Algorytmy i struktury danych 2025/26 — kolokwium II

1. Wykonując minimalną liczbę zamian między elementami tablicy `A[k]` ↔ `A[j]` należy utworzyć z niej kopiec:

15	10	8	20	12	17	5	4	18	11	7	3	14	13	9	16	6	2	19	8	15	5

Można posłużyć się reprezentacją kopca w postaci drzewa binarnego.

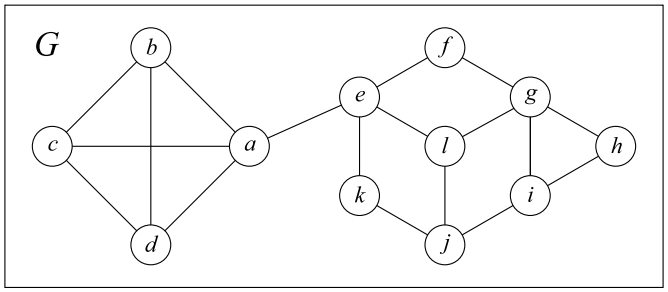
2. Sortowanie przez scalanie, *merge sort*, polega na podziale ciągu danych na dwie równe części (jeśli `n` jest nieparzyste, pierwsza zawiera o jeden element więcej), rekurencyjnym wykonaniu tego sortowania dla każdej z nich, a następnie połączeniu posortowanych połówek w jeden uporządkowany ciąg. Rekursja kończy się, gdy przekazana jej część zawiera nie więcej niż jedną liczbę i porządkowanie staje się trywialne.

Mając dwie funkcje (nie trzeba pisać ich kodu):
- `podziel(&P, &Q)`, która dzieli listę `P` na dwie połowy, pierwszą połówkę wskazuje `P`, a drugą `Q` (`P` zawiera minimalnie 1 element, natomiast `Q` może być pusta);
- `połącz(&P, &Q)`, która tworzy z uporządkowanych list `P` i `Q` jedną listę uporządkowaną, po wyjściu z tej funkcji wskazywaną przez `P`, natomiast `Q` staje się pusta (tak jak w jednym z zadań domowych);

napisz kod funkcji `merge_sort(&P)` porządkującej listę `P`, która korzysta z wywołań funkcji `podziel` i `połącz`.

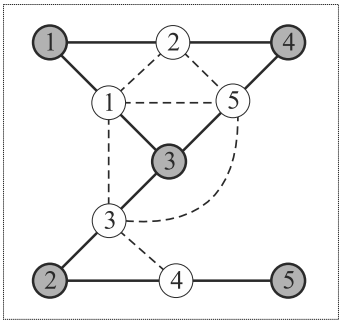
3. Graf G, którego wierzchołki oznaczone są literami `a, b, c, ...,l` opisany jest przez listy sąsiedztwa

sąsiedzi `a`: `b, c, d, e`	sąsiedzi `e`: `a, k, l, f`	sąsiedzi `i`: `g, j, h`
sąsiedzi `b`: `c, d, a`	sąsiedzi `f`: `e, g`	sąsiedzi `j`: `k, l, i`
sąsiedzi `c`: `a, b, d`	sąsiedzi `g`: `h, l, f, i`	sąsiedzi `k`: `e, j`
sąsiedzi `d`: `c, b, a`	sąsiedzi `h`: `g, i`	sąsiedzi `l`: `g, j, e`.



Narysuj drzewa generowane przez algorytmy DFS i BFS, gdy startujemy z wierzchołka `j`. W każdym przypadku podaj kolejność, w jakiej odwiedzane będą wierzchołki grafu (UWAGA: istotna jest kolejność na listach sąsiedztwa).

4. Graf *krawędziowy* $G^* = (V^*, E^*)$ (białe wierzchołki) grafu nieskierowanego $G = (V, E)$ (szare wierzchołki) określony jest następująco: wierzchołki $p \in V^*$ reprezentują krawędzie $e \in E$ oryginalnego grafu G, przy czym p i q połączone są krawędzią w G^* wtedy i tylko wtedy gdy odpowiadające im krawędzie e i f w G są incydentne (tj. mają wspólny koniec $v \in V$), por. rysunek. Napisz funkcję, która wyznaczy macierz sąsiedztwa $B_{(m\times m)}$ grafu G^* na podstawie danej macierzy sąsiedztwa $A_{(n\times n)}$ grafu G.



WSKAZÓWKA: Trzeba zacząć od wybrania jakiejś numeracji krawędzi e_j w G: np. najpierw numerujemy krawędzie wychodzące z wierzchołka 1 w kolejności wyznaczonej przez numery ich drugich końców. Dalej, w podobnej kolejności te wychodzące z 2, które nie otrzymały jeszcze numerów w poprzednim kroku itd. Numery te można wstępnie zapisywać wprost w macierzy A: `if (A[i,j]==1) A[i,j]=A[j,i]=++k;`. Mając tak ustaloną numerację krawędzi, można już określić wartości elementów `B[k,l]`.

5. Czy można zmodyfikować algorytm Dijsktry wyznaczania najkrótszej drogi w grafie tak, by znajdował najdłuższą drogę prostą (czyli nie zawierającą cykli) między zadanymi wierzchołkami? Odpowiedź powinna zawierać przekonywujące uzasadnienie.

Rozwiązania

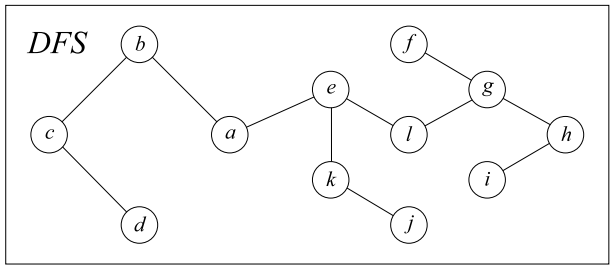
1. Tablica oryginalna (pierwszy wiersz) i po zamianie na kopiec (drugi wiersz):

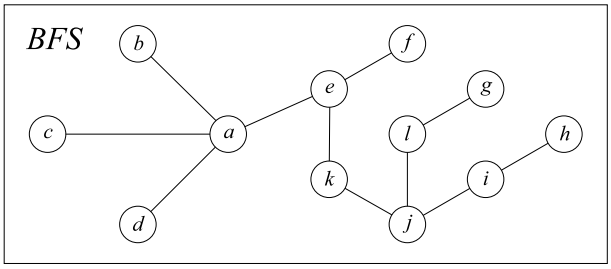
15	10	8	20	12	17	5	4	18	11	7	3	14	13	9	16	6	2	19	8	15	5
20	19	17	18	15	14	13	16	15	12	7	3	8	5	9	4	6	2	10	8	11	5

Tworzenie kopca zaczynamy od elementu $n/2$ przesiewając go w tablicy w prawo, w obrazie drzewa — od przedostatniego poziomu przesiewając w dół do potomków. Kolejno cofamy się do wcześniejszych elementów. Chodzi o to by podczas przetwarzania danego elementu struktury jego poddrzew były już kopcami. Błędem jest więc zaczynanie tego procesu od korzenia czyli od pierwszego elementu w tablicy.

2.
```
void merge_sort(wel *P){
   wel Q=NULL;
   if ((*P!=NULL) && (*P->nast!=NULL)) {
      podziel(P, &Q);
      merge_sort(P);
      merge_sort(&Q);
      połącz(P, &Q);
   }
}
```

3. Kolejność odwiedzania wierzchołków w DFS: `j, k, e, a, b, c, d, l, g, h, i, f`
Kolejność odwiedzania wierzchołków w BFS: `j, k, l, i, e, g, h, a, f, b, c, d`



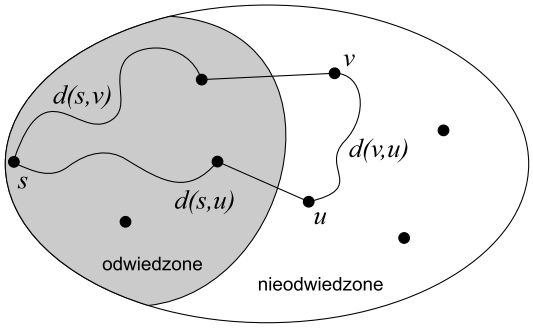


4. Dane: macierz sąsiedztwa `A[n][n]` grafu G oraz liczba wierzchołków `n` i liczba krawędzi `m`.

```
int B[m][m] = {0};
k = 0;
for (u=1; u<=n; u++)
   for (v=u+1; v<=n; v++)                    //  for (v=1; ...) to błąd, bo A[u][v] przetwarzane są
      if ( A[u][v]==1 ) A[u][v]=A[v][u]=++k;  //  wówczas 2-krotnie z różnymi wartościami k

for (u=1; u<=m; u++)
   for (v=1; v<=m; v++)
      if (A[u][v]!=0){
         for (t=v+1; t<=m; t++)
            if (A[u][t]!=0) E[A[u][v]][A[u][t]]=E[A[u][t]][A[u][v]]=1;
      }
```

5. W oryginalnym algorytmie Dijkstry do zbioru wierzchołków odwiedzonych (dla których znana jest już najkrótsza droga z wierzchołka startowego s) trafia w kolejnym kroku ten z nieodwiedzonych jeszcze wierzchołków u, który ma aktualnie najmniejszą odległość $d(s,u)$ od s wzdłuż ścieżki prowadzącej przez wierzchołki już odwiedzone. Fakt, że krawędzie mają z założenia nieujemne wagi gwarantuje, że krótszej ścieżki z s do u nie znajdziemy. Gdyby istniała, musiała by prowadzić przez co najmniej jeden z nieodwiedzonych jeszcze wierzchołków $v \neq u$, dla którego z założenia mamy $d(s,v) \geq d(s,u)$. Dodatkowo odcinek tej ścieżki z v do u musi mieć także nieujemną długość $d(v,u)$, stąd też cała ścieżka z s przez v do u byłaby dłuższa niż $d(s,u)$ wbrew przypuszczeniu.



Łatwo zauważyć, że wyznaczanie najdłuższej ścieżki nie może opierać się na podobnym rozumowaniu. Mając bowiem najdłuższą ścieżkę z s do u przez już odwiedzone wierzchołki, nie można wykluczyć istnienia dłuższej przez wierzchołki jeszcze nieodwiedzone. Przez symetrię z oryginalnym algorytmem potrzebne byłoby założenie o tym, że krawędzie powinny mieć ujemne długości. Nie ma więc dobrego kryterium wyboru kolejnego wierzchołka do odwiedzenia.
Oto bardziej zaawansowane wyjaśnienie (niektórzy z Państwa napisali o tym — brawo!). Algorytm Dijkstry jest przykładem metody opartej o strategię zachłanną (wybór kolejnego wierzchołka u o minimalnej wartości $d(s,u)$). Metody tego typu działają poprawnie o ile problem ma tzw. własność optymalnej podstruktury. W przypadku najkrótszych ścieżek oznacza to, że jeśli ścieżka s do t jest minimalna, wówczas każdy jej fragment, łączący pośrednie wierzchołki na niej, jest także minimalną ścieżką. Problem najdłuższych dróg prostych w grafie tej własności nie posiada. W istocie jest to problem algorytmicznie bardzo trudny. Nie znamy dla niego metody o wielomianowej złożoności.