi trasversali C Tutti gli altri archi. Possono connettere i vertici nello stesso albero DF, purché un vertice non sia un antenato dell'altro, oppure possono connettere vertici di alberi DF differenti

L'algoritmo DFS possiede le informazioni necessarie per classificare gli archi che incontra.

L'idea chiave è che ogni arco (u, v) può essere classificato in base al colore del vertice v che viene raggiunto quando l'arco viene ispezionato per la prima volta:

- Se $v.color$ = WHITE, allora l'arco è un arco d'albero T
- Se $v.color$ = GRAY, allora l'arco è un arco all'indietro B
- Se $v.color$ = BLACK, allora è un arco F o un arco C. Se $u.d < v.d$ è un arco F, se $u.d > v.d$ è invece un arco C e i due vertici non sono connessi

Teorema 16.3. *In una visita in profondità di un grafo non orientato G, gli archi di G possono essere archi d'albero T o archi all'indietro B. Non possono esserci archi F o C perché tutti i nodi sono raggiunti da un attraversamento.*

16.4 Ordinamento topologico

Un **ordinamento topologico** di un grafo aciclico o dag (directed acyclic graph) $G = (V, E)$ è un ordinamento lineare di tutti i suoi vertici tali che, se G contiene un arco (u, v), allora u appare prima di v nell'ordinamento (da qualche parte, non per forza subito prima). Può essere visto come un ordinamento dei suoi vertici lungo una linea orizzontale in modo che tutti gli archi orientati siano diretti da sinistra a destra. È utile per gestire oggetti che hanno un **ordinamento parziale**: se $a > b$ e $b > c$ allora $a > c$ ma si può avere a e b tali che non si sa se $a > b$ o $b > c$. Si può sempre avere un ordinamento **totale** ($a > b$ o $b > a$ $\forall a \neq b$) da un ordinamento parziale. Se il grafo non è aciclico non si può effettuare un ordinamento lineare.

Teorema 16.4. *Un grafo diretto G è* ***aciclico*** *se e solo se una visita* DFS *di G non genere archi all'indietro B.*

Dimostrazione. Bisogna dimostrare in entrambi le direzioni:

- **Se c'è un arco all'indietro allora c'è un ciclo**: si suppone che ci sia un arco all'indietro (u, v). Allora il vertice v è un antenato del vertice u nella foresta DF. Quindi esiste un percorso $v \rightsquigarrow u$ nel grafo G che, insieme all'arco all'indietro (u, v), completa il ciclo.
- **Se c'è un ciclo allora c'è un arco all'indietro**: si suppone che il grafo G contenga un ciclo c. Si dimostra che una visita in profondità di G genera un arco all'indietro. Sia v il primo vertice che viene scoperto in c e sia (u, v) l'arco precedente in c. Al tempo $v.d$ i vertici di c formano un cammino bianco $v \rightsquigarrow u$ (poiché v è il primo vertice scoperto in c). Per il teorema del cammino bianco, il vertice u diventa un discendente di v nella foresta DF. Dunque (u, v) è un arco all'indietro.

□

TOPOLOGICAL-SORT(G)
1 Chiama DFS(G) per calcolare i tempi $v.f$ per ogni vertice v
2 Completata l'ispezione di un vertice,
  inserisce il vertice in testa a una lista concatenata
3 **return** la lista concatenata dei vertici

Commenti:

1 Non interessa $v.d$

3 Equivale a "emetti i vertici in ordine di tempo di terminazione decrescente"

È possibile eseguire un ordinamento topologico nel tempo $\Theta(V + E)$, perché la visita in profondità impiega un tempo $\Theta(V + E)$ e occorre un tempo $O(1)$ per inserire ciascuno dei $|V|$ vertici in testa alla lista concatenata.

16.4.1 Correttezza

Teorema 16.5. TOPOLOGICAL-SORT(G) *produce un ordinamento topologico di un grafo orientato aciclico G.*

Dimostrazione. Si suppone che la procedura DFS venga eseguita su un dato dag $G = (V, E)$ per determinare i tempi di completamento dei suoi vertici. È sufficiente dimostrare che se $(u, v) \in E$, allora $v.f < u.f$. Si considera un arco qualsiasi (u, v) ispezionato dalla procedura DFS. Quando l'arco viene ispezionato u è grigio. Il vertice v non può essere grigio, perché altrimenti v sarebbe un antenato di u e (u, v) sarebbe un arco all'indietro, contraddicendo il lemma precedente (dag non ha archi all'indietro). Quindi, il vertice v deve essere bianco o nero. Se v è bianco, diventa un discendente di u e, dal teorema delle parentesi, si ha $u.d < v.d < v.f < u.f$ cioè $v.f < u.f$. Se v è nero, la sua ispezione è stata già completata, quindi il valore di $v.f$ è già stato impostato. Poiché si sta ancora ispezionando dal vertice u, si deve ancora assegnare un'informazione temporale a $u.f$ e, quando verrà fatto, si avrà ancora $v.f < u.f$. Quindi, per qualsiasi arco (u, v) nel dag, si ha $v.f < u.f$, e questo dimostra il teorema. □

16.5 Componenti fortemente connesse

Una componente fortemente connessa di un grafo orientato $G = (V, E)$ è un insieme massimale di vertici $C \subseteq V$ tale che per ogni coppia di vertici u e v in C, si ha $u \rightsquigarrow v$ e $v \rightsquigarrow u$; ovvero i vertici u e v sono raggiungibili l'uno dall'altro.

L'algoritmo per trovare le componenti fortemente connesse di $G = (V, E)$ utilizza il grafo trasposto di G, che è definito come il grafo $G^T = (V, E^T)$, dove $E^T = \{(u, v) : (v, u \in E)\}$. Ovvero E^T è formato dagli archi di G con direzioni inverse. Data una rappresentazione con liste di adiacenza di G, il tempo richiesto per creare G^T è $O(V + E)$. I grafi G e G^T hanno esattamente le stesse componenti fortemente connesse: i vertici u e v sono raggiungibili l'uno dall'altro in G, se e soltanto se sono raggiungibili l'uno dall'altro in G^T.

L'idea che sta alla base dell'algoritmo per il calcolo delle componenti fortemente connesse di un grafo deriva da una proprietà fondamentale del **grafo delle componenti** $G^{SCC} = (V^{SCC}, E^{SCC})$. Supponendo che un grafo G abbia le componenti fortemente connesse C_1, C_2, ..., C_k, l'insieme dei vertici V^{SCC} è $\{v_1, v_2, ..., v_k\}$ e contiene un vertice v_i per ogni componente fortemente connessa C_i di G. Esiste un arco $(v_i, v_j) \in E^{SCC}$ se G contiene un arco orientato (x, y) per qualche $x \in C_i$ e qualche $y \in C_j$. Ovvero, E^{SCC} ha un arco se c'è un arco fra le corrispondenti componenti fortemente connesse di G. In altri termini, contraendo tutti gli archi i cui vertici incidenti sono all'interno della stessa componente fortemente connessa di G, si ottiene G^{SCC}.

La proprietà fondamentale è che il grafo delle componenti fortemente connesse è un grafo orientato aciclico, cioè un dag, che implica il seguente lemma

Lemma 16.1. *Siano C e C' due componenti fortemente connesse distinte nel grafo orientato G. Se $u, v \in C$ e $u', v' \in C'$ e supponendo che ci sia un cammino $u \rightsquigarrow u'$ in G, allora non può esistere anche un cammino $v' \rightsquigarrow v$ in G.*

Dimostrazione. Se esiste un cammino $v' \rightsquigarrow v$ in G, allora esistono i cammini $u \rightsquigarrow u' \rightsquigarrow v'$ e $v' \rightsquigarrow v \rightsquigarrow u$ in G. Quindi, u e v' sono raggiungibili l'uno dall'altro, contraddicendo l'ipotesi che C e C' siano SCC e distinte. □

L'algoritmo STRONGLY-CONNECTED-COMPONENTS calcola con tempo lineare $\Theta(V + E)$ le componenti fortemente connesse di un grafo orientato G utilizzando due visite in profondità, una su G e una su G^T.

STRONGLY-CONNECTED-COMPONENTS(G)
1 Chiama DFS(G) per calcolare i tempi di completamento $v.f$
  per ogni vertice v
2 Calcola G^T
3 Chiama DFS(G^T), nel ciclo in DFS considera i vertici in
  ordine decrescente rispetto ai tempi $u.f$ (calcolati nella riga 1)
4 Genere l'output dei vertici di ciascun albero della foresta DF
  che è stata prodotta nella riga 3 come una singola
  componente fortemente connessa

Esaminando i vertici nella seconda visita in profondità in ordine decrescente rispetto ai tempi di completamento che sono stati calcolati nella prima visita in profondità si stanno visitando i vertici del grafo delle componenti (corrispondenti a una componente fortemente connessa) secondo un ordinamento topologico.

I valori $u.d$ e $u.f$ si riferiscono ai tempi di scoperta e di completamento che sono stati calcolati dalla prima visita in profondità nella riga 1. Si estende la notazione dei tempi di scoperta e di completamento agli insiemi di vertici. Se $U \subseteq V$, si definiscono

$$d(U) = \min_{u \in U}\{u.d\}$$

e

$$f(U) = \max_{u \in U}\{u.f\}$$

Ovvero $d(U)$ e $f(U)$ sono, rispettivamente, il primo tempo di scoperta e l'ultimo tempo di completamento di un vertice qualsiasi in U.

16.5.1 Teoremi sulle SCC

Teorema 16.6. *Siano C e C' delle componenti fortemente connesse e distinte nel grafo orientato $G = (V, E)$. Si suppone che esista un arco $(u, v) \in E$, dove $u \in C$ e $v \in C'$. Allora $f(C) > f(C')$.*

Dimostrazione. Due casi, a seconda di quale componente fortemente connessa, C o C', contiene il primo vertice che viene scoperto durante la visita in profondità. Se $d(C) < d(C')$, si indica con x il primo vertice scoperto in C. Al tempo $x.d$, tutti i vertici in C e C' sono bianchi. Esiste un cammino in G da x a ciascun vertice di C che è formato soltanto da vertici bianchi. Poiché $(u, v) \in E$, per un vertice qualsiasi $w \in C'$, al tempo $x.d$ esiste anche un cammino da x a w in G che è formato soltanto da vertici bianchi: $x \rightsquigarrow u \to v \rightsquigarrow w$. Per il

teorema del cammino bianco, tutti i vertici in C e C' diventano discendenti di x nell'albero DF. Per il teorema delle parentesi: $x.f = f(C) > f(C')$. Se, invece, $d(C) > d(C')$, si indica con y il primo vertice scoperto in C'. Al tempo $y.d$, tutti i vertici in C' sono bianchi ed esiste un cammino in G da y a ciascun vertice in C' che è formato soltanto da vertici bianchi. Per il teorema del cammino bianco, tutti i vertici in C' diventano discendenti di y nell'albero DF e $y.f = f(C')$. Al tempo $y.d$, tutti i vertici in C sono bianchi. Poiché c'è un arco (u, v) da C a C', per il lemma (16.1) non può esistere un cammino da C' a C. Pertanto, nessun vertice in C è raggiungibile da y. Al tempo $y.f$ quindi tutti i vertici in C sono ancora bianchi. Dunque, per un vertice qualsiasi $w \in C$, si ha $w.f > y.f$ e questo implica che $f(C) > f(C')$. □

Corollario 16.2. *Siano C e C' due componenti fortemente connesse e distinte nel grafo orientato $G = (V, E)$. Si suppone che esista un arco $(u, v) \in E^T$, dove $u \in C$ e $v \in C'$. Allora $f(C) < f(C')$.*

Dimostrazione. Poiché $(u, v) \in E^T$, si ha $(v, u) \in E$. Poiché le componenti fortemente connesse di G e G^T sono le stesse, si ottiene $f(C) < f(C')$. □

Corollario 16.3. *Siano C e C' due componenti fortemente connesse nel grafo orientato G. Si suppone $f(C) > f(C')$. Allora non ci può essere un arco tra C e C' in G^T.*

Dimostrazione. Supponendo per assurdo che esista l'arco, allora si avrebbe $f(C) < f(C')$ (per il corollario) contraddicendo l'ipotesi $f(C) > f(C')$. □

16.5.2 Correttezza

La DFS su G^T all'interno di STRONGLY-CONNECTED-COMPONENTS inizia con la componente fortemente connessa C tale che $f(C)$ è massimo. La visita inizia da $x \in C$ e visita tutti i vertici in C. Poiché, da corollario, $f(C) > f(C')$ per ogni $C' \neq C$, non ci sono archi da C a C' in G^T quindi DFS visita solo i vertici in C. Il vertice successivo è $y \in C'$ tale che $f(C')$ è massimo in tutte le componente fortemente connesse diverse da C. DFS visita tutti i vertici in C' e i soli archi che escono da C' vanno in C che però è già stato visitato (è tutto nero). Si continua induttivamente per ogni componente fortemente connessa. Quando si sceglie un vertice si possono raggiungere solo i vertici nella sua componente fortemente connessa oppure vertici in componenti fortemente connesse già visitate nel secondo DFS.

Capitolo 17

Strutture dati per insiemi disgiunti

Alcune applicazioni richiedono di raggruppare n elementi distinti in una collezione di insiemi disgiunti, la possibilità di trovare l'unico insieme che contiene un determinato elemento e unire due insiemi.

Una **struttura dati per insiemi disgiunti** è una struttura dati che mantiene una collezione $S = \{S_1, S_2, \ldots, S_k\}$ di insiemi dinamici disgiunti. Ciascun insieme è identificato da un **rappresentante** che è un elemento dell'insieme. In alcune applicazioni non è importante quale elemento sarà utilizzato come rappresentante; l'unica condizione che si impone è che, se si richiede due volte il rappresentante di un insieme dinamico disgiunto senza modificare l'insieme fra le due richieste, si deve ottenere la stessa risposta entrambe le volte. In altre applicazioni ci potrebbe invece essere una regola prestabilita per sceglierlo.

17.1 Operazioni con gli insiemi disgiunti

Ogni elemento di un insieme è rappresentato da un oggetto. Indicando con x un oggetto, si vogliono supportare una serie di operazioni:

MAKE-SET(x) crea un nuovo insieme il cui unico elemento (e rappresentante) è x. Poiché gli insiemi sono disgiunti, x non può trovarsi in qualche altro insieme.

UNION(x, y) unisce gli insiemi dinamici che contengono x e y, per esempio S_x e S_y, in un nuovo insieme che è l'unione di questi due insiemi. Si suppone che i due insiemi siano disgiunti prima dell'operazione. Il rappresentante dell'insieme risultante è un elemento qualsiasi di $S_x \cup S_y$, sebbene molte implementazioni di UNION scelgano specificamente il rappresentante di S_x o quello di S_y come nuovo rappresentante. Poiché si richiede che gli insiemi nella collezione siano disgiunti, si distruggono gli insiemi S_x e S_y, eliminandoli dalla collezione S. Nella pratica, spesso gli elementi di uno degli insiemi vengono assorbiti dall'altro insieme.

FIND-SET(x) Restituisce un puntatore al rappresentante dell'insieme (unico) che contiene x.

I tempi di esecuzione delle strutture dati per gli insiemi disgiunti dipendono da due parametri: il numero n di operazioni MAKE-SET, ed il numero totale m di operazioni MAKE-SET, UNION e FIND-SET. Poiché gli insiemi sono disgiunti, ciascuna operazione UNION riduce di un'unità il numero degli insiemi. Dopo $n-1$ operazioni UNION, quindi, resta un solo insieme. Ne consegue che il numero di operazioni MAKE-SET sono incluse nel numero totale di operazioni m, allora $m \geq n$. Si suppone che le n operazioni MAKE-SET siano le prime n operazioni eseguite.

17.1.1 Applicazioni delle strutture dati per insiemi disgiunti

Una delle tante applicazioni delle strutture dati per insiemi disgiunti consiste nel determinare le componenti connesse di un grafo non orientato.

La procedura CONNECTED-COMPONENTS usa le operazioni degli insiemi disgiunti per calcolare le componenti connesse di un grafo. Dopo che CONNECTED-COMPONENTS ha preelaborato il grafo, la procedura SAME-COMPONENT è in grado di determinare se due vertici sono nella stessa componente connessa.

```
CONNECTED-COMPONENTS(G)
1  for ogni vertice v ∈ G.V
2      MAKE-SET(v)
3  for ogni arco (u, v) ∈ G.E
4      if FIND-SET(u) ≠ FIND-SET(v)
5          UNION(u, v)

SAME-COMPONENT(u, v)
1  if FIND-SET(u) = FIND-SET(v)
2      return TRUE
3  else return FALSE
```

Inizialmente, la procedura CONNECTED-COMPONENTS pone ciascun vertice v nel proprio insieme. Poi, per ogni arco (u, v), unisce gli insiemi che contengono u e v. Dopo che tutti gli archi sono stati elaborati, due vertici si trovano nella stessa componente connessa, se e soltanto se i corrispondenti oggetti si trovano nello stesso insieme. Dunque, CONNECTED-COMPONENTS calcola gli insiemi in modo tale che la procedura SAME-COMPONENTS possa determinare se due vertici si trovano nella stessa componente connessa.

17.2 Rappresentazione di insiemi disgiunti tramite liste concatenate

Un semplice modo di implementare una struttura dati per gli insiemi disgiunti consiste nel rappresentare ciascun insieme attraverso una lista concatenata. L'oggetto di ciascun insieme ha gli attributi *head*, che punta al primo oggetto della lista, e *tail*, che punta all'ultimo oggetto. Ogni oggetto nella lista contiene un elemento dell'insieme, un puntatore al successivo oggetto della lista e un puntatore che ritorna all'oggetto dell'insieme. All'interno di ciascuna lista

concatenata, gli oggetti possono apparire in qualsiasi ordine. Il rappresentante è l'elemento dell'insieme nel primo oggetto della lista.

Con questa rappresentazione mediante liste concatenate, entrambe le operazioni MAKE-SET e FIND-SET sono semplici da realizzare e richiedono un tempo $O(1)$. Per realizzare l'operazione MAKE-SET(x), si crea una nuova lista concatenata il cui unico oggetto è x. Per l'operazione FIND-SET(x), basta seguire il puntatore da x per arrivare all'oggetto del suo insieme e poi ritornare all'elemento nell'oggetto cui punta *head*.

17.2.1 Implementazione dell'operazione di unione

La più semplice implementazione dell'operazione UNION che usa la rappresentazione degli insiemi mediante liste concatenate richiede molto più tempo rispetto all'operazione MAKE-SET o FIND-SET. L'operazione UNION(x, y) aggiunge la lista di y alla fine della lista di x. Il rappresentante della lista x diventa il rappresentante dell'insieme risultante. Tuttavia, si deve aggiornare il puntatore all'oggetto dell'insieme per ogni oggetto che originariamente si trovava nella lista y; questo richiede un tempo lineare nella lunghezza della lista di y.

Indicato con m il numero totale di operazioni eseguite, si suppone di avere n oggetti $x_1, x_2, \ldots, x_n$, quindi $m \geq n$. Per n oggetti verranno eseguite n operazioni MAKE-SET seguite, per esempio, da $n-1$ operazioni UNION, quindi $m = 2n - 1$. Si impiega un tempo $\Theta(n)$ per eseguire le n operazioni MAKE-SET. Poiché l'i-esima operazione UNION aggiorna i oggetti, il numero totale di oggetti aggiornati da tutte le $n-1$ operazioni UNION è

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

Nel caso peggiore, la precedente implementazione della procedura UNION richiede un tempo medio $\Theta(n)$ per chiamata, perché si potrebbe appendere una lista più lunga a una più corta; si deve aggiornare il puntatore all'oggetto dell'insieme per ogni elemento della lista più lunga.

17.2.2 Euristica dell'unione pesata

Supponendo che ogni lista includa anche la lunghezza della lista e supponendo di appendere sempre la lista più piccola a quella più lunga, risolvendo in modo arbitrario i casi di liste aventi la stessa lunghezza. In questo modo (euristica dell'unione pesata), una singola operazione UNION può ancora impiegare un tempo $\Omega(n)$, se entrambi gli insiemi hanno $\Omega(n)$ elementi (entrambi gli insiemi hanno $n/2$ elementi). Tuttavia:

Teorema 17.1. *Una sequenza di m operazioni* MAKE-SET, UNION *e* FIND-SET, *su n elementi, impiega un tempo $O(m + n \lg n)$.*

Dimostrazione. Per ogni $k \leq n$, dopo che il puntatore di x viene aggiornato $\lceil \lg k \rceil$ volte, l'insieme risultante deve avere almeno k elementi. Dato che l'insieme più grande ha al più n elementi, il puntatore di ciascun oggetto è stato aggiornato al più $\lceil \lg n \rceil$ volte in tutte le operazioni UNION. Quindi il tempo totale speso per aggiornare i puntatori degli oggetti durante le operazioni UNION è $O(n \lg n)$. Per m operazioni si ottiene un tempo $O(m + n \lg n)$. □

Capitolo 18

Heap

Un **heap (binario)** è una struttura dati composta da un array che si può considerare come un albero binario quasi completo. Ogni nodo dell'albero corrisponde a un elemento dell'array. Tutti i livelli dell'albero sono completamente riempiti, tranne eventualmente l'ultimo che può essere riempito da sinistra verso destra fino a un certo punto. Un array A che rappresenta un heap è un oggetto con due attributi: $A.length$ indica il numero di elementi nell'array; $A.heap\text{-}size$ indica il numero degli elementi dell'heap che sono registrati nell'array. Cioè, anche se ci possono essere dei numeri memorizzati in tutto l'array $A[1 \ldots A.length]$, soltanto i numeri in $A[1 \ldots A.heap\text{-}size]$, dove $0 \leq A.heap\text{-}size \leq A.length$, sono elementi validi dell'heap. La radice dell'albero è $A[1]$. Se i è l'indice di un nodo, gli indici di suo padre PARENT(i), del figlio sinistro LEFT(i) e del figlio destro RIGHT(i) possono essere facilmente calcolati.

```
PARENT(i)
1  return ⌊i/2⌋

LEFT(i)
1  return 2i

RIGHT(i)
1  return 2i + 1
```

Nella maggior parte dei casi, la procedura LEFT può calcolare $2i$ con una sola istruzione, facendo scorrere semplicemente di una posizione a sinistra la rappresentazione binaria di i. Analogamente, la procedura RIGHT può rapidamente calcolare $2i + 1$, facendo scorrere la rappresentazione binaria di i di una posizione a sinistra e aggiungendo 1 come bit meno significativo. La procedura PARENT può calcolare $\lfloor i/2 \rfloor$ con uno scorrimento di una posizione a destra della rappresentazione di i.

Ci sono due tipi di heap binari: max-heap e min-heap. In entrambi i tipi, i valori nei nodi soddisfano una **proprietà dell'heap**, le cui caratteristiche dipendono dal tipo di heap. In un max-heap, la **proprietà del max-heap** è che per ogni nodo i diverso dalla radice, si ha:

$$A[\text{PARENT}(i)] \geq A[i]$$

ovvero il valore di un nodo è al massimo il valore di suo padre. Quindi, l'elemento più grande di un max-heap è memorizzato nella radice e il sottoalbero di un nodo contiene valori non maggiori di quello contenuto nel nodo stesso. Un min-heap è organizzato nel modo opposto; la **proprietà del min-heap** è che per ogni nodo i diverso dalla radice, si ha:

$$A[\textsc{Parent}(i)] \leq A[i]$$

Il più piccolo elemento in un min-heap è nella radice. I min-heap sono talvolta utilizzati nelle code di priorità.

Considerando un heap nella forma di albero, si definsice **altezza di un nodo** il numero di archi nel cammino semplice più lungo che dal nodo scende fino a una foglia. Si definisce **altezza di un heap** l'altezza della sua radice. Poiché un heap di n elementi è basato su un albero binario completo, la sua altezza è $\Theta(lg\, n)$. Le operazioni fondamentali sugli heap vengono eseguite in un tempo che è al massimo proporzionale all'altezza dell'albero e, quindi, richiedono un tempo $O(lg\, n)$.

- La procedura Max-Heapify, che è eseguita nel tempo $O(lg\, n)$, è la chiave per conservare la proprietà del max-heap
- La procedura Build-Max-Heap, che è eseguita in tempo lineare, genera un max-heap da un array di input non ordinato
- La procedura Heapsort, che è eseguita nel tempo $O(n\, lg\, n)$, ordina sul posto un array
- Le quattro procedure Max-Heap-Insert, Heap-Extract-Max, Heap-Increase-Key e Heap-Maximum, che sono eseguite nel tempo $O(lg\, n)$, consentono a un heap di essere utilizzato come una coda di priorità

In un array che rappresenta un heap di n elementi, le foglie sono i nodi con indici $\lfloor n/2 \rfloor + 1$, $\lfloor n/2 \rfloor + 2$, ... , n.

18.1 Conservare la proprietà dell'heap

Per mantenere la proprietà di max-heap si utilizza la procedura Max-Heapify. I suoi input sono un array A e un indice i dell'array. Quando viene chiamata, Max-Heapify assume che gli alberi binari con radici in $\textsc{Left}(i)$ e $\textsc{Right}(i)$ siano max-heap, ma che $A[i]$ possa essere più piccolo dei suoi figli, violando così la proprietà del max-heap. Max-Heapify fa scendere il valore $A[i]$ nel max-heap in modo che il sottoalbero con radice di indice i diventi un max-heap.

A ogni passo, viene determinato il più grande tra gli elementi $A[i]$, $A[\textsc{Left}(i)]$ e $A[\textsc{Right}(i)]$; il suo indice viene memorizzato in *massimo*. Se $A[i]$ è più grande, allora il sottoalbero con radice nel nodo i è un max-heap e la procedura termina. Altrimenti, uno dei due figli ha l'elemento più grande e $A[i]$ viene scambiato con $A[massimo]$; in questo modo, il nodo i e i suoi figli soddisfano la proprietà del max-heap. Il nodo con indice *massimo*, però, adesso ha il valore originale $A[i]$ e, quindi, il sottoalbero con radice in *massimo* potrebbe violare la proprietà del max-heap. Di conseguenza, deve essere chiamata ricorsivamente la subroutine Max-Heapify per questo sottoalbero.

```
MAX-HEAPIFY(A, i)
1   l ← LEFT(i)
2   r ← RIGHT(i)
3   if l ≤ A.heap-size and A[l] > A[i]
4       massimo ← l
5   else massimo ← i
6   if r ≤ A.heap-size and A[r] > A[massimo]
7       massimo ← r
8   if massimo ≠ i
9       scambia A[i] con A[massimo]
10      MAX-HEAPIFY(A.massimo)
```

Il tempo di esecuzione di MAX-HEAPIFY in un sottoalbero di dimensione n con radice in un nodo i è pari al tempo $\Theta(1)$ per sistemare le relazioni fra gli elementi $A[i]$, $A[\text{LEFT}(i)]$ e $A[\text{RIGHT}(i)]$, più il tempo per eseguire MAX-HEAPIFY in un sottoalbero con radice in uno dei figli del nodo i. I sottoalberi dei figli hanno ciascuno una dimensione che non supera $2n/3$ - il caso peggiore si verifica quando l'ultima riga dell'albero è piena esattamente a metà - e il tempo di esecuzione di MAX-HEAPIFY può quindi essere descritto dalla ricorrenza

$$T(n) \leq T(2n/3) + \Theta(1)$$

La soluzione di questa ricorrenza, per il secondo caso del teorema dell'esperto, è $T(n) = O(lg\, n)$. In alternativa, si può indicare con $O(h)$ il tempo di esecuzione di MAX-HEAPIFY in un nodo di altezza h.

18.2 Costruire un heap

Si può utilizzare la procedura MAX-HEAPIFY dal basso verso l'alto per convertire un array $A[1 \ldots n]$ con $n = A.length$, in un max-heap. Tutti gli elementi nel sottoarray $A[(\lfloor n/2 \rfloor + 1) \ldots n]$ sono foglie dell'albero e quindi ciascuno di essi è un heap di un solo elemento che si può usare come punto di partenza. La procedura BUILD-MAX-HEAP attraversa i restanti nodi dell'albero ed esegue MAX-HEAPIFY in ciascuno di essi.

```
BUILD-MAX-HEAP(A)
1   A.heap-size ← A.length
2   for i = ⌊A.length/2⌋ downto 1
3       MAX-HEAPIFY(A, i)
```

Per verificare che BUILD-MAX-HEAP funziona correttamente, si utilizza la seguente invariante di ciclo:

> *All'inizio di ogni iterazione del ciclo* ***for****, righe 2 - 3, ogni nodo* $i+1, i+2, \ldots, n$ *è la radice di un max-heap*

Ogni chiamata di MAX-HEAPIFY costa un tempo $O(lg\, n)$ e ci sono $O(n)$ di queste chiamate. Quindi, il tempo di esecuzione è $O(n\, lg\, n)$.

18.3 Code di priorità

Una delle applicazioni più diffuse dell'heap consiste nell'implementazione delle code di priorità. Analogamente agli heap, ci sono due tipo di code di priorità: code di max-priorità e code di min-priorità.

Una **coda di priorità** è una struttura dati che serve a mantenere un insieme S di elementi, ciascuno con un valore associato detto **chiave**. In particolare, una **coda di max-priorità** supporta le seguenti operazioni:

INSERT(S, x) inserisce l'elemento x nell'insieme S, che equivale all'operazione: $S = S \cup \{x\}$.

MAXIMUM(S) restituisce l'elemento di S con la chiave più grande.

EXTRACT-MAX(S) rimuove e restituisce l'elemento di S con la chiave più grande.

INCREASE-KEY(S, x, k) aumenta il valore della chiave dell'elemento x al nuovo valore k, che si suppone sia almeno grande quanto il valore corrente della chiave dell'elemento x.

Tra le applicazioni delle code di max-priorità vi è quella di programmare i lavori su un computer condiviso.

In alternativa, una **coda di min-priorità** supporta le operazioni INSERT, MINIMUM, EXTRACT-MIN e DECREASE-KEY. Una coda di min-priorità può essere utilizzata in un simulatore controllato da eventi.

La procedura HEAP-MAXIMUM implementa l'operazione MAXIMUM nel tempo $\Theta(1)$.

```
HEAP-MAXIMUM(A)
1  return A[1]
```

La procedura HEAP-EXTRACT-MAX implementa l'operazione EXTRACT-MAX.

```
HEAP-EXTRACT-MAX(A)
1  if A.heap-size < 1
2      error "underflow dell'heap"
3  max ← A[1]
4  A[1] ← A[A.heap-size]
5  A.heap-size ← A.heap-size - 1
6  MAX-HEAPIFY(A, 1)
7  return max
```

Il tempo di esecuzione di HEAP-EXTRACT-MAX è $O(lg\, n)$, perché svolge una quantità costante di lavoro oltre al tempo $O(lg\, n)$ di MAX-HEAPIFY.

HEAP-INCREASE-KEY implementa l'operazione INCREASE-KEY. L'elemento della coda di priorità la cui chiave deve essere aumentata è identificato da un indice i nell'array. Innanzitutto, la procedura aggiorna la chiave dell'elemento $A[i]$ con il suo nuovo valore. Successivamente, poiché l'aumento della chiave di $A[i]$ potrebbe violare la proprietà del max-heap, la procedura segue un cammino semplice da questo nodo verso la radice per trovare un posto appropriato alla

nuova chiave. Durante questo attraversamento, HEAP-INCREASE-KEY confronta ripetutamente un elemento con suo padre e scambia le loro chiavi se la chiave dell'elemento è più grande; questa operazione termina se la chiave dell'elemento è più piccola, perché in questo caso la proprietà del max-heap è soddisfatta.

```
HEAP-INCREASE-KEY(A, i, key)
1   if key < A[i]
2       error "la nuova chiave è più piccola di quella corrente"
3   A[i] ← key
4   while i > 1 and A[PARENT(i)] < A[i]
5       scambia A[i] con A[PARENT(i)]
6       i ← PARENT(i)
```

Il tempo di esecuzione di HEAP-INCREASE-KEY con un heap di n elementi è $O(lg\, n)$, in quanto il cammino verso la radice percorso dal nodo aggiornato nella riga 3 ha lunghezza $O(lg\, n)$.

La procedura MAX-HEAP-INSERT implementa l'operazione INSERT. Prende come input la chiave del nuovo elemento da inserire nel max-heap A. La procedura prima espande il max-heap aggiungendo all'albero una nuova foglia la cui chiave è $-\infty$; poi chiama HEAP-INCREASE-KEY per impostare la chiave di questo nuovo nodo al su valore corretto e mantenere la proprietà del max-heap.

```
MAX-HEAP-INSERT(A, key)
1   A.heap-size ← A.heap-size + 1
2   A[A.heap-size] ← −∞
3   HEAP-INCREASE-KEY(A, A.heap-size, key)
```

Il tempo di esecuzione della procedura MAX-HEAP-INSERT con un heap di n elementi è $O(lg\, n)$. In sintesi, un heap può svolgere ciascuna operazione con le code di priorità nel tempo $O(lg\, n)$ su un insieme di dimensione n.

Capitolo 19

Alberi di connessione minimi

Dato un grafo connesso non orientato $G = (V, E)$, ad ogni arco $(u, v) \in E$ è associato un peso $w(u, v)$. Si vuole trovare un sottoinsieme aciclico $T \subseteq E$ che collega tutti i vertici, il cui peso totale

$$w(T) = \sum_{(u,v)\in T} w(u, v)$$

sia minimo. Poiché T è aciclico e collega tutti i vertici, deve anche formare un albero, che è detto **albero di connessione minimo** perché connette il grafo G (può non essere unico). Il problema di trovare l'albero T è detto **problema dell'albero di connessione minimo**. L' albero di connessione minimo ha $|V| - 1$ archi e non ha cicli.

Tra gli algoritmi che risolvono il problema dell'albero di connessione minimo ci sono l'algoritmo di Kruskal e l'algoritmo di Prim. Entrambi gli algoritmi possono essere eseguiti nel tempo $O(E \lg V)$ utilizzando dei normali heap binari. I due algoritmi sono algoritmi golosi, ma applicano l'approccio goloso in modo differente. La strategia golosa prevede di fare la scelta che in quel particolare momento è ritenuta la migliore. In generale, questa strategia non garantisce che vengano trovate le soluzioni globalmente ottime per i sottoproblemi. Tuttavia, per il problema dell'albero di connessione minimo, è possibile dimostrare che alcune strategie golose forniscono un albero di connessione con peso minimo.

19.1 Creare un albero di connessione minimo

Si suppone di avere un grafo connesso non orientato $G = (V, E)$ con una funzione peso $w : E \to \mathbb{R}$ e di volere trovare un albero di connessione minimo per il grafo G. La strategia golosa utilizzata dagli algoritmi di Kruskal e Prim può essere sintetizzata in un metodo generico che fa crescere l'albero di connessione minimo di un arco alla volta. Il metodo generico gestisce un insieme di archi A (inizialmente $A = \emptyset$), conservando la seguente invariante di ciclo:

> *Prima di ogni iterazione, A è un sottoinsieme di qualche albero di connessione minimo*

Ad ogni passo, si determina un arco (u, v) che può essere aggiunto ad A senza violare questa invariante, nel senso che $A \cup \{(u, v)\}$ è anche un sottoinsieme di un albero di connessione minimo. Tale arco è detto **arco sicuro** per A, perché può essere tranquillamente aggiunto ad A preservando l'invariante.

```
GENERIC-MST(G, w)
1  A ← ∅
2  while A non forma un albero di connessione
3     trova un arco (u, v) che è sicuro per A
4     A ← A ∪ {(u, v)}
5  return A
```

Per verificare la sua correttezza si utilizza l'invariante di ciclo:

Inizializzazione Dopo la riga 1, l'insieme A soddisfa l'invariante di ciclo perché l'insieme vuoto è un sottoinsieme di MST (minimum spanning tree)

Conservazione Il ciclo nelle righe 2 - 4 conserva l'invariante perché aggiunge soltanto archi sicuri; quindi A rimane un sottoinsieme di MST

Conclusione Tutti gli archi aggiunti ad A si trovano in un albero di connessione minimo, quindi l'insieme A restituito nella riga 5 deve essere un albero di copertura che è anche un albero di connessione minimo

Un arco sicuro deve esistere, perché quando viene eseguita la riga 3, l'invariante impone che ci sia un albero di connessione T tale che $A \subseteq T$. All'interno del corpo del ciclo **while**, A deve essere un sottoinsieme proprio di T, quindi deve esistere un arco $(u, v) \in T$ tale che $(u, v) \notin A$ e (u, v) è un arco sicuro per A.

Un **taglio** (S, V - S) di un grafo connesso non orientato $G = (V, E)$ è una partizione di V in due insiemi disgiunti S e S - V, tra loro complementari. Si dice che un arco $(u, v) \in E$ **attraversa** il taglio (S, V - S) se una delle sue estremità si trova in S e l'altra in V - S.

Si dice che un taglio **rispetta** un insieme A di archi se nessun arco di A attraversa il taglio. Un arco è un **arco leggero** per un taglio se il suo peso è minimo fra i pesi degli archi che attraversano il taglio. Ci possono essere più archi leggeri che attraversano un taglio nel caso di pesi uguali. Più in generale, un arco è un arco leggero per una data proprietà se il suo peso è il minimo fra tutti gli archi che soddisfano tale proprietà.

Teorema 19.1. *Sia $G = (V, E)$ un grafo non orientato con una funzione peso w a valori reali definita in E. Sia A un sottoinsieme di un qualche albero di connessione minimo per G, sia (S, V - S) un taglio qualsiasi di G che rispetta A e sia (u, v) un arco leggero che attraversa (S, V - S). Allora, l'arco (u, v) è sicuro per A.*

Dimostrazione. Sia T un albero di connessione minimo che contiene A, si suppone che T non contenga l'arco leggero (u, v), perché se lo contenesse il teorema sarebbe dimostrato. Si costruisce un altro albero di connessione minimo T' che include $A \cup \{(u, v)\}$, dimostrando così che (u, v) è un arco sicuro per A.

L'arco (u, v) forma un ciclo con gli archi nel cammino semplice p che va da u a v in T. Poiché u e v si trovano su lati opposti del taglio (S, V - S), c'è almeno un altro arco in T che appartiene al cammino semplice p e che

attraversa il taglio. Sia (x, y) uno di questi archi. L'arco (x, y) non appartiene ad A, perché il taglio rispetta A. Poiché (x, y) si trova nel cammino semplice unico da u a v in T, eliminando (x, y), l'albero T si spezza in due componenti. Aggiungendo (u, v), le due componenti si ricongiungono per formare un nuovo albero di connessione $T' = T - \{(x, y)\} \cup \{(u, v)\}$.

Si dimostra adesso che T' è un albero di connessione minimo. Poiché (u, v) è un arco leggero che attraversa (S, V - S) e anche l'arco (x, y) attraversa questo taglio, allora $w(u, v) \leq w(x, y)$. Quindi si ha:

$$\begin{aligned} w(T') &= w(T) - w(x, y) + w(u, v) \\ &\leq w(T) \end{aligned}$$

Ma T è un albero di connessione minimo, quindi $w(T) \leq w(T')$; di conseguenza, anche T' (che è un albero di copertura) deve essere un albero di connessione minimo.

Si deve ora dimostrare che (u, v) è effettivamente un arco sicuro per A. Si sa che $A \subseteq T'$, in quanto $A \subseteq T$ e $(x, y) \notin A$; quindi $A \cup \{(u, v)\} \subseteq T'$. Di conseguenza, poiché T' è un albero di connessione minimo, (u, v) è un arco sicuro per A. □

Durante l'esecuzione del metodo, l'insieme A è sempre aciclico; altrimenti un albero di connessione minimo che include A conterebbe un ciclo, e ciò sarebbe una contraddizione. In qualsiasi momento dell'esecuzione, il grafo $G_A = (V, A)$ è una foresta e ciascuna delle componenti connesse di G_A è un albero (inizialmente sono solo dei nodi, via via si costruiscono gli alberi). Inoltre, qualsiasi arco sicuro (u, v) per A collega componenti distinte di G_A, perché $A \cup \{(u, v)\}$ deve essere aciclico.

Il ciclo **while** nelle righe 2 - 4 di GENERIC-MST viene eseguito $|V| - 1$ volte, in quanto i $|V| - 1$ archi di un albero di connessione minimo vengono determinati uno dopo l'altro. Inizialmente, quando $A = \emptyset$, ci sono $|V|$ alberi in G_A e ogni iterazione riduce questo numero di uno. Quando la foresta contiene un albero soltanto, il metodo termina.

Corollario 19.1. *Sia $G = (V, E)$ un grafo connesso non orientato con una funzione peso w a valori reali definita in E. Sia A un sottoinsieme di E che è contenuto in qualche albero di connessione minimo per G e sia $C = (V_C, E_C)$ una componente connessa (un albero) nella foresta $G_A = (V, A)$. Se (u, v) è un arco leggero che collega C a qualche altra componente in G_A, allora (u, v) è sicuro per A.*

Dimostrazione. Il taglio (V_C, $V - V_C$) rispetta A e (u, v) è un arco leggero per questo taglio. Quindi (u, v) è sicuro per A. □

19.2 Algoritmo di Kruskal

Gli algoritmi di Kruskal e Prim (elaborazioni del metodo generico) usano una regola specifica per determinare un arco sicuro nella riga 3 di GENERIC-MST. In Kruskal l'insieme A è una foresta i cui vertici sono tutti quelli del grafo. L'arco sicuro aggiunto ad A è sempre un arco di peso minimo nel grafo che collega due componenti distinte. In Prim l'insieme A è sempre un arco di peso minimo che collega l'albero con un vertice che non appartiene all'albero.

L'algoritmo di Kruskal trova un arco sicuro da aggiungere alla foresta in costruzione scegliendo, fra tutti gli archi che collegano due alberi qualsiasi nella foresta, un arco (u, v) di peso minimo. Si indicano con C_1 e C_2 i due alberi che sono collegati da (u, v). Poiché (u, v) deve essere un arco leggero che collega C_1 a qualche altro albero, il corollario (19.1) implica che (u, v) è un arco sicuro per C_1. L'algoritmo di Kruskal è un algoritmo goloso, perché a ogni passo aggiunge alla foresta un arco con il minor peso possibile.

Usa una struttura dati per insiemi disgiunti per mantenere vari insiemi disgiunti di elementi. Ogni insieme contiene i vertici di un albero della foreste corrente. L'operazione FIND-SET(u) restituisce un rappresentante dell'insieme che contiene u. Quindi, si può determinare se due vertici u e v appartengono allo stesso albero verificando se FIND-SET(u)è uguale a FIND-SET(v). L'unione degli alberi è effettuata dalla procedura UNION.

```
MST-KRUSKAL(G, w)
1   A ← ∅
2   for ogni vertice v ∈ G.V
3       MAKE-SET(v)
4   ordina gli archi di G.E in senso non decrescente rispetto al peso w
5   for ogni arco (u, v) ∈ G.E, preso in ordine di peso non decrescente
6       if FIND-SET(u) ≠ FIND-SET(v)
7           A ← A ∪ {(u, v)}
8           UNION(u, v)
9   return A
```

Commenti:

3 Crea una componente connessa per ogni vertice

4 Se alcuni hanno lo stesso peso stanno nello stesso insieme

6 Se fosse uguale non sarebbe più un albero

Si potrebbe aggiungere un test per fermarsi dopo $|V| - 1$ operazioni UNION. Le righe 1 - 3 inizializzano l'insieme A come un insieme vuoto e creano $|V|$ alberi, uno per ogni vertice. Il ciclo **for** nelle righe 5 - 8 esamina gli archi nell'ordine dal più leggero al più pesante. Il ciclo verifica, per ogni arco (u, v), se l'estremità u e v appartengono allo stesso albero; in caso affermativo, l'arco (u, v) non può essere aggiunto alla foresta senza generare un ciclo, quindi l'arco viene scartato. Altrimenti, i due vertici appartengono ad alberi differenti. In questo caso, l'arco (u, v) viene aggiunto ad A nella riga 7 e i vertici dei due alberi vengono fusi nella riga 8. L'inizializzazione dell'insieme A nella riga 1 richiede un tempo $O(1)$; il tempo per ordinare gli archi nella riga 4 è $O(E \, lg \, E)$. Il ciclo **for** nelle righe 5 - 8 esegue $O(E)$ operazioni FIND-SET e UNION sulla foresta degli insiemi disgiunti. Si tiene conto anche delle $|V|$ operazioni MAKE-SET. Si ottiene un tempo totale pari a $O(E \, lg \, V)$.

19.3 Algoritmo di Prim

L'algoritmo di Prima ha la proprietà che gli archi nell'insieme A formano sempre un albero singolo. L'albero inizia da un arbitrario vertice radice r e si sviluppa

fino a coprire tutti i vertici in V.A ogni passo viene aggiunto all'albero A un arco leggero che collega A con un vertice isolato - un vertice che non sia estremo di qualche arco in A. Per il corollario (19.1), questa regola aggiunge soltanto gli archi che sono sicuri per A; cosicché, quando l'algoritmo termina, gli archi in A formano un albero di connessione minimo. Questa strategia è golosa perché l'albero cresce includendo ad ogni passo un arco che contribuisce con la quantità più piccola possibile a formare il peso dell'albero.

Nella procedura MST-PRIM, il grafo connesso G e la radice r dell'albero di connessione minimo da costruire sono gli input per l'algoritmo. Durante l'esecuzione dell'algoritmo, tutti i vertici che non si trovano nell'albero risiedono in una coda di min-priorità Q basata su un campo *key*. Per ogni vertice v, l'attributo $v.key$ è il peso minimo di un arco qualsiasi che collega v a un vertice nell'albero; per convenzione, $v.key = \infty$ se tale arco non esiste. L'attributo $v.\pi$ indica il padre di v nell'albero. Quando l'algoritmo termina, la coda di min-priorità Q è vuota.

```
MST-PRIM(G, w, r)
1   for ogni u ∈ G.V
2       u.key ← ∞
3       u.π ← NIL
4   r.key ← 0
5   Q ← G.V
6   while Q ≠ ∅
7       u ← EXTRACT-MIN(Q)
8       for ogni v ∈ G.Adj[u]
9           if v ∈ Q and w(u, v) < v.key
10              v.π ← u
11              v.key ← w(u, v)
```

Commenti:

3 Peso minimo per raggiungere ciascun vertice dell'insieme (albero)

4 All'inizio nessun vertice è raggiungibile, sono quindi tutti ∞

7 Alla prima iterazione estrae la radice r

Le righe 1 - 5 impostano la chiave di ciascun vertice a ∞ ad eccezione della radice r, la cui chiave è impostata a 0 (in questo modo la radice sarà il primo vertice ad essere elaborato), assegnano al padre di ciascun vertice il valore NIL e inizializzano la coda di min-priorità Q in modo che contenga tutti i vertici. La riga 7 identifica un vertice $u \in Q$ incidente su un arco leggero che attraversa la taglio (V - Q, Q) (ad eccezione della prima iterazione in cui $u = r$ per la riga 4). Quando il vertice u viene eliminato dall'insieme Q, viene aggiunto all'insieme $V - Q$ dei vertici dell'albero, quindi $(u, u.\pi)$ viene aggiunto ad A. Il ciclo **for** nelle righe 8 - 11 aggiorna i campi *key* e π di qualsiasi vertice v adiacente a u, ma che non appartiene all'albero.

Le prestazioni dell'algoritmo di Prim dipendono dal modo in cui viene implementata la coda di min-priorità Q. Se Q è implementata come un min-heap binario, si può utilizzare la procedura BUILD-MIN-HEAP per eseguire nel tempo $O(V)$ l'inizializzazione nelle righe 1 - 5. Il corpo del ciclo **while** viene eseguito $|V|$ volte e, poiché ogni operazione EXTRACT-MIN, DECREASE-KEY e

INSERT richiedono un tempo $O(lg\,V)$, il tempo totale per tutte le chiamate di EXTRACT-MIN è $O(V\,lg\,V)$. Il ciclo **for** (righe 8 - 11) viene eseguito $O(E)$ volte complessivamente, in quanto la somma delle lunghezze di tutte le liste di adiacenza è $2|E|$. L'assegnazione nella riga 11 richiede implicitamente un'operazione DECREASE-KEY sul min-heap (eseguita al massimo $|E|$ volte), che può essere implementata con un min-heap binario nel tempo $O(lg\,V)$. Quindi, il tempo totale dell'algoritmo di Prim è $O(V\,lg\,V + E\,lg\,V) = O(E\,lg\,V)$, che è asintoticamente uguale a quello dell'implementazione di Kruskal.

L'algoritmo di Prim conserva la seguente invariante di ciclo:

*Prima di ogni iterazione del ciclo **while** (righe 6 - 11)*:

- $A = \{(v,\, v, \pi) : v \in V - \{r\} - Q\}$
- I vertici già inseriti nell'albero di connessione minimo sono quelli che appartengono a $V - Q$
- Per ogni vertice $v \in Q$, se $v.\pi \neq$ NIL, allora $v.key < \infty$ e $v.key$ è il peso di un arco leggero $(v,\, v, \pi)$ che collega v a qualche vertice che si trova già nell'albero di connessione minimo (insieme A)

Alla fine, $V_A = V$, quindi $Q = \emptyset$. e l'albero di connessione minimo è $A = \{(v,\, v, \pi) : v \in V - \{r\}\}$.