Università degli Studi di Bologna
Facoltà di Scienze Matematiche, Fisiche e Naturali
Corso di Laurea in Informatica

Appunti del corso di

Algoritmi e Strutture Dati

tenuto dal dott. Roberto Segala nell'A.A. 2000/01

"Un buon algoritmo è come un coltello affilato: fa esattamente ciò che si suppone debba fare applicando una quantità minima di forza. Invece usare un algoritmo sbagliato per risolvere un problema è come tagliare una bistecca con un cacciavite: si può anche arrivare ad un risultato accettabile, ma facendo sicuramente uno sforzo più grande di quanto non fosse necessario, e inoltre il risultato non sarà certo elegante."[1]

by Alessandro Bondi

[1] Pagina 14 del testo [2]

Questi appunti sono stati scritti a scopo puramente personale per la preparazione dell'esame di "Algoritmi e Strutture Dati" con il dott. Roberto Segala unendo

[1] Appunti delle lezioni in aula

[2] *Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest*, "Introduzione agli algoritmi", Jackson Libri, 1999 (Parti 1-7)

[3] *John C. Martin*, "Introduction to Languages and the Theory of Computation", Mc Graw Hill (Capitoli 16,17)

[4] *Nancy Lynch, Ronald L. Rivest*, "Basic Complexity Theory", Dispense delle lezioni del corso di "Automata, Computability and Complexity" del MIT di Boston

La maggior parte di essi è stata quindi copiata, tradotta e adattata dai suddetti testi, senza l'intenzione di ledere alcun diritto d'autore: l'utilizzo di tali testi è parte integrante di questa preparazione.

Non mi assumo nessuna responsabilità sulla correttezza dei contenuti di questi appunti e da ciò che può derivare dal loro utilizzo da parte di terzi.

Sommario

Sommario ... **3**

Introduzione ... **8**

Gli algoritmi ... *8*
Algoritmi randomizzati ... 8

Analisi di algoritmi ... *8*
Limite asintotico superiore ... 9
Limite asintotico inferiore ... 9
Limite asintotico stretto ... 9

Progettazione di algoritmi ... *9*
Approccio incrementale ... 9
Approccio "divide-et-impera" ... 9
Analisi di algoritmi "divide-et-impera" ... 9
Richiamo sui metodi di risoluzione per le equazioni di ricorrenza ... 10
Metodo di sostituzione ... 10
Metodo iterativo ... 10
Master Theorem ... 10

Algoritmi di ordinamento ... **11**
Problema dell'ordinamento ... 11
Proprietà ... 11
Altri algoritmi di ordinamento ... 11

Insertion sort ... *11*
Algoritmo ... 11
Complessità ... 11
Dimostrazione $\sum_{i=1..n} i = \frac{n \bullet (n+1)}{2}$ ... 12

Merge sort ... *12*
Algoritmo ... 12
Dimostrazione ... 12
Complessità ... 12
Proprietà ... 12

Heapsort ... *12*
Heap ... 12
Algoritmo ... 13
Mantenimento della proprietà di uno heap ... 13
Costruzione di uno heap ... 13
Complessità ... 14
Heapify ... 14
Build-Heap ... 14
HeapSort ... 14
Code con priorità tramite heap ... 14

Quicksort ... *14*
Algoritmo ... 14
Partizionamento dell'array ... 15
Complessità ... 15

Partizionamento peggiore 15
Partizionamento migliore 15
Partizionamento bilanciato 15

Ordinamento per confronto 16
Teorema del limite inferiore per l'ordinamento 16
Dimostrazione 16
Ordinamenti lineari 16

Counting sort 17
Ipotesi 17
Algoritmo 17
Complessità 17

Radix sort 17
Esempio di funzionamento 17
Algoritmo generale 18
Dimostrazione 18
Complessità 18

Bucket Sort 18
Ipotesi 18
Algoritmo 18
Dimostrazione 18
Complessità 19

Algoritmi di selezione su array non ordinato 20
Problema della selezione 20

Selezione del minimo e del massimo 20
Selezione simultanea 20
Gioco dei piccioni 20

Algoritmo con tempo medio lineare (probabilistico) 20
Idea 20
Algoritmo 20
Complessità 21

Algoritmo con tempo peggiore lineare 21
Idea 21
Algoritmo 21
Complessità 21

Strutture di dati avanzate 23

Richiami sulle Strutture di dati 23
Insiemi dinamici 23
Elementi di un insieme dinamico 23
Operazioni su un insieme dinamico 23
Interrogazioni 23
Operazioni di modifica 23

Revisione sugli Alberi binari di ricerca 24
Proprietà 24
Visita 24
Interrogazioni 24
Ricerca 24
Minimo e massimo 24

Successore e predecessore 24
Operazioni di modifica 25
Inserimento 25
Cancellazione 25

RB-Alberi 25
Idea 25
Proprietà 26
Complessità 26
Rotazioni 26
Inserimento 26
Cancellazione 27

Estensioni di strutture di dati 27
Utilizzo 27
Modalità generali 27
Estensioni di RB-alberi 28
Alberi di selezione 28
Determinazione del rango di un elemento 28
Ricerca di un elemento con un dato rango (selezione dell'*i*-esimo elemento) 29
Alberi di intervalli 29
Gli intervalli 29
Definizione 29
Passo 1: struttura dati di base 29
Passo 2: informazione aggiunta 29
Passo 3: mantenimento dell'informazione aggiunta 29
Passo 4: sviluppo delle nuove operazioni 30

Heap aggregabili e binomiali 30
Heap aggregabili 30
Heap binomiali 30

Teoria della complessità 31

La Macchina di Turing 31
Introduzione 31
Il computer come persona umana 31
L'idea di Alan Turing e Teorema di Church-Turing 31
Definizione Informale della MdT 31
Definizione Formale della MdT 32
Configurazione 33
Transizione 33
Esecuzioni erronee 33
Combinazione di MdT 33
Convenzioni 34
Varianti delle Macchine di Turing 34
La MdT a nastro infinito 34
La MdT con doppio nastro semi-infinito e doppia testina 34
La MdT Universale 34
Esempi e esercizi 35
Cancellazione del simbolo puntato 35
Sovrascrittura di una stringa x col simbolo 0 36
Riconoscere se la sottostringa "aba" è presente in una stringa x composta dai simboli [a, b] 37
Somma di due numeri binari 37

Introduzione 37
Problemi facili e difficili 37
Complessità di un linguaggio 37

Il “Triple matching” 37
Modello di riferimento 38

Classi di complessità *38*
TIME(T) 38
Linear Speedup Theorem 38
Dimostrazione 38
Classe P 39
Invarianza per modelli 39
Classe NP 39
MdT non deterministica 39
NTIME 39
La classe NP 40
NP-completezza 40

Problemi decisionali classici *40*
Teoria dei numeri 40
COMPOSITE = { w | w è un numero binario composto} 40
UCOMPOSITE = { w | w è un numero unario composto } 40
MOLTIPLICA = { (w, u, v) | binari, w=uv } 40
Teoria dei grafi 40
PATH = { (G, a, b) | a e b sono nodi di G ed esiste un cammino da a a b) 40
CLIQUE = { (G, k) | esiste un sottografo completo di k nodi } 41
VertexCover = { (G,l) | G ha un sottoinsieme di l vertici che tocca tutti gli archi } 41
EULT = { G | G ha un ciclo euleriano } 41
HAMC = { G | G ha un ciclo hamiltoniano } 41
TravelingSalesmanProblem (Problema del commesso viaggiatore) 41
Colorazione di mappe / grafi 41
MAP_2_COL 41
MAP_3_COL 41
MAP_4_COL 41
Matching 41
2_MATCH = { G | esistono $|V|/2$ archi che toccano un nodo esattamente una volta } 41
3_MATCH = { G | ammette claque di 3 nodi } 41
Logica 41
VAL = { ϕ, δ | ϕ [δ] = true } 41
GSAT = { ϕ | esiste un assegnamento δ che rende ϕ vera (ϕ è soddisfacibile) } 41
2SAT = { $\phi \in$ GSAT | ϕ è in 2CNF } 41
3SAT = { $\phi \in$ GSAT | ϕ è in 3CNF } 42
TAUTOLOGY = { ϕ | ϕ è una tautologia } 42

Strumenti di analisi *42*
CHOOSE 42
Problemi classici con la CHOOSE 42
GSAT(ϕ) 42
3COL(G) 42
CLIQUE(G, k) [Prima soluzione] 42
CLIQUE(G,k) [Seconda soluzione] 42
HAMC(G) 42
Riducibilità polinomiale 43
Riduzioni polinomiali classiche 43
VC $\leq_p$ CLIQUE e VC $\leq_p$ CLIQUE 43
3SAT $\leq_p$ VC 43
3SAT $\leq_p$ HAMC 44
GSAT $\leq_p$ 3SAT 44
3SAT $\leq_p$ CLIQUE e CLIQUE $\leq_p$ SAT 44

3COL $\leq_p$ 4COL 44
Esempio di un L $\leq_p$ GSAT 44

Introduzione

Gli algoritmi[2]

Un algoritmo è una qualsiasi procedura computazionale ben definita che prende alcuni lavori, o un insieme di valori, in input e produce alcuni valori, o un insieme di valori, come output. È quindi una sequenza di passi computazionale che trasformano l'input nell'output.

Un algoritmo può essere considerato uno strumento per risolvere un ben definito problema computazionale; infatti la definizione del problema specifica, in termini generali, la relazione che deve valere tra input e output e l'algoritmo descrive una procedura computazionale specifica per raggiungere tale relazione tra input e output.

Un'istanza di un problema consiste in tutti i dati di ingresso (che soddisfano i vincoli imposti nella definizione del problema) necessari a calcolare una soluzione.

Un algoritmo si dice corretto se, per ogni istanza di input, si ferma con l'output corretto; si dice che un algoritmo corretto risolve il problema computazionale dato. (Un algoritmo scorretto può essere utile solo se il tasso di errore può essere controllato: in generale, utilizziamo solo algoritmi corretti).

Algoritmi randomizzati

Un algoritmo si dice randomizzato se il suo comportamento è determinato non solo dall'input ma anche da valori prodotti da un generatore di numeri casuali (nella realtà si tratta di generatori di numeri pseudo-casuali). Tutti gli algoritmi randomizzati godono quindi della proprietà che nessun input particolare provoca il caso peggiore nel comportamento dell'algoritmo, il quale dipende dal generatore di numeri: quindi vi sono pochissime permutazioni dell'input che causano un comportamento simile a quello del caso peggiore.

Una strategia randomizzata è utile tipicamente quando vi sono molti modi in cui un algoritmo può procedere, ma è difficile determinare quello che promette di essere migliore.

Analisi di algoritmi[3]

Analizzare un algoritmo ha il significato di prevedere le risorse che l'algoritmo richiede; molto più spesso è interessante misurare il tempo computazionale.

In generale, il tempo richiesto da un algoritmo cresce con la dimensione dell'input, per cui descriveremo il tempo di esecuzione di un programma come una funzione della dimensione del suo input.

La dimensione dell'input dipende dal problema che dobbiamo studiare: nei problemi come l'ordinamento la misura più naturale è il numero degli elementi in input, nella moltiplicazione tra interi è il numero totale di bit, nei grafi può anche essere espressa con due numeri (numero di nodi e numero di archi).

Il tempo di esecuzione di un algoritmo per un particolare input è rappresentato dal numero di operazioni elementari o passi eseguiti per il calcolo dell'output corrispondente: esso è pari alla somma dei tempi di esecuzione di ciascuna istruzione.

Ordini di grandezza[4]

Anche con input delle stesse dimensioni, il tempo di esecuzione può dipendere da quale tipo di input è dato: quindi è possibile distinguere tra il caso migliore, peggiore e medio. Viene utilizzato più

[2] Pagina 1 del testo [2]

[3] Pagina 5 del testo [2]

[4] Pagina 8 del testo [2]

comunemente il tempo di esecuzione nel caso peggiore (cioè il tempo di esecuzione più lungo per qualsiasi input di dimensione n) per i seguenti motivi

- rappresenta una limitazione superiore sui tempi di esecuzione per qualsiasi input
- in genere si verifica abbastanza frequentemente
- il caso medio è in genere "cattivo" come quello peggiore (il tempo di esecuzione nel caso medio, detto anche atteso, sarà esaminato in casi particolari: rimane comunque il problema di riconoscere un input medio per un dato problema)

Spesso si assumerà che tutti gli input della stessa dimensione siano equamente probabili; nella pratica questa ipotesi può essere violata (tuttavia gli algoritmi randomizzati possono talvolta rendere vera, forzatamente, questa ipotesi)

Si può considerare il tasso di crescita o ordine di grandezza del tempo di esecuzione. A tale scopo si considera solo il termine principale di una formula, dato che i termini di ordine inferiore sono relativamente non significativi per valori di n molto grandi.

GRAFICI

Limite asintotico superiore

$$f(n) = O(g(n)) \Leftrightarrow \exists c, n_0 : \forall n > n_0 \; 0 \le f(n) \le c \bullet g(n)$$

Limite asintotico inferiore

$$f(n) = \Omega(g(n)) \Leftrightarrow \exists c, n_0 : \forall n > n_0 \; 0 \le c \bullet g(n) \le f(n)$$

Limite asintotico stretto

$$f(n) = \Theta(g(n)) \Leftrightarrow f(n) = O(g(n)) \wedge f(n) = \Omega(g(n)) \Leftrightarrow \exists c_1, c_2, n_0 : \forall n > n_0 \; 0 \le c_1 \bullet g(n) \le f(n) \le c_2 \bullet g(n)$$

Progettazione di algoritmi[5]

Approccio incrementale

È quello utilizzato da insertion sort: dopo aver ordinato il sottoarray *A[1...j-1]* si inserisce il singolo elementi *A[j]* nel proprio posto producendo il sottoarray *A[1..j]* ordinato.

Approccio "divide-et-impera"

Si applica ad algoritmi con struttura ricorsiva: per risolvere un determinato problema essi richiamo se stessi per gestire sottoproblemi analoghi a quello dato. Essi generalmente seguono l'approccio divide-et-impera che si struttura in tre passi

- Divide: il problema è suddiviso in un certo numero di sottoproblemi
- Impera: i sottoproblemi sono risolti ricorsivamente. Tuttavia, se la dimensione dei sottoproblemi è piccola a sufficienza, essi sono risolti direttamente.
- Combina (o fusione): le soluzioni dei sottoproblemi sono combinati per ottenere la soluzione del problema originale

Analisi di algoritmi "divide-et-impera"

Quando un algoritmo contiene una chiamata ricorsiva a se stesso, il suo tempo di esecuzione può spesso essere descritto con una equazione di ricorrenza o ricorrenza, che descrive il tempo di esecuzione complessivo di un problema di dimensione n in termini del tempo di esecuzione per input più piccoli.

Una ricorrenza per un algoritmo divide-et-impera è basata sui tre passi del paradigma di base.

$$T(n) = \begin{cases} \Theta(1) & n \le c \\ a \bullet T(n/b) + D(n) + C(n) & altrimenti \end{cases}$$

[5] Pagina 10 del testo [2]

Se la dimensione di un problema è sufficientemente piccola ($n \le c$ per qualche costante *c*) allora la risoluzione banalmente impiega un tempo costante. Altrimenti dividiamo il problema in *a* sottoproblemi di dimensione *n/b* impiegando tempo *D(n)* e tempo *C(n)* per combinare le soluzioni dei sottoproblemi nella soluzione del problema principale

Richiamo sui metodi di risoluzione per le equazioni di ricorrenza[6]

Metodo di sostituzione

Si tenta un limite e si usa l'induzione matematica per dimostrare che è corretto.

Metodo iterativo

Si sviluppa la ricorrenza e la si esprime come somma di termini dipendenti solo da *n* e dalle condizioni iniziali.

Master Theorem

Siamo $a \ge 1 \wedge b \ge 1$ (costanti). Data la funzione $T(n) = a \bullet T(n/b) + f(n)$ essa può essere asintoticamente limitata come segue (si suppone che $\varepsilon > 0$, costante):

- $f(n) = O(n^{\log_b a+\varepsilon}) \Rightarrow T(n) = \Theta(n^{\log_b a})$
- $f(n) = \Theta(n^{\log_b a+\varepsilon}) \Rightarrow T(n) = \Theta(n^{\log_b a} \bullet \log\ n)$
- $f(n) = \Omega(n^{\log_b a+\varepsilon}) \wedge a \bullet f(n/b) < c \bullet f(n) \Rightarrow T(n) = \Theta(f(n))$

[6] Pagina 51 del testo [2]

Algoritmi di ordinamento[7]

Problema dell'ordinamento

Il problema dell'ordinamento è il seguente: data in input una sequenza di n numeri $a_1, a_2, ..., a_n$, restituirne una permutazione (riordinamento) $a_1', a_2', ..., a_n'$ tale che $a_1' \leq a_2' \leq ... \leq a_n'$. In altri termini, un vettore si dice ordinato $\Leftrightarrow i < j \rightarrow A[i] \leq A[j]$.

Gli elementi di questo vettore sono intesi come record, contenente ciascuno una chiave a cui facciamo riferimento per l'ordine.

Il rango di un elemento è la sua posizione nell'insieme ordinato.

Proprietà

Ordinamento in loco – Non utilizziamo memoria aggiuntiva o in quantità costante (heap sort)

Stabilità – Non scambiamo elementi con chiave uguale (merge sort)

Altri algoritmi di ordinamento

- Bubble sort (confronti tra elementi contigui due a due) $\Omega(n^2)$
- Shake sort (come Bubble ma bidirezionale)

Insertion sort

Algoritmo[8]

L'insertion sort risulta efficiente per ordinare un piccolo numero di elementi. Esso si basa sulla ricerca del minimo.

```
for i=1 to length(A)
   /* ricerca del minimo */
   min = i
   for j=i+1 to length(A)
      if (A[j]<A[min]) then min←j
   /* shift e inserimento nella posizione corrispontente */
   x=a[min]
   for j=min downto i+1
      A[j] = A[j-1]
   A[i]=x
```

Se all'interno dell'if utilizziamo un ≤ invece che il <, l'algoritmo continua a funzionare ma perde la stabilità: la perde anche se facciamo uno scambio al posto dello shift facciamo (dovrei supporre che non esistono elementi uguali per non perderla).

Complessità[9]

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

Il caso migliore l'abbiamo quando gli elementi sono già ordinati: in tal caso il costo è lineare.

Il caso peggiore l'abbiamo quando gli elementi sono ordinati in ordine inverso.

[7] Pagina 129 del testo [2]

[8] Pagina 2 del testo [2]

[9] Pagina 7 del testo [2]

<u>Dimostrazione</u> $\sum_{i=1..n} i = \frac{n \bullet (n+1)}{2}$

Proviamo a sommare due sommatorie identiche

1a sommatoria	1	2	3	…	n-1	n-2	+
2a sommatoria	n	n-1	n-2	…	2	1	=
Totale	n+1	n+1	n+1	…	n+1	n+1	= n(n+1)

Pertanto basta dividere il risultato per 2.

Merge sort

Algoritmo[10]

Segue in modo stretto il paradigma divide-et-impera
- Divide gli *n* elementi della sequenza da ordinare in due sottosequenze di *n/2* elementi ciascuna
- Ordina, usando ricorsivamente il merge sort, le due sottosequenze
- Combina, con un merge, le due sottosequenze
-

```
MergeSort(A,p,r)
   if (p==r) return(A[p])
   q=p+⌈(r-p)/2⌉
   B1=MergeSort(A,p,q)
   B2=MergeSort(A,q+1,r)
   return(Merge(B1,B2))
```

Dimostrazione

(per induzione su *n*) - *n*=0, *n*=1 Ok

- *n*>1 B_1 e B_2 sono ordinati (per induzione). La `Merge`, dati due array ordinati, restituisce un array ordinato.

Complessità[11]

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 2 \bullet T(n/2) + \Theta(n) & n > 1 \end{cases} = O(n \bullet \log_2 n) \quad \text{(supponendo che } TMerge(n) = n\text{)}$$

Proprietà

Esso non ordina in loco, ma è stabile? Se *n*=1 non abbiamo nessun problema. Se *n*>1, B1 e B2 sono ordinati in modo stabile (per induzione su n), ma la Merge preserva la stabilità? Dipende da come è implementata: per essere stabile, occorre che sia presente il ≤ nel blocco

```
if (B1[i] ≤ B2[i]) then copy_from_B1 else copy_from_B2
```

Heapsort[12]

Heap

Un <u>heap binario</u> è un albero binario quasi completo: è riempito completamente su tutti i livelli tranne, eventualmente, il più basso che è riempito da sinistra in poi.

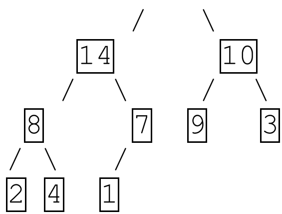
16

Un heap può comodamente essere memorizzato in un array: la radice

[10] Pagina 11 del testo [2]

[11] Pagina 13 del testo [2]

[12] Pagina 133 del testo [2]



dell'albero è $A[1]$ e dato il nodo $A[i]$ il padre *parent*(i)=$A[\lfloor i/2 \rfloor]$, il figlio sinistro *left*(i)=$A[2i]$, il figlio destro *right*(i)=$A[2i+1]$. La lunghezza del vettore è *length*(A) e la lunghezza dello heap all'interno del vettore è *heap-size*(A).

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
16	14	10	8	7	9	3	2	4	1

La proprietà di ordinamento parziale dello heap (brevemente proprietà dello heap) dice che per ogni nodo i diverso dalla radice $A[Parent(i)] \geq A[i]$, cioè che il valore di un nodo è minore o uguale al valore del padre. Quindi l'elemento più grande dello heap è memorizzato nella radice.

L'altezza di un nodo in un albero il numero di archi sul più lungo cammino semplice discendente che va dal nodo a una foglia e si definisce altezza di un albero l'altezza della sua radice: essa vale $\lfloor \log n \rfloor$ poiché si basa su un albero binario completo.

Algoritmo

Costruiamo uno heap sull'array $A[1\ldots n]$ (`Build-Heap`). L'elemento massimo è sulla radice e possiamo collocarlo nella sua posizione corretta scambiandolo con l'elemento $A[n]$. Ora rimane da ordinare la parte del vettore $A[1\ldots n\text{-}1]$: su questa parte si può facilmente ripristinare la proprietà dello heap (`Heapify`). Dopodiché possiamo tranquillamente ripetere il processo fino allo heap di dimensione 2.

```
HeapSort(A)
   Build-Heap(A)
   for i=length(A) downto 2
      swap(A[1],A[i])
      heap-size(A)--
      Heapify(A,1)
```

Mantenimento della proprietà di uno heap

Dato un array A e un indice i, tale che gli alberi binari con radice *left*(i) e *rigth*(i) siano heap ma che $A[i]$ possa essere più piccolo dei suoi figli, essa lascia "scendere" il valore di $A[i]$ nello heap così che il sottoalbero con radice di indice i diventi uno heap.

```
Heapity(A,i)
   l=left(i)
   r=rigth(i)
   if (l<=heap-size(A))&&(A[l]>A[i])
      then  largest=l
      else  largest=i
   if (r<=heap-size(A))&&(A[r]>A[i])
      then  larghest=r
   if largest!=i
      then  swap(A[i],A[largest])x
            Heapify(A,largest)
```

Ad ogni passo è determinato il più grande tra il nodo in posizione i e i suoi figli e il suo indice è memorizzato in *largest*. Se il più grande è $A[i]$ (i==largest) allora siamo a posto. Altrimenti vuol dire che uno dei due figli ha un elemento più grande e occorre quindi scambiare $A[i]$ con $A[largest]$: quindi il nodo i e i suoi figli soddisfano la proprietà dello heap ma ora occorre verificare se il sottoalbero $A[largest]$ la soddisfa, richiamando ricorsivamente la Heapify appunto sul nodo *largest*.

Costruzione di uno heap

Sappiamo che gli elementi del sottoarray $A[\lfloor n/2 \rfloor+1\ldots n]$ sono foglie dell'albero e quindi ognuno di essi p uno heap di un solo elemento. A questo punto non ci basta che attraversare i restanti nodi eseguendo Heapify su ognuno di essi (l'ordine in cui essi sono trattati garantisce che i sottoalberi con radice nei figli di un nodo i sono heap, prima che Heapify venga eseguita su quel nodo).

```
Build-Heap(A)
   heap-size(A)=length(A)
   for i=⌊length(A)/2⌋ downto 1
```

```
        Heapify(A,1)
```

Heapsort ordina in loco ma non è stabile.

Complessità

Heapify

Il tempo di esecuzione totale è dato dal tempo utilizzato per stabilire la relazione tra gli elementi $A[i]$, $A[left(i)]$, $A[right(i)]$, cioè $\Theta(1)$, più il tempo necessario per far eseguire Heapify su un sottoalbero con radice in uno dei due figli del nodo i. Ogni sottoalbero ha dimensioni al più di $2n/3$ (il caso peggiore si verifica quando l'ultimo livello dell'albero è pieno esattamente a metà): quindi il tempo di esecuzione può essere descritto dalla ricorrenza $T(n) \leq T(2n/3) + \Theta(1) = O(\log n)$, secondo il Master Theorem.

Build-Heap

Sarebbe corretto affermare che un limite superiore sul tempo di esecuzione è $O(n \log n)$ ma questo limite non è asintoticamente stretto. Si può osservare che il tempo per far eseguire l'Heapify su un nodo varia con l'altezza del nodo nell'albero e le altezze di molti nodi sono piccole. Sappiamo che uno heap di n nodi ha altezza $\lfloor \log n \rfloor$ e al più $\lceil n/2^{h+1} \rceil$ nodi di altezza h. Quindi il tempo richiesto da Heapify quando è chiamata su un nodo di altezza h è $O(h)$, pertanto il costo totale di Build-Heap è uguale a

$$\sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil \bullet O(h) = O\left(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} \right) \leq O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) = O(n \bullet 2) = O(n)$$

per il teorema[13] $\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$

HeapSort

$T_{HeapSort}(n) = T_{Build-Heap}(n) + (n-1) \bullet T_{Heapify}(n) = O(n \bullet \log n)$

Code con priorità tramite heap[14]

Argomento non trattato a lezione ma che comunque può essere utile.

Quicksort[15]

Quicksort è un algoritmo di ordinamento il cui tempo di esecuzione nel caso peggiore è $\Theta(n^2)$, ma spesso è la migliore scelta pratica perché in media è notevolmente efficiente (il suo tempo medio di esecuzione è $\Theta(n \log n)$ e i fattori costanti sono sufficientemente piccoli). Inoltre ordina in loco e funziona bene in ambienti con memoria virtuale.

Algoritmo

Il processo "divide-et-impera" per ordinata un sottoarray $A[p \ldots r]$ richiede queste tre fasi:

- L'array è ripartito (risistemato) in due sottoarray non vuoti $A[p \ldots q]$ e $A[q+1 \ldots r]$ in modo tale che ogni elemento di $A[p \ldots q]$ sia minore o uguale ad ogni elemento di $A[q+1 \ldots r]$. L'indice q è calcolato da una procedura di partizionamento (`Partition`)
- I due sottoarray sono ordinati con chiamate ricorsive a Quicksort
- Poiché i sottoarray sono ordinati in loco, l'intero array $A[p \ldots r]$ è subito ordinato

```
Quicksort(A,p,r)
   if p<r
      then  q=Partition(A,p,r)
```

[13] Pagina 41 del testo [2]

[14] Pagina 141 del testo [2]

[15] Pagina 145 del testo [2]

```
            Quicksort(A,p,q)
            Quicksort(A,q+1,r)
```

Per ordinare l'intero array la chiamata da effettuare è `Quicksort(A,1,length(A))`.

Partizionamento dell'array

Il cuore dell'algoritmo è la procedura Partition che risistema il sottoarray $A[p \ldots r]$ in loco

```
Partition(A,p,r)
   x=A[p]
   i=p-1
   j=r+1
   while true
      repeat j--
      until A[j]<=x
      repeat i++
      until A[i]>=x
      if i<j
         then  swap(A[i],A[j])
         else  return j
```

Prima viene selezionato un elemento $x=A[p]$ come elemento "perno" attorno al quale partizionare il sottoarray. Quindi vengono fatte crescere le due regioni $A[p \ldots i]$ e $A[j \ldots r]$ rispettivamente dall'inizio e dalla fine di $A[p \ldots r]$ in modo tale che ogni elemento in $A[p \ldots i]$ sia minore o uguale a x e ogni elemento in $A[j \ldots r]$ sia maggiore o uguale a x. Inizialmente le due regioni sono vuote.

Nel corpo del *while*, l'indice j viene decrementato e l'indice i viene incrementato fino a che $A[i] \geq x \geq A[j]$. Assumendo che queste disuguaglianze siano strette (non conta), $A[i]$ è troppo piccolo per appartenere alla regione alta. Così scambiando $A[i]$ e $A[j]$ si possono estendere le due regioni.

Questo corpo si ripete fino a che $i \geq j$; a questo punto l'intero array $p \ldots r$ è partizionato e il valore di q che serve al Quicksort è j.

Complessità

Innanzitutto la complessità della Partition è $\Theta(n)$.

La complessità totale dipende dal fatto che il partizionamento sia bilanciato o sbilanciato, che dipende da quale elementi sono usati per il partizionamento. Se il partizionamento è bilanciato tende verso $O(n \log n)$, se è sbilanciato tende verso $O(n^2)$.

Partizionamento peggiore

Avviene quando la Partition produce una regione con n-1 elementi e una con un solo elemento. Se questo partizionamento sbilanciato avviene ad ogni passo, il tempo di esecuzione è pari a

$$T(n) = T(n-1) + \Theta(n) = \sum_{k=1}^{n} \Theta(k) = \Theta\left(\sum_{k=1}^{n} k\right) = \Theta(n^2)$$

Partizionamento migliore

Avviene quando la Partition produce due regioni di dimensioni $n/2$. La ricorrenza è pari a

$$T(n) = 2 \bullet T(n/2) + \Theta(n) = \Theta(n \log n) \qquad \text{(per il Master Theorem)}$$

Partizionamento bilanciato

Il tempo di esecuzione del Quicksort nel caso medio è molto più vicino al caso migliore che al caso peggiore. Si supponga, per esempio, che l'algoritmo di partizionamento produca sempre una suddivisione 9 a 1, che sembra sbilanciata: allora si ottiene la ricorrenza

$$T(n) = T(9n/10) + T(n/10) + n = \Theta(n \log n) \qquad \text{(per il metodo iterativo)}$$

Quicksort probabilistico

Modificando la procedura Partition si può progettare una versione randomizzata del Quicksort che usi la seguente strategia di scelta casuale. Ad ogni passo dell'algoritmo Quicksort, prima di partizionare l'array, si scambia l'elemento $A[p]$ con un elemento scelto a caso in $A[p\ldots r]$. Questa modifica assicura che l'elemento perno $x=A[p]$ sia in modo equiprobabile uno qualunque degli r-p+1 elementi del sottoarray. DI conseguenza ci si aspetta che la suddivisione dell'array in input sia, in media, ragionevolmente ben bilanciata. Anche l'algoritmo randomizzato basato sulla permutazione casuale dell'array in input funziona bene in media, ma è po' più difficile da analizzare di questa versione.

```
Randomized-Partition(A,p,r)
   // effettua lo scambio prima dell'effettivo partizionamento
   i=Random(p,r)
   swap(A[p],A[i])
   return Partiton(A,p,r)

Randomized-Quicksort(A,p,r)
   if p<r
      then  q=Randomized-Partition(A,p,r)
            Randomized-Quicksort(A,p,q)
            Randomized-Quicksort(A,q+1,r)
```

Evitiamo in questa sede di analizzare l'algoritmo randomizzato, come per l'analisi completa della versione non randomizzata, essendo essa molto complicata e difficile dal punto di vista matematico.

Ordinamento per confronto[16]

Insertion sort, merge sort, heapsort e quicksort sono ordinamento che operano per confronto: determinano la posizione finale degli elementi confrontandoli l'un l'altro.

Teorema del limite inferiore per l'ordinamento

"Qualsiasi algoritmo di ordinamento per confronti ha una complessità minima pari a $O(n \log\ n)$"

Dimostrazione

Dato un array, una sola tra le *n!* permutazioni possibili è il nostro risultato. Occorre quindi tentare di eliminare le altre permutazioni. È come se creiamo un albero di decisione in cui ad ogni confronto riduciamo il numero possibile di permutazioni (lo spazio delle soluzioni), ottenendo ad ogni livello dei sottoinsiemi sempre più piccoli. Alla fine otteniamo un albero ternario (ad ogni confronto possiamo avere i casi '<', '=' e '>') le cui foglie (di numero *n!*) sono insiemi composti da una sola permutazione. Praticamente il nostro algoritmo deve decidere quale foglia prendere come ordinamento corretto, scendendo per l'albero (ad ogni confronto decide quale ramo prendere): la sua complessità è pari alla profondità massima dell'albero che, nel caso migliore di un albero bilanciato, è pari a $\log\ n! \cong \log\ n^n = n \log\ n$ (c.v.d).

Ordinamenti lineari

Un algoritmo di ordinamento può andare in $O(n) \Leftrightarrow$ non compie confronti diretti (ciò è possibile solo se conosciamo a fondo le relazioni d'ordine che ci sono tra i vari elementi).

[16] Pagina 163 del testo [2]

Counting sort[17]

Ipotesi

Il counting sort si basa sull'ipotesi che ognuno degli n elementi di input sia un intero nell'intervallo da $1 \dots k$, per qualche intero k. Quando $k=O(n)$, l'ordinamento viene eseguito in tempo $O(n)$.

Algoritmo

L'idea di base è di determinare, per ogni elemento x dell'input, il numero di elementi minori di x: questa informazione può essere usata per porre l'elemento x direttamente nella sua posizione nell'array di output (ad esempio, se ci sono 17 elementi minori di x, allora x va messo nella posizione 18). Nel caso in cui gli elementi possano assumere lo stesso valore, occorre modificare leggermente questo schema. Supponiamo di voler ordinare un array A di elementi compresi tra 1 e k (va bene anche da $a \dots a+k$ con l'opportuna sottrazione) e di voler mettere l'output in B:

```
Counting-Sort(A,B,k)
   // inizializziamo il vettore C (memoria temporanea) a 0 [O(k)]
   for i=1 to k
      C[i]=0
   // inseriamo in C[i] il numero di elementi uguali a i [O(n)]
   for j=1 to length(A)
      C[A[j]]++
   // inseriamo in C[i] il numero di elementi <= i [O(k)]
   for i=2 to k
      C[i]+=C[i-1]
   // inseriamo in B l'output ordinato [O(n)]
   for j=length(A) downto 1
      B[C[A[j]]]=A[i]
      C[A[j]]=C[A[j]]-1
```

Ordinamento stabile ma non in loco.

Esso funziona per qualsiasi struttura dati a cui possiamo associare una cardinalità finita fissata a priori. Se facciamo prima una ricerca del valore massimo all'interno dell'array, rischiamo di trovare un valore k esageratamente grande: se $k>n$, $T(n) \geq O(n^2)$.

Complessità

$$T(n) = O(k) + O(n) + O(k) + O(n) = O(n+k) = O(n) \qquad k = n \Rightarrow O(n+n) = O(n)$$

Il counting sort abbatte il limite inferiore di $\Theta(n \log n)$ perché non è un ordinamento per confronti. Esso usa i valori effettivi degli elementi per indicare direttamente un elemento all'interno di un array.

Radix sort[18]

Esempio di funzionamento

A_0 = (329, 457, 657, 839, 436, 720, 355)

Se applicassimo un counting sort con k=999 otterrei una complessità ben maggiore di n^2 (7x999>>7x7).

Ordiniamo A_0 per counting sort sulle cifre meno significative

A_1 = (720, 355, 436, 457, 657, 329, 839)

Ora ordiniamo A_1 sulla seconda cifra meno significativa

A_2 = (720, 329, 436, 839, 355, 457, 657)

Questo vettore è ordinato secondo le ultime due cifre (per la stabilità, che in questo caso è fondamentale per il funzionamento dell'algoritmo). Quindi ordiniamo per la cifra più significativa

[17] Pagina 166 del testo [2]

[18] Pagina 168 del testo [2]

```
A3 = (329, 355, 436, 457, 657, 720, 839)
```

e otteniamo così il vettore ordinato.

Algoritmo generale

Dato un array *A* di *n* elementi abbia *d* cifre, dove la cifra 1 è quella di ordine più basso mentre la cifra *d* è quella di ordine più alto

```
Radix-Sort(A,d)
   for i=1 to d
      usa un ordinamento stabile per ordinare l'array A sulla cifra i
```

Dimostrazione

(per induzione sulle colonne) *d* = numero di cifre
- se *d*=1 → ok, è un counting sort
- se *d*=(*d*-1)+1 → ok, per induzione

Complessità

se *d* è costante e $k=O(n)$ → $T(n) = d(n+k) = O(n)$

Ovviamente se *k* è basso, conviene utilizzare il counting sort.

Bucket Sort[19]

Ipotesi

Il bucket sort assume che l'input sia generato da un processo casuale che distribuisce gli elementi in modo uniforme sull'intervallo [0,1) (cioè, preso un qualsiasi sottointervallo la probabilità che un evento cada in uno di essi è uguale per tutti). L'idea è di dividere l'intervallo [0,1) in *n* sottointervalli uguali (bucket, secchi) e di distribuire gli *n* numeri nei bucket. Poiché gli elementi sono uniformemente distribuiti non ci si aspetta che molti numeri cadano nello stesso bucket. Per produrre l'output si ordinano semplicemente i numeri di ogni bucket e quindi si elencano gli elementi di ogni bucket prendendoli in ordine.

Algoritmo

In input abbiamo un array *A* di *n* elementi tale che $\forall i\ 0 \le A[i] < 1$. Il codice richiede un array ausiliario $B[0..n-1]$ di liste concatenate (bucket) e assume che vi sia un meccanismo per mantenere tali liste.

```
Bucket-Sort(A)
   n=length(A)
   for i=1 to n
      inserisci A[i] nella lista B[⌊n*A[i]⌋]
   for i=0 to n-1
      ordina la lista B[i]
   concatena insieme le liste B[0], B[1],…B[n-1] in quest'ordine
```

Dimostrazione

Prendiamo due elementi $A[i]$ e $A[j]$. Se essi cadono nello stesso bucket, allora compariranno ordinati in output dato che ogni bucket viene ordinato. Se essi cadono in bucket diversi, $B[i']$ e $B[j']$ (assumiamo senza perdita di generalità che $i'<j'$, eventualmente vale il processo inverso). Quando le liste di *B* vengono concatenate gli elementi del bucket $B[i']$ vengono prima di quelli del bucket $B[j']$ quindi $A[i]$ apparirà prima di $A[j]$. Sappiamo per certo (per la definizione dell'algoritmo) che $i' = \lfloor nA[i] \rfloor$ e che $j' = \lfloor nA[j] \rfloor$. Assumendo per assurdo che $A[i] > A[j]$, si ha che $\lfloor nA[i] \rfloor \ge \lfloor nA[j] \rfloor \rightarrow i' \ge j'$ che è assurdo per l'ipotesi iniziale di $i'<j'$.

[19] Pagina 170 del testo [2]

Complessità[20]

La dimostrazione richiede elementi di calcolo delle probabilità. Comunque, nonostante l'ordinamento dei singoli bucket possa avere costo quadratico, il tempo di esecuzione totale è $T(n)=O(n)$.

Gioco dell'ordinamento dei punti del cerchio per la distanza. Vedi appunti

[20] Pagina 172 del testo [2]

Algoritmi di selezione su array non ordinato[21]

Problema della selezione

Il problema della selezione è il seguente: abbiamo un array A di n numeri distinti e un numero i: $1 \leq i \leq n$. A noi interessa l'elemento $x \in A$ che è più grande di esattamente altri i-1 elementi di A, cioè l'i-esimo elemento nell'ordinamento di A.

Questo problema può essere risolto in tempo $O(n \log n)$, ordinando prima l'array con un algoritmo con costo $O(n \log n)$ e poi indirizzare l'i-esimo elemento dell'output.

Selezione del minimo e del massimo

Questi due algoritmi sono semplici e hanno costo lineare: basta esaminare tutti gli n elementi (facendo n-1 confronti) e tenere traccia dell'elemento più piccolo (o più grande) incontrato fino a quel momento.

```
Minimun(A)
   min=A[1]
   for i=1 to length(A)
      if min>A[i] then min=A[i]
   return(min)
```

```
Maximun(A)
   max=A[1]
   for i=1 to length(A)
      if max<A[i] then max=A[i]
   return(max)
```

Questi due algoritmi sono ottimi. Ulteriore raffinazione dell'analisi è la valutazione del numero medio di volte che viene eseguita la variazione del minimo (o del massimo) che è pari a $\Theta(\log n)$.

Selezione simultanea

Non è difficile ideare un algoritmo che possa trovare sia il minimo che il massimo usando un numero di confronti pari a $\Omega(n)$ asintoticamente ottimale. Basta trovare il minimo e il massimo in modo indipendente, usando n-1 confronti per ognuno, per un totale di $2n$-2 confronti.

In realtà sono sufficienti solo $3\lceil n/2 \rceil$ confronti. Manteniamo gli elementi minimo e massimo via via incontrati, ma piuttosto che analizzare ogni elemento dell'input confrontandolo sia col minimo che col massimo (2 confronti per elemento), si analizzano gli elementi in coppia: si confrontano i due numeri della coppia in input, quindi si confronta il più piccolo con il minimo e il più grande con il massimo (3 confronti ogni due elementi, invece che 4).

Gioco dei piccioni

E' SUGLI APPUNTI MA NON SUL LIBRO.

CHIEDERE COL BIZ DATO CHE NON CI CAPISCO UNA SEGA

Algoritmo con tempo medio lineare (probabilistico)[22]

Idea

Questo algoritmo segue lo stesso modello del Quicksort: l'idea è di partizionare ricorsivamente l'array di input, ma lavorando su un solo lato della partizione.

Algoritmo

Restituisce l'i-esimo elemento più piccolo dell'array $A[p\ldots r]$

```
Randomized-Select(A,p,r,)
   if p==r
```

[21] Pagina 175 del testo [2]

[22] Pagina 177 del testo [2]

```
      return(A[p])
   q=Randomized-Partition(A,p,r)
   k=q-p+1
   if i<=k
      then  return(Randomized-Select(A,p,q,i))
      else  return(Randomized-Select(A,q+1,r,i-k)
```

Dopo la Randomized-Partition, l'array è diviso in due sottoarray non vuoti tali che ogni elemento del primo è minore di ogni elemento del secondo. In k mettiamo il numero degli elementi del primo sottoarray. Determiniamo in quale dei due sia presente l'i-esimo elemento più piccolo: se $i \leq k$ allora è nel primo, altrimenti è nel secondo. Quindi eseguiamo la Randomized-Select fino a che non arrivo all'elemento ricercato.

Complessità

Nel caso peggiore è quadratica, ma nel caso medio è lineare.

L'ha fatta?

Algoritmo con tempo peggiore lineare[23]

Idea

Cerchiamo l'elemento desiderato sempre tramite partizionamenti ricorsivi dell'array in input: l'idea di base è però di garantire una buona suddivisione. Utilizziamo la Partition modificata in modo da prendere come parametro di input l'elemento perno attorno al quale partizionare.

Algoritmo

`Select` determina l'i-esimo elemento più piccolo di un array di input di n elementi eseguendo i seguenti passi

1. Divide gli n elementi dell'array di input in $\lfloor n/5 \rfloor$ gruppi di 5 elementi ciascuno e al più un gruppo costituito dai rimanenti n mod 5 elementi.
2. Trova il mediamo di ognuno degli $\lceil n/5 \rceil$ gruppi, ordinando con l'insertion sort gli elementi (max 5) di ogni gruppo.
3. Usa `Select` ricorsivamente per trovare il mediano x degli $\lceil n/5 \rceil$ mediano trovati al passo 2
4. Partiziona l'array di input attorno al mediano dei mediani x usando una versione modificata di Partition. Sia k il numero di elementi sul lato basso della partizione, così che n-k sia il numero di elementi sul lato alto
5. Usa `Select` ricorsivamente per trovare l'i-esimo elemento più piccolo sul lato basso se $i \leq k$ oppure l'(i-k)-esimo elemento più piccolo sul lato alto se $i > k$.

Complessità

Determiniamo un limite inferiore sul numero di elementi più grande dell'elemento partizionante x. Almeno metà dei mediani trovati al passo 2 sono maggiori o uguali dell'elemento mediano dei mediani x. Così almeno metà degli $\lceil n/5 \rceil$ contribuisce con 3 elementi più grandi di x, eccetto quel gruppo che ha meno di 5 elementi se n non è divisibile per 5 e quel gruppo contiene x stesso. Detraendo questi due gruppi, segue che il numero di elementi più grandi di x è almeno $3\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{5} \right\rceil \right\rceil\right) - 2 \geq \frac{3n}{10} - 6$. Analogamente il numero di elementi minori di x è almeno $3n/10$ - 6. Perciò nel caso peggiore Select è richiamata ricorsivamente su al più $7n/10 + 6$ elementi al passo 5.

[23] Pagina 179 del testo [2]

I passi 1, 2 e 4 richiedono tempo $O(n)$. Il passo 3 richiede tempo $T(\lceil n/5 \rceil)$. Il passo 5 richiede al più tempo $T(7n/10 - 6)$, assumendo che T sia monotona crescente. Si noti che $7n/10 - 6 < n$, per n>20 e che qualunque input di al più 80 elementi richiede tempo $O(1)$. Otteniamo la ricorrenza

$$T(n) \leq \begin{cases} \Theta(1) & n \leq 80 \\ T(\lceil n/5 \rceil) + T(7n/10 + 6) + O(n) & n > 80 \end{cases} \leq cn = O(n)$$

assumendo che $T(n) \leq cn$ per ogni $n>80$ e qualche costante c e dimostrando il tutto per sostituzione, poiché si può scegliere c sufficientemente grande in modo che $c(n/10\text{-}7)$ sia maggiore della funzione descritta dal termine $O(n)$ per ogni $c>80$.

Strutture di dati avanzate

Richiami sulle Strutture di dati

Insiemi dinamici

Un insieme dinamico è un insieme che viene manipolato da degli algoritmi che nel tempo può crescere, contrarsi o cambiare in altri modi.

Un dizionario è un insieme dinamico che offre la possibilità di inserire e cancellare elementi e di verificare l'appartenenza ad un insieme.

Elementi di un insieme dinamico

Ogni elemento è rappresentato da un oggetto i cui campi possono essere esaminati e manipolati se si ha un puntatore all'oggetto. Alcuni insiemi dinamici assumono che uno dei campi dell'oggetto sia un campo chiave che lo identifica (se sono tutte diverse, si può pensare ad un insieme di valori chiave). L'oggetto può contenere dati satellite che lo accompagnano ma che non sono utilizzati nella realizzazione dell'insieme.

Operazioni su un insieme dinamico

Interrogazioni

Restituiscono semplicemente informazioni sull'insieme

- *Search(S,k)*

Dato un insieme *S* e un valore chiave *k*, restituisce un puntatore *x* a un elemento in *S* tale che *key*[*x*]=k oppure *nil* se un tale elemento non appartiene a *S*.

- *Minimun(S)*

Dato un insieme totalmente ordinato *S*, restituisce l'elemento di *S* con chiave più piccola.

- *Maximun(S)*

Dato un insieme totalmente ordinato *S*, restituisce l'elemento di *S* con chiave più grande.

- *Successor(S,x)*
- Dato un elemento *x* la cui chiave appartiene a un insieme totalmente ordinato *S*, restituisce il successivo elemento più grande in *S* o *nil* se *x* è l'elemento minimo.
- *Predecessor(S,x)*

Dato un elemento *x* la cui chiave appartiene a un insieme totalmente ordinato *S*, restituisce il successivo elemento più piccolo in *S* o *nil* se *x* è l'elemento minimo.

Operazioni di modifica

Modificano l'insieme

- *Insert(S,k)*

Inserisce nell'insieme *S* l'elemento puntato da *x*. Di solito si assume che tutti i campi dell'elemento *x* necessari alla realizzazione dell'insieme siano già stati inizializzati.

- *Delete(S,k)*

Dato un puntatore *x* ad un elemento dell'insieme *S*, toglie *x* da *S*. (questa funzione utilizza un puntatore e non un valore chiave).

Revisione sugli Alberi binari di ricerca[24]

Proprietà

Per qualunque nodo x, le chiavi nel sottoalbero sinistro di x sono minori o uguali alla chiave di x e le chiavi nel sottoalbero destro di x sono maggiori o uguali alla chiave di x.

Per molte operazioni di ricerca il tempo di esecuzione nel caso peggiore è proporzionale all'altezza dell'albero, indicata con h. Se l'albero è bilanciato, h=log$n$.

Visita

Questa proprietà permette di elencare tutte le chiavi in modo ordinato attraverso un semplice algoritmo ricorsivo, la visita in ordine simmetrico (`Inorder-Tree-Walk(root(T))`) Questo algoritmo deriva il suo nome dal fatto che la chiave della radice di un sottoalbero è elencata tra i valori del suo sottoalbero sinistro e quelli del suo sotto albero destro. Analogamente, una visita in ordine anticipato elenca prima la radice quindi i valori di entrambi i sottoalberi e una visita in ordine differito elenca i valori dei sottoalberi della radice e dopo la radice stessa.

```
Inorder-Tree-Walk(x)
   if x!=nil
      then  Inorder-Tree-Walk(left[x])
            print(key[x])
            Inorder-Tree-Walk(right[x])
```

La correttezza dell'algoritmo deriva per induzione direttamente dalla proprietà dell'ABR. La sua complessità è $\Theta(n)$ dato che dopo la chiamata iniziale la procedura è chiamata ricorsivamente esattamente due volte per ciascun nodo dell'albero.

Interrogazioni

Teorema: Le operazioni si un insieme dinamico Search, Minimun, Maximun, Successor e Predecessor possono essere eseguite su un ABR di altezza h in tempo $O(h)$.

Ricerca

```
Tree-Search(x, k)
   if x==nil o k==key[x] then return(x)
   if k<key[x] then return(Tree-Search(left[x],k))
               else return(Tree-Search(right[x],k))
```

$T(n)=O(h)$

Minimo e massimo

```
Tree-Minimum(x)
   while left[x]!=nil
      x=left[x]
   return(x)
```

```
Tree-Maximun(x)
   while right[x]!=nil
      x=right[x]
   return(x)
```

$T(n)=O(h)$

Successore e predecessore

```
Tree-Successor(x)
   if right[x]!=nil
      then return(Tree-Minimun(right[x])
   y=p
   while y!=nil e x=right[y]
      x=y
      y=p[y]
   return(y)
```

[24] Pagina 299 del testo [2]

Operazioni di modifica

Teorema: Le operazioni Insert e Dolete su un insieme dinamico possono essere eseguite in tempo $O(h)$ utilizzando un ABR di altezza h.

Inserimento

```
Tree-Insert(T,z)
   y=nil
   x=root[T]
   while x!=nil
      y=x
      if key[z]<key[x]
         then  x=left[x]
         else  x=rigth[x]
   p[z]=y
   if y==nil
      then  root[T]=z
      else  if key[z]<key[y]
            then  left[y]=z
            then  right[y]=z
```

Cancellazione

```
Tree-Delete(T,z)
   if left[z]==nil o rigth[z]==nil
      then  y=z
      else  y=Tree-Successor(z)
   if left[y]!=nil
      then  x=left[y]
      else  x=rigth[y]
   if x!=nil
      then p[x]=p[y]
   if p[y]==nil
      then  left[p[y]]=x
      else  rigth[p[y]]=x
   if y!=z
      then  key[z]=key[y]
            // Se y ha altri campi copia anche questi
   return(y)
```

Cancellazione di un nodo *z* in un ABR:

- Se z non ha figli viene rimosso
- Se z ha un solo figlio si estrae z
- Se z ha due figli, si estrae il suo successore y, che ha al più un figlio, e quindi si sostituisce il contenuto di z con il contenuto di y

RB-Alberi[25]

Idea

Un RB-albero è un ABR con in più un campo binario in ogni nodo, il suo colore, che può essere RED oppure BLACK. Nessun cammino di un RB-albero risulti più lungo del doppio di qualsiasi altro: l'albero è approssimativamente bilanciato.

Ciascun nodo contiene i campi *color*, *key*, *left*, *right* e *p* [*father*]. Se i figli o il padre non esiste, il campo vale NIL: questi valori sono considerati come puntatori a foglie dell'ABR.

[25] Pagina 247 del testo [2]

Un numeri di nodi neri su un cammino da un nodo x, non incluso, ad una foglia è chiamato b-altezza del nodo ed è indicata con $bh(x)$. La b-altezza della radice è la b-altezza dell'albero. $bh(NIL)=0$.

Proprietà

Un ABR è un RB-albero se soddisfa le seguenti proprietà RB:

1. Ciascun nodo è rosso o nero
2. Ciascuna foglia è nera
3. Se un nodo è rosso, allora entrambi i figli sono neri
4. Ogni cammino semplice da un nodo ad una foglia sua discendente ha la stessa b-altezza.

Complessità

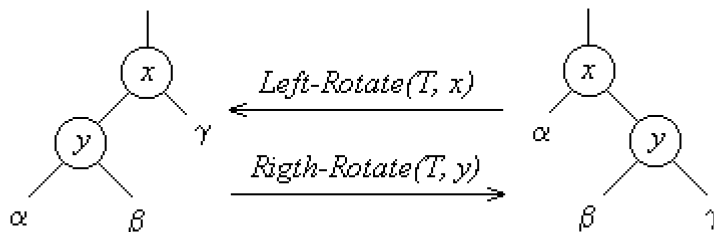
Un RB-albero con n nodi interni ha al più un altezza di $2\log(n+1)$.

> Dimostrazione: il sottoalbero radicato in qualsiasi nodo x contiene almeno $2^{bh(x)}-1$ nodi interni (per induzione sull'altezza di x: se $x=0$ allora x è una foglia e $2^0-1=0$. Altrimenti possiamo considerare che x abbia un'altezza positiva e che abbia due figli, ciascuno con una b-altezza di $bh(x)$ (rosso) o $bh(x)-1$ (nero): quindi per ipotesi induttiva ogni figlio ha almeno $2^{bh(x)-1}-1$ e il sottoalbero radicato in x contiene almeno $(2^{bh(x)-1}-1)+(2^{bh(x)-1}-1)+1=2^{bh(x)}-1)$. Sia h l'altezza dell'albero: per la proprietà 3 la b-altezza della radice deve essere almeno $h/2$, per cui $n\geq 2^{h/2}-1$ da cui si deriva che $h\leq 2\log(n+1)$.

Pertanto le interrogazioni sugli RB-alberi hanno $T(n)=O(\log n)$. Per le operazioni di modifica vale lo stesso ragionamento, ma occorre garantire che l'ABR modificato sia ancora un RB-albero.

Rotazioni

Per ripristinare le proprietà RB dopo le operazioni di modifica è necessario cambiare il colore a qualche nodo e modificare la struttura dei puntatori: tale operazione si chiama rotazione.



Le due operazioni cambiano un numero costante di puntatori - $O(1)$. Le lettere α, β e γ rappresentano sottoalberi arbitrati. Un'operazione di rotazione mantiene l'ordinamento delle chiavi secondo la visita in ordine simmetrico.

```
Left-Rotate(T, y)
   y=right(x)
   right(x)=left(y)   //rot. del subtree sx di y nel dx di x
   if left(y)!=NIL then p(left(y))=x
   p(y)=p(x)
   if p(x)==NIL then root(T)=y
                else if x==left(p(x)) then left(p(x))=y
                                      else right(p(x))=y
   left(y)=x
   p(x)=y
```

Inserimento[26]

Si inserisce il nodo x in un albero T come se fosse un normale ABR e lo si colora di rosso: si deve poi sistemare l'albero modificato ricolorando i nodi ed eseguendo le rotazioni

```
RB-Insert(T, x)
   Tree-Insert(T, x)
```

[26] Pagina 252 del testo [2]

```
color(x)=red
while (x!=root(T))&&(color(p(x))==red)    // fino a che c'è l'anomalia
   if p(x)=left(p(p(x))      // il padre di x è il figlio sx
   then  y=right(p(p(x))
         if color(y)==red
         then  color(p(x))=black          // caso 1
               color(y)=black             // caso 1
               color(p(p(x)))=red         // caso 1
               x=p(p(x))                  // caso 1
         else  if x==right(p(x))
               then  x=p(x)               // caso 2
                     Left-Rotate(T, x)    // caso 2
               color(p(x))=black          // caso 3
               color(p(p(x)))=red         // caso 3
               Right-Rotate(T, p(p(x)))   // caso 3
   else  {analogo al ramo then con "right" e "left" scambiati}
color(root(T))=black
```

L'unica proprietà che potrebbe essere violata nelle prime due istruzioni dell'algoritmo è la 3, che dice che un nodo rosso non può avere figli rossi; più precisamente questa proprietà è violata se il padre di x è rosso. Lo scopo del ciclo è di far "risalire" quest'anomalia: vi sono sei casi da analizzare, ma tre sono simmetrici agli altri tre a seconda che il $p(x)$ sia figlio sinistro o destro di $p(p(x))$. Le differenze tra il caso 1 e i casi 2, 3 nascono dal colore di y, lo "zio" di x: se è rosso si passa al caso 1, altrimenti ai casi 2 e 3. In ogni caso il "nonno" di x è nero.

Il caso 1 è eseguito quando sia $p(x)$ che y sono rossi. Dato che il nonno di x è nero, si colorano $p(x)$ e y di rosso (per la proprietà 4), ma potrebbe essere violata la proprietà 3 se il bisnonno di x è rosso: quindi si ripete il ciclo assegnando a x il valore di suo nonno.

Il caso 2 è eseguito quando y è nero e x è un figlio destro di suo padre, per cui si esegue una rotazione a sinistra per trasformare il caso 2 in 3 (senza influenzare la b-altezza dei nodi né la validità della proprietà 4).

Arrivando al caso 3 (direttamente o tramite il caso 2), sappiamo che y è nero: dobbiamo effettuare alcuni cambiamenti di colore dei nodi ed una rotazione a destra che preserva la proprietà 4, dopodiché l'esecuzione termina in quanto non ci sono più nodi rossi adiacenti (non viene ripetuto il ciclo perché $p(x)$ è nero).

Cancellazione

Chiedere quale cancellazione ha fatto

Estensioni di strutture di dati[27]

Utilizzo

In alcuni casi avremmo bisogno di utilizzare strutture di dati diverse da quelle esistenti: spesso sarà però sufficiente estendere quelle già esistenti aggiungendovi ulteriori informazioni e definire su di esse le nuove operazioni che permetteranno di ottenere il comportamento richiesto. Tale estensione non è sempre semplice da compiere: le informazioni aggiunte dovranno essere mantenute e aggiornate in modo corretto dalle operazioni della corrispondente struttura di dati di base.

Modalità generali[28]

L'estensione di una struttura di dati può essere suddivisa in quattro passi:

1. Scelta della struttura di dati di base

[27] Pagina 263 del testo [2]

[28] Pagina 268 del testo [2]

2. Determinazione dell'informazione aggiunta che deve essere mantenuta nella struttura di dati di partenza
3. Verifica che l'informazione aggiunta possa essere mantenuta attraverso le usuali operazioni di modifica della struttura di dati di partenza
4. Sviluppo delle nuove operazioni

Nella maggior parte dei casi lo sviluppo dei diversi passi avviene in parallelo.

Estensioni di RB-alberi

Teorema: Sia f un campo che estende un RB-albero T composto da n nodi e si supponga che il valore di f per un nodo x possa essere calcolato usato soltanto le informazioni dei nodi x, *left*(x) e *right*(x), compresi i valori di f. Allora, durante l'inserimento e la cancellazione, si può mantenere l'informazione f su tutti i nodi T senza influenzare asintoticamente il tempo $O(\log n)$ di queste operazioni.

L'idea di base è che una modifica del campo f di un nodo x si propaga nell'albero solo agli elementi di x: dopo che è stata aggiornata la radice dell'albero il processo termina. Quindi, visto che l'altezza di un RB-albero è $O(\log n)$, la modifica di un campo f costa tempo $O(\log n)$ per l'aggiornamento dei nodi che dipendono da tale modifica.

L'inserimento di un nodo x in un albero T è diviso in due fasi. Nella prima fase x viene inserito come figlio di un nodo esistente, quindi il valore di $f(x)$ può essere calcolato in tempo $O(1)$ dato che, per ipotesi, esso dipende solo dal contenuto degli altri campi di x stesso e dalle informazioni associate ai suoi figli (che sono NIL): dopo che $f(x)$ è stato calcolato, la sua modifica deve essere propagata all'indietro con tempo $O(\log n)$. Nella seconda fase l'unica modifica strutturale dipende dalle rotazioni: dato che compiamo al più due rotazione e ciascuna di esse cambia solo due nodi, il tempo impiegato per l'aggiornamento dei campi f è di $O(\log n)$.

Anche la cancellazione è composta da due fasi. Nella prima fase si verifica una modifica solo se il nodo cancellato è sostituito dal suo successore e poi di nuovo quando o il nodo cancellato o il suo successore sono effettivamente estratti. La propagazione degli aggiornamenti di f costa al più $O(\log n)$. Nella seconda fase si richiedono al più tre rotazioni che, come per l'inserimento, richiedono tempo complessivo $O(\log n)$.

<u>Alberi di selezione[29]</u>

Un <u>albero di selezione</u> T è un RB-albero con informazioni addizionali memorizzate in ogni nodo, come il campo *size* che contiene la dimensione del sottoalbero radicato in x (incluso). Definendo *size*(NIL)=0, si ha che *size*(x)=*size*(*left*(x))+*size*(*right*(x))+1.

Questo campo soddisfa le condizioni del teorema di estensione degli RB-alberi. Altri campi che lo soddisfano sono *sum*(x)=*sum*(*left*(x))+*sum*(*right*(x))+*key*(x) e *max*(x)=(*right*(x)==0?x;*right*(x)). Un campo che non lo soddisfa è *num_min* che indica il numero di elementi minori, dato che di volta in volta dovrei modificare tutti i nodi.

<u>Determinazione del rango di un elemento[30]</u>

Il rango di un elemento è il numero di nodi che precedono x nella visita simmetrica più 1 (per il nodo stesso).

```
Rank(T, y)
   r=size(left(x))+1
   y=x
   while (y!=root(T))
      if y==right(p(y))
      then  r=r+size(left(p(y)))+1
      y=p(x)
   return(r)
```

[29] Pagina 263 del testo [2]

[30] Pagina 265 del testo [2]

Ogni iterazione impiega tempo $O(1)$ e il tempo di esecuzione di *Rank* è nel caso peggiore proporzionale all'altezza dell'albero quindi $O(\log n)$.

Ricerca di un elemento con un dato rango (selezione dell'*i*-esimo elemento)[31]

La procedura *Select*(x, i) restituisce il puntatore al nodo che contiene la i-esima chiave più piccola nel sottoalbero radicato in x.

```
Select(x, i)
   r=size(left(x))+1
   if i==r
   then  return(r)
   else  if i<r
         then  return(Select(left(x),i))
         else  return(Select(right(x),i-r)
```

Il valore di *size*(*left*(x)) è il numero di nodi che precedono x nella visita simmetrica nel sottoalbero radicato in x.

Alberi di intervalli[32]

Gli intervalli

Un intervallo chiuso è una coppia ordinata di numeri reali $[t_1, t_2]$ dove $t_1 \leq t_2$: esso rappresenta l'insieme $\{t\in\Re: t_1 \leq t \leq t_2\}$. Intervalli aperti o semiaperti escludono dall'insieme rispettivamente entrambi o uno dei due estremi. Si può rappresentare un intervallo $[t_1, t_2]$ come un oggetto i con campi $low(i)=t_1$ (l'estremo sinistro) e $high(i)=t_2$ (l'estremo destro). Qualsiasi coppia di intervalli i e i' soddisfa la proprietà di tricotomia degli intervalli, cioè vale esattamente una delle tre proprietà:

a. si sovrappongono - se $i \cap i' \neq \Phi$, cioè se $low(i) \leq high(i')$ e $low(i') \leq high(i)$
b. i viene prima di i' - $high(i) \leq low(i')$
c. i' viene prima di i - $high(i') \leq low(i)$

Gli intervalli sono adatti alla rappresentazione di eventi ciascuno dei quali si manifesta in un periodo continuo di tempo.

Definizione

Un albero di intervalli è un RB-albero che memorizza un insieme dinamico di elementi con ciascun elemento x contenente un intervallo $int(x)$. Su di essi sono previste le seguenti operazioni:

- *Interval-Insert*(T, x) – aggiunge un elemento x, il cui campo *int* si suppone contenga un intervallo, all'albero di intervalli T.
- *Interval-Delete*(T, x) – rimuove l'elemento x
- *Interval-Search*(T, i) – restituisce il puntatore ad un elemento x nell'albero di intervalli T tale che $int(x)$ si sovrappone all'intervallo i, oppure NIL se un tale elemento non è nell'insieme

Passo 1: struttura dati di base

RB-albero con l'intervallo *int* e chiave *low* (l'estremo sinistro dell'intervallo). La visita in ordine simmetrico dell'albero elenca gli intervalli ordinati rispetto all'estremo sinistro.

Passo 2: informazione aggiunta

In aggiunta ogni nodo contiene il valore *max* che è il più grande degli estremi (ovviamente destro) degli intervalli memorizzati nel sottoalbero radicato in x.

Passo 3: mantenimento dell'informazione aggiunta

La condizione del teorema di estensione di RB-alberi è soddisfatta, dato che

$max(x)$=MAX($high(left(x))$,$max(left(x))$,$max(right(x))$)

[31] Pagina 264 del testo [2]

[32] Pagina 271 del testo [2]

Passo 4: sviluppo delle nuove operazioni

L'unica nuova operazione è *Interval-Search*(*T*, *i*).

```
Interval-Search(T, i)
   x=root(T)
   while (x!=NIL)&&(i non si sovrappone a int(x))
      if (left(x)!=NIL)&&(max(left(x))>=low(i))
      then  x=left(x)
      else  x=right(x)
   return(x)
```

Per verificare la correttezza si deve capire perché sia sufficiente esaminare un solo cammino a partire dalla radice: tale dimostrazione dipende dalla proprietà di tricotomia degli intervalli.

Heap aggregabili e binomiali

Heap aggregabili

Le strutture dati conosciute come heap aggregabili forniscono le seguenti operazioni:

Make-Heap() – crea e restituisce un nuovo heap vuoto

Insert(H, x) – inserisce un nodo *x*, la cui chiave è gi

Heap binomiali[33]

-

[33] Pagina 382 del testo [2]

Teoria della complessità

La Macchina di Turing

Introduzione

Con le Macchine di Turing (che ora chiameremo MdT) ci proponiamo di definire un computer semplicissimo (con cui programmare diventa difficilissimo) ma con cui è possibile fare tutto quello che posso fare con un computer moderno e con un linguaggio di programmazione qualunque.

Il computer come persona umana

Negli anni '30/'40 per "computer" si intendeva quella persona che con carta e penna poteva risolvere vari problemi in maniera algoritmica, in particolar modo conti matematici. Le caratteristiche di questo "computer" sono:

- memoria illimitata e stabile (non infinita!)
- memoria di calcolo interna finita
- può leggere e scrivere
- può cambiare la zona di lavoro spostandosi sul foglio sia in direzione orizzontale che in quella verticale.

L'idea di Alan Turing e Teorema di Church-Turing

Alan Mathison Turing
(1912-1954)

Alan Turing formulò una macchina astratta che fosse capace di compiere le stesse operazioni di un "computer umano", capace quindi di risolvere tutti i problemi definiti tramite algoritmi.

Egli limitò l'operato del computer: supponiamo che

- sia possibile lavorare (leggere e scrivere) su un quadretto per volta
- ci si possa spostare in un'unica direzione orizzontale.

Nonostante queste limitazioni il mio "computer" può fare tutte le operazioni che vuole.

I passi primitivi di un'esecuzione sono:

- esaminare un simbolo
- scrivere, sostituire o cancellare un simbolo
- trasferire l'attenzione da una parte all'altra del foglio

Ogni azione successiva dipende da due fattori:

- il simbolo appena esaminato
- l'attuale "stato della mente" (memoria)

La memoria può cambiare ad ogni computazione, ma il punto cruciale di tutto ciò è il fatto che essa ha un numero finito di stati possibili.

Teorema fondamentale di quest'analisi è quello di Church-Turing che afferma che ogni procedura (algoritmo) che può essere svolto da uno o più "computer umani" può essere eseguito da una o più MdT.

Definizione Informale della MdT

La MdT è composta dai seguenti componenti:

- un nastro semi-infinito
- una testina di lettura/scrittura
- un automa a stati finiti (FSA)

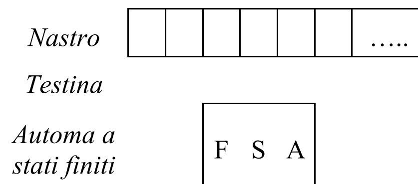


Figura 1 - La Macchina di Turing

Il nastro è semi-infinito, cioè è limitato a sinistra ma infinito a destra, ed è suddiviso in celle. Ogni cella può contenere uno e uno solo simbolo appartenente ad un dato alfabeto finito. Se una cella non contiene alcun simbolo, per convenzione diciamo che essa contiene il simbolo Δ (o BLANK).

È conveniente distinguere tra l'alfabeto di input, che contiene tutti i simboli che possono essere presenti sul nastro come input o output (cioè quello con cui l'utente interagisce con la macchina), e l'alfabeto di nastro, che può contenere dei simboli aggiuntivi che possono essere utilizzati durante l'esecuzione.

La testina di lettura/scrittura è controllata dal FSA: in ogni momento essa agisce su una singola cella del nastro (la cella corrente in quel momento) e può muoversi in qualsiasi direzione all'interno del nastro e può leggere e modificare ogni simbolo che incontra.

Le operazioni di base che la macchina può compiere sono:

- leggere un simbolo dalla cella corrente
- scrivere un simbolo sulla cella corrente
- spostare la testina di una posizione o a destra o a sinistra
- cambiare stato

La decisione di compiere una delle ultime tre operazioni (2,3,4) dipende dal simbolo della cella corrente e dallo stato del FSA.

Definizione Formale della MdT

Una Macchina di Turing (MdT) è una 5-tupla

T = (Q, , , *q0*,)

ove

Q = insieme finito di stati, non contenente lo stato *h* (*halt*) che indica la terminazione

Σ = alfabeto di input, non contenente Δ

Γ = alfabeto di nastro, non contenente Δ

q_0 = lo stato iniziale ($\in$ Q)

δ = funzione di transizione

$\delta: Q \times (\Gamma \cup \{\Delta\}) \rightarrow (Q \cup \{h\}) \times (\Gamma \cup \{\Delta\}) \times D$

E' una funzione parziale (cioè può non essere definita in certi punti) che preso uno stato e un simbolo dell'alfabeto restituisce un altro stato, un altro simbolo dell'alfabeto e un elemento dell'insieme D.

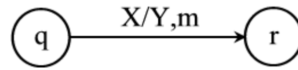
D = {L, R, S}

Sono i movimenti che può compiere la testina (L sposta a sinistra, R a destra, S nessun spostamento)

La nostra MdT ha quindi un nastro con un limite a sinistra ma infinito a destra, diviso in celle. Ogni cella può contenere un simbolo appartenente all'insieme $\cup \{\Delta\}$, ma in ogni momento tutte le celle, tranne che un numero finito di esse che possono contenere solo simboli di Σ, contengono il simbolo Δ. Per convenienza immaginiamo le celle numerate da sinistra a destra, iniziando da 0. La MdT ha anche una testina che può leggere e scrivere in una singola cella e che può muoversi a sinistra e a destra di un passo alla volta.

Se $q \in Q$, $r \in Q \cup \{h\}$, $X, Y \in$ $\cup$ { }, $m \in D$, la formula

$$(q, X) = (r, Y, m)$$



indica che quando la MdT è nello stato q e il simbolo corrente è X, essa sostituisce X con Y all'interno della cella, cambia il suo stato in r e fa compiere alla testina lo spostamento m. Se $r = h$, l'esecuzione termina: pertanto non esisterà mai alcuna definizione di $\delta(h, X)$. Quindi per definire la funzione di transizione occorre scrivere tutte le possibili transizioni della MdT.

Un altro modo per rappresentare δ è di utilizzare il prodotto cartesiano:

$$= \{ (q, X, r, Y, m), (q', X', r', Y', m'), (q'', X'', r'', Y'', \underline{m''}), ... \}.$$

Tramite la FSA posso memorizzare un numero finito di caratteri: il numero di stati necessario per memorizzare n caratteri è la cardinalità di Γ elevata alla n ($|\Gamma|^n$).

Configurazione

Per descrivere completamente una MdT dobbiamo specificare il suo stato, il contenuto del nastro e la posizione della testina. Per fare ciò definiamo come configurazione di una MdT una coppia

$$(q, x\underline{c}y)$$

ove q è lo stato attuale, x è tutto ciò che c'è prima della testina, c il simbolo puntato dalla testina (che deve essere sottolineato) e y tutto ciò che c'è dopo la testina (tutti i caratteri successivi a y sono uguali a Δ).

Se w è una stringa con lunghezza > 1 e scriviamo $(q, x\underline{w}y)$, intendiamo che la testina è posizionata sul primo simbolo di w.

Transizione

Definiamo con

$$(q, x\underline{c}y) \vdash (q', x'\underline{c'}y')$$

una transizione singola, cioè il fatto che la MdT passa dalla configurazione $(q, x\underline{c}y)$ alla configurazione $(q', x'c'y')$ in un solo passo computazionale.

Parallelamente definiamo con

$$(q, x\underline{c}y) \vdash^* (q', x'\underline{c'}y')$$

una transizione multipla, cioè il fatto che la MdT passa dalla configurazione $(q, x\underline{c}y)$ alla configurazione $(q', x'\underline{c'}y')$ in zero o più passi computazionali.

Se scriviamo

$$(q, x\underline{c}y) \vdash^* (h, x'\underline{c'}y')$$

diciamo che la MdT, a partire dallo stato q e con input $x\underline{c}y$, termina con un output pari a $x'\underline{c'}y'$.

Esecuzioni erronee

Come tutti i computer normali, una MdT durante un esecuzione può

- *halt* - terminare
- *loop* - andare in loop infinito
- *crash* - si verifica quando la MdT si trova in uno stato q e con un simbolo corrente X per cui δ(q, X) non è definito.

Per i nostri obiettivi, non ci interessa la gestione degli errori in input dato che essa richiede molto codice aggiuntivo.

Combinazione di MdT

Possiamo pensare di combinare più MdT "eseguendole" consecutivamente. Per capire ciò, basti pensare ad una MdT come un'implementazione di un certo algoritmo, determinato dalla funzione di transizione. Può capitare che la soluzione di un problema possa essere: "Prima esegui l'algoritmo A; poi quello B; poi quello C;..." e così via. Intuitivamente quindi una MdT può essere "lanciata" dopo un'altra: ovviamente esse devono lavorare sullo stesso nastro. Per questa ragione, non assumiamo il fatto che alla partenza di una MdT la testina sia posizionata sulla posizione 0.

Convenzioni

Pertanto, esistono varie convenzioni da seguire quando usiamo le MdT in maniera isolata.

All'inizio del nastro (cella in posizione 0) conviene scrivere il carattere Δ per poter riconoscere il limite sinistro del nastro. Quindi la stringa x viene scritta a partire dalla cella in posizione 1 e facciamo partire la MdT dalla configurazione (q, Δx). Pertanto se essa deve essere utilizzata all'interno di una esecuzione più ampia, l'importante non è che parta dalla posizione 0 ma che da un certo input yΔx, per ogni y.

Varianti delle Macchine di Turing

Dobbiamo verificare se tutte le macchina e i linguaggi di programmazione sono equivalenti alla MdT precedentemente definita, cioè che esse abbiano la stessa potenza di calcolo.

Due macchine si dicono equivalenti quando hanno lo stesso identico comportamento dato un certo input (halt con lo stesso ouput, loop, crash). Potrebbe comunque essere necessario codificare l'input e l'ouput di una macchina, quindi costruire un emulatore.

La MdT a nastro infinito

Supponiamo di avere una MdT (che chiamiamo MdT_∞) che lavora su un nastro non limitato a sinistra, quindi infinito.

Y X

…..	y2	y1	y0	O	x0	x1	x2	…..

Figura 2 - La MdT a nastro infinito

E' la MdT_∞ equivalente con quella classica? Per verificarlo dobbiamo vedere se è possibile definire un emulatore della MdT_∞ che "giri" su una MdT classica.

Supponiamo di memorizzare i dati presenti a destra e a sinistra dell'origine in maniera alternata.

O	y0	x0	y1	x1	y2	x2	…..

Figura 3 - Prima emulazione della MdT a nastro infinito

... UNDER CONSTRUCTION ...

La MdT con doppio nastro semi-infinito e doppia testina

Supponiamo che la nostra MdT lavori su due nastri semi-infiniti e che ognuno di essi abbia una testina indipendente (cioè la testina del primo nastro può trovarsi in una posizione diversa da quello del secondo).

	x0	x1	x2	x3	x4	…..

	y0	y1	y2	y3	y4	…..

Figura 4 - La MdT a doppio nastro semi-infinito

... UNDER CONSTRUCTION ...

La MdT Universale

Von Neumann ebbe l'idea di separare il software dall'hardware, cioè di mettere il programma in memoria. Pertanto occorre costruire una macchina in modo tale che dato il programma e l'input del programma essa restituisca l'output relativo.

Esiste una MdT che possa fare questo? La risposta è sì. Essa è una macchina con 7 stati[34] composta tra 3 nastri: uno che contiene l'input e l'output, uno che contiene lo stato e uno che contiene il programma.

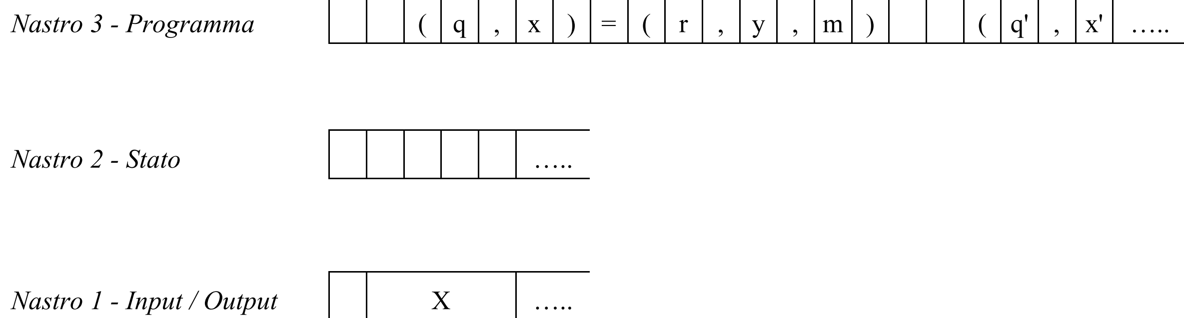


Figura 5 - La MdT universale

... UNDER CONSTRUCTION ...

Esempi e esercizi

Nel cercare una soluzione ad un certo problema, occorre prestare attenzione a vari aspetti:

- non posso utilizzare alcun tipo di contatore: per ricordarmi una certa posizione posso eventualmente raddoppiare l'alfabeto utilizzando oltre ai simboli normali anche i simboli sottolineati
- posso utilizzare dei simboli come segnaposto (ad esempio #), appartenenti a Γ ma non a Σ, per ricordarmi eventuali punti di arrivo
- nel momento in cui sovrascrivo qualche cosa, devo essere sicuro di ricordarmi le informazioni necessarie per il proseguimento dell'esecuzione e gli eventuali punti di arrivo

Cancellazione del simbolo puntato

Vogliamo definire una MdT che cancelli da una stringa il simbolo attualmente puntato dalla testina (quindi con successivo shift dei simboli successivi) e che al termine punti al primo Δ presente a sinistra. Formalmente ciò equivale a dire

$$(q_0, \Delta x\underline{c}y) \vdash^* (h, \Delta x\underline{y})$$

Ecco l'algoritmo informale:

- al posto del carattere da eliminare inserisco un segnaposto (#)
- scandisco tutta la stringa verso destra finché non arrivo alla prima cella contenente
- torno indietro verso sinistra e faccio lo shift (prima di sovrascrivere un simbolo devo ovviamente ricordarmi il suo valore).
- quando trovo il segnaposto #, lo sovrascrivo col carattere letto al passo precedente
- continuo a spostarmi verso sinistra fino a che non trovo il primo Δ

Ed ecco la definizione della mia MdT:

Q = { q, qR, qΔ, q0, q1, qL }
Σ = { 0, 1 }
Γ = { 0, 1, # }
q0 = q (stato iniziale)

e la definizione della funzione di transizione

Inizio dell'esecuzione e scrittura del segnaposto
(q, 0) = (qR, #, R)
(q, 1) = (qR, #, R)
δ(q, #) non si verifica mai
δ(q, Δ) *supponiamo non si verifichi mai*

[34] Secondo Dijistra se ne potrebbe costruire una anche con 5, ma nessuno c'è ancora riuscito.

Avanzamento a destra fino a che non si trova il primo Δ
δ(qR, 0) = (qR, 0, R)
δ(qR, 1) = (qR, 1, R)
δ(qR, Δ) = (qΔ, Δ, L)
quando trovo Δ, me lo ricordo e inizio a tornare indietro
Indietreggiamento a sinistra e shift fino a che non trovo il segnaposto #
δ(qΔ, 0) = (q0, Δ, L)
se prima ho visto Δ e trovo 0 → scrivo Δ, mi ricordo 0 e vado a sinistra
δ(qΔ, 1) = (q1, Δ, L)
se prima ho visto Δ e trovo 1 → scrivo Δ, mi ricordo 1 e vado a sinistra
δ(qΔ, #) = (qL, Δ, L)
se prima ho visto Δ e trovo # → scrivo Δ, e mi ricordo che devo tornare in cima alla stringa
(q0, 0) = (q0, 0, L)
(q0, 1) = (q1, 0, L)
(q0, #) = (qL, 0, L)
come per qΔ
δ(q1, 0) = (q0, 1, L)
(q1, 1) = (q1, 1, L)
(q1, #) = (qL, 1, L)
come per qΔ
Indietreggiamento fino alla cima della stringa
δ(qL, 0) = (qL, 0, L)
se devo tornare in cima e vedo 0 → vado a sinistra
δ(qL, 1) = (qL, 1, L)
se devo tornare in cima e vedo 1 → vado a sinistra
(qL,) = (h, , S)
se devo tornare in cima e vedo Δ → termino l'esecuzione

È leggermente più difficile, ma sicuramente molto più chiaro, scrivere la funzione di transizione tramite un diagramma degli stati (grafo) come automa di Mealy (quello utilizzato nel corso) o di Moore.

<u>Sovrascrittura di una stringa x col simbolo 0</u>

$$(q0, \Delta x) \vdash^* (h, \Delta 0)$$

Soluzione:

Vado in fondo, torno indietro scrivendo Δ, scrivo 0 nella posizione 1 (in questo modo mantengo l'informazione relativa al punto di arrivo, trovando Δ quando torno indietro).

δ = {
inizio
(q0, Δ, qR, Δ, R),
avanza a destra fino alla fine della stringa
(qR, 0, qR, 0, R),
(qR, 1, qR, 1, R),
quando trova Δ, va a sinistra e inizia a tornare indietro
(qR, Δ, qL, Δ, L),
torna indietro a sinistra e scrive Δ su tutte le celle

(qL, 0, qL, Δ, L),
(qL, 1, qL, Δ, L),
quando trova Δ, va a destra
(qL, Δ, qS, Δ, R),
scrive 0 e torna alla cima della stringa a sinistra
(qS, Δ, h, 0, L) }

Riconoscere se la sottostringa "aba" è presente in una stringa x composta dai simboli [a, b]

(q0, Δx) |-- * (h, Δ 0) se la sottostringa "aba" non compare in *x*.
(q0, Δx) |-- * (h, Δ 1) se la sottostringa "aba" compare in *x*.

Σ = { a, b }

Somma di due numeri binari

Somma di due numeri binari *x* e *y*, i quali hanno la stessa lunghezza e il cui bit meno significativo (LSB) è a sinistra.

(q0, x y) ├* (h, (x+y)

Introduzione

Il riconoscimento di linguaggi ricorsivi (insiemi di stringhe costruite secondo certe regole) e le funzioni calcolabili corrispondono ai problemi fondamentali che devono essere risolti da un computer, che mette a disposizione il tempo e la memoria necessari. Analizzare i problemi senza fare assunzioni sul limite dei costi permette di dare delle definizioni più generali; d'altro canto il semplice fatto che un problema è risolvibile non fornisce alcuna informazione sul suo costo computazionale. In queste analisi ci occuperemo di analizzare problemi decisionali relativi a linguaggi, che dicono se un certo oggetto appartiene ad una certo linguaggio o meno.

Problemi facili e difficili

Occorre capire se un certo problema risolvibile è facile o difficile: per questo essi sono racchiusi in classi di complessità che vengono definite in base ai limiti dei loro costi computazionali. Questi costi possono essere valutati in modi differenti: nel campo delle MdT essi possono essere quantificati in base allo spazio occupato in memoria oppure al numero di transizioni (tempo).

Complessità di un linguaggio

La complessità di un linguaggio può essere misurata tramite una funzione che associa alla lunghezza di un input un limite superiore del numero di passi computazionali necessari per calcolare l'output. La sua dipendenza con la lunghezza del problema è ovvia. In generale, una MdT impiega almeno *n*+2 passi per leggere un input di lunghezza *n* e Nper scrivere la risposta (0 o 1).

Dato un linguaggio $L \subseteq \Sigma^*$, una MdT chiamata *M* e una funzione $T : N \to \Re^{\geq 0}$, dire che "*M* valuta *L* in tempo *T*" significa che *M* valuta *L* e termina al massimo in $\max(n+2, \lceil T(n) \rceil)$ passi (ove *n* è la lunghezza dell'input $w \in \Sigma^*$). Questo concetto, se abbiamo una funzione $f : \Sigma^* \to \Delta^*$, può essere applicato anche all'affermazione"*M* calcola *f* in tempo *T*".

Il "Triple matching"

Supponiamo di avere un linguaggio $L = \{w \in \{a,b,c\}^* : N_a(w) = N_b(w) = N_c(w)\}$, cioè composto da tutte le stringhe finite *w* (composte dai caratteri *a*, *b*, *c*) che hanno lo stesso numero di *a*, di *b* e di *c* ($abccba \in L, abba \notin L$) e di voler definire una MdT che indichi le una stringa è corretta (1) o meno (0).

_*aabaccbcb*

In una MdT a un nastro non possiamo contare le lettere (non possiamo mantenere in memoria dei contatori). Potremmo cancellare le terne: ogni volte che trovo una *a*, cancello anche una *b* e una *c*; se alla fine ho cancellato tutti i caratteri, allora la stringa era corretta. Per fare ciò dobbiamo scandire la stringa al massimo $n/3$ volte e ogni volta dobbiamo fare al massimo *n* passi, quindi la complessità è di $O(n^2)$.

Notiamo che input differenti della stessa lunghezza cambiano drasticamente il numero di passi: per questo analizziamo il caso peggiore, dato che non ci interessa calcolare il numero esatto di passi.

Supponiamo ora di utilizzare una MdT a 4 nastri così configurata: _*aabaccbcb*Δ,_,_,_. Scandiamo la stringa e ogni volta che troviamo una *a* la scriviamo sul 2° nastro, una *b* sul 3° e una *c* sul 4°. Raggiungiamo quindi la configurazione Δ*aabaccbcb*_, Δ*aa*<u>*a*</u>,Δ*bb*<u>*b*</u>,Δ*cc*<u>*c*</u> in $O(n)$. A questo punto facciamo indietreggiare le testine dei nastri 2, 3 e 4 e verifichiamo se quando una legge Δ lo leggono anche le altre, in $O(n)$. A questo punto cancelliamo la stringa iniziale e scriviamo 0 o 1, in $O(n)$.

Abbiamo così abbassato il limite inferiore di $O(n^2)$ in $O(n)$.

Modello di riferimento

Abbiamo visto che che non sempre la soluzione più ovvia è quella ottimale: analizzando in maniera intelligente gli algoritmi possiamo anche trovare dei miglioramenti.

Visto che a noi interessa identificare i problemi difficili, dobbiamo definire un <u>modello di riferimento</u>. La scelta è caduta sulle MdT a più nastri (se implemento una MdT a più nastri su una MdT a un nastro la complessità cambia di un ordine di grandezza con una perdita quadratica). Si pensa che qualsiasi modello di calcolo ragionevole (cioè implementabile fisicamente) sia una espressione polinomiale di una MdT. Quindi indicheremo generalmente con MdT una macchina multinastro.

I limiti di complessità dipendono dalla rappresentazione: nelle analisi della complessità è importante scegliere una rappresentazione del problema che fornisca una "giusta" misura della "dimensione" del problema.

Classi di complessità

TIME(*T*)

Sia una funzione $T : N \to \Re^{\geq 0}$, definiamo TIME(*T*) come l'insieme dei linguaggi *L* valutati da una MdT in tempo *T*. (vogliamo costruire una MdT che verifichi se una stringa appartiene al linguaggio *L*, quindi prendo tutti quei linguaggi che vengono riconosciuti in tempo ≤ *T*).

Osserviamo che $T(|w|)$ deve essere un limite superiore nel numero di passi necessari e che $\forall n \in N, T_1(n) \leq T_2(n), L \in TIME(T_1) \Rightarrow L \in TIME(T_2)$.

Dato che la definizione di "*M* valuta *L*" tratta le risposte 1 e 0 simmetricamente, $L \in TIME(T) \Leftrightarrow \overline{L} \in TIME(T)$.

Inoltre due MdT possono essere simulate in parallelo da una MdT con un numero di nastri e di testine uguale alla somma di quelle simulate, cioè $L_1, L_2 \in TIME(T) \Rightarrow L_1 \cup L_2, L_1 \cap L_2 \in TIME(T(n) + 2n + k)$ (occorre anche aggiungere i tempi di copia dell'input più un numero costante di operazioni).

Linear Speedup Theorem

Questo teorema dice che $L \in TIME(T), \varepsilon > 0 \Rightarrow L \in TIME(T'), T' = \varepsilon T(n) + 2n + 14$, cioè che posso abbassare le costanti su $T(n)$ a patto di aggiungere un $2n$ ($27n^2 \to n^2 + 2n$), quindi che le costanti non hanno alcuna importanza nello studio della complessità (dal punto di vista teorico, visto che è più facile lavorare senza di esse: da quello pratico esse sono importanti…).

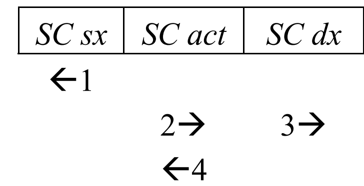
<u>Dimostrazione</u>

Costruiamo una macchina M_2 con lo stesso numero di nastri di *M*. M_2 valuta *L* in tempo T_2. Ogni cella di un nastro di M_2 contiene *m* simboli di *M* (chiamiamole super-celle, SC).

All'inizio dobbiamo "comprimere" l'input di *M* nell'input di M_2: questo impiega tempo $2n+4$.

Vogliamo fare in modo che ogni pdc (passo di computazione) di M_2 emuli m pdc di M. Per fare ciò dobbiamo

- Leggere la SC corrente e quelle adiacenti (sx e dx): (dalla SC corrente in m passi su M potrei al massimo arrivare solo in queste due) (4 pdc di M_2)
- Emulo internamente m pdc di M (0 pdc di M_2)
- Scrivo le SC coinvolte nell'operazione (4 pdc di M_2)
- Sposto eventualmente la testina nella SC successiva (1 pdc di M_2)



per un totale di 9 pdc: per ottenere un risultato accettabile $m \geq 9/\varepsilon$. Il costo di queste operazione è di $9T(n)/m + 9 \leq \varepsilon T(n) + 9$. Alla fine occorre decodificare l'output, per un altro pdc. Il totale è quindi al massimo di $\varepsilon T(n) + 2n + 14$.

Questo teorema funziona perché non facciamo alcuna misura relativa allo spazio occupato, ma in ogni caso è un valido indicatore del fatto che le costanti nono significanti. In questo modo possiamo operare per stime di ordini di grandezza.

Aggiungere osservazioni relative agli ordini di grandezza (pagina 6 della dispensa) con i dati all'inizio degli appunti

Classe P

È conveniente distinguere tra i problemi che hanno complessità polinomiale in n e tra quelli che hanno complessità esponenziale in n (o maggione). Possiamo quindi la classe P, cioè dei linguaggi riconosciti da una MdT con complessità descrivibile da un polinomo, come:

$$P = \bigcup_{k>0} TIME\left(O\left(n^k\right)\right)$$

Questa classe viene utilizzata comunemente per identificare comunemente i problemi "facili" (o "risolvibili" dal punto di vista computazionale). Ovviamente se il polinomio è del tipo $10^{150}n^{107}$, sarà difficile poter chiamare il linguaggio "facile": questa definizione serve per essere certi che se ci troviamo davanti ad una formula esponenziale quel problema sarà sicuramente "difficile". Comunque, in genere, se esiste una soluzione polinomiale ad un certo problema, prima o poi si dovrebbe riuscire a trovare una soluzione meno costosa: a noi interessa far "entrare" un problema nella classe P.

Invarianza per modelli

Questa classe è molto vasta: viene quindi comunemente suddivisa il modelli (1, log$n$, n, n^2, n^3…). Prendiamo un linguaggio in TIME(n^2) e due input, uno di lunghezza n e l'altro di lunghezza $2n$, vediamo che il tempo complessivo vale n^2 per il primo e $4n^2$ per il secondo. Ciò per il fatto che presi due linguaggi appartenenti allo stesso modello le cui rispettive lunghezze variano per un fattore costante, anche i loro tempi computazionali varieranno per un altro fattore costante (ciò non vale per le formule esponenziali).

Classe NP

MdT non deterministica

Si vedrà più avanti che alcuni problemi "difficili" possono diventare "facili" se esiste una funzione non deterministica (dato lo stato della MdT e l'input non possiamo prevedere a priori cosa restituisce, ma solo cosa potrebbe restituire). Occorre quindi definire una MdT non deterministica $NdMdT = \langle Q, \Sigma, \Gamma, q_0, \delta \rangle$ in cui $\delta \subset Q \times \Gamma \times 2^{Q \times \Gamma \times \{L,R,S\}}$: se siamo nella configurazione c, possiamo giungere a più di una configurazione in un solo passo computazionale.

NTIME

Dato un linguaggio $L \subseteq \Sigma^*$, una NdMdT chiamata M e una funzione $T : N \to N$, dire che "M valuta L in tempo T" significa che M accetta L per ogni $w \in L$ al massimo in $T(|w|)$ passi.

$$\text{NTIME}(T) = \{\ L \text{ accettati da una NdMdT in tempo } T(n)\ \}$$

Visto che una MdT può essere considerata come una NdMdt, $TIME(T) \subseteq NTIME(T)$. Inoltre si dimostra che $NTIME(T) \subseteq \bigcup_{r>1} TIME(r^T)$.

La classe NP

Da qui si può definire $NP = \bigcup_{k>0} NTIME(n^k)$. Anche per questa classe vale l'invarianza per modelli. Ancora non si sa se $P = NP \cap co-NP$ o se $P \neq NP$. Questo è uno dei più importanti problemi aperti nel campo dell'informatica teorica.

NP-completezza

Un problema A è NP-hard $\Leftrightarrow \forall B \in NP, B \leq_p A$. Un problema A si dice NP-completo cioè $A \in NPC \Leftrightarrow A \in NP \wedge A$ è NP-hard.

Il teorema di Cook-Levin dimostra che GSAT è *NPC*: quindi qualsiasi problema $B : GSAT \leq_p B$ è in *NPC* ($A \in NPC \wedge B \in NP \wedge A \leq_p B \Rightarrow \forall C \in NP, C \leq_p A \leq_p B \Rightarrow B \in NPC$). Dimostreremo avanti che 3SAT, VC, CLIQUE, HAMC sono problemi in *NPC*.

Se un qualsiasi problema *NPC* è risolvibile in tempo polinomiale, allora *P* = *NP*. In maniera equivalente, se un qualsiasi problema in *NP* non è risolvibile in tempo polinomiale, allora tutti i problemi *NPC* non sono risolvibili in tempo polinomiale (se un problema *A* appartiene sia a *P* che a *NPC*, allora è più difficile di tutti i problemi *NP*, quindi *P*=*NP*; parallelamente se un problema *A* appartiene a *NP* ma non a *P* e un problema *B* appartiene a *NPC*, se *B* appartiene anche a *P* allora *A* appartiene anch'esso a *P*, il che contraddice le ipotesi).

Problemi decisionali classici

Teoria dei numeri

COMPOSITE = { *w* | *w* è un numero binario composto}

Dobbiamo provare tutti i divisori da 1 a $2^{n/2}$. Ogni divisione di un numero binario *w* per un numero binario di lunghezza ≤ *n*/2 impiega tempo kn^2. Il miglior algoritmo conosciuto impiega tempo $n^{\log \log n}$.

$T(n) = kn^2 2^{n/2} \Rightarrow COMPOSITE \in TIME(O(n^2 2^{n/2}))$

Nonostante la sua soluzione sia esponenziale, se fosse vera la congettura di Rieman, relativa alla teoria dei numeri, per cui si riuscirebbe a trovare una soluzione polinomiale, questo problema sarebbe in *P*.

Da questo problema deriva quello della fattorizzazione, che attualmente è un problema difficile su cui si basano tutte le teorie relative alla crittografia.

UCOMPOSITE = { w | w è un numero unario composto }

Per fare ciò, occorre convertire la lunghezza *n* del numero in binario (tempo *kn*) che sarà lungo a sua volta *c*log(*n*). Dopodiché a questo numero si applica COMPOSITE. Problema facile.

$T(n) = kn + k(\log(n))^2 2^{\log(n)/2} = kn + k(\log(n))^2 \sqrt{n} \Rightarrow UCOMPOSITE \in TIME(O(n))$

MOLTIPLICA = { (*w*, *u*, *v*) | binari, *w*=*uv* }

Moltiplicazione classica, problema facile

$T(n) = kn^2 \Rightarrow MOLTIPLICA \in TIME(O(n^2))$

Teoria dei grafi

PATH = { (*G*, *a*, *b*) | *a* e *b* sono nodi di *G* ed esiste un cammino da *a* a *b*)

Problema facile, basta utilizzare l'algoritmo dei cammini minimi o di visita. *TIME*(*O*(n^2)).

CLIQUE = { (G, k) | esiste un sottografo completo di k nodi }

Provo tutti i sottoinsiemi di k vertici (2^k) e ogni volta verifico che i nodi sono tutti collegati tra loro (n^2 coppie per cui necessitano n scansioni). $TIME(O(n^3 2^n)$. Problema difficile.

VertexCover = { (G,l) | G ha un sottoinsieme di l vertici che tocca tutti gli archi }

Provo con tutti i sottoinsiemi di l nodi. $TIME(O(n^2 2^n))$. Problema difficile.

EULT = { G | G ha un ciclo euleriano }

Cioè G è un grafo orientato in cui è presente un ciclo che tocca ogni arco una e una sola volta (es. gioco in cui bisogna disegnare la casa senza staccare la penna dal foglio). Esiste un teorema che dice che un grafi G ammette un ciclo se e solo se il grado entrante di ogni nodo è uguale al suo grado uscente. Problema facile.

HAMC = { G | G ha un ciclo hamiltoniano }

Cioè G è un grafo in cui è presente un ciclo che tocca ogni nodo una e una sola volta. Occorre provare tutte le permutazioni: ci sono $n!$ ($n! = O(n^n) = O(2^{n\log n})$) ordini possibili in cui visitare i nodi e per ognuno di essi si può verificare se è un ciclo hamiltoniano in n^2. Problema difficile.

TravelingSalesmanProblem (Problema del commesso viaggiatore)

Preso un grafo G con gli archi pesati e un numero b, verificare se esiste un ciclo hamiltoniano il cui costo totale è $\leq b$. TSP = { (C,b) | C è una matrice di costi relativa a un grafo G, b intero ≥ 0, G ha un ciclo hamiltoniano ci costo massimo b }. Problema difficile, simile a HAMC.

Colorazione di mappe / grafi

Data una mappa (con un grafo G), verificare se è possibile colorarla con c colori in modo tale che regioni (nodi) adiacenti abbiano colori diversi?

(Nota: se abbiamo un grafo generico, le cose cambiano: GRAPH_2_COL è facile, GRAPH_3_COL e GRAPH_4_COL sono difficili).

MAP_2_COL

Coloriamo arbitrariamente un nodo. Se tutti i nodi adiacenti non sono colorati, li coloriamo con l'altro colore. Se troviamo un nodo adiacente con lo stesso colore, allora non è 2-colorabile. Problema facile.

MAP_3_COL

Problema difficile

MAP_4_COL

Problema facile. Esiste un teorema che dice che qualsiasi mappa è 4-colorabile.

Matching

2_MATCH = { G | esistono $|V|/2$ archi che toccano un nodo esattamente una volta }

Problema della ruota panoramica (dover riempire ogni cabina con esattamente 2 persone). Si utilizza la tecnica dinamica: preso un macthing in cui non prendiamo tutti i vertici, proviamo a creare le altre coppie. Il matching massimale è più complicato perché il grafo non è bipartito. Problema facile.

3_MATCH = { G | ammette claque di 3 nodi }

Problema difficile.

Logica

VAL = { ϕ, δ | ϕ [δ] = true }

Basta provare l'assegnamento e semplificare. Problema facile.

GSAT = { ϕ | esiste un assegnamento δ che rende ϕ vera (ϕ è soddisfacibile) }

Esistono 2^n assegnamenti da provare. Problema difficile.

2SAT = { $\phi \in$ GSAT | ϕ è in 2CNF }

Cioè verificare se è nella forma $(x_1 \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2)$. Problema facile.

<u>3SAT = { ϕ ∈ GSAT | ϕ è in 3CNF }</u>

Cioè verificare se è nella forma $(x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee \overline{x_3})$. Problema difficile.

<u>TAUTOLOGY = { ϕ | ϕ è una tautologia }</u>

Cioè se tutti i possibili assegnamenti rendono ϕ vera. Problema difficile.

Strumenti di analisi

CHOOSE

Abbiamo visto che i problemi "difficili" sono tali perché dobbiamo trovare qualche cosa ma non sappiamo come fare. D'altra parte se qualcuno ci da una risposta, riusciamo ad affermare facilmente se è corretta.

Definiamo la funzione non deterministica $CHOOSE : \{0,1\} \rightarrow \{0,1\}$. Assumiamo che questa funzione abbia complessità costante e che se esiste un risultato che permette al programma di funzionare correttamente, essa lo restituisce (altrimenti avremo un valore a caso). Essa funziona per ogni insieme di valori finiti: se ne ho k, devo fare log$k$ CHOOSE in cascata, una per ogni bit della codifica in binario dei valori.

Problemi classici con la CHOOSE

<u>GSAT(ϕ)</u>

Siano $x_1, x_2, x_3,\ldots,x_n$ variabili in ϕ

$\forall i, x_i \leftarrow CHOOSE(0,1)$

return(EVAL(ϕ,x_1, x_2, x_3…))

<u>3COL(G)</u>

Siano $v_1, v_2, v_3,\ldots,v_n$ nodi di G

$\forall i, v_i \leftarrow CHOOSE(R,G,B)$ // Assegnamento

$\forall i, j \quad if\,(v_i, v_j) \in E \wedge v_i.colore = v_j.colore$ return(false) // Verifica

return(true)

<u>CLIQUE(G, k) [Prima soluzione]</u>

$\forall i \in 1...k, x_i \leftarrow CHOOSE(V(G))$

$\forall i, j$ if $(v_i, v_j) \notin E$ return(false)

$\forall i \neq j$ if$(x_i = x_j)$return(false)

return(true)

<u>CLIQUE(G,k) [Seconda soluzione]</u>

$\forall i \in 1...k, x_i \leftarrow CHOOSE(0,1)$ // indica se un nodo è stato scelto o meno

if $(\Sigma x_i < k)$ return(false)

$\forall i, j$ if $(x_i \wedge x_j \wedge (x_i, x_j) \in E)$ return(false)

return(true)

<u>HAMC(G)</u>

$V \leftarrow V(G)$

$\forall i_{1...|V|} \quad x_i \leftarrow CHOOSE(V)$

$V \leftarrow V - x_i$

$\forall i_{1...|V(G)|-1}$ if $(x_i, x_{i+1}) \notin E$ return(false)

if $(x_i, x_{|V(G)|}) \notin E$ return(false)

return(true)

Riducibilità polinomiale

Un linguaggio L1 può essere ridotto ad un altro linguaggio L2 se una qualsiasi istanza di L1 può essere "riformulata" in poco tempo come un'istanza di L2, e la soluzione di quest'ultima fornisce una soluzione all'istanza di Q. Se un linguaggio L1 si riduce ad un altro linguaggio L2, allora L1 è in un certo senso non più "difficile" di L2; quindi se L1 è facile lo è anche L2, e viceversa.

Un linguaggio L1 è riducibile in tempo polinomiale ad un linguaggio L2 (L1 $\leq_p$ L2) se esiste una funzione calcolabile in tempo polinomiale $f : \{0,1\}^* \rightarrow \{0,1\}^* \mid \forall x \in \{0,1\}^*, x \in L1 \Leftrightarrow f(x) \in L2$. La funzione *f* si chiama funzione di riduzione e un algoritmo *F* di tempo polinomiale che calcola *f* è detto algoritmo di riduzione.

Queste riduzioni sono uno strumento potente per dimostrare che diversi linguaggi appartengono a P. $L1, L2 \subseteq \{0,1\}^* \wedge L1 \leq_p L2 \Rightarrow (L2 \in P \rightarrow L1 \in P)$: se io valuto L1 tramite *F* e L2, so che per valutare sia *F* che L2 occorre tempo polinomiale.

Valgono le seguenti proprietà:

1. In generale, se L1 $\leq_p$ L2 non è detto che L2 $\leq_p$ L1
2. $\underset{f}{L1 \leq_p L2} \quad \underset{g}{L2 \leq_p L3} \quad \Rightarrow \quad \underset{f \circ g}{L1 \leq_p L3}$
3. $L1 \equiv_p L2 \Leftrightarrow L1 \leq_p L2 \wedge L2 \leq_p L1$ (relazione di equivalenza, hanno la stessa difficoltà)

Riduzioni polinomiali classiche

VC $\leq_p$ CLIQUE e VC $\geq_p$ CLIQUE

$f : (G,l) \mapsto (G',k)$ $\qquad$ (*G*,*l*) $\in$ VC $\Leftrightarrow$ (*G'*,*k*) $\in$ CLIQUE

- *Trasformazione*

$V' = V, E' = (V \times V) - E, k = |V| - l$

- *Dimostrazione*

Verso $\Rightarrow$: Supponiamo che *A* sia un VC in *G* e che |*A*|=*l*. Prendiamo ora *B* = *V* – *A*: sappiamo che |*B*|=|*V*|-|*A*|=|*V*|-*l*=*k*. Ora *B* è un CLIQUE in *G'*? Cioè, $\forall a,b \in B, (a,b) \in E'$? Sì, perché $a,b \in B \Rightarrow a,b \notin A \Rightarrow (a,b) \notin E \Rightarrow (a,b) \in E'$.

Verso $\Leftarrow$: Supponiamo che *B* sia un CLIQUE in *G'* e che |*B*|=*k*. Prendiamo ora *A* = *V* – *B*: sappiamo che |*A*|=*l*. Ora *A* è un VC in *G*? Cioè, $\forall (a,b) \in E, (a \in A) \vee (b \in A)$? Sì, perché dati $(a,b) \in E$, supponiamo per assurdo che né *a* né *b* appartengano ad *A*: quindi l'arco $(a,b) \in E'$ quindi *B* non è un CLIQUE (assurdo, contraddice le ipotesi).

In questo caso vale la riduzione inversa. I due problemi sono equivalenti.

3SAT $\leq_p$ VC

- *Trasformazione*

per ogni variabile costruisco due nodi e li unisco con un arco. Per ogni clausola costruiamo una tripla di nodi. Per ogni nodo di ogni clausola mando un arco alla variabile corrispondente (quella di dx se è negata, quella di sx altrimenti).

Prendo *l*=*m*+2*c* (*m* = numero di variabili, *c* = numero di clausole).

$\varphi = (x_2 \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_2} \vee x_2 \vee x_4)$

disegno della trasformazione

Presa una variabile, almeno uno dei due nodi deve essere nel VC; presa una clausola, almeno 2 nodi devono stare nel VC (3 archi): da questo ricaviamo correttamente *l*. Se c'è un VC la formula è soddisfacibile.

- *Dimostrazione*

Verso $\Rightarrow$: φ è soddisfacibile $\Rightarrow \exists \delta : \varphi[\delta] = 1$. Si consideri per ogni variabile x_i il nodo di sx se $\delta(x_i)$=1, altrimenti quello di dx. Per ogni clausola devo prendere i nodi che toccano gli archi non coperti dalle variabili scelte precedentemente (almeno uno dei nodi il cui arco uscente deve essere coperto dai nodi delle variabili).

Verso $\Leftarrow$: Sia *Y* un VC di *l* elementi $\Rightarrow$ ho un nodo per variabile e 2 per clausola $\Rightarrow$ ho un assegnamento δ, se la clausola è soddisfatta $\Rightarrow$ una variabile di una clausola non è stata scelta e se è collegata ad una variabile presa, siamo a posto.

Eeeeeeeeh?

3SAT $\leq_p$ HAMC

Non ho capito nulla su questa riduzione

GSAT $\leq_p$ 3SAT

Da ricopiare

3SAT $\leq_p$ CLIQUE e CLIQUE $\leq_p$ SAT

Da ricopiare

3COL $\leq_p$ 4COL

Da ricopiare

Esempio di un L $\leq_p$ GSAT

Prendiamo per esempio il linguaggio L = { $\varphi \in GSAT \mid \exists \delta$ che soddisfi φ in cui almeno una variabile abbia valore 1 }. L è in *NP* perché occorre un algoritmo non deterministico (simile a quello di GSAT) che verifichi anche se almeno una variabile è vera ($return(EVAL(\ldots) \wedge (x_1 \vee x_2 \ldots \vee x_n))$). L è anche *NPC* perché è riducibile polinomialmente a GSAT utilizzando come funzione $\varphi \wedge (x_1 \vee x_2 \ldots \vee x_n)$.