Algoritmi

Di Gianluca Mondini

1 Algoritmi e strutture dati

1 Algoritmo ricorsivo

Una metodologia per programmare le funzioni ricorsive è la seguente:

1. Individuare i casi base in cui la funzione è definita immediatamente;
2. Effettuare le chiamate ricorsive su un insieme più piccolo di dati, cioè un insieme più vicino ai casi base;
3. Fare in modo che alla fine di ogni sequenza di chiamate ricorsive, si ricada nei casi base;

2 Complessità di un algoritmo

2.1 Definizione

È una funzione (sempre positiva) che associa alla dimensione del problema il costo della sua risoluzione in base alla misura scelta

$T_P(n)$ = Complessità con costo = tempo del programma P al variare di n

2.2 Ordine

$g(n)$ è di ordine $O(f(n))$ se esistono un intero n_0 ed una costante $c>0$ tali che per ogni $n>n_0$ si ha $g(n) \leqslant c\, f(n)$

$O(f(n))$ = insieme delle funzioni di ordine $O(f(n))$

2.3 Regole

2.3.1 Regola dei fattori costanti

Per ogni costante positiva k, $O(f(n)) = O(k\, f(n))$

2.3.2 Regola della somma

Se $f(n)$ è $O(g(n))$, allora $f(n)+g(n)$ è $O(g(n))$

2.3.3 Regola del prodotto

Se $f(n)$ è $O(f1(n))$ e $g(n)$ è $O(g1(n))$, allora $f(n)g(n)$ è $O(f1(n)\, g1(n))$

2.3.4 Altre regole

- Se $f(n)$ è $O(g(n))$ e $g(n)$ è $O(h(n))$, allora $f(n)$ è $O(h(n))$
- Per ogni costante k, k è $O(1)$
- Per $m \leqslant p$, n^m è $O(n^p)$
- Un polinomio di grado m è $O(n^m)$

Esempi

- $2\,n+3\,n+2$ è $O(n)$
- $(n+1)^2$ è $O(n^2)$
- $2\,n+10\,n^2$ è $O(n^2)$

2.4 Teorema

Per ogni k, $n^k \in O(a^n)$, per ogni $a>1$

Una qualsiasi funzione polinomiale ha minore complessità di una qualsiasi funzione esponenziale

3 Complessità di un algoritmo ricorsivo

3.1 Esempio: fattoriale di un numero

La funzione è definita come

```
int fact (int n) {
  if (n == 0) return 1;
  else return n * fact(n-1);
}
```

Consideriamo come base il valore 0:

$$T(0) \in C[[n == 0]] + C[[\text{return } 1]] = O(1) + O(1) = O(1)$$

Per $T(n)$, se $n > 0$, abbiamo un tempo $O(1)$ complessivo per il test, la chiamata ricorsiva e la moltiplicazione più il tempo per l'esecuzione applicata a $n-1$. Quindi

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

A questo punto rimpiazziamo gli $O(1)$ di $T(1)$ e $T(n)$ con simboli generici di costante, diversi fra loro perchè corrispondono a pezzi di programma diversi, e abbiamo la seguente relazione di ricorrenza:

$$T(0) = a$$

$$T(n) = b + T(n-1) \quad \text{per } n > 0$$

Proviamo a calcolare i valori di $T(n)$:

$$T(1) = b + T(0) = b + a$$

$$T(2) = b + T(1) = 2\,b + a$$

In generale, per $i \geqslant 0$, avremo $T(i) = i\,b + a$. Dimostriamo questo risultato con l'induzione naturale

- Base. $i = 0$. Abbiamo $T(0) = 0\,b + a = a$
- Induzione.
  Ipotesi. $T(i) = i\,b + a$
  Tesi. $T(i+1) = (i+1)\,b + a$
  Dim.

$$T(i+1) = b + T(i) \text{ per definizione di T}$$

$$= b + i\,b + a \quad \text{per ipotesi induttiva}$$

$$= (i+1)\,b + a$$

Quindi, poiché $T(n) = n\,b + a$, abbiamo $T(n) \in O(n)$

3.2 Dalla relazione di ricorrenza alla complessità

- $T(0) = a$
- $T(n) = b + T(n-1)$

$$T(n) \in O(n)$$

- $T(1) = a$
- $T(n) = b\,n + T(n-1)$

$$T(n) \in O(n^2)$$

- $T(1) = a$

- $T(n) = b\,n + 2\,T(n/2)$

$$T(n) \in O(n \log(n))$$

- $T(0) = a$
- $T(n) = b + T(n-1)$

$$T(n) \in O(n)$$

- $T(0) = a$
- $T(n) = b + T(n/2)$

$$T(n) \in O(\log(n))$$

- $T(0) = a$
- $T(n) = b + 2\,T(n/2)$

$$T(n) \in O(n)$$

- $T(1) = a$
- $T(n) = b + 2\,T(n-1)$

$$T(n) \in O(2^n)$$

- $T(0) = d$
- $T(n) = b\,n^k + T(n-1)$

$$T(n) \in O(n^{k+1})$$

Esempio: fibonacci() ricorsivo

- $T(0) = T(1) = d$
- $T(n) = b + T(n-1) + T(n-2)$

$$T(n) \in O(2^n)$$

Esempio: fibonacci() non ricorsivo

- $T(0) = d$
- $T(n) = b + T(n-1)$

$$T(n) \in O(n)$$

Esempio: mergeSort()

- $T(0) = T(1) = d$
- $T(n) = b\,n + 2\,T(n/2)$

$$T(n) \in O(n \log(n))$$

Esempio: split()

- $T(0) = T(1) = d$
- $T(n) = b + T(n-2)$

$$T(n) \in O(n)$$

Esempio: merge()

- $T(0) = d$
- $T(n) = b + T(n-1)$

$$T(n) \in O(n)$$

3.3 Limiti inferiori delle funzioni

Definizione 1. *$g(n)$ è di ordine $\Omega(f(n))$ se esistono un intero n_0 ed una costante $c > 0$ tali che per ogni $n \geqslant n_0$: $g(n) \geqslant c\,f(n)$*

Definizione 2. *Un problema è di ordine $\Omega(f(n))$ se non è possibile trovare un algoritmo che lo risolva con complessità minore di $f(n)$ (tutti gli algoritmi che lo risolvono hanno complessità $\Omega(f(n))$)*

Si applica soltanto agli algoritmi

- Basati su confronti
- Che hanno complessità proporzionale al numero di confronti che vengono effettuati durante l'esecuzione dell'algoritmo

3.3.1 Alberi di decisione

Definizione 3. *L'albero di deciione è un albero binario che corrisponde all'algoritmo*

- Ogni foglia rappresenta una soluzione per un particolare assetto dei dati iniziali
- Oni cammino dalla radice ad una foglia rappresenta una esecuzione dell'algoritmo (sequenza di confronti) per giungere alla soluzione relativa alla foglia

Corollario 4. *Ogni algoritmo che risolve un problema che ha $s(n)$ soluzioni ha un albero di decisione con almeno $s(n)$ foglie*

Corollario 5. *Un algoritmo ottimo nel caso peggiore (medio) ha il più corto cammino max(medio) dalla radice alle foglie*

Corollario 6. *Un albero binario con k livelli ha al massimo 2^k foglie (ce l'ha quando è bilanciato)*

Corollario 7. *Gli alberi binari bilanciati minimizzano sia il caso peggiore che quello medio: hanno $\log(s(n))$ livelli.*

4 Algoritmi di ordinamento

- Merge sort
- Quick sort
- Insertion sort

5 Alberi binari

- NULL è un albero binario
- Un nodo p più due alberi binari Bs e Bd forma un albero binario

5.1 Numero di foglie e di nodi

Un albero binario bilanciato con livello k ha

- $2^{k+1} - 1$ nodi
- 2^k foglie

5.2 Alcuni algoritmi

5.2.1 Contare il numero dei nodi

```
int nodes(Node* tree) {
  if (!tree) return 0;
  return 1 + nodes(tree -> left) + nodes(tree -> right);
}
```

5.2.2 Contare il numero delle foglie

```
int leaves(Node* tree) {
  if (!tree) return 0;
  if (!tree -> left && !tree -> right) return 1;
  return leaves(tree -> left) + leaves(tree -> right);
}
```

5.2.3 Cercare un etichetta e resistuire un puntatore

Se il nodo non compare nell'albero, viene restituito NULL. Se l'albero contiene più di un'etichetta, viene restituito un puntatore al primo

```
Node* findNode(Infotype n, Node* tree) {
  // L'albero è vuoto, l'etichetta non può esserci
  if (!tree) return NULL;
  // Trovata l'etichetta, restituisco il puntatore
  if (tree -> label == n) return tree;
  // Cerco a sinistra
  Node* a = findNode(n, tree -> left);
  // Controllo se il puntatore della ricerca "a sinistra"
  // a resistuito qualcosa di interessante, altrimenti cerco a destra
  if (a) return a;
  else return findNode(n, tree -> right);
}
```

5.2.4 Eliminare tutto l'abero

Alla fine il puntatore deve essere NULL

```
void delTree(Node* &tree) {
  if (tree) {
    delTree(tree -> left);
    delTree(tree -> right);
    delete tree;
    tree = NULL;
  }
}
```

5.2.5 Inserire un nodo

Inserisce un nodo (son) come figlio di father, sinistro se $c = 'l'$ oppure destro se $c = 'r'$.

Ritorna 1 se l'operazione ha successo, 0 altrimenti. Se l'albero è vuoto, inserisce il nodo come radice

```
int insertNode(Node* &tree, InfoType son, InfoType father, char c) {
  // Caso in cui l'albero sia vuoto
  if (!tree) {
    tree = new Node;
    tree -> label = son;
    tree -> left = tree -> right = NULL;
    return 1;
  }
  // Caso normale
  // Effettuo la ricerca di father con la funzione
  // di ricerca nodo (vedi sopra)
  Node * a = findNode(father, tree);
  // Se il nodo non è stato trovato, restituisco 0 e mi fermo
  if (!a) return 0;
```

```
  // Inserimento come figlio sinistro
  if (c == 'l' && !a -> left) {
    a -> left = new Node;
    a -> left -> label = son;
    a -> left -> left = a -> left -> right = NULL;
    return 1;
  }
  if (c == 'r' && !a -> right) {
    a -> right = new Node;
    a -> right -> label = son;
    a -> right -> left = a -> right -> right = NULL;
    return 1;
  }
}
```

5.2.6 Compito 2 - Esercizio 4

Scrivere una funzione che, dato un albero binario ad etichette di tipo string, con puntatore alla radice t, restituisca come risultato il numero di nodi che hanno numero dispari di discendenti. Si calcoli la complessità della soluzione proposta in funzione del numero di nodi dell'albero.

```
int dispari(Node* t, int & nodi) {
  if (!t) {
    nodi = 0;
    return 0;
  }
  int cs, cd, nodis, nodid;
  cs = dispari(t -> left, nodis);
  cd = dispari(t -> left, nodid);
  nodi = nodis + nodid + 1;
  return cs + cd + ((nodis + nodid) % 2);
}
```

La funzione è $O(n)$

5.2.7 Compito 1 - Esercizio 4

Scrivere una funzione ricorsiva che, dato un albero binario a etichette intere, conta il numero di nodi che hanno più foglie maggiori o uguali a zero che minori di zero tra i propri discendenti.

```
int conta(const Node * t, int & pos, int & neg) {
  if (!t) {
    pos = 0;
    neg = 0;
    return 0;
  }
  if (!t -> left && !t -> right) {
    pos = (t -> info >= 0) ? 1 : 0;
    neg = (t -> info < 0)  ? 1 : 0;
    return 0;
  }
  int pos_left, pos_right;
  int neg_left, neg_right;
  int conta_left = conta(t -> left, pos_left, neg_left);
  int conta_right = conta(t -> right, pos_right, neg_right);
  pos = pos_left + pos_right;
  neg = neg_left + neg_right;
  return (pos > neg) ? 1 : 0 + conta_left + conta_right;
}
```

6 Alberi generici

- Un nodo p è un albero
- Un nodo più una sequenza di alberi $A_1...A_n$ è un albero

6.1 Alcuni algoritmi

6.1.1 Contare il numero di nodi

Vedi l'algoritmo per gli alberi binari

6.1.2 Contare il numero di foglie

```
int leaves(Node* tree) {
  if (!tree) return 0;
  // Caso della foglia
  if (!tree -> left) return 1 + leaves(tree -> right);
  // "Non caso" della foglia
  return leaves(tree -> left) + leaves(tree -> right);
```

6.1.3 Inserire un nodo in fondo ad una lista di fratelli

```
void addSon(InfoType x, Node* &tree) {
  // Caso in cui la lista sia vuota
  if (!tree) {
    tree = new Node;
    tree -> label = x;
    tree -> left = tree -> right = NULL;
  }
  else {
    addSon(x, tree -> right);
  }
}
```

6.1.4 Inserire un nodo son come ultimo figlio di father

Se l'albero è vuoto, lo inserisce come radice

```
int insert(InfoType son, InfoType father, Node* &tree) {
  if (!tree) {
    tree = new Node;
    tree -> label = son;
    tree -> left = tree -> right = NULL;
    return 1;
  }
  Node* a = findNode(father, tree);
  if (!a) return 0;
  addSon(son, a -> left);
  return 1;
}
```

6.1.5 Compito 1 - Esercizio 5

Scrivere una funzione che, dato un albero generico a etichette intere e memorizzazione figlio-fratello, conta il numero di nodi che hanno più figli maggiori o uguali a zero che minori di zero.

```
int conta(const Node* t) {
  if (!t) return 0;
  const Node * n;
  int pos = 0, neg = 0;
```

```
  for (n = t -> left; n != NULL; n = n -> right) {
    if (n -> info >= 0) pos++;
    else neg++;
  return (pos > neg) ? 1 : 0 + conta(t -> left) + conta(t -> right);
}
```

7 Alberi binari di ricerca

Un albero binario di ricerca è un albero binario tale che per ogni nodo p

- I nodi del sottoalbero sinistro di p hanno etichetta minore dell'etichetta di p
- I nodi del sottoalbero destro di p hanno etichetta maggiore dell'etichetta p

7.1 Proprietà

- Non ci sono doppioni
- La visita simmetrica elenca le etichette in ordine crescente

7.2 Alcuni algoritmi

7.2.1 Cercare un nodo

```
Node* findNode(InfoType n, Node* tree) {
   if (!tree) return 0;
   if (n == tree -> label) return tree;
   if (n < tree -> label) {
     return findNode(n, tree -> left);
   }
   return findNode(n, tree -> right);
}
```

7.2.2 Inserire un nodo

```
void insertNode(InfoType n, Node* &tree) {
  // Albero vuoto
  if (!tree) {
    tree = new Node;
    tree -> label = n;
    tree -> left = tree -> right = NULL;
    return;
  }
  // Caso n < radice
  if (n < tree -> label) {
    insertNode(n, tree -> left);
  }
  if (n > tree -> label) {
    insertNode(n, tree -> right);
  }
}
```

L'algoritmo ha complessità $O(\log(n))$

7.2.3 Restituire l'etichetta del nodo più piccolo di un albero ed eliminare il nodo che la contiene

```
void deleteMin(Node *& tree, InfoType &m) {
  if (tree -> left) // C'è un nodo più piccolo
    deleteMin(tree -> left, m);
  else {
```

```
    m = tree -> label; // restituisco l'etichetta
    Node * a = tree;
    // connetto il sottoalbero destro di
    // m al padre di m
    tree = tree -> right;
    // elimino il nodo
    delete a;
  }
}
```

7.2.4 Cancellare un nodo ?

```
void deleteNode(InfoType n, Node *& tree) {
  if (tree) {
    // n è minore della radice
    if (n < tree -> label) {
      deleteNode(n, tree -> left);
      return;
    }
    // n è maggiore della radice
    if (n > tree -> label) {
      deleteNode(n, tree -> right);
      return;
    }
    // n non ha figlio sinistro
    if (!tree -> left) {
      Node * a = tree;
      tree = tree -> right;
      delete a;
      return;
    }
    // n non ha figlio destro
    if (!tree -> right) {
      Node * a = tree;
      tree = tree -> left;
      delete a;
      return;
    }
    // n ha entrambi i figli
    deleteMin(tree -> right, tree -> label);
}
```

Questo algoritmo ha complessità $O(\log(n))$

8 Heap

Definizione 8. *Un heap è un albero binario quasi bilanciato con le seguenti proprietà:*

- *i nodi dell'ultimo livello sono addossati a sinistra*
- *in ogni sottoalbero l'etichetta della radice è maggiore o uguale a quella di tutti i discendenti*

L'heap viene memorizzato in un array

8.1 Calcolare le parentele

- Figlio sinistro di $i = 2\,i + 1$
- Figlio destro di $i = 2\,i + 2$

- Padre di $i = \frac{i-1}{2}$

8.2 Classe Heap

```
class Heap {
  private:
    int * h;
    int last;
    void up(int);
    void down(int);
    void exchange(int i, int j);
  public:
    Heap(int);
    ~Heap();
    void insert(int);
    int extract();
};
```

8.2.1 Costruttore

```
Heap::Heap(int n) {
  h = new int[n];
  last = -1;
}
```

8.2.2 Distruttore

```
Heap::~Heap() {
  delete h [n];
}
```

8.2.3 Inserimento

- Memorizza l'elemento nella prima posizione libera dell'array
- Fa risalire l'elemento tramite scambi figlio-padre per mantenere la proprietà dello heap

```
void Heap::insert(int x) {
  h[++last] = x;
  up(last);
}

// i è l'indice dell'elemento da far risalire
void Heap::up(int i) {
  // Se non sono sulla radice
  if (i > 0) {
    // Se l'elemento è maggiore del padre
    if (h[i] > h[(i-1)/2]) {
      // Scambia il figlio con il padre
      exchange(i, (i-1)/2);
      // chiama up sulla nuova posizione
      up((i-1)/2);
    }
  }
}
```

La funzione termina in due casi:

- viene chiamata con l'indice 0 (radice)

- L'elemento è inferiore al padre

La complessità è $O(\log(n))$ perché ogni chiamata risale di un livello

8.2.4 Estrazione

- Restituisce il primo elemento dell'array
- Mette l'ultimo elemento al posto della radice e decrementa last
- Fa scendere l'elemento tramite scambi padre-figlio per mantenere la proprietà dello heap

```
int Heap::extract() {
  int r = h[0];
  h[0] = h[last--];
  down(0);
  return r;
}

// i è l'indice dell'elemento da far scendere
void Heap::down(int i) {
  // son = indice del figlio sinistro (se esiste)
  int son = 2*i+1;
  // se i ha un solo figlio (è l'ultimo dell'array)
  if (son == last) {
    // se il figlio è maggiore del padre
    if (h[son] > h[i]) {
      // fai lo scambio, altrimenti termina
      exchange(i, last);
    }
  }
  // se i ha entrambi i figli
  else if (son < last) {
    // son = indice del figlio maggiore tra i due
    if (h[son] < h[son+1]) son++;
    // se il figlio è maggiore del padre
    if (h[son] > h[i]) {
      // fai lo scambio
      exchange(i, son);
      // chiama down sulla nuova posizione
      down(son);
      // altrimenti termina (termina anche se i non ha figli)
    }
  }
}
```

L'algoritmo ha complessità $O(\log(n))$

9 Ricerca hash

- È un metodo di ricerca in array
- Non è basato su confronti
- Molto efficiente

9.1 Alcuni algoritmi

9.1.1 Ricerca tramite hash

```
bool hashSearch (int* A, int k, int x) {
```

```
  int i = h(x);
  if (A[i] == 1) return true;
  else return false;
}
```

9.2 Metodo hash ad accesso non diretto

È possibile rilasciare l'iniettività e permettere che due elementi diversi abbiano lo stesso indirizzo hash. Si ha una collisione quando

$$h(x_1) = h(x_2)$$

Bisogno gestire le seguenti situazioni:

- Come cercare un elemento se il suo posto è occupato da un altro
- Come inserire gli elementi

9.2.1 Soluzione: hash modulare

Si scrive una funzione $h()$

$$h(x) = (x \ \% \ k)$$

In modo tale da essere sicuri di generare tutti e soli gli indici dell'array

Corollario 9. ***Legge di scansione lineare*** *Se non si trova l'elemento al suo posto, lo si cerca nelle posizioni successive fino a trovarlo o ad incontrare una posizione vuota.*

L'inserimento è fatto con lo stesso criterio

Definizione 10. *Un agglomerato è un gruppo di elementi con indirizzi hash diversi*

La presenza di collisioni ed agglomerati aumenta il tempo di ricerca

Corollario 11. *La diversa lunghezza del passo di scansione riduce gli agglomerati, ma è necessario controllare che la scansione visiti tutte le possibili celle vuote dell'array, per evitare che l'inserimento fallisca anche in presenza di array non pieno.*

9.2.2 Tempo medio di ricerca

Il tempo medio di ricerca (numero medio di confronti) dipende dal rapporto $\alpha = n \,/\, k$ (sempre minore di 1)

Funzione di ricerca con scansione lineare

```
bool hashSearch(int *A, int k, int x) {
  int i = h(x);
  for (int j=0; j<k; j++) {
    int pos = (i+j) % k;
    if (A[pos] == -1) return false;
    if (A[pos] == x) return true;
  }
  return false;
}
```

Funzione di inserimento in presenza di cancellazioni

```
int hashInsert(int *A, int k, int x) {
  int i = h(x);
  for (int j=0; j < k; j++) {
    int pos = (i+j) % k;
    // -1: posizione vuota
```

```
      // -2: posizione disponibile
      if ((A[pos] == -1) || (A[pos] == -2)) {
        A[pos] = x;
        return 1;
      }
    }
    return 0;
  }
```

10 Programmazione dinamica

Si può usare quando non è possibile applicare il metodo del divide et impera (non si sa con esattezza quali sottoproblemi risolvere e non è possibile partizionare l'insieme in sottoinsiemi disgiunti)

Metodo: si risolvono **tutti** i sottoproblemi a partire dal basso e si conservano i risultati ottenuti per poterli usare successivamente (strategia **bottom-up**)

La complessità del metodo dipende dal numero di sottoproblemi

Si può applicare:

- **Sottostruttura ottima:** una soluzione ottima del probleme contiene la soluzione ottima dei sottoproblemi
- **Sottoproblemi comuni:** un algoritmo ricorsivo richiederebbe di risolvere lo stesso problema più volte

11 Algoritmi greedy (golosi)

Definizione 12. *Negli algoritmi greedy la soluzione ottima si ottiene mediante una sequenza di scelte*

In ogni punto dell'algoritmo, viene scelta la strada che in quel momento sembra la migliore.

La **scelta locale** deve essere in accordo con la **scelta globale**: scegliendo ad ogni passo l'alternativa che sembra la migliore non si perdono alternative che potrebbero rivelarsi migliori nel seguito.

11.1 Metodo top-down

Non sempre si trova la soluzione ottima ma in certi casi si può trovare uan soluzione approssimata (esempio del problema dello zaino)

11.2 Algoritmo di Huffman

Gli alberi sono memorizzati in uno heap (con ordinamento inverso: la radice è il più piccolo)

Si fa un ciclo dove in ogni iterazione:

1. Vengono estratti i due alberi con radice minore
2. Vengono fusi in un nuovo albero avente come etichetta della radice la somma delle due radici
3. L'albero risultante è inserito nello heap

Il ciclo ha n iterazioni ed ogni iterazione ha complessità $O(\log(n))$ (si eseguono 2 estrazioni ed un inserimento). In totale abbiamo una complessità $O(n\log(n))$

12 Grafi

$$\text{Grafo orientato}=(N, A)$$

dove

- $N =$ insieme di nodi
- $A \subseteq N \times N =$ insieme di archi (coppie ordinate di nodi)

Un grafo orientato con n nodi ha al massimo n^2 archi

Corollario 13. *Un grafo non orientato è connesso se esiste un cammino fra due nodi qualsiasi del grafo*

Definizione 14. *La **componente connessa** è il **sottografo connesso***

Definizione 15. *Nella componente connessa massimale nessun nodo è connesso ad un altra componente connessa*

Definizione 16. *L'**albero di copertura** è l'insieme delle componenti connesse massimali acicliche*

Definizione 17. *Nel **minimo albero di copertura** la somma dei pesi degli archi è minima*

12.1 Rappresentazioni in memoria di grafi

12.1.1 Tramite liste di adiacenza

```
struct Node {
  int NodeNumber;
  Node * next;
};

Node* graph[N];
```

12.1.2 Tramite matrici di adiacenza

```
int graph[N][N];
```

12.2 Rappresentazioni in memoria di grafi con nodi ed archi etichettati

12.2.1 Tramite liste di adiacenza

```
struct Node {
  int NodeNumber;
  ArcType archLabel;
  Node * next;
};

Node * graph[N];

NodeType nodeLabels[N];
```

NodeType = char
ArcType = int

12.2.2 Tramite matrici di adiacenza

```
ArcType graph[N][N];
NodeType nodeLabels[N];
```

12.3 Cammino più breve

12.3.1 Algoritmo di Dijkstra

- Si applica ai grafi orientati
- Trova i cammini minimi **da un nodo a tutti gli altri**

- È basato sulla metodologia **greedy**

Vengono utilizzate due tabelle:

- **dist** (distanza)
- **pred** (predecessore)

con n elementi.

Vengono eseguiti n passi, e ad ogni passo:

1. Si sceglie il nodo con distanza minore in **dist**
2. Si aggiornano **pred** e **dist** per i suoi immediati successori

13 Esercizi

13.1 Più lunga sottosequenza comune (PLSC)

Metodo pratico

1. Si inseriscono gli 0
$$L(0,0) = L(i,0) = L(0,j) = 0$$
2. Riempio la tabella partendo dagli indici più piccoli considerando i due casi:
   - Se $a_i = \beta_j$
   $$L(i,j) = L(i-1, j-1) + 1$$
   - Se $a_i \neq \beta_j$
   $$L(i,j) = \max\left(L(i, j-1), L(i-1, j)\right)$$

Equazione di ricorrenza: $T(n) = b + 2\,T(n-1)$

		c	b	a	b	a	c
	0	0	0	0	0	0	0
a	0	0	0	1	1	1	1
b	0	0	1	1	2	2	2
c	0	1	1	1	2	2	3
a	0	1	1	2	2	3	3
b	0	1	2	2	3	3	3
b	0	1	2	2	3	3	3
a	0	1	2	3	3	4	4

Risalire la matrice

1. Andare nell'angolo in basso a destra e segnare la lettera della colonna
2. Muoversi verso sinistra o verso l'altro fino a quando non ci sarà lungo la diagonale il numero attuale decrementato di 1 (esempio: da 3 a 2)
3. Annotare la lettera della colonna (come nel passo 1)
4. Ripetere i passi 2 e 3 fino a quando non si arriva ad avere il numero 0 (sulla colonna di sinistra)
5. Per ottenere la stringa bisogna invertire il percorso effettuato (ovvero leggere il percorso dalla fine all'inizio); il risultato sarà la stringa desiderata

Nota: se ci si può muovere in due direzioni possibili, occorre farlo e controllare se si può arrivare alla prima colonna da destra seguendo i passaggi di sopra

14 NP-Completezza

14.1 Alcuni problemi

Commesso viaggiatore. Date n città, è possibile partire da una città, attraversare ogni città esattamente una volta e tornare alla città di partenza, percorrendo una distanza complessiva non superiore ad un intero k

N-Regine. Data una scacchiera con $n \times n$ caselle, è possibile posizionare su di essa n regine in modo che nessuna possa "mangiare" un altra?

La complessità è esponenziale in n

Soddisfattibilità di una formula logica. Data una formula F con n variabili, esiste una combinazione di valori booleani che, assegnati alle variabili di F, la rendono vera?

La complessità è esponenziale: $O(2^n)$

14.2 Algoritmi non deterministici

Si aggiunge il comando choice(I), dove I è un insieme.

choice(I) sceglie non deterministicamente un elemento dell'insieme I

Algoritmo non deterministico di ricerca in array

```
int nSearch(int* a, int n, int x) {
  int i = choiche({0..n-1});
  if (a[i] == x) {
    return 1;
  }
  else {
    return 0;
  }
}
```

Complessità: $O(1)$

14.2.1 Relazione tra determinismo e nondeterminismo

Per ogni algoritmo nondeterministico ne esiste uno deterministico che lo simula, esplorando lo spazio delle soluzioni, fino a trovare un successo.

Se le soluzioni sono in numero esponenziale l'algoritmo deterministico avrà complessità esponenziale.

Definizione 18. *P = Insieme di tutti i problemi decisionali risolubili in tempo polinomiale con un algoritmo deterministico*

Definizione 19. *NP = Insieme di tutti i problemi decisionali risolubili in tempo polinomiale con un algoritmo nondeterministico (NP: Nondeterministico Polinomiale)*

Definizione 20. *Un problema P1 si riduce ad un altro problema P2 se ogni soluzione di P1 può ottenersi deterministicamente in tempo polinomiale da una soluzione di P2*

14.3 Teorema di Cook

Teorema 21. ***Teorema di Cook*** *per qualsiasi problema R in NP vale che R è riducibile al problema della soddisfattibilità della formula logica*

$$R \leqslant P_S$$

Se si trovasse un algoritmo polinomiale per P_S allora tutti i problemi in NP sarebbero risolubili in tempo polnomiale e quindi P sarebbe uguale ad NP

14.4 NP-completezza

Un problema R è NP-completo se

- R appartiene ad NP

e

- $P_s \leqslant R$

Se si trovasse un algoritmo polinomiale per un problema NP-completo, allora tutti i problemi in NP sarebbero risolubili in tempo polinomiale e quindi P sarebbe uguale ad NP

14.5 Problemi NP-Completi

È stato dimostrato che i seguenti problemi e altri sono NP-Completi

- Commesso viaggiatore
- Colorazione di mappa
- Zaino
- n-regine

Quindo uno qualsiasi di questi problemi può essere usato al posto di P_S nella dimostrazione di NP-completezza

14.5.1 Dimostrare che un problema *R* è *NP*-Completo

Per dimostrare che un problema R è NP-Completo:

- **R appartiene ad NP** individuare un algoritmo polinomiale nondeterministico per risolvere P
- **esiste un problema NP-Completo che si riduce a R** se ne sceglie uno fra i problemi NP-Completi noti che sia facilmente riducibile a R

Perché ci serve dimostrare che un problema è NP-Completo?

Perchè non riusciamo a risolverlo con algoritmo polinomiale e vogliamo dimostrache che on ci si riesce a meno che P non sia uguale a NP, problema tuttora non risolto

14.5.2 Caratterizzazione alternativa dei problemi NP-Completi

Problemi NP-Completi: problemi con certificato[1] verificabile in tempo polinomiale

Esempio: per il problema della soddisfattibilità della formula logica si può controllare se un assegnamento di valori booleani alle variabili è una soluzione

15 Templates

15.1 Esempi di definizione

```
template <class T1>
T1 somma(T1 a, T2 b) {
  return a + b;
}
```

15.2 Variabili statiche

Ogni "tipo di funzione" possiede la propria variabile statica indipendente

```
template<class Data>
void typeCounter(Data a) {
        static int c;
        c++;
        cout << "Il counter vale " << c << endl;
```

1. certificato: soluzione del problema

```
}
```

16 Derivazione

16.1 Specificatori d'accesso

- I campi **privati** di una classe non sono accessibili dalle sottoclassi nè dall'esterno
- I campi **protetti** di una classe sono accessibili dalle sottoclassi, ma non dall'esterno
- I campi **pubblici** di una classe sono accessibili anche dall'esterno
- I campi privati, protetti e pubblici rimangono tali in tutta la gerarchia

16.2 Costruzione degli oggetti

Quando un oggetto viene costruito si costruisce prima la parte **BASE** ed in seguito quella **DERIVATA**.

Viene quindi prima chiamato il costruttore della classe base e poi quello della classe derivata.

Se la classe base ha dei costruttori, il costruttore di una classe derivata deve chiamarne uno nella lsita di inizializzazione. Può non chiamarlo esplicitamente se la classe base ha un costruttore di default, che in questo caso viene chiamato automaticamente.

16.3 Funzioni virtuali