.

Appunti Corso Olimpiadi
Anno Scolastico 2007/2008

Prof. M. Cristina Pinotti
Dipatimento di Matematica e Informatica
Perugia

Avvertenza: queste note sono nate per tenere traccia degli argomenti trattati nel corso organizzato nell'anno scolastico 2007/2008 presso l'ITCS "Vittorio Emanuele" di Perugia nell'ambito delle attivita' di formazione per le olimpiadi di informatica in Umbria. Sono solo appunti che richiedono ancora molto lavoro prima di poterli considerare liberi da errori (di stampa e non). Queste note non sono pertanto pronte per essere distribuite ma ho accettato di renderle disponibili - ancora grezze - agli studenti che hanno seguito il corso per facilitare il loro lavoro di approfondimento.

1 Array

Questa sezione considera la struttura *array/vettore di interi*, implementato in $C++$ come un puntatore ad un vettore di *dim* interi.

La procedura `createArray(int)` legge da terminale la dimensione *dim* dell'array e ritorna il puntatore ad un array di *dim* interi.

```
int dim;
int* createArray(int _dim) {
dim = _dim;
int* A = new int[dim];
return A;
}
```

La procedura `insertArray (int* A)` legge da terminale gli interi da memorizzare nell'array.

```
void insertArray(int* A) {
        for (int i = 0; i < dim; i++) {
                cout << "A[" << i << "] = ";
                cin >> A[i];
        }
        cout << endl;
        }
```

La procedura `insertArray (int* A)` legge da video gli interi da memorizzare nell'array.

```
void printArray(int* A) {
        for (int i = 0; i < dim; i++) {
                cout << " | " << A[i];
        }
        cout << " |" << endl;
}
```

La procedura `insertionsort` ordina il vettore a considerando gli elementi in a con indice via via crescente e trovando per ciascun elemento la sua posizione nell'array ordinato degli elementi che lo precedono. Precisamente all'i-esima iterazione, con $1 \leq i < dim$, l'array $A[1], \ldots, A[i-1]$ é ordinato e l'elemento $A[i]$ é inserito in esso cosí da ottenere l'array $A[1], \ldots, A[i]$ ordinato.

```
void insertionsort(int* A) {
        for (int i = 1; i < dim; i++) {
                int j=i;
                int temp=A[j];
                while ((temp < A[j-1]) && (j>0)) do {
                        A[j]=A[j-1];
                        j--;
                        }
                A[j]=temp;
                                }
                }
```

La procedura **ricorsiva** `mergesort (int* a, int sx, int rx)` ordina gli interi memorizzati in $A[\text{sx}], \ldots, A[\text{rx}]$. Questo algoritmo di ordinamento ha complessità ottima in tempo, ossia richiede un numero di operazioni proporzionale a $n \log n$, se n sono gli elementi da ordinare. Ad esempio per ordinare $n = 2048$ elementi, mergesort richiede un numero di operazioni dell'ordine delle decine di migliaia, mentre l'algoritmo insertion sort, richiede un numero di operazioni dell'ordine di milione di operazioni.

```
void mergesort(int* a, int sx, int rx) {
        if (sx < rx) {
```

```
                int center = (sx+rx)/2;
                mergesort(a, sx, center);
                mergesort(a, center+1, rx);
                merge(a, sx, center, rx);
        }
}


void merge(int* a, int sx, int center, int rx) {
        int i, j;

        static int *aux = createArray(dim);
        for (i = center+1; i > sx; i--)
                aux[i-1] = a[i-1];
// i rimane inizializzato a sx
        for (j = center; j < rx; j++)
                        aux[rx+center-j] = a[j+1];
// a e' copiato in  aux in ordine decrescente
// aux[rx]=a[center+1] ... aux[center]=a[rx]
// j rimane inizializzato con rx
// i rimane inizializzaro a sx
        for (int k = sx; k <= rx; k++) {
                if (aux[j] < aux[i])
                        a[k] = aux[j--];
                else
                        a[k] = aux[i++];
        }
}
```

La procedura `merge` copia l'array a in un array ausiliario di appoggio aux. Precisamente, in $aux[sx], \ldots, aux[center]$ sono copiati gli elementi $A[\text{sx}], \ldots, A[\text{center}]$ procedendo da sx a center cosí che $aux[\text{sx}], \ldots, aux[\text{center}]$ sia ordinato in ordine crescente e l'indice i punti dopo l'operazione di copia a sx. Successivamente, in $aux[\text{rx}], \ldots, aux[\text{center}+1]$ sono copiati gli elementi $a[\text{center}+1], \ldots, a[\text{rx}]$, cosí che $aux[\text{rx}], \ldots, aux[\text{center}+1]$ sia ordinato in ordine crescente e l'indice j punti dopo l'operazione di copia a rx. L'ultimo `for` costruisce in a il vettore ordinato.

3-max Progettare un algoritmo *divide-et-impera* per il calcolo del massimo di un vettore A non ordinato contenente n interi positivi. Tale algoritmo deve dividere A in tre parti e organizzare ricorsivamente la ricerca. In particolare:

Al fine di costruire un algoritmo basato sulla tecnica ricorsiva di programmazione divide-et-impera, dobbiamo individuare un *caso base* per il quale il problema ammetta soluzione senza ricorsione. Se il vettore A é vuoto o contiene un solo elemento, allora $\text{fmax}(A)$ é calcolato direttamente.

$$\text{fmax}(A, i, f) = \begin{cases} -1 & \text{if } |A| = f - i + 1 \leq 0; \\ A[i] & \text{if } |A| = f - i + 1 = 1; \end{cases}$$

Se $|A| > 1$, il problema é scomposto in 3 sottoproblemi di dimensioni bilanciate n_1, n_2, e n_3 tali che $n_1 + n_2 + n_3 = n$ e $n_1 \approx n_2 \approx n_3$. Ossia, con $n_1 = n_2 = \lfloor \frac{f-i}{3} \rfloor$, e $n_3 = n - n - 1 - n - 2$, si ha

$$\text{fmax}(A, i, f) = \max(\text{fmax}(A, i, i + n_1), \text{fmax}(A, i + n_1 + 1, i + n_1 + n_2), \text{fmax}(A, i + n_1 + n_2 + 1, f))$$

```
#include <array>
```

```
int fmax(array*, int, int);
int max3(int, int, int);


int main() {

        array *P = new array();
        int n;
        cout << "Inserisci la dimensione dell'array: ";
        cin >> n;

        createArray(P, n);
        insertArray(P);

        int max = fmax(P, 0, (n-1));
        cout << "Il massimo e': " << max << endl;
}

int max3( int a, int b, int c){
        if ((a >= b) && ( a >= c)) {return (a);}
           else if ((b>=c) && (b>=a)) {return(b);}
                 else {return(c);}
                                 }
int fmax(array* P, int i, int f) {
        if (i >= f) {    if (i==f) {
                         cout << P->V[i] << endl;
                         return (P->V[i]);}
                              else {return(-1);}
                 }
         else { int q = ((f-i)/3);
                 cout << q << endl;
                return(max3(fmax(P, i+q, q), fmax(P, i+q+1, i+2*q),
                                          fmax(P, i+2*q+1, f)));}
                }
```

E' interessante notare che l'algoritmo calcola il massimo procedendo con una computazione ad *albero*, tipica della tecnica di programmazione ricorsiva *divide-et-impera*. Dapprima il problema e' scomposto in sottoproblemi via via piu' piccoli fino ad arrivare alle foglie dell'albero che rappresentano array con al piu' un elemento. Questa e' la fase *divide* dell'algoritmo. Alle foglie il problema e' risolto direttamente. Nella fase *impera*, si risale verso la radice componendo i risultati parziali. Ogni nodo interno dell'albero calcola il massimo degli elementi memorizzati nelle sue foglie a partire dai tre massimi locali calcolati dai suoi tre figli.

Occorrenze E' dato un array A **ordinato** di n interi, compresi fra 1 e 5. Si consideri il problema di contare il numero di occorrenze del numero 2 in A.

```
#include <array>

int esercizio1(array*, int, int);
bool esercizio2(graph*);

int main() {

        int n;
```

```
        cout << "Inserisci la dimensione dell'array: ";
        cin >> n;

        array *A = new array();
        createArray(A, n);
        insertArray(A);
        mergesort(A, 0, n-1);

        int occ = esercizio1(A, 0, n-1);
        cout << "Occorrenze del numero 2: " << occ << endl;
        cout << "------------------------------------------------------" << endl;
        }


int esercizio1(array* A, int i, int f) {

        if (i == f) {
                    if (A->V[i] == 2)
                       {return (1);}
                 else {return (0);}}
                    else {
 return (esercizio1(A, i, ((i+f)/2))+ esercizio1(A, ((i+f)/2 +1), f)); }
                       }
```

L'algoritmo proposto non tiene conto che l'array di partenza e' ordinato e controlla ciascun elemento dell'array. Pertanto, l'algoritmo richiede un numero di operazioni proporzionale alla dimensione dell'array stesso.
E' possibile pensare ad un algoritmo piu' efficiente che sfrutti l'ordinamento iniziale dell'array. La soluzione proposta cerca la prima, detta `first-occ` e l'ultima occorrenza, detta `last-occ`, dell'elemento 2 nell'array. Poiche' l'array e' ordinato, tutti gli elementi $A[\text{first} - \text{occ}], \ldots, A[\text{last} - \text{occ}]$ hanno valore 2.

```
int esercizio1first(array*, int, int, int);
int esercizio1last(array*, int, int, int);

int main() {

        int first_occ = esercizio1first(A, 0, n-1, 2);
        cout << first_occ << endl;
        int last_occ = esercizio1last(A, 0, n-1, 2);
        cout << last_occ << endl;
        if (first_occ !=-1)
        {cout << "Occorrenze del numero 2: " <<
                last_occ - first_occ +1 << endl;}
        else {cout << "non trovato" << endl;}
        cout << "------------------------------------------------------" << endl;

        }

int esercizio1first(array* A, int i, int f, int elem) {
        if (i == f) {
                if (A->V[i] == elem)
                {return (i);}
                else {return (-1);};
```

```
                    }
        else {int q = (i+f)/2;
             if (A->V[q] >= elem) {return(esercizio1first(A, i, q, elem));}
                        else {return (esercizio1first(A, q+1,f, elem));}}
}

int esercizio1last(array* A, int i, int f, int elem) {
        if (i == f) {
                if (A->V[i] == elem)
                {return (i);}
                else {return (-1);};
                     }
        else {int q = (i+f+1)/2;
             if (A->V[q] > elem) {return(esercizio1last(A, i, q-1, elem));}
                        else {return (esercizio1last(A, q,f, elem));}}
}
```

Alcune considerazioni riguardo la terminazione dell'algoritmo.

La funzione `esercizio1first`, dopo aver valutato il valore $A[q]$, ricorre su uno dei due intervalli (A, i, q) o $(A, q+1, f)$. Ciascuna partizione dell'array descresce la propria ampiezza, e quindi la terminazione e' garantita, se si sceglie $q = (i+f)/2 = \lfloor \frac{i+f}{2} \rfloor$. Infatti, quando $i = f-1$, q coincide con l'estremo inferiore dell'intervallo, ossia $q = i$, e pertanto entrambi gli intervalli su cui si puo' ricorrere hanno ampiezza 1. Similmente, la procedure `esercizio1last`, dopo aver valutato il valore $A[q]$, ricorre su uno dei due intervalli $(A, i, q-1)$ o (A, q, f). Ciascuna partizione dell'array descresce la propria ampiezza, se si sceglie $q = (i+f+1)/2 = \lceil \frac{i+f}{2} \rceil$, e pertanto l'algoritmo termina. Infatti, quando $i = f-1$, q coincide con l'estremo superiore dell'intervallo, ossia $q = f$, ed entrambi gli intervalli su cui si puo' ricorrere hanno ampiezza 1.

Remark. Quando il caso base della ricorsione lavora su array di cardinalita' minore di o uguale ad 1, si noti che se l'elemento $A[q]$ analizzato puo' essere escluso dai successivi livelli di ricorsione, ossia se si ricorre su $A, i, q-1$ e $A, q+1, f$, siamo garantiti che la dimensione dei sottoproblemi decresce sempre e qualunque scelta di q porta alla terminazione dell'algoritmo. Se invece nelle chiamate ricorsive non si puo' escludere nessun elemento dell'array, conviene seguire questa semplice regola che garantire di raggiungere il caso base. Se richiami fra A, i, q e $A, q+1, f$, allora $q = \lfloor (i+f)/2 \rfloor$, mentre se richiami fra $A, i, q-1$ e A, q, f, allora $q = \lceil (i+f)/2 \rceil = (i+f+1)/2$.

Coppie sottili In un vettore A di interi, si dice *spessore* della coppia $(A[i], A[j])$ la differenza fra il massimo e il minimo della coppia. Inoltre, si dice *distanza* della coppia $(A[i], A[j])$ la differenza, in modulo, fra gli indici. Per esempio, dati $A[7] = 6$ e $A[3] = 1$, la coppia $(A[7], A[3])$ ha spessore $i6 - 1 = 5$ e distanza $7 - 3 = 4$. Progettare un algoritmo, che preso un vettore A di $n = 2^k$ interi distinti e un intero C, trovi la coppia $(A[i], A[j])$ di distanza massima tra quelle di spessore al piú C. La complessitá dell' algoritmo deve essere $O(n \log n)$.

Una soluzione *brute-force* del problema enumera in successione tutte le coppie del vettore. Per ogni coppia ammissibile, cioé con spessore al piú C, calcola la distanza e aggiorna la soluzione se la coppia correntemente esaminataha distanza massima.

Per risolvere in tempo $O(n \log n)$ il problema proposto, innanzitutto si copi A in un vettore appoggio B e si memorizzino le posizioni iniziali in A degli elementi in PA e PB, ossia $\forall 1 \leq i \leq n, B[i] = A[i]$ e $PB[i] = PA[i] = i$.

Si applichi poi un algoritmo divite-et-impera.

Caso base: Se A ha al piu' un elemento (i.e.,$\text{fine} - \text{inizio} + 1 \leq 1$), allora se A contiene almeno un elemento tale elemento e' copiato in B e la sua posizione originale in A e' copiata in PB. In ogni caso la distanza e' inizializzata a -1.

Caso generale: Se A ha piú di un elemento (i.e.,$\text{fine} - \text{inizio} + 1 \geq 2$), sia q il punto medio di A. Dopo aver ricorsivamente chiamato l'algoritmo su $A, inizio, q$ e $A, q+1, fine$, si riordinano gli

elementi in senso crescente utilizzando il vettore di appoggio B. Si utilizzano due puntatori l e r, che scorrono rispettivamente la parte sinistra e destra di A.

If $A[l] < A[r]$, ed $A[l]$ é confrontato con i valori memorizzati nella posizione $A[r]$ e in quelle successive (alla destra di r), fino a che non é rilevata la prima coppia non sottile. Si noti che per le ipotesi fatte che tutti i valori memorizzati in A siano interi e distinti al piú C coppie sono candidate ad essere sottili per ogni valore di l. Conseguentemente, per ogni coppia si testa se e' sottile e la distanza é opportunamente aggiornata. Infine, $A[l]$ é inserito in B e l é incrementato.

Similmente, if $A[r] < A[l]$, al piú le C posizioni $A[l], \ldots A[l+C]$ sono candidate ad essere coppie sottili e possono aggiornare la soluzione del prolema. Infine, $A[r]$ é inserito in B e r é incrementato.

Infine la coppia sottile a massima distanza cosi' individuata e' confrontata con quella ritornata dalle due invocazioni ricorsive dell'algoritmo, e fra le tre distanze e' ritornata la distanza massima.

```
#include <array>
int coppiesottili(array*, int, int, int);

int main() {
        array *A = new array();
        array *P = new array();
        int dim;
        cout << "Inserisci la dimensione dell'array: ";
        cin >> dim;

        createArray(A, dim);
        createArray(PA, dim);
        insertArray(A);
        insertArray(PA);
        //l'iesimo elemento contiene i, ossia PA[i]=i


        int dmax = coppiesottili (A, PA, 0, dim-1,C);
                cout << dmax << endl;
        cout << "————————————————————————————————————————————" << endl;
        }

int coppiesottili(array* A, array *PA, int inizio, int fine, int C){
        static int *B = createArray(dim);
        static int *PB = createArray(dim);
        int t, dtemp;

if (inizio >= fine) {
                if (inizio=fine) {B[inizio]=A[inizio];
                                  PB[inizio]=PA[inizio]};
                copiesottili= -1;}
                else {
                int q =(i+f)/2;
                int dleft= copiesottili (A, inizio, q, C);
                int dright= copiesottili (A, q+1, fine, C);
                int l = inizio; int r = q+1;
                int n=inizio; int dtemp=-1;
                while (n <= fine) do {
                if (A[l] <= A[r]){B[n]=A[l];
                                  PB[n]=PA[l];
                                  t = r;
                                  while (A[t] -A[l] <= C) {
```

```
                                        dtemp = max(dtemp, abs (PA[t]-PA[l]));
                                        t++;}
                                        else
                                        {B[n]=A[r];
                                         PB[n]=PA[r];
                                        t = l;
                                       while (A[t] -A[r] <= C) {
                                       dtemp = max(dtemp, abs (PA[t]-PA[l]));
                                         t++;}
                                        }
                                       n++;}
                    n=inizio;
                    while (n <= fine) do {
                    A[n]=B[n];
                    PA[n]=PB[n];
                    n++;
                    }
                    copiesottili=max(dleft,dright,dtemp);
          }
}
```

Ricerca Binaria Sono dati un array A **ordinato** di n interi ed un intero x. Determinare se x appartiene ad A.

```
int ricerca_binaria (array* A, int i, int f, int x){
if (i=f) {if (A[i] = x) {return 1;}
          else {return 0;}}
    else {int q=(i+f)/2;
          if (x > A[q]) {return ricerca_binaria(A, q+1, f);}
                else   {return ricerca_binaria(A, i, q);}}
          }
```

Se si intende ritornare la posizione di x in A quando x appartiene ad A, l'algoritmo puo' essere facilmente modificato. Infatti e' sufficiente modificare il caso base come segue. Si noti che se l'algoritmo ritorna -1 si indica che $x \not in A$.

```
if (i=f) {if (A[i] = x) {return i;}
          else {return -1;}}
```

Elemento Mancante E' dato un array A **ordinato** di n interi distinti, compresi fra 1 e $n+1$. Si determini l'intero m mancante in A.

```
int trova-numero (array* A, int i, int f){

if ((i=f) && (A[i] = i)) {return i+1}
      else {if ((i=f) && (A[i] = i+1)) {return i}
                else { int q=(i+f)/2;
                   if (A[q]=q) {return trova-numero(A, q+1, f);}
                         else   { return trova-numero (A, i, q);}
                                }
                }
```

Rotazione di un array crescente E' dato un array A **ordinato** di n interi distinti, rotato ciclicamente di k posizioni a destra. Determinare k.

```
int trova-discont(array* A, int i, int f){

if ((i=f-1) && (A[i] < A[f])) {return -1}
        else { if ((i=f-1) && (A[i] > A[f])) {return f}
                        else { int q=(i+f)/2;
                           if (A[i]<A[q]) { return trova-discont(A, q, f);}
                                  else    { return trova-discont(A, i, q);}
                              }
            }
 }
```

Remark Quando un array e' rotato di k posizioni verso destra, l'elemento di A originariamente in posizione i appare dopo la rotazione in posizione $(i+k) \bmod n$. Se l'array e' rotato ciclicamente di $k = 0$ posizioni verso destra, l'array e' ordinato.

Quando un array e' rotato di k posizioni verso sinistra, l'elemento di A originariamente in posizione i appare dopo la rotazione in posizione $(i-k) \bmod n$. E' facile vedere che se un array e' rotato ciclicamente di k posizioni verso destra, allora sara' rotato di $n-k$ posizioni verso sinistra. Inoltre, un array rotato ciclicamente verso sinistra di n posizioni e' ordinato.

Picco di un vettore unimodale Un array A di n, con $n \geq 3$, interi distinti e' detto*unimodale* se esiste un indice m, con $m \geq 1$, tale che $A[0] < \ldots < A[m-1] < A[m] > A[m+1] > \ldots > A[n]$. Dato un array unimodale, si determini m.

```
int trova_picco(array* A, int i, int f){

if ((i=f-1) && (A[i] < A[f])) {return f}
        else { if ((i=f-1) && (A[i] > A[f])) {return i}
                else { int q=(i+f)/2;
                     if (A[q-1]<A[q]) { return trova-picco(A, q, f);}
                            else     { return trova-picco(A, i, q);}
                     }
              }
 }
```

Remark. Quando il caso base della ricorsione lavora su array di cardinalita' 2. Se l'elemento $A[q]$ analizzato non puo' essere escluso dai successivi livelli di ricorsione, conviene seguire questa semplice regola: fissato $q = \lfloor (i+f)/2 \rfloor$, si richiami fra A, i, q e A, q, f.

Salto doppio In un array A di n, con $n \geq 2$, interi distinti vi e' un *salto-doppio* se $A[i+1] - A[i] \geq 2$, con $0 \leq i \leq n-1$. Progettare un algoritmo che, dato un array A con $n > 2$ interi tale che $A[n-1] - A[0] \geq n$, trovi un salto-doppio.

Innanzitutto, si provi che se $A[n-1] - A[0] \geq n$ vi e' almeno un salto in A. Procediamo per assurdo. Se non vi fosse un salto-doppio, ogni elemento memorizzato in A e' minore di o uguale al precedente valore piu' 1. E' facile percio' dimostrare per induzione che $A[n-i] \leq A[0] + (n-i)$ e quindi, se non vi e' alcun salto-doppio, deve valere $A[n-1] - A[0] \leq (n-1)$.

```
int salto-doppio(array* A, int i, int f){

if (i > f-1) then {return -1}
        else { if ((i=f-1) && (A[f]-A[i-1] \le 1)) {return -1}
                                        else {return i}
         else {int q=(i+f)/2;
               if (A[q]-A[i] \le  q-i) {return salto-doppio(A, q+1, f);}
```

```
                    else {return salto-doppio (A, i, q);}
                                        }
                    }
                }
```

2 Grafi

Questa sezione considera la struttura *grafo* $G = (V, E)$, ove $V = \{1, 2, \ldots, n\}$ e' l'insieme dei *nodi* del grafo ed E e' l'insieme degli *archi* di G. Ciascun arco consiste in una coppia $\{(i, j)\}$ *non ordinata* (resp. *ordinata*) di vertici se il grafo e' *non-orientato* (resp., *orientato*). Sono proposte due implementazioni del grafo in C: la rappresentazione con la *matrice delle adiacenze* e quella con il *vettore delle adiacenze*. La rappresentazione con matrice delle adiacenze di un grafo orientato $G = (V, E)$ e' una matrice booleana di dimensioni $|V| \times |V|$, ove $M[i][j] = true$ se $(i, j) \in E$. La rappresentazione con matrice delle adiacenze di un grafo non-orientato $G = (V, E)$ e' una matrice booleana di dimensioni $|V| \times |V|$, ove $M[i][j] = M[j][i] = true$ se $(i, j) \in E$, cioe' l'arco (i, j) di un grafo non-orientato e' rappresentato dai due archi orientati (i, j) and (j, i). Chiaramente, indipendedentemente dal numero di archi presenti nel grafo, la rappresentazione richiese spazio proporzionale al quadrato del numero dei vertici.

L'operazione di base sulla struttura grafo e' la sua *visita*, ossia esplorare tutti i vertici e gli archi di G a partire da un vertice qualsiasi, detto *sorgente*, costruendo un *albero di copertura* T_G. A tale scopo, per ogni vertice $v \in V$, sono mantenute le seguenti informazioni: $G->visitato[v]$ indica se il vertice v e' gia' stato incontrato durante la visita del grafo, $G->B[v]$ indica il vertice *padre* di v nell'albero T_G, $G->inftime[v]$ e $G->ftime[v]$ i tempi di inizio e fine visita del vertice v, e $G->d[v]$ la distanza del vertice v dalla sorgente.

```
void createGraphMatrix(graph* G) {
        G->M = new bool*[G->dimV];
        for (int i = 0; i < G->dimV; i++)
                G->M[i] = new bool[G->dimV];

        G->visitato = new bool[G->dimV];
        G->B = new int[G->dimV];
        G->d = new int[G->dimV];
        G->time = 0;
        G->intime = new int[G->dimV];
        G->ftime = new int[G->dimV];
}
```

Seguono le due procedure per leggere da terminale un grafo G e per stampare a terminale la sua matrice delle adiacenze.

```
void insertGraphMatrix(graph* G) {
        for (int i = 0; i < G->dimV; i++) {
                cout << "  Inserisci archi uscenti da " << i << ": ";
                int arch;
                cin >> arch;
                for (int j = 0; j < arch; j++) {
                        cout << "    Inserisci il vicino di " << i << ": ";
                        int adj;
                        cin >> adj;
                        G->M[i][adj] = true;
                }
        }
}
```

```
void printGraphMatrix(graph* G) {
        cout << "M -> matrice adiacenza" << endl << endl;
        for (int i = 0; i < G->dimV; i++) {
                for (int j = 0; j < G->dimV; j++) {
                        cout << " | " << G->M[i][j];
                }
                cout << endl;
        }
}
```

La procedura *Depth First Search (DFS)* visita i nodi di G richiamando la visita ricorsivamente su ogni nodo raggiunto per la prima volta. La visita segue il paradigma *L'ultimo nodo visitato e' il primo ad essere esplorato.* La variabile time e' inizializzata a 0.

```
void DFSMatrix(graph* G, int s) {
        G->time++;
        G->intime[s] = G->time;
        G->visitato[s] = true;
        cout << s << " | ";

        for (int j = 0; j < G->dimV; j++) {
                if ((G->M[s][j] == 1) &&(G->visitato[j] == false)) {
                        G->B[j] = s;
                        DFSMatrix(G, j);
                }
        }

        G->time++;
        G->ftime[s] = G->time;
}
```

Prima di iniziare la visita e' necessario inizializzare la struttara dati come segue. Si noti che se un nodo non e' stato ancora visitato, suo padre e' inizializzato a -1. Infine si noti che i campi non utilizzati dalla procedura *DFS* sono ignorati.

```
void resetDFS(graph* G) {
        for (int j = 0; j < G->dimV; j++) {
                G->visitato[j] = false;
                G->B[j] = -1;
                G->intime[j] = -1;
                G->ftime[j] = -1;
        }
}
```

La visita di G e' completa solo quando tutti i nodi sono stati visitati. Poiche' eseguendo la procedura *DFS* a partire dal nodo i si raggiungono tutti e solo i nodi raggiungibili da i, puo' essere necessario completare la visita di G ripartendo da un nodo non ancora raggiunto.

In particolare, se il grafo G e' non-orientato, $DFS(G, i)$ visita la componente connessa di G a cui appartiene i. Pertanto, la visita in profondita' di G per visitare tutti i nodi di G richiede di invocare la procedura *DFS* a partire da tanti nodi distinti quanti sono le componenti connesse in G.

```
void DFSTotaleMatrix(graph* G) {
        for (int i = 0; i < G->dimV; i++) {
                if (G->visitato[i] == false) {
                        DFSMatrix(G, i);
                        G->time = 0;
```

```
                }
        }
}
```

La procedura *Breadth First Search (BFS)* visita i nodi di G in profondita', completando prima la visita di tutti gli archi uscenti dal nodo che si sta esaminando e sospendendo la visita dei nodi raggiunti. I nodi saranno poi visitati nello stesso ordine in cui sono stati sospesi. Per mantenere tale ordine fra i nodi, ossia per rispettare che *il primo nodo sospeso sara' il primo nodo ad essere esplorato*, la procedura *BFS* utilizza una coda Q d'appoggio. Nella coda Q gli oggetti sono inseriti ed estratti seguendo la politica *First In First Out*. Nel seguito si implementa una coda come una struttura allocata dinamicamente, e realizzata con puntatori.

```
void creaCoda(lista* Q) {
        Q->coda = NULL;
        Q->testa = NULL;
        Q->count = 0;
}

void inCoda(int elem, lista* Q) {
        nodo* n = new nodo();
        n->key = elem;

        if (Q->count == 0) {
                Q->coda = n;
                Q->testa = n;
        } else {
                Q->coda->next = n;
                Q->coda = n;
        }

        Q->count++;
}

void fuoriCoda(lista* Q) {
        if (Q->testa != NULL) {
                nodo* tmp = Q->testa;
                Q->testa = Q->testa->next;
                delete tmp;
        }

        Q->count--;
}

bool codaVuota(lista* Q) {
        if (Q->count == 0)
                return true;
        else
                return false;
}

int leggiCoda(lista* Q) {
        if (!codaVuota(Q))
                return Q->testa->key;
```

```
        return -1;
}

void stampaCoda(lista* Q) {
        nodo* p = Q->testa;

        while (p != NULL) {
                cout << " | " << p->key;
                p = p->next;
        }
        cout << endl;
}
```

Infine, la procedura *BFSMatrix* esegue la visita in profondita' del grafo G a partire dal vertice s. Una proprieta' molto interessante di questa procedura e' che i nodi di G sono visitati in ordine non decrescente della lora distanza da s. Precisamente, il campo $G->d[v]$ mantiene la distanza minima del vertice v da s $\forall v \in V$, ossia $G->d[v]$ contiene il numero di archi sul cammino piu' breve da s a v.

```
void BFSMatrix(graph* G, int s) {
        for (int j = 0; j < G->dimV; j++) {
                G->visitato[j] = false;
                G->B[j] = -1;
                G->d[j] = INT_MAX;
        }

        G->visitato[s] = true;
        G->d[s] = 0;

        creaCoda(G->Q);
        inCoda(s, G->Q);
        cout << s << " | ";

        do {
                int i = leggiCoda(G->Q);
                for (int j = 0; j < G->dimV; j++) {
                        if ((G->M[i][j] == 1) && (G->visitato[j] == false)) {
                                G->visitato[j] = true;
                                inCoda(j, G->Q);
                                cout << j << " | ";
                                G->B[j] = i;
                                G->d[j] = G->d[i]+1;
                        }
                }
                fuoriCoda(G->Q);

        } while (!(codaVuota(G->Q)));
}
```

Le due procedure seguenti stampano per ogni nodo la sua distanza dalla sorgente e il padre nell'albero di copertura, rispettivamente.

```
void dBFS(graph* G) {
        cout << "vettore delle distanze" << endl;
        for (int i = 0; i < G->dimV; i++)
                cout << i << ": " << G->d[i] << endl;
```

```
}

void BBFS(graph* G) {
        cout << "vettore dei padri" << endl;
        for (int i = 0; i < G->dimV; i++)
                cout << i << ": " << G->B[i] << endl;
}
```

Durante la visita in profondita' di un grafo gli archi di una componente connessa sono partizionati in archi appartenenti all'albero di copertura *edgeTree*, archi in avanti *edgeAhead*, archi all'indietro *edgeback*, e archi di attraversamento *edgeCross*. Ogni insieme di archi si immagina rappresentato da una coda.

```
void checkEdgeMatrix(graph* G) {
        for (int i = 0; i < G->dimV; i++) {
                        for (int j = 0; j < G->dimV; j++) {
                    if (G->M[i][j] == 1) {
                            int u = i;
                            int v = j;
                     if (u == G->B[v])
                         inEdgeQueue(u, v, G->edgeTree);
                             else {
         if ((G->intime[u] < G->intime[v]) && (G->ftime[v] < G->ftime[u]))
                        inEdgeQueue(u, v, G->edgeAhead);
         if ((G->ftime[v] > G->ftime[u]) && (G->intime[u] > G->intime[v]))
                        inEdgeQueue(u, v, G->edgeBack);
         if ((G->ftime[v] < G->intime[u])
                        inEdgeQueue(u, v, G->edgeCross);
                    }
                }
            }
            }
```

Questa caratterizzazione degli archi e' molto utile perche' aiuta a risolvere problemi elementari su un grafo. Ad esempio, poiche' un ciclo semplice ha un arco all'indietro, e' possibile verificare se un grafo orientato e' aciclico testando se non ha archi all'indietro. Ricordando che un grafo orientato aciclico e' detto *Directed Acyclic Graph (DAG)*, assumendo che tutti i vertici di G siano raggiungibili dal vertice 0, si ha:

```
bool isDagMatrix(graph* G) {
resetDFS(G);
cout << "un DAG non ha archi all'indietro, quindi si deve calcolare la DFS" << end
DFSMatrix(G, 0);
cout << "DFS eseguita" << endl;
cout << endl << endl;
checkEdgeMatrix(G);
cout << "controllo archi eseguito" << endl;
if (emptyQueue(G->edgeBack))
                return true;
return false;
```

Il seguente algoritmo stampa, se esiste, un ciclo di G.

```
void ciclo (graph* G) {
        if (isDagMatrix(G))
            cout << "Il grafo non ha cicli" << endl;
                else {
```

```
                cout << "Il grafo ha cicli" << endl;
                        int u, v;
                       u = G->edgeBack->head->u;
                cout << u << " | ";
                        v = G->edgeBack->head->v;
                        int c = G->B[u];
            while (c != v) {
                        cout << c << " | ";
                        c = G->B[c];
            }
              cout << v << " | ";
              cout << u << " | " << endl;
        }
         }
```

Si definisce *sort topologico* o *linearizzazione* di un grafo orientato (non necessariamente connesso) un ordinamento dei nodi tale che nessun nodo u preceda nell'ordinamento un altro nodo v se (v, u) e' un arco del grafo G. Nel grafo ordinato topologicamente, tutti gli archi sono orientati da sinistra verso destra. Il primo nodo non ha archi entranti e ogni nodo v e' preceduto nell'ordine topologico da tutti i nodi che hanno archi entranti in v. Applicando la definizione appena vista, l'algoritmo *topsortMatrix* restituisce una linearizzazione di G.

```
void topSortMatrix(graph* G) {
        for (int i = 0; i < G->dimV; i++)
                G->degreeIn[i] = 0;
            for (int i = 0; i < G->dimV; i++)
                for (int j = 0; j < G->dimV; j++)
                    if (G->M[i][j] == 1)
                        G->degreeIn[j]++;
              createQueue(G->Q);
              int numOrd = 0;
              for (int i = 0; i < G->dimV; i++) {
                      if (G->degreeIn[i] == 0)
                      inQueue(i, G->Q);
                              }
                do {
                 int v = readQueue(G->Q);
                         outQueue(G->Q);
                         G->L[numOrd] = v;
                         numOrd++;
               for (int j = 0; j < G->dimV; j++)
                       if (G->M[v][j] == 1) {
                           int w = j;
                          G->degreeIn[w]--;
                       if (G->degreeIn[w] == 0)
                              inQueue(w, G->Q);
              }
              } while (!emptyQueue(G->Q));
              if (numOrd < G->dimV-1)
                 cout << "Non esiste un ordine topologico" << endl;
            else {
                 for (int i = 0; i < G->dimV; i++)
                        cout << " | " << G->L[i];
              }
```

```
}
```

Un elegante algoritmo per l'ordinamento topologico esegue la visita in profondita' del grafo ed ordina i nodi in ordine crescente di $ftime$. In pratica, quando la procedura DFS conclude la visita dell'i-esimo nodo, tale nodo diviene $(n - i + 1)$-esimo nodo dell'ordine topologico dei vertici e memorizza tale posizione nel campo $G->SortOrder$. Per facilitare la stampa del grafo linearizzato e' utilizzato un array `SortTop` in cui i nodi sono memorizzati nell'ordine in cui appaiono nella linearizzazione. `Order` e' una variabile globale inizializzata a $|V|$, usata per indicizzare l'array SortTop e memorizzata alla fine della visita di ciascun nodo v in $G->SortOrder[v]$.

```
void DFSTotaleMatrix(graph* G) {
        for (int i = 0; i < G->dimV; i++) {
                if (G->visitato[i] == false) {
                        DFSMatrix(G, i);
                        G->time = 0;
                        }
                }
        }

void DFSMatrix(graph* G, int s) {
        G->time++;
        G->intime[s] = G->time;
        G->visitato[s] = true;
        for (int j = 0; j < G->dimV; j++) {
                if ((G->M[s][j] == 1) &&(G->visitato[j] == false)) {
                                        G->B[j] = s;
                                        DFSMatrix(G, j);
                                                }
                                        }
                G->time++;
                G->ftime[s] = G->time;
                G->sortorder[s]= order;
                SortTop[order]=s;
                order--;
                }
```