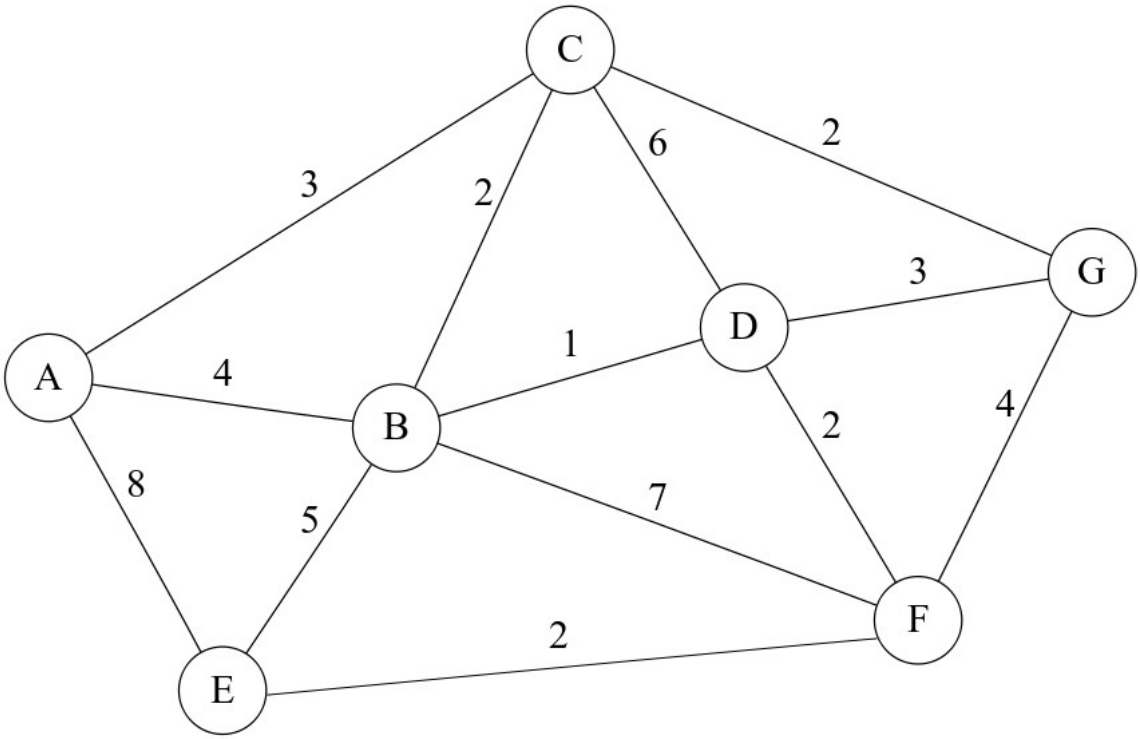
Algoritmi e Basi di dati – Modulo di Algoritmi e Strutture dati

11 giugno 2010 - ANNO ACCADEMICO 2009/10

NOME: **COGNOME:** **MATRICOLA:**

Esercizio 1

Sia dato il seguente grafo non orientato:



Applicare l'algoritmo di Dijkstra per calcolare i cammini minimi dal nodo A verso tutti gli altri nodi

Q	distanze A	B	C	D	E	F	G	predec A	B	C	D	E	F	G
A B C D E F G	0	inf	inf	inf	inf	inf	inf	-	-	-	-	-	-	-
B C D E F G	0	4	3	inf	8	inf	inf	-	A	A	-	A	-	-
B D E F G	0	4	3	9	8	inf	5	-	A	A	C	A	-	C
D E F G	0	4	3	5	8	11	5	-	A	A	B	A	B	C
D E F	0	4	3	5	8	9	5	-	A	A	B	A	G	C
E F	0	4	3	5	8	7	5	-	A	A	B	A	D	C
E	0	4	3	5	8	7	5	-	A	A	B	A	D	C

1	*2*	*3*	*4*	*5*

Esercizio 2
Sia data la seguente funzione:

```
int f(Node* t) {
     if (t) {
          t->label = t->label * t->label;
          if (!t->left && !t->right) return 1;
          return f(t->left)+f(t->right);
     }
     return 0;
}
```

a) Se viene applicata ad un albero binario ad etichette intere, cosa restituisce e come modifica l'albero?

b) Se viene applicata ad un albero binario di ricerca ad etichette intere, l'albero risultante è comunque un albero binario di ricerca? Se sì, mostrare che le proprietà dell'albero binario di ricerca sono preservate, altrimenti mostrare un controesempio.

a) *Restituisce il numero di foglie dell'albero e eleva al quadrato le etichette di ogni nodo*

b)

L'albero risultante non è necessariamente un albero binario di ricerca. Infatti, la funzione $f(x) = x^2$*, che viene applicata ad ogni etichetta non è monotona sul dominio dei numeri interi e quindi la proprietà che ogni nodo deve essere maggiore dei discendenti di sinistra e minore dei discendenti di destra può venire violata. Ad esempio, se applicata all'albero binario di ricerca con due nodi formato da 2 (radice) e -3 (figlio sinistro) restituisce l'albero con 4 come radice e 9 come figlio sinistro, violando la proprietà dell'albero binario di ricerca.*

Esercizio 3
Calcolare la complessità del blocco (indicando le relazioni di ricorrenza di tempo e risultato per ogni funzione) in funzione del numero di nodi dell'albero t, supponendo che sia quasi bilanciato.

```
{
  int a = 0;
  for (int i=0; i <= g(f(t)); i++)
     a += Nodes(t);
}
```

Le funzioni **f** e **g** sono definite come segue:

```
int f(Node* tree) {
  if (!tree) return 1;
  int a=Nodes(tree),j=a;
  for (int i=1;i<=j;i++) a+=j;
  int b = f(tree->left)
          +f(tree->right);
  return a+b;
}
```

```
int g(int x) {
  if (x<=1) return 1;
  cout << g(x/2) + g(x/2) + g(x/2);
  int a=1+g(x/2);
  return a;
}
```

Tf(0)=cost
Tf(n)=dn+2Tf(n/2) O(nlogn)

Rf(0)=1
Rf(n)=dn^2+2Rf(n/2) O(n^2)

Tg(0)=cost
Tg(n)=d+4Tg(n/2) O(n^2)

Rg(0)=1
Rg(n)=1+Rg(n/2) O(logn)

Numero iterazioni for: Rg(Rf(n)) = (O(log(n^2))= O(logn)
Complessità di una iterazione: C[f(t)] + C[g(n^2)] + C[Nodes(t)]= O(nlogn) + O(n^4) + O(n) = O(n^4)
Complessità del blocco: O(n^4 * logn)

Esercizio 4
Sia dato un albero binario t ad etichette intere. Si scriva una funzione duplica che restituisce un puntatore ad un altro albero binario identico a t, ma con le etichette raddoppiate. Si calcoli la complessità della soluzione proposta in funzione del numero n di etichette di t.

```
Node* duplica(Node *t) {
 if (!t) return NULL;
 Node *np = new Node();
 np->label = 2* t->label;
 np->left = duplica(t->left);
 np->right = duplica(t->right);
 return np;
}
```

Complessità: O(n)

Esercizio 5
Sia dato un albero generico memorizzato figlio-fratello ad etichette intere. Si scriva una funzione cfp che trasforma l'albero andando a cancellare tutti i nodi che sono figli in posizione pari. Ad esempio, se il nodo A ha come figli B,C,D,E,F, dopo l'applicazione di cfp, il nodo A avrà come figli B,D, F. Una volta che si decide di cancellare un nodo, è necessario cancellare anche tutti i suoi discendenti.

```
void cancella(Node*& t) {
  if (t) {
    Node* tmp = t;
    deleteTree(t->left);
    t = tmp->right;
    delete tmp;
    if (t) cancella(t->right);
  }
}
void cfp(Node* t) {
  if (t) {
    if (!t->left) {
      cfp(t->right);
      return;
    }
    cancella(t->left->right);
    cfp(t->left);
    cfp(t->right);
  }
}
void deleteTree(Node*& t) {
  if (t) {
    deleteTree(t->left);
    deleteTree(t->right);
    delete t; t = NULL;
  }
}
```