UNIVERSITÀ DEGLI STUDI DI MILANO
Dipartimento di Tecnologie dell'Informazione
Polo Didattico e di Ricerca di Crema

Algoritmi Iterativi

Appunti del corso di
Complementi di Algoritmi
del Professor Roberto Cordone

Vittoria Polimeni (670659)

– Anno Accademico 2005/06 –

Indice

Prefazione

Queste note, che si devono alla certosina pazienza di Vittoria Polimeni, nascono da una mia triplice insoddisfazione di fondo rispetto al modo in cui molti studenti di Informatica si abituano a concepire gli algoritmi.

Primo: molti studenti descrivono gli algoritmi partendo dalle strutture dati. Il fatto è che parecchi validi testi di Algoritmi e Strutture Dati sono in realtà testi di Strutture Dati e Algoritmi, giacché si dedicano ai secondi solo dopo aver diffusamente trattato le prime. L'approccio è intuitivo (per imparare a far case, si impari prima a far mattoni) e crea un legame con il corso di Programmazione. Ma induce la falsa idea che le strutture dati siano il cuore degli algoritmi, quando invece ne determinano tutt'al più la complessità, non certo la correttezza. Tant'è che i matematici greci facevano algoritmi molto prima che di strutture dati si sentisse il bisogno, e che non si suole ribattezzare un algoritmo quando se ne mutano le strutture dati.

Secondo: molti studenti sanno come *un algoritmo funziona, ma non* perché*. Vedono la dimostrazione di correttezza (dove ancora si insegni) come un orpello ostico e immotivato, e dunque la dimenticano subito. Tuttavia, imparare a memoria una sequenza di istruzioni senza coglierne l'intima logica è la via migliore per ricordarla in modo inesatto e disseminare il proprio codice di bachi.*

Terzo: molti studenti credono che algoritmi e programmazione siano sinonimi. Di qui la loro difficoltà a ricordare gli algoritmi, a descriverli, ad analizzarli, a progettarli, il cui sintomo rivelatore è la confusione fra codice e pseudocodice.

Queste note intendono porre gli algoritmi al cuore del discorso, presentandoli come macchine per manipolare oggetti matematici*. Come ogni macchina, un algoritmo funziona perché le sue parti svolgono il proprio ruolo e interagiscono attraverso una struttura corretta. Se non funziona, o una parte non svolge il suo ruolo o la struttura complessiva è errata: di qui, una comoda lista dei controlli da fare per scoprire l'errore. Poiché manipolano oggetti matematici, gli algoritmi si possono realizzare fisicamente come codice in un linguaggio di programmazione, circuiti in una scheda elettronica, schemi in un sistema di ingranaggi o segni tracciati su carta, ma non sono nulla di tutto questo:* gli algoritmi sono idee*.*

La visione è certo astratta, e dapprima faticosa. Ma imparare a riconoscere una forma comune negli algoritmi più disparati sicuramente paga nel medio periodo. Auspicabilmente, al termine di queste pagine, lo studente non descriverà più un algoritmo con le parole "... usa una pila..." o "... prima fa questo e poi quello..." o "... while $i < n$ do...", ma "... fa questo per ottenere quel risultato...".

Non mi resta che congedarmi, ringraziando la curatrice di queste note e augurando buona lettura a chi prosegue.

Roberto Cordone

Capitolo 1

Introduzione

Un *algoritmo* è un procedimento matematico eseguito per risolvere un problema. Parte da un insieme di oggetti matematici che definiscono lo specifico problema da risolvere (*istanza*). Talvolta, aggiunge loro altri oggetti matematici ausiliari. In base alle proprietà degli oggetti correnti, decide se fermarsi o modificare gli oggetti applicando una trasformazione scelta in modo deterministico. Ogni applicazione si chiama *operazione elementare* e il numero di operazioni elementari prima della fermata deve essere finito. Al termine dell'algoritmo, una parte degli oggetti matematici costituisce la *soluzione* del problema. Un algoritmo può manipolare oggetti matematici di qualsiasi genere: numeri, simboli, insiemi, funzioni e relazioni, sequenze ordinate, strutture complesse (ad es. grafi).

Definiamo *stato della computazione* il particolare insieme di oggetti matematici che sussistono in un determinato momento e *computazione* l'esecuzione di un algoritmo, cioè l'evoluzione nel tempo dello stato. Essa è rigorosamente deterministica: lo stato ad ogni passo è determinato dallo stato al passo precedente. Gli aspetti rilevanti al fine di analizzare o progettare un algoritmo sono:

- lo stato iniziale contiene l'istanza del problema;
- alcuni stati contengono la soluzione del problema;
- ogni stato determina se fermarsi o eseguire un'operazione e quale;
- ogni operazione trasforma lo stato corrente in un altro;
- alcuni stati sono raggiungibili da quello iniziale attraverso poche semplici operazioni;
- alcuni stati godono di proprietà matematiche che sono interessanti perché ne godono anche gli stati che contengono la soluzione.

- l'algoritmo deve raggiungere uno stato che contenga la soluzione e fermarvisi;

Queste definizioni valgono in astratto. Se si vuole far eseguire un algoritmo ad una macchina, bisogna realizzare una *codifica* degli oggetti matematici in stringhe di simboli (se si usa la macchina di Turing), in sequenze di numeri interi (se si usa la macchina RAM), o in strutture dati più o meno avanzate (se si usa un linguaggio di programmazione). Lo stato della computazione diventerà qualcosa di solo apparentemente diverso:

- per una macchina di Turing: lo stato della macchina più il contenuto dell'intero nastro e la posizione della testina;
- per una macchina RAM: il contenuto di tutti i registri più il programma e l'istruzione corrente.

Per descrivere, analizzare e progettare algoritmi, tuttavia, l'approccio più corretto è astrarre dalle caratteristiche della macchina che lo esegue e rifarsi alla definizione originaria, cioè vedere un algoritmo come un procedimento matematico. Il linguaggio e le strutture dati con cui esso verrà implementato sono aspetti da prendere in considerazione solo in un secondo tempo.

1.1 L'algoritmo di Euclide

Si voglia calcolare il Massimo Comune Divisore (MCD) di due numeri interi. L'algoritmo di Euclide procede come segue:

1. Considerare i due numeri dati a e b come numeri correnti a_0 e b_0
2. Confrontare i due numeri correnti a_i e b_i

   2a. Se $a_i = b_i$, andare al punto 3

   2b. Se $a_i \neq b_i$ sostituire al maggiore fra a_i e b_i la loro differenza e lasciare invariato il minore dei due numeri; tornare al punto 2
3. I due numeri correnti a_i e b_i coincidono fra loro e con il MCD dei due numeri originari a e b.

Consideriamo l'esempio $a = 147$ e $b = 102$.

1. $|a_0 - b_0| = 45 \quad \text{e} \quad \min(a_0, b_0) = 102 \quad \Longrightarrow \quad a_1 = 45 \quad \text{e} \quad b_1 = 102$
2. $|a_1 - b_1| = 57 \quad \text{e} \quad \min(a_1, b_1) = 45 \quad \Longrightarrow \quad a_2 = 45 \quad \text{e} \quad b_2 = 57$

3. $|a_2 - b_2| = 12$ e $\min(a_2, b_2) = 45 \implies a_3 = 45$ e $b_3 = 12$

4. $|a_3 - b_3| = 33$ e $\min(a_3, b_3) = 12 \implies a_4 = 33$ e $b_4 = 12$

5. $|a_4 - b_4| = 21$ e $\min(a_4, b_4) = 12 \implies a_5 = 21$ e $b_5 = 12$

6. $|a_5 - b_5| = 9$ e $\min(a_5, b_5) = 12 \implies a_6 = 9$ e $b_6 = 12$

7. $|a_6 - b_6| = 3$ e $\min(a_6, b_6) = 9 \implies a_7 = 9$ e $b_7 = 3$

8. $|a_7 - b_7| = 6$ e $\min(a_7, b_7) = 3 \implies a_8 = 6$ e $b_8 = 3$

9. $|a_8 - b_8| = 3$ e $\min(a_8, b_8) = 3 \implies a_9 = 3$ e $b_9 = 3$

La soluzione è $\text{MCD}(a, b) = a_8 = b_8 = 3$. Si può verificare esaustivamente che l'algoritmo di Euclide fornisce la soluzione corretta per l'istanza $(147, 102)$. Ma perché?

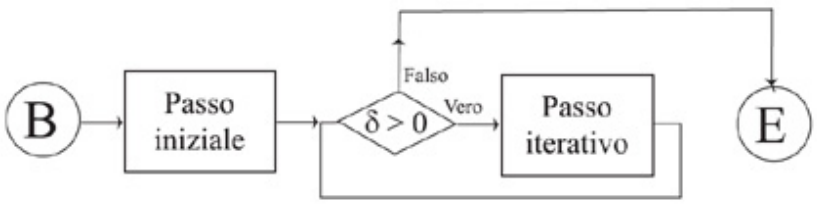
E, soprattutto, continuerà a funzionare su tutte le coppie di numeri?

Questa affermazione non si può dimostrare sperimentalmente, poiché le coppie di numeri sono infinite. Si può invece dimostrare formalmente, e questo ci darà la certezza assoluta che l'algoritmo è corretto e ci farà capire perché.

1.2 Algoritmi iterativi

L'algoritmo di Euclide segue lo schema semplicissimo rappresentato in Figura 1.1: alcune operazioni (*passo iniziale*), la verifica di una *condizione di termine*, altre operazioni da ripetere ciclicamente finché non si verifica la condizione di termine (*passo iterativo*). Tutti gli algoritmi che seguono questo schema vengono definiti *algoritmi iterativi*.

Figura 1.1: Schema generale di algoritmo iterativo



Gli algoritmi iterativi si basano sull'idea che la soluzione di un problema possa essere ottenuta ripetendo ciclicamente l'applicazione della stessa sequenza di operazioni (ovviamente su oggetti che via via si modificano) fino al verificarsi di una certa condizione di termine. Sullo schema di Figura 1.1 si possono costruire innumerevoli varianti: il ciclo può contenere istruzioni condizionali e altri cicli, può essere seguito da altre istruzioni (*passo finale*), e vi possono essere più cicli in sequenza. Ad ogni modo, molti algoritmi fondamentali seguono lo schema base.

Introduciamo ora alcuni concetti basilari per il progetto e l'analisi di algoritmi iterativi. Come si è detto, eseguire un algoritmo significa manipolare un insieme di oggetti matematici, facendoli passare da uno stato a un altro. Alcuni stati sono particolarmente interessanti perché alcuni degli oggetti in essi contenuti hanno proprietà in comune con la soluzione. Se godessero di tutte le proprietà della soluzione, avremmo risolto il problema e l'algoritmo potrebbe terminare. È quindi utile definire una misura quantitativa di quanto manchi affinché le altre proprietà siano verificate (*distanza*). Ciò fatto, abbiamo tutti gli ingredienti necessari.

Un algoritmo iterativo contiene:

- un *passo iniziale* che costruisce a partire dallo stato iniziale uno stato in cui alcune proprietà della soluzione sono vere;
- un *test di fine* che verifica se la distanza delle altre proprietà dall'essere vere è nulla;
- un *passo iterativo* che:
  - conserva le proprietà istituite nel passo iniziale;
  - riduce di un fattore finito la distanza dalle proprietà mancanti;

Eventualmente, si può avere un *passo finale*, che estrae la soluzione dallo stato raggiunto al termine del ciclo. Le proprietà istituite nel passo iniziale e poi conservate si chiamano *invarianti di ciclo*, perché rimangono vere, e quindi non variano, durante il ciclo.

Ogni algoritmo iterativo ha quindi uno *pseudocodice* di questo genere:

1. *istituisci una proprietà invariante* P
2. *finché distanza* $\delta > 0$

   2a) *conserva invariante* P

   2b) *riduci distanza* δ

Questa struttura si riflette immediatamente sul modo di dimostrare la correttezza di un algoritmo iterativo, sul modo di valutarne la complessità, e sul modo di progettarne uno.

Analisi di correttezza

La dimostrazione di correttezza di un algoritmo iterativo consiste in quattro dimostrazioni:

1. che il passo iniziale istituisca l'invariante P
2. che il passo iterativo conservi l'invariante P
3. che il passo iterativo riduca la distanza δ
4. che l'invariante (il quale è vero anche alla fine, grazie alla dimostrazione 2), combinato con la condizione $\delta = 0$, garantisca la soluzione del problema

$$\left.\begin{array}{l} P \\ \delta = 0 \end{array}\right\} \Rightarrow \text{ soluzione del problema}$$

Analisi di complessità

L'analisi di complessità di un algoritmo iterativo consiste nel valutare:

1. la complessità del passo iniziale $T_s(n)$
2. la complessità del generico passo iterativo $T_i(n)$, che dipende anche dall'iterazione i
3. il numero di iterazioni $i_{\max}$ che hanno luogo prima che si verifichi la condizione di termine
4. la complessità del passo finale $T_e(n)$

$$T(n) = T_s(n) + \sum_{i=1}^{i_{\max}} T_i(n) + T_e(n)$$

1.3 Analisi dell'algoritmo di Euclide

Applichiamo quanto visto sopra all'algoritmo di Euclide. L'algoritmo funziona perché le operazioni che compie sui due numeri non sono casuali, ma pensate per:

1. istituire la proprietà che due numeri ausiliari a_i e b_i abbiano lo stesso MCD dei due numeri dati a e b
2. conservare tale proprietà in ogni iterazione (invariante di ciclo)
3. garantire che i due numeri correnti a_i e b_i diminuiscano da un'iterazione alla successiva (riduzione della distanza)
4. garantire che l'invariante, combinato con l'annullarsi della distanza, fornisca la soluzione.

Analisi di correttezza

Parte 1 Poiché il passo iniziale pone $a_0 = a$ e $b_0 = b$, banalmente istituisce la proprietà $P : \text{MCD}(a_i, b_i) = \text{MCD}(a, b)$ per $i = 0$.

Parte 2 Dimostriamo che la proprietà si conserva durante l'esecuzione del passo iterativo. Sia $m = \text{MCD}(a_i, b_i)$; quindi

$$\begin{cases} a_i = h_i m \\ b_i = k_i m \end{cases} \quad \text{con } h_i \text{ e } k_i \in \mathbb{N}, \text{ primi tra loro.}$$

Senza perdita di generalità, supponiamo $a_i > b_i \Rightarrow h_i > k_i$. Al passo successivo b non cambia ($b_{i+1} = b_i$), mentre a_i viene sostituito da

$$a_{i+1} = a_i - b_i = (h_i - k_i)m$$

Quindi m è divisore comune ad a_{i+1} e b_{i+1}. Resta da dimostrare che sia il massimo divisore comune.

Supponiamo per assurdo che non lo sia, cioè che

$$\exists m' > m : \begin{cases} a_{i+1} = h_{i+1} m' \\ b_{i+1} = k_{i+1} m' \end{cases} \quad \text{con } h_{i+1}, k_{i+1} \in \mathbb{N}$$

Ma allora $a_i = a_{i+1} + b_i = a_{i+1} + b_{i+1} = (h_{i+1} + k_{i+1})\, m'$. Quindi a_i e b_i ammettono $m' > m$ come divisore comune. Questo è assurdo, perché m è il massimo divisore comune. Quindi l'ipotesi è falsa e $\text{MCD}(a_{i+1}, b_{i+1}) = \text{MCD}(a_i, b_i)$. Da cui si deduce che la proprietà

$$\text{MCD}(a_i, b_i) = \text{MCD}(a, b)$$

è vera per ogni i durante l'esecuzione, cioè è un invariante di ciclo.

Parte 3 Quanto sopra non basta a garantire la correttezza dell'algoritmo. Infatti, la proprietà si conserverebbe anche se sostituissimo $|a - b|$ al minore dei due numeri, anziché al maggiore. In tal caso, però, l'algoritmo non terminerebbe mai. Ad esempio siano $a = 24$ e $b = 18$

1. $|a_0 - b_0| = 6 \quad \text{e} \quad \min(a_0, b_0) = 18 \implies a_1 = 24 \quad \text{e} \quad b_1 = 6$
2. $|a_1 - b_1| = 18 \quad \text{e} \quad \min(a_1, b_1) = 6 \implies a_2 = 24 \quad \text{e} \quad b_2 = 18$
3. $|a_2 - b_2| = 6 \quad \text{e} \quad \min(a_2, b_2) = 18 \implies a_3 = 24 \quad \text{e} \quad b_3 = 6$
4. ...

È un requisito fondamentale che le operazioni del ciclo riducano sempre il più grande dei due numeri, perché questo garantisce che prima o poi i due numeri diventino uguali. Nell'esempio precedente ($a = 24$ e $b = 18$)

1. $|a_0 - b_0| = 6 \quad \text{e} \quad \min(a_0, b_0) = 18 \implies a_1 = 18 \quad \text{e} \quad b_1 = 6$
2. $|a_1 - b_1| = 12 \quad \text{e} \quad \min(a_1, b_1) = 6 \implies a_2 = 12 \quad \text{e} \quad b_2 = 6$
3. $|a_2 - b_2| = 6 \quad \text{e} \quad \min(a_2, b_2) = 6 \implies a_3 = 6 \quad \text{e} \quad b_3 = 6$

La terza parte della dimostrazione ha come tesi che la distanza δ cali monotonicamente e si annulli in un numero finito di iterazioni.

Come definire la distanza?

Una buona definizione è:

$$\delta_i = \max(a_i, b_i) - \text{MCD}(a, b)$$

Può sembrare strano che la definizione di distanza contenga la soluzione stessa del problema $\text{MCD}(a, b)$, che è un termine incognito. Accantoniamo la stranezza per un istante e verifichiamo che la definizione sia valida. Ad ogni iterazione, $\delta_i = \max(a_i, b_i) - \text{MCD}(a, b)$ viene sostituito da

$$\delta_{i+1} = \max(a_{i+1}, b_{i+1}) - \text{MCD}(a, b) = \max[\,|\, a_i - b_i \,|, \min(a_i, b_i)] - \text{MCD}(a, b)$$

che è $< \delta_i$, perché $a_i \neq b_i$. Si noti che δ è intero e non può diventare negativo perché l'invariante di ciclo garantisce che a_i e b_i siano entrambi sempre multipli di $\text{MCD}(a, b)$. Poiché δ è un numero naturale che cala strettamente ad ogni iterazione, arriverà a 0 in un numero finito di iterazioni.

Parte 4 All'uscita dal ciclo

$$\delta_i = 0 \Rightarrow \max(a_i, b_i) = \mathrm{MCD}(a, b)$$

che fornisce direttamente la soluzione del problema. È la quarta e ultima parte della dimostrazione.

Rimane solo la difficoltà che il valore della distanza δ è incognito, per cui non è chiaro come realizzare il test di fine. Non è una difficoltà insormontabile: esiste una condizione che non contiene grandezze incognite ed è del tutto equivalente all'annullarsi di δ.

Grazie all'invariante di ciclo, $\mathrm{MCD}(a, b) = \mathrm{MCD}(a_i, b_i)$. Inoltre, per definizione $a_i \geq \mathrm{MCD}(a_i, b_i)$ e $b_i \geq \mathrm{MCD}(a_i, b_i)$. Quindi, $\delta = 0 \Leftrightarrow \max(a_i, b_i) = \mathrm{MCD}(a_i, b_i) \Leftrightarrow a_i = b_i$. La condizione $a_i = b_i$, se vale l'invariante di ciclo, equivale a $\delta_i = 0$

Riassumendo, l'algoritmo di Euclide funziona perché:

1. il passo iniziale istituisce la proprietà invariante P

$$\mathrm{MCD}(a_i, b_i) = \mathrm{MCD}(a, b) \quad \text{per } i = 0$$

2. il passo iterativo conserva la proprietà invariante P:

$$\mathrm{MCD}(a_i, b_i) = \mathrm{MCD}(a, b) \Longrightarrow \mathrm{MCD}(a_{i+1}, b_{i+1}) = \mathrm{MCD}(a, b)$$

3. il passo iterativo riduce δ almeno di 1 ad ogni iterazione

$$\max(a_{i+1}, b_{i+1}) - \mathrm{MCD}(a, b) < \max(a_i, b_i) - \mathrm{MCD}(a, b)$$

4. invariante e distanza nulla implicano la soluzione

$$\left.\begin{array}{c} \mathrm{MCD}(a_i, b_i) = \mathrm{MCD}(a, b) \\ \delta_i = 0 \Leftrightarrow a_i = b_i \end{array}\right\} \Rightarrow a_i = b_i = \mathrm{MCD}(a, b)$$

Capitolo 2

Il problema della ricerca

Un *insieme indicizzato* è un insieme i cui elementi sono associati univocamente a numeri interi, detti *indici*, attraverso i quali è possibile leggere o modificare gli elementi. Senza perdita di generalità, assumeremo che gli indici siano consecutivi e compresi fra s e d. Di conseguenza, per definire un insieme indicizzato occorre conoscere anche i valori di s e d.

In genere, un insieme indicizzato si rappresenta con un *vettore* (o *array*), cioè una struttura dati statica (i suoi elementi non variano di numero durante l'esecuzione dell'algoritmo) e omogenea (gli elementi sono tutti dello stesso tipo). Questo perché tale struttura dati consente la lettura e la modifica di un elemento attraverso l'indice in tempo costante ($O(1)$) rispetto al numero di elementi del vettore. Dal nostro punto di vista, però, la struttura dati è di importanza secondaria, purché consenta l'accesso agli elementi attraverso l'indice.

Problema della ricerca Dati un insieme indicizzato A e un valore x, detto *chiave*, stabilire se il valore è identico a un elemento dell'insieme, riportando in caso affermativo l'indice di tale elemento.

2.1 Ricerca sequenziale

L'algoritmo di ricerca sequenziale si può usare su insiemi indicizzati generici. Scandisce sequenzialmente gli elementi dell'insieme, confrontandoli uno a uno con la chiave, finché non trova un elemento identico ad essa o finché non raggiunge la fine dell'insieme. In caso di successo l'algoritmo restituisce l'indice dell'elemento identico alla chiave.

Esempio 1

Istanza: $x = 24$ e $A = [2, 4, 21, 24, 28, 31, 32, 47, 93]$ con $s = 1$ e $d = 9$

Soluzione: La chiave si trova in posizione $i = 4$.

Esempio 2

Istanza: $x = 34$ e $A = [2, 4, 21, 24, 28, 31, 32, 47, 93]$ con $s = 1$ e $d = 9$

Soluzione: La chiave non è presente nel vettore.

Cerchiamo di applicare la metodologia di progetto descritta nell'introduzione riguardo all'algoritmo di Euclide. Per farlo ci serve:

1. una definizione della soluzione come insieme di oggetti matematici dotati di certe proprietà;
2. alcune proprietà facili da imporre subito;
3. una misura di distanza dello stato corrente dal soddisfare le proprietà mancanti.

La proprietà invariante, facile da imporre e di cui si richiede la conservazione. riguarda un oggetto matematico ausiliario (come i due numeri a_i e b_i usati da Euclide), cioè un insieme indicizzato A': vogliamo che A' sia un sottoinsieme di A e che contenga la chiave x se e solo se $x \in A$.

Invariante di ciclo

$$A' \subseteq A \qquad \text{e} \qquad x \in A' \iff x \in A$$

Istituzione dell'invariante

Per costruire un vettore A' che gode di queste proprietà, basta porre $A' = A$.

Distanza

Una misura di distanza dello stato corrente dalla soluzione è il numero di elementi di A' meno uno ($\delta = |A'| - 1$). Infatti, quando $\delta = 0$ risulta che A' contiene un solo elemento. In tale situazione, è facile ottenere la soluzione: basta un semplice passo finale, nel quale si confronta l'unico elemento di A' con la chiave. Grazie all'invariante, se elemento e chiave coincidono, A contiene la chiave nella stessa posizione; altrimenti, né A né A' contengono la chiave.

Conservazione dell'invariante e riduzione della distanza

Per ridurre la distanza δ, basta eliminare almeno un elemento di A'.

Per conservare l'invariante, bisogna non introdurre in A' elementi spuri (e questo è banale) e non eliminare la chiave. Questo richiede di confrontare con la chiave gli elementi che si intende eliminare.

Pseudocodice

Dato un insieme indicizzato A', siano $s_{A'}$ e $d_{A'}$ i suoi indici estremi (sinistro e destro). Indichiamo inoltre con $A'[\,i, \ldots, j\,]$ il sottoinsieme di A' compreso fra gli elementi di indice i e j e con $'A[\,i\,]$ l'elemento di indice i (con $s_{A'} \leq i \leq j \leq d_{A'}$).

Ricerca sequenziale(A, x)

$A' = A$; { Passo iniziale }

While $|A'| > 1$ **do** { Test di fine: $\delta =\mid A' \mid -1 > 0$ }

  If $A'[\,s_{A'}\,] = x$ { Passo iterativo }
    then $A' := A'[\,s_{A'}, \ldots, s_{A'}\,]$
    else $A' := A'[\,s_{A'} + 1, \ldots, d_{A'}\,]$;

EndWhile

If $A'[\,s_{A'}\,] = x$ { Passo finale }
  then Return $s_{A'}$
  else Return FALSE;

Possiamo scrivere anche uno pseudocodice più vicino al codice, gestendo esplicitamente gli indici estremi, tanto di A (s e d) quanto di A' (i e j):

Ricerca sequenziale(A, s, d, x)

$i := s$; $j := d$; { Passo iniziale: $A' = A$ }

While $i < j$ **do** { Test di fine: $\delta =\mid A' \mid -1 = j - i > 0$ }

  If $A[\,i\,] = x$ **then** $j := i$ **else** $i := i + 1$; { Passo iterativo }

If $A[\,i\,] = x$ **then** Return i **else** Return FALSE; { Passo finale }

Analisi di correttezza

1. L'invariante viene istituito. Infatti $A' = A$ garantisce che
$$A' \subseteq A$$
$$x \in A' \Longleftrightarrow x \in A$$

2. L'invariante viene conservato. Infatti, se $A[\,i\,] = x$, l'algoritmo elimina da A' tutti gli altri elementi, e quindi sia A sia A' contengono x. D'altra parte, se $A[\,i\,] \notin x$, l'algoritmo elimina $A[\,i\,]$, e non x, da A'; quindi rimane vero che $x \in A \Leftrightarrow x \in A'$.

3. La distanza si riduce perché, se $A[\,i\,] = x$, δ cala bruscamente a 0, invece se $A[\,i\,] \notin x$, δ cala di 1.

4. Combinando l'invariante ($A' \subseteq A$ e $x \in A' \Leftrightarrow x \in A$) con la condizione di termine ($\mid A' \mid = 1$), si ha che l'unico elemento di A' appartiene ad A ed è uguale alla chiave se e solo se la chiave appartiene ad A. Quindi, basta un semplice confronto fra la chiave e l'unico elemento di A' per ottenere la soluzione.

Analisi di complessità

$$T(n) = T_s(n) + \sum_{i=1}^{i_{\max}} T_i(n) + T_e(n)$$

1. Il passo iniziale consiste nel porre $A' = A$, stabilendo $i = s$ e $j = d$. Quindi, ($T_s(n) \in O(1)$).

2. Il passo iterativo è un accesso all'insieme indicizzato, un confronto e un assegnamento. Se l'insieme è rappresentato con un *array*, $T_i(n) \in O(1)$.

3. Nel caso pessimo (valore non presente nel vettore) si eseguono $i_{\max} = n$ iterazioni.

4. Il passo finale è un confronto: $T_e(n) \in O(1)$.

La complessità è quindi

$$T_{(n)} = O(1) + \sum_{i=1}^{n} O(1) + O(1) = O(n)$$

Nel caso ottimo, cioè quando la chiave coincide con il primo elemento del vettore, si esegue una sola iterazione e la complessità è:

$$T(n) = O(1) + O(1) + O(1) = O(1)$$

2.2 Ricerca binaria

L'algoritmo di *ricerca binaria* (o *dicotomica*) è corretto solo quando gli elementi dell'insieme sono ordinati per valori non decrescenti secondo un'opportuna relazione di ordine totale. Lo schema è analogo a quello della ricerca sequenziale, ma sfrutta in più la proprietà di ordinamento per accelerare la riduzione della distanza.

Questo algoritmo richiama il metodo che un essere umano impiega per trovare una parola sul dizionario: sapendo che questo è ordinato alfabeticamente, non si comincia la ricerca dalla prima parola, ma da quella mediana, cioè a metà del dizionario. Si confronta l'elemento mediano con la chiave: se l'elemento mediano precede la chiave secondo la relazione d'ordine, la chiave sta nella seconda metà dell'insieme e si elimina dalla ricerca immediatamente la prima metà; altrimenti, si fa il contrario. Ogni volta viene esclusa la parte dell'insieme A in cui la chiave x certamente non può trovarsi e si ripete il ragionamento sull'altra metà. La ricerca continua finché non rimane un solo elemento: l'ultimo confronto ci darà l'informazione richiesta o rivelerà che l'elemento non si trova nel dizionario. L'algoritmo sfrutta la tecnica del *divide et impera*, dato che divide in due gli elementi del problema, risolve entrambi i problemi (uno dei due ha banalmente soluzione FALSE) e ricombina le soluzioni (cioè restituisce quella del sottoproblema non banale).

Questo metodo si basa sulla seguente proprietà fondamentale:

$$\begin{cases} x \leq A[\,m\,] \Rightarrow x \notin A[\,m+1, ..., d\,] \\ x > A[\,m\,] \Rightarrow x \notin A[\,s, ..., m\,] \end{cases}$$

Invariante di ciclo

L'invariante di ciclo di questo algoritmo è lo stesso della ricerca sequenziale, cioè l'esistenza di un insieme ausiliario A' tale che

$$A' \subseteq A$$

$$x \in A' \Longleftrightarrow x \in A$$

Istituzione dell'invariante

Per istituire l'invariante, poniamo ancora $A' = A$.

Distanza

La misura di distanza è ancora il numero di elementi del vettore A' meno uno: $\delta =\mid A' \mid -1$.

Conservazione dell'invariante e riduzione della distanza

Sia m l'indice mediano di A'. Si considera l'elemento $A'[m]$,

- se $x \leq A'[m]$ si elimina il sottoinsieme di destra (escluso $A'[\,m\,]$)
- se $x > A'[\,m\,]$, si elimina il sottoinsieme di sinistra (compreso $A'[\,m\,]$)

Pseudocodice

Ricerca Binaria(A, s, d, x)

$i := s;\ j := d;$ { Passo iniziale: $A' = A$}

While $i < j$ **do** { Test di fine: $\delta = j - i$ }

$m := \lfloor \frac{i+j}{2} \rfloor;$ { Passo iterativo }

If $x \leq A[\,m\,]$ **then** $j := m$ **else** $i := m + 1;$

if $x = A[\,i\,]$ **then** Return i **else** Return FALSE; { Passo finale }

Si noti che passo iniziale, test di fine e passo finale sono identici a quelli della ricerca sequenziale: cambia solo il passo iterativo.

Analisi di correttezza

1. L'invariante viene istituito. Infatti, porre $i = s$ e $j = d$ significa porre $A' = A$, e quindi

$$A' \subseteq A \quad \text{e} \quad x \in A' \Leftrightarrow x \in A$$

2. L'invariante viene conservato perché:

   2a. se $x \leq A[\,m\,]$ sicuramente $x \notin A[\,m+1, ..., d\,]$. Quindi

   $$x \in A' = A[i, ..., m] \Leftrightarrow x \in A[\,1, ..., m\,]$$

   2b. se $x > A[\,m\,]$ sicuramente $x \notin A[\,s, ..., m\,]$. Quindi

   $$x \in A' = A[m+1, ..., j] \Leftrightarrow x \in A[\,m+1, d\,]$$

3. La distanza si riduce. Infatti:

3a. nel caso $x \leq A[\,m\,]$, δ cala da $(j-i)$ a $(\lfloor \frac{i+j}{2} \rfloor - i)$ ed è:

$$j - i > \left\lfloor \frac{i+j}{2} \right\rfloor - i \Leftarrow j > \left\lfloor \frac{i+j}{2} \right\rfloor \Leftarrow 0 > \left\lfloor \frac{i-j}{2} \right\rfloor$$

dato che $i < j$.

3b. Nel caso $x > A[\,m\,]$, δ cala da $(j-i)$ a $(j - \lfloor \frac{i+j}{2} \rfloor - 1)$ ed è

$$j - i > j - \left\lfloor \frac{i+j}{2} \right\rfloor - 1 \Leftarrow \left\lfloor \frac{i+j}{2} \right\rfloor - i + 1 > 0 \Leftarrow \left\lfloor \frac{j-i}{2} \right\rfloor + 1 > 0$$

dato che $j > 1$

4. Combinando l'invariante con la condizione di fine, si ha $A' \subseteq A$, $x \in A' \Leftrightarrow x \in A$ e $\delta = 0$ cioè $A' = A[\,i, ..., i\,]$. Quindi $x \in A \Leftrightarrow A[\,i\,] = x$.

Osservazione Se si prendesse $m = \lceil \frac{i+j}{s} \rceil$ e $x \leq A[\,m\,]$ l'invariante continuerebbe a conservarsi, ma la distanza δ non diminuirebbe sempre.

Infatti siano $A = [\,7 \;\; 12\,]$ e $x = 10$. E' $i = 1$ e $j = 2$ cioè $\delta = 1$. L'indice mediano è $m = 2$. Poiché $x \leq A[\,m\,] \Longrightarrow j = m = 2$, cioè all'iterazione successiva $\delta = 1$ come prima e si entra in un ciclo infinito.

Analisi di complessità

$$T(n) = T_s(n) + \sum_{i=1}^{i_{\max}} T_i(n) + T_e(n) = O(1) + \sum_{i=1}^{\log_2 n} O(1) + O(1) = O(\log_2 n)$$

Infatti $i_{\max} = \log_2 n$ perché la dimensione del vettore A' viene ripetutamente divisa a metà fino a restare con un unico elemento da confrontare con il valore cercato. Denotato con n il numero di elementi iniziali, dopo $i_{\max}$ iterazioni, si ottiene un vettore A' di dimensione $n(\frac{1}{2})^{i_{\max}}$. Affinché sia

$$n\left(\frac{1}{2}\right)^{i_{\max}} \leq 1 \Longrightarrow i_{\max} \geq \log_2 n \Longrightarrow i_{\max} = \lceil \log_2 n \rceil.$$

L'algoritmo di ricerca binaria è quindi più efficiente di quello di ricerca sequenziale. Tuttavia, esso è meno generale, in quanto può essere applicato solo a insiemi indicizzati ordinati.

Capitolo 3

Il problema dell'ordinamento

Dato un insieme indicizzato A, il problema dell'ordinamento consiste nel restituirne uno che contenga gli stessi elementi, permutati in modo da rispettare un'opportuna relazione di ordine totale (tipicamente, si parla di numeri interi e della relazione $\leq$).

Il progetto di algoritmi di ordinamento efficienti è molto importante in ambito informatico e per queste classi di algoritmi sono stati dimostrati diversi teoremi che ne definiscono i limiti. Il più importante è che la complessità temporale di qualsiasi algoritmo di ordinamento basato su operazioni di confronto fra elementi è $\Omega(n \log n)$, dove n è il numero degli elementi da ordinare. Questo teorema fissa il limite inferiore di *performance* di tali algoritmi: nessun algoritmo di ordinamento per confronto avrà complessità inferiore.

3.1 Insertion Sort

L'algoritmo *Insertion Sort*, in italiano ordinamento per inserimento, è definito dalla seguente strategia, non molto diversa dal modo in cui un essere umano ordina un mazzo di carte.

1. Si parte osservando che il primo elemento $A[s]$ costituisce un sottoinsieme ordinato di A, che chiameremo A'
2. Se tutti gli elementi di A sono inseriti in A' l'algoritmo si arresta.
3. Altrimenti si considera un elemento di A non contenuto in A' e si inserisce tale elemento nel sottoinsieme ordinato A' mantenendolo ordinato, poi si torna al punto 2.

È un algoritmo *in place*, cioè ordina l'insieme senza richiedere un altro insieme di appoggio. Quindi risparmia memoria.

Invariante di ciclo

$$A' \subseteq A$$

A' e' ordinato

Istituzione dell'invariante

Si pone $A' = A[s, ..., s]$ cioè A' contiene solo il primo elemento di A. Quindi $A' \subseteq A$ e A' è ordinato.

Distanza

$$\delta = |A \setminus A'|$$

Conservazione dell'invariante e riduzione della distanza

Si aggiunge ad $A' = A[s, ..., j-1]$ l'elemento $A[j]$ con $j = s+1, ..., d$. La distanza si riduce perché A' si allarga. L'invariante si conserva perché

1. $A' \subseteq A$ in quanto il nuovo elemento di A' viene da A
2. $A[j]$ viene aggiunto nella posizione corretta affinché A' resti ordinato

La posizione corretta va cercata. Inoltre, se si vuole rappresentare A' con una porzione di A, bisogna aggiornare gli indici dei suoi elementi, in modo che $A[j]$ occupi la posizione giusta: occorre spostare in avanti gli elementi di A' che sono maggiori di $A[j]$ e inserire $A[j]$ nel vuoto che così si forma.

Pseudocodice

InsertionSort(A, s, d)

$j := s + 1$; { Passo iniziale: $A' = A[s, ..., s]$ }

While $j \leq d$ **do** { Test di fine: $\delta = |A \setminus A'|$ }

  $x := A[j]$; { Passo iterativo }

  {Scala tutti gli elementi maggiori del nuovo}

  $i := j - 1$;

  While $i \geq 1$ and $A[i] > x$ **do**

    $A[i+1] := A[i]$;

    $i := i - 1$;

$A[i+1] := x$ {Inserisce il nuovo elemento al posto giusto}

$j := j + 1;$

Return A; { Passo finale }

Analisi di correttezza

1. l'invariante viene istituito correttamente
2. l'invariante viene conservato
3. la distanza viene ridotta di 1 ad ogni iterazione
4. combinando invariante e annullamento della distanza si ottiene $A' \subseteq A$, A' ordinato e $\mid A \setminus A' \mid = \emptyset \Rightarrow A' = A \Rightarrow A$ è ordinato

Analisi di complessità

$$T(n) = T_s(n) + \sum_{i=1}^{i_{\max}} T_i(n) + T_e(n)$$

1. $T_s(n) = O(1)$ perché è un solo assegnamento.
2. Il numero di iterazioni $i_{\max}$ è sempre $n - 1$.
3. $T_i(n)$ può andare da $O(1)$ nel caso migliore (il nuovo elemento è maggiore di tutti quelli in A') a $O(i)$ nel caso peggiore (il nuovo elemento è minore o uguale di tutti quelli in A', per cui occorre esaminare e scalare a destra tutti gli elementi).
4. $T_e(n) = \emptyset$.

Quindi nel caso migliore

$$T(n) = O(1) + \sum_{i=1}^{n-1} O(1) = O(n)$$

mentre nel caso peggiore

$$T(n) = O(1) + \sum_{i=1}^{n-1} O(i) = O(n^2)$$

3.2 Selection Sort

L'ordinamento per selezione (*Selection Sort*) opera in modo simile all'ordinamento per inserimento: gestisce un sottoinsieme $A' \subseteq A$ ordinato, a cui aggiunge via via elementi di A, finché $A' = A$. La differenza è che l'invariante di ciclo è più forte: A' non è solo un sottoinsieme ordinato di A, ma contiene anche elementi tutti non superiori agli elementi rimasti fuori. Per aggiornare A', quindi, l'algoritmo seleziona ogni volta l'elemento minimo in $A \setminus A'$ e lo aggiunge in fondo ad A', anziché selezionare il primo ed inserirlo nella posizione corretta. Si rallenta la scelta dell'elemento, si accelera il suo inserimento. Esiste, e viene spesso presentato nei libri di testo, una variante in cui si seleziona ogni volta l'elemento massimo di $A \setminus A'$ e lo si aggiunge in cima ad A'. Come prima, rappresentiamo A' come $A[\, s, ..., j-1\,]$. I passi sono i seguenti

1. Si parte con $A' = A[\, s, ..., s-1\,]$, cioè $A' = \emptyset$
2. Se $A' = A$, l'algoritmo si arresta
3. Altrimenti, si cerca il più piccolo elemento di $A \setminus A' = A[\, j, ..., d\,]$, si scambia tale elemento con il primo di $A' \setminus A$, cioè $A[\, j\,]$ e si allarga A' incrementando j. Quindi si torna al punto 2.

Invariante di ciclo

$$A' \subseteq A$$

$$A' \text{ e' ordinato}$$

$$a_i \leq a_j \quad \forall\, a_i \in A', a_j \in A \setminus A'$$

Distanza

$$\delta = |A \setminus A'| = d - j + 1$$

Istituzione dell'invariante

Si pone $A' = \emptyset$, cioè $j = s$.

Conservazione dell'invariante e riduzione della distanza

Per ridurre δ, si aggiunge un elemento di $A \setminus A'$ ad A'. Siccome questo elemento stava in $A \setminus A'$, è maggiore o uguale a tutti gli elementi di A'. Per conservare A' ordinato, quindi, il nuovo elemento va in fondo ad A'.

Siccome deve ancora essere $a_i \leq a_j \, \forall a_i \in A', a_j \in A \setminus A'$, il nuovo elemento deve essere minore o uguale agli altri elementi di $A \setminus A'$, cioè deve essere l'elemento minimo.

Pseudocodice

SelectionSort(A, s, d)

$j = s$; { Passo iniziale: $A' = A[s, \ldots, s-1] = \emptyset$ }

while $j \leq d$ { Test di fine: $\delta = |A \setminus A'|$ }

$i_{\min} = \arg \min_{i : a_i \in A \setminus A'} a_i$; { Passo iterativo }

{ Scambio gli elementi $A[j]$ e $A[i_{\min}]$ }

$t = A[i_{\min}]$;

$A[i_{\min}] = A[j]$;

$A[j] = t$;

Return A { Passo finale }

Analisi di correttezza

1. L'invariante viene istituito perché:

$$A' = \emptyset \Longrightarrow \begin{cases} A' \subseteq A \\ A' \text{ ordinato} \\ a_i \leq a_j \, \forall a_i \in A', a_j \in A \setminus A' \end{cases}$$

2. La distanza cala perché A' acquista un elemento $\Rightarrow \delta = |A \setminus A'|$ diminuisce di 1.

3. L'invariante si conserva perché il nuovo elemento è il minimo di $A \setminus A'$ e va in fondo ad A'.

4. Se la distanza è nulla, $|A \setminus A'| = \emptyset$. Essendo $A' \subseteq A$, risulta $A' = A$. Quindi, se A' è ordinato, A è ordinato.

Analisi di complessità

$$T(n) = T_s(n) + \sum_{i=1}^{i_{\max}} T_i(n) + T_e(n)$$

1. $T_s(n) = O(1)$ perché consiste in un assegnamento $(i = 1)$
2. $T_i(n) = O(n - i)$ perché consiste in uno scorrimento del sottoinsieme $A \setminus A'$.
3. il numero delle iterazioni $i_{\max}$ è n
4. $T_e(n) = \emptyset$

da cui

$$T(n) = O(1) + \sum_{i=1}^{n} O(n - i) = O(n^2)$$

Il tempo di esecuzione dell'algoritmo di ordinamento per selezione non dipende dal grado di ordinamento in cui si trova il vettore. La ricerca del minimo elemento durante una scansione del vettore non viene usata per ottenere informazioni circa la posizione del prossimo minimo nella scansione successiva.

Nonostante non sia particolarmente efficiente, l'ordinamento per selezione ha un'importante applicazione: poiché ciascun elemento viene spostato al più una volta, questo è l'algoritmo da preferire quando si devono ordinare file costituiti da record molto grandi e da chiavi molto piccole. Per queste applicazioni, infatti, il costo dello spostamento dei dati prevale sul costo dei confronti e nessun algoritmo è in grado di ordinare un file con spostamenti di dati sostanzialmente inferiori a quelli dell'ordinamento per selezione. In tali casi, le costanti moltiplicative possono essere tali da rendere questo algoritmo competitivo rispetto ad altri, asintoticamente migliori.

3.3 Merge

L'algoritmo di ordinamento *Merge Sort* utilizza un processo di risoluzione ricorsivo, ma contiene la procedura *Merge* che è iterativa. L'idea alla base del *Merge Sort* si inquadra nella tecnica nota come *divide et impera*, che consiste nella suddivisione del problema in sottoproblemi via via più piccoli e nella ricomposizione delle relative soluzioni. In particolare, il *Merge Sort* divide l'insieme da ordinare in due parti uguali e procede ricorsivamente a ordinarle. Quando la divisione dà luogo a insiemi di un singolo elemento, tali

microscopici insiemi vengono ordinati con la forza bruta (non facendo nulla, dato che un insieme di un solo elemento è già ordinato di per sé). Quindi si procede alla fusione dei sottoinsiemi ordinati in un insieme complessivo ordinato. Questo è il compito della funzione *Merge*. Riassumendo

- *divide*: si divide l'insieme a metà;
- *impera*: si risolvono ricorsivamente i problemi sui due sottoinsiemi;
- *combina*: si fondono i due sottoinsiemi ordinati in un insieme ordinato.

Problema della fusione di insiemi ordinati: Dati due insiemi ordinati B e C, restituire un insieme ordinato $A = B \cup C$.

Invariante di ciclo

L'istanza è costituita dai vettori B (con indici estremi s_B e d_B) e C (con indici estremi s_C e d_C). Ci appoggiamo a un vettore ausiliario A, che cresce via via sino a includere tutti gli elementi di $B \cup C$.

$$A \subseteq B \cup C$$

$$A \text{ ordinato}$$

$$B \setminus A \text{ e } C \setminus A \text{ ordinati}$$

$$a_i \leq a_j \quad \forall\ a_i \in A, a_j \in B \setminus A$$

$$a_i \leq a_j \quad \forall\ a_i \in A, a_j \in C \setminus A$$

Istituzione dell'invariante

Per soddisfare istantaneamente tutte le condizioni dell'invariante, basta porre:

$$A = \emptyset$$

Per implementare tutto ciò possiamo descrivere A come un vettore con indici estremi s_A e d_A, $B \setminus A$ come il sottovettore $B[i_B, ..., d_B]$ e $C \setminus A$ come il sottovettore $C[i_C, ..., d_C]$. Inizialmente, poniamo

$$\begin{cases} d_A := s_A - 1 \\ i_B := s_B \\ i_C := s_C \end{cases}$$

Distanza

È il numero di elementi ancora non portati in A

$$\delta = |(B \cup C) \setminus A|$$

Riduzione della distanza e conservazione dell'invariante

Prendere un qualsiasi elemento x da $(B \cup C) \setminus A = (B \setminus A) \cup (C \setminus A)$ e metterlo in A riduce la distanza. Dato che x sta in $(B \cup C) \setminus A$, è $x \geq a_i, \forall a_i \in A$. Quindi, per mantenere A ordinato x va aggiunto in fondo. Resta il problema di scegliere x:

- per garantire che $a_i \leq a_j \ \forall \ a_i \in A$ e $a_j \in B \setminus A$, bisogna che sia $x \leq a_j, \forall a_j \in B \setminus A$;
- per garantire che $a_i \leq a_j \ \forall \ a_i \in A$ e $a_j \in C \setminus A$, bisogna che sia $x \leq a_j, \forall a_j \in C \setminus A$.

Quindi serve l'elemento minimo di $(B \setminus A) \cup (C \setminus A)$

$$\min_{(B \setminus A) \cup (C \setminus A)} a_i = \min \left[\min_{B \setminus A} a_i \ ; \ \min_{C \setminus A} a_i \right]$$

Poiché conserviamo $B \setminus A$ e $C \setminus A$ ordinati (vedi invariante), il termine $\min_{B \setminus A} a_i$ è il primo elemento di $B \setminus A$ e il termine $\min_{C \setminus A} a_i$ è il primo elemento di $C \setminus A$: x è il più piccolo dei due.

Pseudocodice

$d_A := s_A - 1$; $i_B := s_B$; $i_C := s_C$; { Passo iniziale: $A = \emptyset$ }

While $(i_B < d_B)$ **and** $(i_C < d_C)$ { Test di fine: $\delta = |(B \cup C) \setminus A|$ }

- $d_A := d_A + 1$; { Si allarga A per far posto a x }
- **If** $(i_B = d_B)$ { $B \setminus A = \emptyset$: si estrae x da $C \setminus A$}
  - $A[d_A] := C[i_C]$;
  - $i_C := i_C + 1$;
- **else If** $(i_C = d_C)$ { $C \setminus A = \emptyset$ e $B \setminus A \neq \emptyset$: si estrae x da $B \setminus A$}
  - $A[d_A] := B[i_B]$;
  - $i_B := i_B + 1$;
- **else If** $(B[i_B] < C[i_C])$ { $\min_{B \setminus A} a_i < \min_{C \setminus A} a_i$: si estrae x da $B \setminus A$}

$A[d_A] := B[i_B];$
$i_B := i_B + 1;$

else $\{ \min_{B \setminus A} a_i \geq \min_{C \setminus A} a_i$: si estrae x da $C \setminus A\}$

$A[d_A] := C[i_C];$
$i_C := i_C + 1;$

Analisi di correttezza

1. L'invariante viene istituito perché, ponendo $A = \emptyset$, si ha
$$A \subseteq B \cup C$$
$$A \text{ ordinato}$$
$$B \setminus A = B \text{ e } C \setminus A = C \text{ ordinati}$$
$$a_i \leq a_j \quad \forall\ a_i \in A, a_j \in B \setminus A$$
$$a_i \leq a_j \quad \forall\ a_i \in A, a_j \in C \setminus A$$
2. La distanza cala perché ad ogni passo si aggiunge un elemento ad A e quindi $\delta =\mid B \cup C \mid - \mid A \mid$ cala di 1.
3. L'invariante si conserva perché il nuovo elemento aggiunto ad A è maggiore o uguale a tutti i precedenti e viene inserito in fondo ad A. Inoltre è minore o uguale a tutti gli elementi di $(B \cup C) \setminus A$.
4. Al termine $\delta = 0 \Rightarrow \mid B \cup C \setminus A \mid = 0 \Rightarrow A \supseteq B \cup C$, ma siccome $A \subseteq B \cup C$ risulta $A = B \cup C$ e l'invariante garantisce che A è ordinato.

Analisi di complessità

$$T(n) = T_s(n) + \sum_{i=1}^{i_{\max}} T_i(n) + T_e(n)$$

1. $T_s(n) = O(1)$ perché consiste in tre assegnamenti.
2. $T_i(n) = O(1)$ perché comprende un numero dato di confronti e assegnamenti.
3. $i_{\max} = n$
4. $T_e(n) = 0$

Quindi
$$T(n) = O(1) + \sum_{1}^{n} O(1) = O(n)$$

3.4 Partition

Anche il *Quick Sort* è un algoritmo di ordinamento ricorsivo. A partire da un elemento *perno* (*pivot*) divide l'insieme da ordinare in due parti: la prima contiene gli elementi minori o uguali al perno (escluso quest'ultimo), la seconda contiene gli elementi maggiori del perno. Quindi, riapplica ricorsivamente l'algoritmo alle due parti e concatena i due vettori ordinati, ponendo il perno in posizione intermedia. Quando l'insieme contiene un solo elemento, non fa nulla. La procedura che scompone l'insieme nelle due parti è iterativa e la indichiamo col nome *Partition*.

Problema della partizione: Dato un insieme indicizzato A e un suo elemento x, restituire il sottoinsieme B degli elementi di A non superiori a x (escluso) e il sottoinsieme C degli elementi di A superiori a x.

Per usare efficientemente lo spazio in memoria e per eseguire rapidamente il concatenamento dei due insiemi, conviene gestire B e C come parti di A, permutandone gli elementi di A in modo che B corrisponda agli indici più bassi e C a quelli più alti.

$$A = B \cup \{x\} \cup C$$

$$a_i \leq x < a_j \quad \forall \quad a_i \in B \text{ e } a_j \in C$$

Poiché il primo sottoinsieme ha elementi tutti minori o uguali a quelli del secondo, basta ordinare separatamente i due sottoinsiemi e concatenarli per ottenere l'insieme di partenza ordinato.
Per rappresentare B e C, introduciamo due sottoinsiemi di A definiti come

$$B = A[\, s, ..., q-1 \,] \quad e \quad C = A[\, q+1, ..., d \,]$$

e poniamo x in posizione q-esima. Se riusciamo ad ottenere questo risultato, abbiamo risolto il problema.

Invariante di ciclo

Costruiamo B e C poco per volta, a partire da due insiemi B' e C'. Imponiamo sin dal principio che

$$\begin{cases} B' \subseteq A \\ C' \subseteq A \\ a_i \leq x < a_j \quad \forall \quad a_i \in B', \quad a_j \in C' \end{cases}$$

Istituzione dell'invariante

Possiamo definire B' e C' come sottoinsiemi di A ponendo $B' = A[s, ..., p]$ e $C' = A[p+1, ..., q-1]$. Per imporre che siano vuoti, basta mettere

$$\begin{cases} p := s - 1 \\ q := s \end{cases} \quad \begin{cases} B' = \emptyset \\ C' = \emptyset \end{cases}$$

E' facile verificare che questi due vettori vuoti godono delle proprietà richieste.

Distanza

$$\delta = |A| - \mid B' \mid - \mid C' \mid -1 = d - q$$

Riduzione della distanza e conservazione dell'invariante

Per ridurre la distanza δ, basta prendere un elemento a da A e metterlo in B' o C'. La scelta dell'elemento è arbitraria: per conservare l'invariante è sufficiente metterlo in B' se $a \leq x$ e in C' se $a > x$.
Per conservare le definizioni di B' e C' come sottovettori di A, però, è opportuno che si prenda non un elemento qualsiasi, ma l'elemento $A[\,q\,]$. Questo significa che:

- se $A[\,q\,]$ va posto in C', è sufficiente incrementare q;
- se invece va posto in B' basta incrementare p, scambiare $A[\,p\,]$ con $a[\,q\,]$ e incrementare q.

Pseudocodice

Partition(A, s, d)

$x := A[\,d\,]$;

$p := s - 1$;

$q := s$;

while $q < d$ **do**

    if $A[\,q\,] \leq x$

    $p := p + 1$;

scambia $A[\,p\,]$ con $A[\,q\,]$

$q := q + 1$;

scambia $A[\,p+1\,]$ con $A[\,d\,]$

return $p + 1$

Una nota variante dell'algoritmo *Partition* gestisce diversamente i due sottoinsiemi. Definisce infatti $B' = A[\,s, ..., p\,]$ e $C' = A[\,q, ..., d-1\,]$ e quindi

1. nel passo iniziale si pone $p = 0$ $(B' = \emptyset)$ e $q = d$ $(C' = \emptyset)$;
2. ad ogni passo si aggiunge l'elemento $a = A[\,p+1\,]$ a B' o C': anzi tutto, si incrementa p; se a va messo in B' $(a \leq A[\,d\,])$, non fa altro; se a va messo in $C'(a > A[\,d\,])$, si abbassa q e si scambia a con $A[q-1]$.

In questa variante, la distanza δ è $\mid A \mid - \mid B \mid - \mid C \mid -1 = q - p$ e quindi $\delta = 0 \iff p = q$, che diventa la condizione di termine.

Analisi di correttezza

1. L'invariante viene istituito perché ponendo $p = s - 1$ e $q = s$ è $B' = C' = \emptyset$ e quindi

$$\begin{cases} B' \subseteq A \\ C' \subseteq A \\ a_i \leq x < a_j \quad \forall \; a_i \in B', \;\; a_j \in C' \end{cases}$$

2. La distanza cala perché ad ogni passo un elemento nuovo entra in B' o in C'

$$\delta = \mid A \mid - \mid B' \mid - \mid C' \mid -1$$

3. L'invariante si conserva perché ad ogni passo si prende un elemento da A e lo si mette in B' se $a \leq x$ e in C' se $a > x$. E' opportuno che si prenda l'elemento $A[\,q+1\,]$, per conservare le definizioni di B' e C' come sottoinsiemi di A.

4. All'uscita dal ciclo, $q = d$ $(\delta = 0)$ e i due sottoinsiemi $B' = A[\,s, ..., p\,]$ e $C' = [\,p+1, ..., d-1\,]$ contengono, rispettivamente, tutti gli elementi di A minori o uguali ad $A[\,d\,]$ e tutti quelli maggiori di $A[\,d\,]$. Questa non è ancora la soluzione desiderata, dato che si voleva $A = B \cup \{x\} \cup C$, ovvero in posizione intermedia. Per ottenere la soluzione, basta però scambiare $A[\,d\,]$ col primo elemento di B', cioè $A(p+1)$. A questo punto la posizione del pivot x è $p + 1$.

Analisi di complessità

La complessità della funzione *Partition* è data dalla somma della complessità delle inizializzazioni e della complessità del ciclo. Quest'ultimo ha una complessità $O(n)$ in quanto l'istruzione di return, che determina la fine del ciclo viene raggiunta dopo l'esecuzione di un numero di istruzioni dell'ordine di n.

Ricerca del pivot $\Longrightarrow O(1)$
Partition $\Longrightarrow O(n)$

Nel caso ottimo, cioè nel caso in cui le due liste sono di uguale dimensione: $T(n) = 2T\left(\frac{n}{2}\right) + cn = O(n \lg n)$.

Nerl caso pessimo, cioè nel caso in cui le liste sono sbilanciate al massimo: $T(n) = T(n-1) + T(1) + cn = O(n^2)$; il caso peggiore si ha quando un sottoinsieme ha n-1 elementi e l'altro zero elementi. Tale situazione si presenta quando l'array è ordinato in maniera crescente o in maniera decrescente.

Nel caso medio, la particolarità del QuickSort è che anche con una partizione sfavorevole, esempio 9 a 1, mantiene proprietà interessanti, esempio per Partition:

$$T(n) = T\left(\frac{9n}{10}\right) + T\left(\frac{n}{10}\right) + cn$$

In definitiva la ricorsione termina alla profondità di:

$$\lg_{\frac{9}{10}} n \; O(\lg n)$$

per cui il QuickSort ha complessità $O(n \lg n)$ e la avrebbe anche se la partizione fosse 99 a 1.

Capitolo 4

Il problema dell'albero ricoprente minimo

Dato un grafo non orientato $G(V, E)$ e una funzione di costo definita sui lati $c : E \longrightarrow R$, il problema consiste nel trovare un sottografo $T = (U, X)$ di G che sia:

- ricoprente ($U = V$)
- connesso e aciclico (e quindi sia un albero)
- di costo $c_X = \sum_{c \in X} c_e$ minimo rispetto a quello di tutti gli altri alberi ricoprenti.

Descriviamo nel seguito due algoritmi iterativi, basati sui concetti esposti in precedenza, con numerose varianti che dipendono dalle strutture dati impiegate per eseguire le diverse operazioni.

4.1 Algoritmo di Kruskal

L'idea è la solita: partire da un oggetto matematico che ha alcune delle proprietà della soluzione e introdurre pian piano le altre, conservando le prime. La soluzione è caratterizzata dal fatto di essere

1. ricoprente
2. aciclica
3. connessa

4. di costo minimo rispetto a tutti i sottografi che godono delle proprietà 1, 2, 3

Noi partiremo da un sottografo $F = (U, X)$ che gode delle proprietà 1 e 2 e (sotto forma opportuna) 4, e modificheremo F in modo da introdurre via via la proprietà 3. Per farlo, occorre anzitutto una misura quantitativa di distanza. Per costruirla, definiamo k il numero di componenti connesse che costituiscono il sottografo F. La proprietà 3 si può riscrivere dicendo che $k = 1$, perché un grafo connesso è un grafo che ha una sola componente connessa. Il vantaggio è che esprimendo la proprietà in modo numerico possiamo misurare di quanto essa sia violata.
Inoltre, esprimeremo la proprietà 4 dicendo che la soluzione corrente $F = (U, X)$ ha costo minimo rispetto a tutti i sottografi con le proprietà 1 e 2 e con $k = 1$.

Invariante di ciclo

L'invariante di ciclo per l'algoritmo di Kruskal è:

1. $F = (U, X)$ è ricoprente
2. $F = (U, X)$ è aciclico
3. $F = (U, X)$ ha costo minimo rispetto ai sottografi ricoprenti e aciclici con k componenti connesse

Istituzione dell'invariante

Possiamo istituire l'invariante ponendo:

1. $U = V$, per cui $F = (U, X)$ è ricoprente
2. $X = \emptyset$, per cui $F = (U, X)$ non contiene cicli
3. $(V, \emptyset)$ è l'unico grafo che gode di queste due proprietà e in più del fatto di avere $k = n$ componenti connesse. Quindi è il sottografo di costo minimo che gode di tali proprietà.

Distanza

Poiché inizialmente abbiamo $k = n$ componenti connesse e vogliamo ottenerne una sola, si può definire la distanza come:

$$\delta = k - 1$$

Riduzione della distanza e conservazione dell'invariante

Per ridurre δ bisogna abbassare k. Lo faremo un passo alla volta sfruttando il seguente teorema.

Il numero di vertici n, di lati i e di componenti connesse k di un grafo aciclico $F = (U, X)$ sono legati dalla relazione:

$$i + k = n$$

Siccome n è costante per la proprietà 1, che fa parte dell'invariante, il teorema garantisce che per abbassare k bisogna alzare i. Di conseguenza, il ciclo aggiunge ad ogni passo un lato a X. Il lato deve essere tale da far rimanere F ricoprente, aciclico e di costo minimo. L'invariante è conservato se:

- non si eliminano vertici da U;
- non si aggiungono lati che chiudono cicli in X ;
- non si distrugge la minimalità di X rispetto agli altri sottografi ricoprenti e aciclici di $i = n - k$ lati.

Inoltre, si può sostituire la condizione $\delta > 0 \Leftrightarrow k > 1$ con $\mid X \mid = i < n - 1$.

Pseudocodice

$U := V$;

$X := \emptyset$

While $|X| < n - 1$ **do**

Estrae un lato e^* valido rispetto a X

X:=X $\cup\{e^*\}$

return (U, X)

Come scegliere il lato e^*?
Deve essere un lato che non chiude cicli con quelli di X e deve mantenere minimo il costo. Proviamo col lato di costo minimo che non chiude cicli con i lati in X.

Kruskal(V, E, c)

$U := V$; $X := \emptyset$;

While $|X| < |V| - 1$

{Determina il nuovo arco da aggiungere}

$E' := E \setminus X$;

Repeat $e^* := \arg\min_{e \in E'} c_e$

$E' := E' \setminus \{e^*\}$;

until Aciclico$(X \cup \{e^*\})$;

$X := X \cup \{e^*\}$;

Return (U, X);

dove l'insieme E' serve per evitare di estrarre sempre lo stesso lato.
In realtà l'inizializzazione di E' si può spostare fuori dal ciclo *While* perché, siccome X cresce sempre, se un lato chiude ciclo con X in un dato passo, lo chiuderà anche più avanti. Inoltre, l'istruzione $U = V$ si può togliere perché non interviene in nessun passo dell'algoritmo. Infine, si può riscrivere lo pseudocodice fondendo il ciclo *repeat ... until* col ciclo *while* che lo contiene.

$X := \emptyset$

$E' := E$

while $\mid X \mid < n - 1$ **do**

$e^* = \arg\min_{e \in E'} c_e$

if Aciclico $(X \cup \{e\})$ **then** $X := X \cup \{e^*\}$

$E' := E' \setminus \{e^*\}$

ottenendo la forma comunemente usata per l'algoritmo di Kruskal.

La differenza tra le due versioni è che nella prima è chiaro lo schema generale degli algoritmi iterativi, dato che la distanza diminuisce strettamente di 1 ad ogni passo. La seconda permette, d'altra parte, di valutare più facilmente la complessità.

Analisi di correttezza

1. L'invariante viene istituito correttamente con $i = 0$
   - $U := V$ garantisce la ricopertura

- $X := \emptyset$ garantisce l'aciclicità
- $(V, \emptyset)$ è l'unico grafo ricoprente aciclico di $i = 0$ lati, per cui è quello di costo minimo.

2. La distanza si riduce, perché ad ogni passo i cresce di 1 e quindi k cala di 1 (dato che il grafo F resta aciclico e ricoprente)

3. L'invariante viene conservato perché

   - non si eliminano mai vertici da U, per cui F resta ricoprente
   - non si aggiungono mai lati che rendono X ciclico, per cui F resta aciclico
   - se $c_{x_i} \leq c_x$ per ogni X aciclico di i lati, allora $c_{X_{i+1}} \leq c_X$ per ogni X aciclico di $i+1$ lati

La terza proprietà (quella di minimalità) non è banale. Dimostriamo ora che si conserva. Si supponga per assurdo che al passo i, F sia minimo, e al passo $i+1$ non lo sia più; esiste quindi un grafo aciclico $\bar{F} = (V, barX)$ di $i+1$ lati che costa meno. Ordiniamo i lati di F nell'ordine con cui sono stati aggiunti dall'algoritmo a X e i lati del grafo $\bar{F}$ in modo che quelli che stanno anche in X siano nella stessa posizione e gli altri, in ordine di costo crescente, riempiano le lacune. Fino ad un certo passo h i due grafi coincidono. Dopo differiscono: $\bar{F}$ contiene $\bar{e}$, mentre F contiene e. Osserviamo che $\bar{e}$ non chiude ciclo coi primi h lati di F, dato che sono gli stessi di $\bar{F}$. Inoltre, $c_e \leq c_{\bar{e}}$ perché altrimenti Kruskal avrebbe preso $\bar{e}$ in considerazione e l'avrebbe aggiunto a X. Allora si può aggiungere e a $\bar{F}$. A questo punto, vi sono due casi possibili:

1. $\bar{X} \cup \{e\}$ contiene un ciclo. Almeno un lato di tale ciclo non è fra i primi h di $\bar{F}$, per cui costa più di $c_{\bar{e}}$, e quindi di c_e. Tolto quel lato, si ottiene un grafo aciclico di $i+1$ lati che costa meno di $\bar{F}$. Ma se $\bar{F}$ era ottimo, questo caso è assurdo.

2. $\bar{X} \cup \{e\}$ ha una componente connessa in meno rispetto a X_{i+1}. Se si elimina il lato più costoso di $\bar{X} \cup \{e\}$ otteniamo ancora un grafo aciclico di $i+1$ lati che costa meno di $\bar{F}$. Ma se $\bar{F}$ era ottimo, anche questo caso è assurdo.

Analisi di complessità

Valutiamo la complessità riferendoci alla seconda versione dello pseudocodice.

$$T(n) = T_s(n) + \sum_{i=1}^{i_{\max}} T_i(n) + T_e(n)$$

- $T_s(n) = O(1)$
- il numero di iterazioni massimo è uguale a $\mid E \mid = M$ perché E' contiene inizialmente tutti i lati, e ne perde uno ad ogni iterazione
- $T_e(n) = 0$

Per quanto riguarda $T_i(n)$, le operazioni del ciclo consistono in

1. $T_{i1}(n)$: trovare il lato e di costo minimo
2. $T_{i2}(n)$: verificare se $X \cup \{e\}$ è aciclico
3. $T_{i3}(n)$: in caso affermativo, aggiungere e ad X

Mentre il terzo passo richiede $O(1)$, i primi due richiedono una descrizione più approfondita.

Ricerca del lato di costo minimo in E'

Presentiamo tre possibili implementazioni.

1. Implementazione banale: Siccome l'insieme E' contiene al massimo m elementi, trovarne l'elemento di costo minimo per semplice scorrimento costa $T_{i1}(n) = O(m)$.

2. Ordinamento preventivo: E' viene ordinato nel passo iniziale. Questo complica l'invariante di ciclo, dato che gli si aggiunge la proprietà che E' sia ordinato per costi crescenti. L'invariante si conserva, perché eliminare da E' il lato e di costo minimo non distrugge l'ordinamento. La complessità del passo iniziale cresce da $T_s(n) = O(1)$ a $T_s(n) = O(m \log m)$. D'altra parte la complessità dell'estrazione cala da $T_{i1}(n) = O(m)$ a $T_{i1}(n) = O(1)$.

3. Ordinamento parziale tramite *heap*: E' viene descritto da uno *heap*, costituito nel passo iniziale. L'invariante di ciclo viene integrato con la proprietà che E' è un min-heap. Affinché l'invariante si conservi, dopo aver eliminato il lato e^* di costo minimo bisogna aggiornare lo heap E'. Questo modifica lo pseudocodice come segue

$X := \emptyset$

$E' := BuildHeap(E)$

while $\mid X \mid < n - 1$ **do**

$e^* = \text{ExtractMin}(E')$
if Aciclico $(X \cup \{e^*\})$ $X := X \cup \{e^*\}$
$E' := E' \setminus \{e^*\}$

La complessità del passo iniziale cresce da $T_s(n) = O(1)$ a $T_s(n) = O(n)$.
La complessità dell'estrazione cala da $T_{i1}(n) = O(m)$ a $T_{i1}(n) = O(1) + O(\log m)$, dove $O(1)$ deriva dall'individuazione del minimo e $O(\log m)$ dall'aggiornamento di E' per mantenere la proprietà heap. Questa complessità è identica alla precedente, ma bisogna osservare che l'espressione $O(\log m)$ di T_i si riferisce di volta in volta ai lati non ancora considerati, che sono in realtà meno di m, mentre l'espressione $O(m \log m)$ della versione precedente si riferisce a tutti i lati. Inoltre T_i non viene eseguita m volte in tutte le istanze, dato che la condizione di termine può scattare anche molto prima.

Verifica di aciclicità

Se l'insieme di lati X è aciclico, $X \cup \{e^*\}$ è aciclico se e solo se $e^* = (u^*, v^*)$, non chiude cicli con i lati di X, cioè se e e solo se gli estremi di e^* non sono già connessi fra loro in X.

1. Implementazione banale Per verificare che u^* e v^* non sono già connessi in X, basta visitare il sottografo (V, X) partendo da u^* e controllare se v^* è raggiungibile. L'operazione costa $T_{i2}(n) = O(\mid X \mid) = O(n)$

2. Implementazione tramite Merge-Find Set Le strutture dati *Merge-Find Set* descrivono una collezione di sottoinsiemi disgiunti di un insieme dato. Consentono di verificare a quale sottoinsieme appartiene ciascun elemento e di fondere due sottoinsiemi. Se rappresentiamo le componenti connesse del sottografo (U, X) (più precisamente i sottoinsiemi di vertici che sorrispondono a ciscuna componente connessa) con i *Merge-Find Set*, possiamo verificare se u^* e v^* non siano già connessi in X semplicemente verificando che appartengono a sottoinsiemi distinti. Ogni volta che si aggiunge un lato a X, invece, occorrerà fondere i sottoinsiemi a cui u^* e v^* appartengono.
Questo complica il passo iniziale, in cui bisogna costruire i *Merge-Find Set* (un sottoinsieme per ciascun vertice), portando la sua complessità da $T_s(n) = O(1)$ a $T_s(n) = O(n)$.
D'altra parte, la verifica di aciclicità richiede il confronto dei due sottoinsiemi di appartenenza di u^* e v^*. Se si applica la euristica di fusione per rango e quella di compressione dei cammini, tale operazione richiede $O(\alpha(m, n))$, dove $\alpha(m, n)$ è una funzione così lentamente crescente da assumere valori ≤ 4 per ogni grafo di al più ... vertici.

Bisogna poi mantenere aggiornata la struttura, fondendo le due componenti ogni volta che si aggiunge un lato a X. Questo richiede $O(1)$ operazioni.

$X := \emptyset$

$E' := E$

Build Heap (E')

Foreach $v \in V$ do

Make Set (V, v)
while $\mid X \mid < n - 1$ do
$e^* = (u^*, v^*) =$ ExtractMin(E')
$r^* =$ FindSet(u^*, V)
$s^* =$ FindSet$v^*, V)$
If $(r^* \neq s^*)$
Union(r^*, s^*, V)
$X := X \cup \{e^*\}$
$E' := E' \setminus \{e^*\}$

Complessivamente il termine che domina $T_i(n)$ è $T_{i1}(n) = O(\log m)$, per cui la complessità dell'intero algoritmo di Kruskal diviene $O(m \log n)$.

4.2 Algoritmo di Prim

L'algoritmo di *Prim* si basa sull'idea di partire istituendo le proprietà 2 e 3 (sottografo aciclico e connesso) e, in qualche modo 4, (sottografo di costo minimo rispetto ad un'opportuna famiglia di sottografi) e poi cercare di introdurre la proprietà 1 (sottografo ricoprente il grafo dato).
Esprimiamo la proprietà che il sottografo sia ricoprente con $|U| = n$, per cui $\delta = n - \mid U \mid$ misura la distanza dal soddisfare pienamente questa proprietà.
Un grafo che banalmente soddisfa le proprietà 2 e 3 è costituito da un solo vertice e nessun lato. Vi sono n grafi di questo genere, ma hanno tutti lo stesso costo, per cui si può sceglierne uno qualsiasi per partire.

Per ridurre la distanza dalla soluzione, bisogna aggiungere un nuovo nodo ad ogni passo. Siccome vogliamo conservare l'aciclicità e la connessione, dobbiamo aggiungere un nodo e un lato e il lato deve collegare il nuovo nodo ai vecchi. Infatti, se gli estremi del nuovo lato fossero entrambi già in U, il nuovo lato chiuderebbe un ciclo a causa del Teorema 1. Se i vertici estremi fossero entrambi nuovi, il sottografo risultante sarebbe sconnesso, anzi non sarebbe neppure un grafo, perché uno degli estremi del nuovo lato non apparterrebbe all'insieme dei vertici.
Serve quindi un lato in $\Delta(U)$. Ma quale scegliere?
Viene spontaneo pensare al lato di costo minimo in $\Delta(U)$.

Invariante di ciclo (versione scorretta)

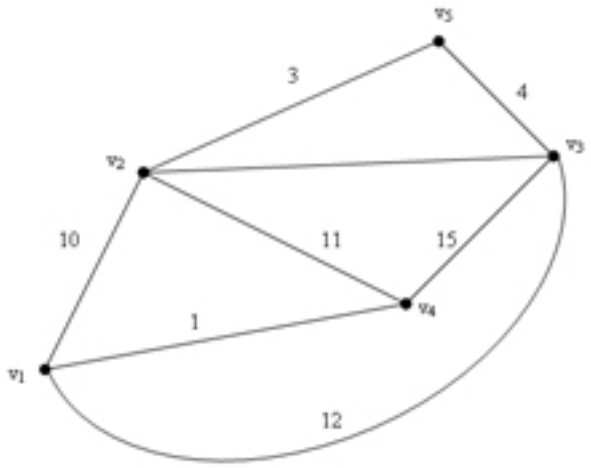
Una prima ipotesi di invariante di ciclo è quindi la seguente:

1. $T_k(U_k, X_k)$ connesso
2. $T_k(U_k, X_k)$ aciclico
3. $c_{X_k} \leq c_X$ per ogni X di k lati connesso e aciclico

Per $k = 0$, è facile istituire queste proprietà ponendo $X_0 = \emptyset$ e $U_0 = \{v\}$ qualunque sia $v \in V$.
Il problema è che questo invariante di ciclo non si riesce a conservare.
Se si parte da un grafo connesso, aciclico e minimo di k lati e gli si aggiunge il lato di costo minimo in $\Delta(U_k)$, il nuovo grafo è ancora connesso e aciclico (cioè è un albero), ma non è in generale il meno costoso fra gli alberi di $k+1$ lati.

Esempio: In figura 4.1 (v_1, v_4) è il sottografo connesso e aciclico di costo minimo fra quelli di $k = 1$ lati; e (v_1, v_2) è il lato di costo minimo nel taglio

Figura 4.1: Esempio di esecuzione dell'algoritmo di Prim



$\Delta(U_1)$. D'altra parte, aggiungere (v_1, v_2) dà un albero di $k+1 = 2$ lati che non è il minimo. Infatti, l'albero di costo minimo di due lati è $\{(v_2, v_5)(v_3, v_5)\}$. Questo è anche dimostrato dal fatto che il problema di trovare l'albero di k lati di costo minimo (k-Cardinality Tree Problem) è NP-difficile.

Ci viene in soccorso un'idea fondamentale in algoritmica e in matematica, cioè il *paradosso dell'inventore*.

Spesso è più facile dimostrare una proprietà forte che una meno forte.

Possiamo descrivere il concetto con una metafora. Eseguire algoritmi iterativi è come salire da un punto A ad un punto B per gradi. Il passo iniziale è il primo scalino, a cui segue una serie di altri scalini.
E' possibile che alcuni di questi scalini siano troppo alti, ovvero, che l'algoritmo non riesca a estendere l'invariante di ciclo dal gradino k al gradino $k+1$. Se però si ha un primo scalino più alto, è possibile che tutti i gradini successivi siano più bassi e che si finisca per aggiungere un punto B' più alto di B. In altre parole l'idea consiste nell'introdurre e conservare una proprietà più forte di quella effettivamente richiesta. Il primo passo diventa più difficile, ma nei successivi potremo dare per scontato che già valga una proprietà più forte e può essere che risulti facile estenderla.

Invariante di ciclo (versione corretta)

L'invariante che useremo sarà:

$$\begin{cases} T_k(U_k, X_k) \text{ connesso e aciclico} \\ U'_k \subset U_k \\ \mid U'_k \mid = \mid U_k \mid -1 \\ c_{X_k} \geq c_X \text{ rispetto a tutti i sottografi } T_k \text{ aciclici, connessi di } k \text{ lati contenenti } U'_k \end{cases}$$

Distanza

$\delta = n - \mid U \mid$

Conservazione dell'invariante e riduzione della distanza

Si aggiunge ogni volta a X_k il lato di costo minimo in $\Delta(U_k)$ e a U_k l'estremo di tale lato che non appartiene ancora a U_k.

Pseudocodice

$U = \{v_1\}$

$X = \emptyset$

While $U \subset V$ **do**

$e^* = (u, v) = \arg\min_{u \in U, v \in V \setminus U} C_{uv}$
$X = X \cup \{e^*\}$
$U = U \cup \{v\}$

Analisi di correttezza

1. L'invariante viene istituito, perché si pone $U'_0 = \emptyset$ e $U_0 = \{v\}$ con v vertice qualsiasi. Quindi $(\{v\}, \emptyset)$ è connesso e aciclico, U_0 contiene $U'_\emptyset$, ha un elemento in più ed è di costo nullo, rispetto a tutti i sottografi dello stesso tipo.

2. Si riduce $\delta = n - \mid U \mid$ perché si aggiunge un nuovo vertice a U.

3. Si conserva l'invariante perché X ingloba il lato di costo minimo di $\Delta(U)$ e U ingloba il vertice corrispondente. Inoltre U' diventa il vecchio U, per cui continua ad essere interno a U e di cardinalità pari a $\mid U \mid -1$. Resta solo da dimostrare che il nuovo grafo è di costo minimo rispetto a quelli che rispettano l'invariante.

4. Se $\delta = \emptyset$, significa che $\mid U \mid = n$, cioè $T(U, X)$ contiene n vertici e $n-1$ lati. Inoltre è aciclico, connesso e minimo rispetto a tutti i sottografi aciclici connessi di $n-1$ lati (dunque gli alberi ricoprenti) che contengono U'. Ora, gli alberi ricoprenti che contengono U' sono tutti gli alberi ricoprenti per cui l'ipotesi restrittiva che abbiamo introdotto all'inizio (e che è effettivamente restrittiva per $k = \emptyset$, ..., $n-2$), non lo è più per $k = n-1$.

Vediamo ora la dimostrazione di minimalità.

For v = 1 to n **do**

Uinc[v] = 0
Cmin[v] = $+\infty$
Pred[v] = 0

$\{U = \{v_1\}; X = 0\}$

Uinc[1] = 1

Cmin[1] = 0

For $v \in Adj(1)$ **do**

Cmin[v] = c_{1_v}
Pred[v] = 1

$m = 0$

While $m < n-1$ **do** $\{U \subset V\}$

$vMin = 0$
min = $+\infty$
For $v = 2$**to**n**do**
If $(Uinc[v] = 0)$**and**$(Cmin[v] < min)$
min = $Cmin[v]$
$vMin = v$
$Uinc[vMin] = 1$ $\{U = U \cup \{v\}\}$
For $v \in Adj(vMin)$ **do**
If $(Uinc[v] = 0)$ **and** $(c_{vvMin} < Cmin[v])$
$Cmin[v] = c_{vvMin}$
$Pred[v] = vMin$

Analisi di complessità

$$T(n) = T_s(n) + \sum_{i=1}^{i_{\max}} T_i(n) + T_e(n)$$

- $T_s(n) = O(1)$, si tratta solo di fissare $V = \{v\}$ e $X = \emptyset$
- il numero massimo di iterazioni $i_{\max}$ è uguale a $n - 1$
- Il passo iterativo ha complessità data da
  1. Ricerca del lato di costo minimo in $\Delta(U)$
  2. Aggiunta del lato a X
  3. Aggiunta degli estremi a U
- $T_e(n) = 0$

Come per l'algoritmo di Kruskal, la complessità dipende dal modo in cui si realizza nel passo iterativo la ricerca del lato di costo minimo.
Questo aspetto merita una discussione più approfondita.

Ricerca del lato di costo minimo in $\Delta(U)$

Presentiamo tre diverse implementazioni.

1. Implementazione banale Si può banalmente conservare l'insieme $\Delta(U)$ e scorrerlo per determinare l'elemento di costo minimo. In generale $\Delta(U)$ contiene $O(m)$ elementi, per cui $T_{i_1}(m) = O(m)$. Ovviamente, ogni volta che un nuovo vertice entra in U, tutti i lati in esso incidenti e diretti a vertici esterni ad U vanno aggiunti a $\Delta(U)$, mentre i lati incidenti diretti a vertici interni ad U vanno eliminati da $\Delta(U)$. Un'alternativa ancora più banale evita queste operazioni: basta scorrere l'intero insieme dei lati E e, testare ogni volta se il lato corrente appartiene a $\Delta(U)$ o no (cioè se i suoi estremi stanno uno in U e uno fuori, oppure enrambi dalla stessa parte).

2. Conservazione del lato minimo incidente in ogni vertice Dell'insieme $\Delta(U)$ ci interessano in realtà solo i lati che possono risultare di costo minimo in qualche passo dell'algoritmo: è inutile conservare gli altri. Supponiamo che $\Delta(U)$ sia strutturato in n sottoinsiemi, ciascuno dei quali contiene i lati incidenti in un diverso vertice u. Di ogni sottoinsieme $\Delta_u(U)$ interessa solo il lato di costo minimo. Inoltre, interessano solo i sottoinsiemi per cui

$u \neq U$. Quindi elimineremo gli altri lati e gli altri sottoinsiemi. Questa struttura complica il passo iniziale perché bisogna determinare per ogni u il lato di costo minimo fra u e l'unico elemento v di U. Ovviamente, esiste al massimo un lato di tale genere, per cui queste operazioni costano $O(n)$. La ricerca del lato di costo minimo in $\Delta(U)$ richiede ora $T_{i1}(n) = O(n)$.
Inoltre, quando si aggiunge il nuovo lato e^* a X e il nuovo vertice u^* a U, bisogna aggiornare gli insiemi $\Delta_u(U)$: bisogna eliminare $\Delta_{u^*}(U)$ e bisogna eventualmente modificare tutti gli altri $\Delta_u(U)$ verificando se il lato (u^*, u) che fa ora parte di $\Delta_u(U)$, ha costo minore di quello precedentemente conservato. Questa operazione costa anch'essa $O(n)$ (a essere rigorosi $O(\delta_{u^*})$, dato che va effettuata solo se il lato (u^*, u) esiste).
Il risultato complessivo è $T_i(n) = O(n)$, e quindi questa versione dell'algoritmo di Prim costa $O(n^2)$.

Pseudocodice

$U = \{v_1\}$;

For v = 1 to n **do**

    Cmin[v] = $+\infty$
    Pred[v] = 0

$\{X = 0\}$

Cmin[v_1] = 0

For $v \in Adj(v_1)$ **do**

    Cmin[v] = $c_{v_1 v}$
    Pred[v] = v_1

$m = 0$

While $m < n - 1$ **do** $\{U \subset V\}$

    $vMin = 0$
    min = $+\infty$
    For $v = 2$**to**n**do**
        If $(v \notin U)$**and**$(Cmin[v] < min)$
            min = $Cmin[v]$
            $vMin = v$

$U := U \cup \{vMin\}$

For $v \in Adj(vMin)$ **do**

If $(v \notin U)$ **and** $(c_{vvMin} < Cmin[v])$

$Cmin[v] = c_{vvMin}$

$Pred[v] = vMin$

Da notare che $\Delta_u(U)$ è descritto da due vettori:

- *Pred* che contiene per ogni $u \notin U$ il vertice più vicino ad u in U
- *Cmin* che contiene il costo del lato $(u, Pred(U))$

3. Implementazione tramite Heap E' anche possibile conservare gli n lati di costo minimo in uno heap binario, in modo da accelerare l'iterazione del lato di costo minimo. Ovviamente, questo aggiunge un ulteriore termine $O(n)$ al passo iniziale, riduce la ricerca del lato minimo di $O(n)$ a $O(1)$, ma aggiunge l'aggiornamento dei valori di ΔU per ciascuno dei lati incidenti nel nuovo nodo u_i aggiunto ad U. Ogni aggiornamento costa $O(\log n)$ e ne vengono eseguiti al più δ_{u_i} (grado del nodo n). Complessivamente, ne vengono eseguiti al più m perché ogni lato è considerato una sola volta.

- $T_s = O(n)$
- $T_i(n) = O(1) + O(\delta_{u_i})O(\log n)$
- $T_e(n) = \emptyset$

Quindi

$$T(n) = O(n) + \sum_{i=1}^{n-1} O(\delta_{u_i} \log n) = O(m \log n)$$