KOLOKWIUM I

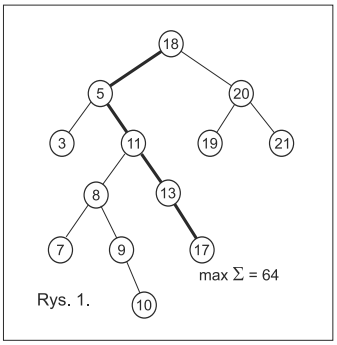
1. Napisz funkcję logiczną `int sublist(wel P, wel Q)`, której parametrami są wskaźniki na *nieuporządkowane* listy jednokierunkowe `P` i `Q` złożone z liczb typu `int`, zwracającą wrtość 1, gdy lista `Q` zawarta jest w `P`, to jest gdy `P = P1` $\rightarrow$ `Q` $\rightarrow$ `P2`, gdzie `P1` i `P2` są listami (które mogą być puste lub nie). W przeciwnym razie funkcja zwraca wartość 0. W szczególności lista pusta `Q=NULL` jest zawarta w *każdej* liście `P`.

2. Napisz funkcję usuwającą powtórzenia z prostej listy 2-kierunkowej.

3. Napisz procedurę `max_powt(C)`, która zwraca maksymalną liczbę powtórzeń tej samej wartości w jednokierunkowej cyklicznej liście `C`.

4. Drzewo uporządkowane zawiera liczby całkowite. Należy znaleźć największą wśród sum tych liczb liczonych wzdłuż ścieżek z korzenia do liści (Rys. 1).

5. Dla danego drzewa binarnego należy wyznaczyć jego *defekt*, to jest różnicę między jego wysokością a minimalną głębokością liścia. Kod powinien być kompletny, z kodami funkcji pomocniczych jeśli ich używasz.



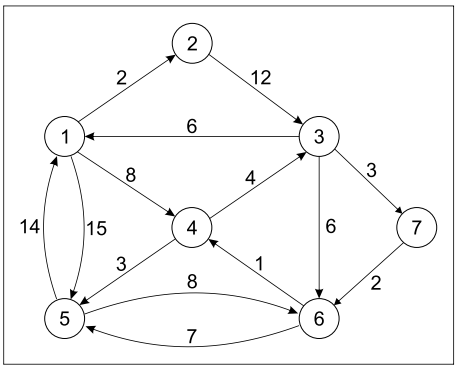
Rys. 1.

KOLOKWIUM 2

1. Jakie warunki powinny spełniać dane, które zamierzamy poddać sortowaniu metodą kubełkową?

2. Porównać złożoność metody sortowania przez wstawianie realizowane w tablicy z podobnym algorytmem realizowanym w dwukierunkowej liście. Opisać w szczególności czy — a jeśli tak, to w jakim stopniu — zastosowanie listy może zmniejszyć liczbę porównań i/lub podstawień niezbędnych do posortowania danych.

3. Przeprowadź symulację działania algorytmu Dijkstry na skierowanym grafie `G` o strukturze przedstawionej na rysunku. Wierzchołkiem startowym jest a) `s=1`, b) `s=5`. W każdym przypadku podaj kolejność, w jakiej wierzchołki kwalifikowane będą jako odwiedzone. Wypisz przebieg najkrótszych ścieżek z `s` do pozostałych wierzchołków. Struktura list sąsiedztwa jest następująca:



`S[1]` $\rightarrow$ `2, 5, 4` `S[2]` $\rightarrow$ `3` `S[3]` $\rightarrow$ `1, 6, 7`
`S[4]` $\rightarrow$ `3, 5` `S[5]` $\rightarrow$ `6, 1` `S[6]` $\rightarrow$ `5, 4` `S[7]` $\rightarrow$ `6`

4. *Ekscentrycznością* $\mathcal{E}$ wierzchołka v w nieskierowanym grafie G nazywamy długość najdłuższej wśród najkrótszych ścieżek z v do wszystkich innych wierzchołków,

$$\mathcal{E}(v) \;=\; \max_{x \in V(G)} d_{\min}(v,x)\,, \qquad d_{\min}(v,x) = \text{długość najkrótszej ścieżki z } v \text{ do } x\,.$$

Centrum $\mathcal{C}$ grafu G jest to zbiór wierzchołków G o minimalnej ekscentryczności. Wśród zastosowań można wymienić np. algorytm wyboru wsi na siedzibę gminnej przychodni zdrowia. Wybór wsi z centrum grafu połączeń drogowych jest najbardziej sprawiedliwy dla mieszkańców całej gminy, także tych z jej obrzeży.

Załóżmy, że mamy funkcję `float dmin(int u, int v)`, zwracającą długość najkrótszej ścieżki z u do v w grafie. Napisz program, który wyznaczy ekscentryczności `E[v]` wszystkich wierzchołków, a następnie centrum danego grafu.

BONUS (1 PKT): Zaproponuj jak można dodatkowo uwzględnić liczbę mieszkańców poszczególnych wiosek w sprawiedliwym wyborze lokalizacji przychodni.

5. Graf nazywamy dwudzielnym, jeśli zbiór jego wierzchołków dzieli się rozłącznie na 2 części $V = V_1 \cup V_2$, tak, że wszystkie krawędzie grafu są połączeniami między V_1 i V_2. Odwrotnie: żadna para wierzchołków w V_1 ani też w V_2 nie jest połączona krawędzią. Napisz pesudokod programu bazujący na schemacie BFS, który sprawdza czy zadany graf jest dwudzielny. Przyjmujemy, że struktura grafu opisana jest przez listy sąsiedztwa $S[u]$, $u \in V$.

Rozwiązania

I.1. Funkcja pomocnicza `subl` sprawdza czy lista `Q` jest identyczna z początkowym segmentem listy `R`. Funkcja `sublist` wywołuje `subl` ustawiając `R` na kolejnych elementach listy `P`.

```
int subl(wel R, wel Q){
   while ((Q!=NULL) && (R!=NULL) && (Q->x == R->x)){
      Q=Q->nast; R=R->nast;
   }
   return Q==NULL;
}

int sublist(wel P, wel Q){
   wel R = P;
   int s = 0;
   while ((R!=NULL) && (s==0)){
      s = subl(R, Q);
      R = R->nast;
   }
   return s;
}
```

I.2.
```
void usun_powt2(wel2 L}{
   wel2 *q;
   while (L!=NULL){
      q = &(L->nast);
      while (*q != NULL)
         if ((*q)->x == L->x) { usunp(q); } else { q=&((*q)->nast); }
      L=L->nast;
   }
}

void usunp{wel2 *q){
   wel2 r=*q;
   if (r!=NULL){
      *q=r->nast;
      if (*q!=NULL) (*q)->pop = r->pop;
      free(r);
   }
}
```

I.3.
```
int max_powt(wel L){
   wel p=L, q;
   int k, mx=0;
   if (L != NULL){
      do {
         q = p->nast;
         k = 1;
         while (q != L){
            if ( q->x == p->x ) k++;
            q = q->nast;
         }
         if (k > mx) mx = k;
         p = p->nast;
      } while (p->nast != L);
   }
   return mx;
}
```

I-4.
```
int max_suma(weld T}{
   int j,k;
   if (T==NULL) return 0;
   j = max_suma(T->l);
   k = max_suma(T->p);
   return T->x + (j>k) ? j : k;
}
```

I.5.
```
int wys(weld T){
   if (T==NULL) return 0;
   return 1+max(wys(T->l),wys(T->p));
}
```

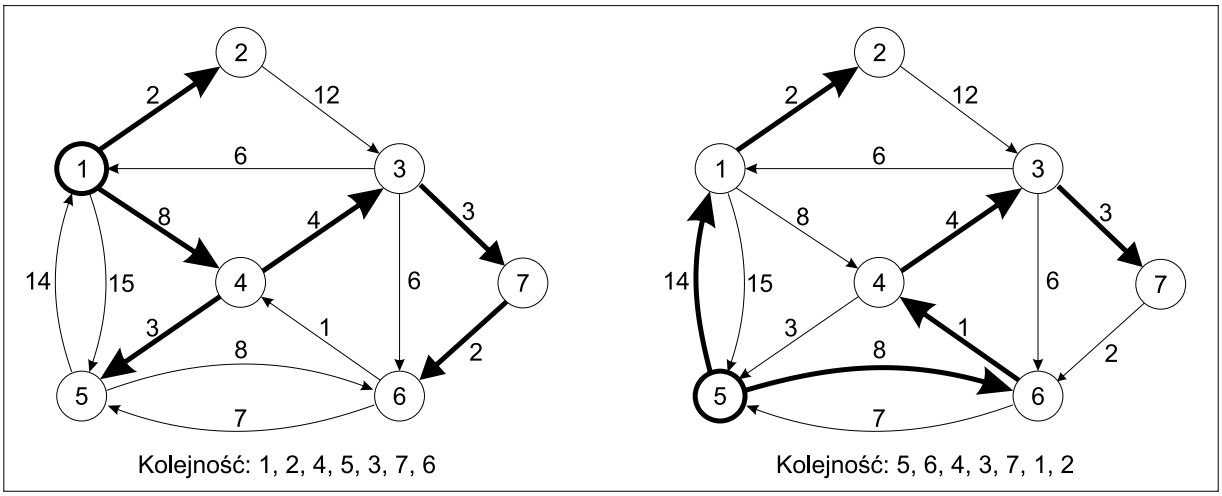
```
int mingll(weld T){
   if (T==NULL) return 0;
   return 1+min{mingll(T->l),mingll(T->p)};
}
```

```
int defekt(weld T){
   return wys(T)-mingll(T);
}
```

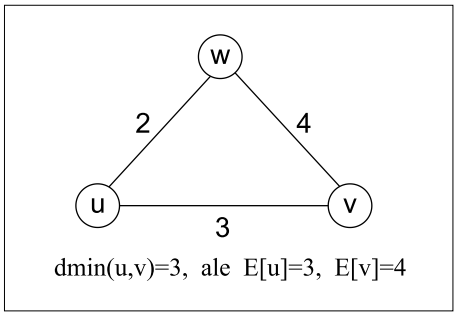
II.1. Sortowanie kubełkowe wymaga aby: a) z góry znany był zakres wielkości sortowanych danych $A \le x_i \le B$ oraz b) by rozkład danych x_i w przedziale $[A, B]$ był zbliżony do równomiernego. Chodzi o to by a) możliwe było wyznaczenie podziału zakresu danych na kubełki i b) by na jeden kubełek przypadało mniej więcej tyle samo danych x_i, tj. by wszystkie listy realizujące kubełki miały podobną długość.

II.2. Sortowanie przez wstawianie w tablicy w pesymistycznym przypadku wymaga około $\frac{n^2}{2}$ porównań i tyle samo podstawień. Sortowanie przez wstawianie nowych elementów we właściwych miejscach zarówno 1- jak i 2-kierunkowej listy w podobnym przypadku wymaga także około $\frac{n^2}{2}$ porównań, ale liczba podstawień jest rzędu n, na każdy sortowany element x_i jedno utworzenie nowego elementu listy, zapisanie w nim danej x_i i aktualizację 2 łączników `nast` i `poprz`.

II.3.



II.4.
```
ME = INT_MAX;
for (u=1; u<=n; u++){
   E[u] = 0;
   for (v=1; v<=n; v++){
      d = dmin(u,v);
      if (E[u] < d) E[u] = d;
   }
   if (E[u] < ME) ME = E[u];
}
printf("Centrum: ");
for (u=1; u<=n; u++) if (E[u]==ME) printf("%i  ", u);
printf("\n minimalna ekscentryczność %i\n", ME);
```



Mimo że w grafie nieskierowanym `dmin(u,v)`=`dmin(v,u)`, to na ogół `E[u]`≠`E[v]` (por. rysunek).

BONUS: Optymalizacja uwzględniająca liczbę mieszkańców wiosek powinna operować czymś w rodzaju "kosztów transportu", które prócz odległości `dmin` powinny być proporcjonalne do liczby dojeżdżających osób. Zamiast funkcji `dmin(u,v)` należałoby użyć np. $\texttt{kmin}(u,v) = \texttt{dmin}(u,v)\cdot \texttt{m}[v]$, gdzie `m[v]` jest liczbą mieszkańców wioski `v`. Zauważmy, że funkcja `kmin` nie jest już teraz symetryczna, $\texttt{kmin(u,v)} \neq \texttt{kmin(v,u)}$. Jeśli np. w `u` mieszka 100 osób, a w `v` 150, to przy odległości $\texttt{dmin(u,v)} = 20$ km mielibyśmy $\texttt{kmin(u,v)} = 3000$ i $\texttt{kmin(v,u)} = 2000$. Gdyby nie było innych wiosek, obliczenie zmodyfikowanej ekscentryczności wskazałoby — słusznie – `v` jako centrum.

Alternatywnie można też użyć "łącznych kosztów transportu" do danej miejscowości

$$\texttt{K}[u] = \sum_{v\neq u} \texttt{dmin}(u,v)\cdot \texttt{m}[v]^{\gamma}$$

i wybrać jako centrum wierzchołki o minimalnej wartości `K`. Opcjonalna "korekta gamma" przy eksperymentalnie dobranym wykładniku $\gamma > 1$ mogłaby wprowadzić dodatkową preferencję dla większych miejscowości.

II.5. W schemacie BFS zamiast tablicy `Odwiedzony[u]` użyjemy tablicy `Kolor[u]`, w której znajdzie się 0 jeśli wierzchołek `u` nie był odwiedzony, a w przypadku wierzchołka odwiedzonego przypisany mu "kolor" -1 lub $+1$. Działanie programu polega na przypisaniu kolejnym piętrom drzewa BFS naprzemiennie kolorów -1 i $+1$. Jeśli w trakcie przypisywania koloru napotkamy wierzchołek już odwiedzony o przeciwnym od aktualnego kolorze, jest to sygnał, że graf nie jest dwudzielny: funkcja zwraca wówczas 0. Zakładamy że liczba wierzchołków to `n`, a tablica `S` zawiera listy sąsiedztwa. `Q` jest pomocniczą kolejką realizowaną jako lista. Pętla `while ...` w programie głównym wywoła funkcję `BFS` kilka razy jeśli graf nie jest spójny.

```
int BFS(wel S[], int Kolor[], int u){
   int v, kol;  wel Q=NULL;
   push(u,Q);   Kolor[u] = 1;
   while (Q!=NULL){
       u = pop(Q);
       kol = -Kolor[u];
       for (v in S[u]){
          if (Kolor[v]==0){
             Kolor[v] = kol;
             push (v,Q);
          }
          else {
             if (Kolor[v]!=kol) return 0;
          }
       }
   }
   return 1;
}

for (u=1; u<=n; u++) Kolor[u]=0;
u = 1; f = 1;
while (u<=n && f!=0){
   if (Kolor[u]==0) f = BFS(S,Kolor,u);
   u++;
}
if (f==1) printf("Graf jest dwudzielny\n");
```