informatyka+

Algorytmika i programowanie

Bazy danych

Multimedia, grafika i technologie internetowe

Sieci komputerowe

Tendencje w rozwoju informatyki i jej zastosowań

*Człowiek – najlepsza inwestycja*

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

Warszawska Wyższa Szkoła Informatyki

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.

informatyka+

# Kuźnia Talentów Informatycznych: Algorytmika i programowanie

## Zaawansowane algorytmy

*Bolesław Kulbabiński,*

*Tomasz Kulczyński, Błażej Osiński*

*Człowiek – najlepsza inwestycja*

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

Warszawska Wyższa Szkoła Informatyki

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.

# Zaawansowane algorytmy

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

Warszawska Wyższa Szkoła Informatyki

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego

**Rodzaj zajęć:** Kuźnia Talentów Informatycznych
**Tytuł:** Zaawansowane algorytmy
**Autor:** Bolesław Kulbabiński, Tomasz Kulczyński, Błażej Osiński

**Redaktor merytoryczny:** prof. dr hab. Maciej M Sysło

Zeszyt dydaktyczny opracowany w ramach projektu edukacyjnego **Informatyka+** — ponadregionalny program rozwijania kompetencji uczniów szkół ponadgimnazjalnych w zakresie technologii informacyjno-komunikacyjnych (ICT).
**www.informatykaplus.edu.pl**
**kontakt@informatykaplus.edu.pl**

**Wydawca:** Warszawska Wyższa Szkoła Informatyki
ul. Lewartowskiego 17, 00-169 Warszawa
**www.wwsi.edu.pl**
**rektorat@wwsi.edu.pl**

Projekt graficzny: FRYCZ I WICHA

Warszawa 2010
Copyright © Warszawska Wyższa Szkoła Informatyki 2010
Publikacja nie jest przeznaczona do sprzedaży.

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

WARSZAWSKA WYŻSZA SZKOŁA INFORMATYKI

UNIA EUROPEJSKA
EUROPEJSKI FUNDUSZ SPOŁECZNY

Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

WARSZAWSKA WYŻSZA SZKOŁA INFORMATYKI

UNIA EUROPEJSKA
EUROPEJSKI FUNDUSZ SPOŁECZNY

Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego

# Zaawansowane algorytmy

**Bolesław Kulbabiński, Tomasz Kulczyński, Błażej Osiński**
Uniwersytet Warszawski

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI
WARSZAWSKA WYŻSZA SZKOŁA INFORMATYKI
UNIA EUROPEJSKA
EUROPEJSKI FUNDUSZ SPOŁECZNY
Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego

## Streszczenie

Niniejszy kurs ma na celu pomoc w zgłębianiu najtrudniejszych zagadnień algorytmicznych, które pojawiają się na konkursach informatycznych. Przeznaczony jest dla uczniów, którzy opanowali już najważniejsze podstawowe techniki rozwiązywania problemów informatycznych. Szczególnie zalecane jest wcześniejsze zapoznanie się z kursami „Przegląd podstawowych algorytmów" oraz „Struktury danych i ich zastosowania".

Kurs jest zdecydowanie nastawiony na przygotowanie do konkursów. Jeden z rozdziałów zawiera nawet rady co do samej taktyki startowania w zawodach.

Spora część kursu zakłada znajomość języka C++, nie jest ona jednak wymagana do zrozumienia większości idei, które staramy się przekazać.

## Spis treści



KAPITAŁ LUDZKI NARODOWA STRATEGIA SPÓJNOŚCI
Warszawska Wyższa Szkoła Informatyki
UNIA EUROPEJSKA EUROPEJSKI FUNDUSZ SPOŁECZNY
Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI
Warszawska Wyższa Szkoła Informatyki
UNIA EUROPEJSKA
EUROPEJSKI FUNDUSZ SPOŁECZNY
Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego

# 1 Taktyka

Jako że kurs jest prowadzony pod kątem rozwiązywania zadań konkursowych, chcielibyśmy przedstawić kilka naszych przemyśleń co do samego startowania w zawodach informatycznych.

Oczywiście, nawet najlepsza taktyka nie pomoże, jeśli nie potrafimy zrobić zadań, które są do rozwiązania. Często jednak, dobre podejście do zawodów może znacznie poprawić nasz wynik.

Rozdział będzie składał się ze zbioru dobrych rad dotyczących różnych rodzajów konkursów.

## 1.1 Przed zawodami

Ważnym elementem, o którym często się zapomina, jest przygotowanie jeszcze przed dniem konkursu. Warto spojrzeć na poprzednie edycje, zadania, wyniki. Dostarcza to cennych informacji, czego można się spodziewać na samych zawodach. Upewnijmy się, że wiemy ile będzie zadań, jak oceniane, jak jest tworzony ranking. Nie ma czasu, by się nad tym zastanawiać później, podczas trwania rywalizacji. Bardzo ważna jest też ustalenie, czego musimy dokonać, żeby być z siebie zadowolonym. Chcemy rozwiązać jak najszybciej wszystkie zadania, a może spokojnie i powoli, byle się nie pomylić? A może w ogóle nie chcemy ryzykować i zakładamy od razu, że lepiej zrobić jedno zadanie, ale na pewno dobrze, niż dwa niepewnie. Na te pytania nie ma uniwersalnej odpowiedzi. Zapamiętajmy przede wszystkim następującą radę:

> Ustalmy sobie wyraźne cele jeszcze przed zawodami. Nie dajmy się niczym zaskoczyć.

## 1.2 Wybór zadania

Pierwszy problem, jaki napotykamy po rozpoczęciu zawodów, to dobry wybór zadania, którym się zajmiemy. Tutaj strategia zależy prawdopodobnie od charakteru konkursu. Jeśli są to krótkie zawody, w których liczy się czas, może opłacać się czytać najpierw zadania o najkrótszej treści i jeśli wydają się być rozwiązywalne, starać się je zrobić. Jeśli natomiast zawody są dłuższe, a czas nie gra większej roli, to polecamy na początku przeczytać wszystkie zadania, żeby mieć dobry obraz całości i lepiej zadecydować, czym się po kolei zająć. Warto wziąć pod uwagę pierwsze wrażenie co do trudności zadania, jak również własne preferencje tematyczne („geometria nie", „teoria liczb tak sobie", itp.).

> Niezależnie od formy konkursu, w zdecydowanej większości przypadków warto przeczytać wszystkie zadania, ale nie koniecznie na początku zawodów.

Kluczowym elementem strategii, na którego braku najłatwiej traci się punkty czy pozycję w tego typu konkursach, jest tzw. „moment odpuszczenia". Otóż, jeśli z jakichś powodów nie możemy rozwiązać zadania, którym się zajmujemy, należy je zostawić i przejść do następnego. Naturalnie nie chodzi tutaj o poddawanie się przy pierwszej napotkanej trudności. I właśnie dlatego musimy sobie określić, kiedy decydujemy się na zmianę zadania. Prawdopodobnie będzie to czas w minutach spędzony nad danym problemem. Trzeba przy tym pamiętać, że inaczej traktujemy problemy implementacyjne (z nimi zazwyczaj opłaca się walczyć do końca).

> Jeśli nie potrafimy znaleźć dobrego rozwiązania zadania po pewnym, ustalonym z góry czasie, zostawiamy je i próbujemy rozwiązać następne.

## 1.3 Ponowne czytanie

Bardzo wiele błędów wynika ze złego zrozumienia treści zadania. Jeśli tylko specyfika zawodów na to pozwala (np. nie są bardzo krótkie), gorąco polecamy następującą metodę:

> Przeczytajmy **całą** treść zadania jeszcze raz **po** jego rozwiązaniu. Zróbmy to także, jeśli od dłuższego czasu nie możemy wymyślić dobrego rozwiązania.

Po chwili zastanawiania się nad zadaniem (obojętne, czy z sukcesem, czy nie) możemy nabrać nowego spojrzenia, albo po prostu lepiej zrozumieć stawiany przed nami problem. Być może właśnie wtedy ponowne przeczytanie treści da nam poprawny obraz tego, czego się od nas oczekuje.

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI
Warszawska Wyższa Szkoła Informatyki
UNIA EUROPEJSKA
EUROPEJSKI FUNDUSZ SPOŁECZNY
Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego

## 1.4 Testowanie

Jak powszechnie wiadomo, tworzenie programów, które będą od razu działały graniczy z cudem. Bardzo często popełniamy błędy, szczególnie w fazie kodowania rozwiązania. Nie powinno nas to zniechęcać, trzeba tylko umieć sobie z tym problemem radzić.

Jedną z metod jest (czasem wielokrotne) czytanie kodu po jego napisaniu. Niestety, podczas konkursu, gdzie jesteśmy zdani tylko na siebie, jest to zawodna metoda, gdyż ciężko krytycznie spojrzeć na swoje dzieło i znaleźć w nim usterkę.

Dużo lepszym sposobem jest testowanie programu lub jego fragmentów. Podajmy naszemu programowi konkretne dane i sprawdźmy, czy oblicza poprawny wynik. Warto się przy tym zastanowić i wybrać dane podchwytliwe czy złośliwe dla naszego algorytmu. Ważne jest, aby samemu obliczyć wynik, najlepiej inną metodą niż ta zaimplementowana w programie i porównać go z wynikiem programu.

Zrobienie kilku testów jest najpewniejszą metodą sprawdzenia kodu. **Uwaga!** Testy dostarczane wraz z treścią zadania są na bardzo wielu konkursach bardzo proste i nie należy wnioskować, że program działający na nich jest poprawny.

## 1.5 „Bruty"

Na zawodach w których mamy odpowiednio dużo czasu na takie zabiegi, bardzo dobre efekty przynosi pisanie „brutów" — programów, które działają szybko tylko dla bardzo małych testów, ale są proste zarówno pod względem samej idei rozwiązania, jak i napisanego kodu.

Mając takiego „bruta" możemy łatwo sprawdzać nasz program na wielu testach. Nie musimy się bowiem martwić o liczenie odpowiedzi na kartce, wystarczy spytać „bruta" o dobrą odpowiedź. W bardzo wielu zadaniach możemy wręcz losować testy automatycznie (napisać kolejny prosty programik do takiego losowania) i sprawdzać, czy nasze rozwiązanie i „brut" dają jednakowe wyniki. Jeśli nie, to jest to sygnał do szukania błędów. Taka metoda szukania błędów przynosi stosunkowo najpewniejsze rezultaty.

„Bruta" możemy też wykorzystać w inny sposób. Jeśli nie jesteśmy pewni w stu procentach naszego rozwiązania i nie mamy czasu go już poprawić, sprawdzić, itp., możemy zastosować metodę łączenia dwóch programów w jeden. Po wczytaniu danych nasz nowy program może zadecydować, czy test jest bardzo mały i w związku z tym bezpieczniej rozwiązywać go „brutem", czy też jest na tyle duży, że „brut" nie da rady i trzeba zaryzykować i posłużyć się niepewnym ale szybszym rozwiązaniem. Takie wyjście może spowodować przyznanie większej liczby punktów za dane zadanie na konkursach, gdzie otrzymuje się punkty za każdy poprawnie rozwiązany test. Rzadko jednak spowoduje poprawę wyniku na zawodach, w których trzeba mieć całe zadanie dobrze rozwiązane, aby uzyskać jakiekolwiek punkty.

Podczas trwania tego kursu będziesz miał okazję kilkakrotnie przetestować te rady w praktyce i wypracować swoją własną taktykę na startowanie w zawodach.

# 2 Standard Template Library

Prawdopodobnie przy tej czy innej okazji spotkałeś się już z biblioteką STL, a pewnie także wykorzystywałeś ją w swoich programach. Zawiera ona wiele gotowych do użycia implementacji struktur danych oraz algorytmów.

Niestety ta biblioteka często jest nadużywana. Na przykład, ktoś używa `multimap` tam, gdzie w zupełności starczyłaby zwykła tablica. Dlatego też biblioteką STL zajmujemy się szerzej dopiero w kursie dla zaawansowanych, gdy wiecie już, co można zrobić prostymi metodami.

Intencją tego rozdziału jest usystematyzowanie wiedzy o tej bibliotece i poznanie sposobów dobrego jej wykorzystywania w rozwiązaniach zadań algorytmicznych. W kilku miejscach pokażemy też stosowane przez wielu zawodników sztuczki (np. dyrektywy preprocesora), dzięki którym możemy pisać trochę mniej kodu.

Oczywiście nikt nie powinien się uczyć na pamięć różnych metod z biblioteki STL. Na większości konkursów (np. na Olimpiadzie Informatycznej) dostępna jest dla zawodników dokumentacja biblioteki podobna do tej zamieszonej na stronie `http://www.sgi.com/tech/stl/`. Dlatego też warto się z nią zapoznać i nauczyć się z niej korzystać.

## 2.1 Kontener vector

Przegląd możliwości biblioteki STL zaczniemy od chyba najbardziej popularnego kontenera: `vector`, który łączy w sobie cechy stosu i tablicy. Poświęcimy mu dość dużo czasu, gdyż większość jego cech odnosi się też do innych kontenerów.

Aby móc go wykorzystać w programie zwykle piszemy:

```
#include<vector>
using namespace std;
```

Co powoduje drugi wiersz? Jest to wprowadzenie *przestrzeni nazw* `std`: „pełna nazwa" kontenera `vector` to `std::vector`. Zatem aby zaoszczędzić sobie pisania tego `std::` przed każdym wystąpieniem czegoś z przestrzeni nazw biblioteki standardowej polecamy kompilatorowi, by robił to za nas. W praktyce zawodowego programisty nie zawsze jest to dobre rozwiązanie, natomiast na konkursach informatycznych jest bardzo popularne.

Bardzo ważną cechą kontenerów z biblioteki STL jest to, że możemy ich użyć do przechowywania obiektów dowolnego typu. Stosuje się w tym celu zaawansowany mechanizm języka C++ jakim są szablony. Na przykład napisanie `vector<int> v;` powoduje utworzenie przez kompilator specjalnej wersji kontenera `vector`, przystosowanej do przechowywania zmiennych całkowitoliczbowych. Jeżeli zdefiniujemy klasę `MojaKlasa` to nic nie stoi na przeszkodzie, by zadeklarować `vector<MojaKlasa> vmk`. Możemy nawet zadeklarować `vector<vector<MojaKlasa> > vvmk` — jest to `vector`, który będzie przechowywał kontenery `vector` zmiennych typu `MojaKlasa`. **Uwaga:** spacja między dwoma znakami > > jest istotna — inaczej kompilator myliłby je z operatorem przesunięcia bitowego.

Przyjrzyjmy się teraz kilku ważniejszym metodom oferowanym przez obiekty `vector<T> v`:

- `void v.push_back(a)` — dodawanie elementu `a` na końcu `v`. Jest to na tyle często używana metoda, że wiele osób na początku programu używa dyrektywy: `#define PB push_back`, dzięki której może pisać tylko `v.PB(a)`.
- `void v.pop_back()` — usunięcie ostatniego elementu.
  **Uwaga:** Ze względów wydajnościowych metoda ta jest typu `void`, czyli nic nie zwraca. Jeżeli chcielibyśmy zapamiętać zdejmowany element, to możemy przed jego zdjęciem skorzystać z metody `v.back()`, która zwróci referencję do ostatniego elementu.
- `v[i]` to odwołanie do $i$-tego elementu `v`, zupełnie tak samo jak dla tablicy. Operator ten działa w czasie stałym.

Metody `push_back` i `pop_back` działają w czasie zamortyzowanym stałym. Warto wiedzieć, w jaki sposób uzyskuje się taką wydajność. Otóż `vector` pamięta wskaźnik na tablicę elementów odpowiedniego typu. Początkowo jest ona mała, ma na przykład $5$ elementów. Kiedy po kilkukrotnym uruchomieniu `push_back` brakuje w niej miejsca na nowy element, wówczas alokowana jest nowa, dwa razy większa tablica, a zawartość starej jest doń kopiowana.

**Ćwiczenie 1.** Wrzucamy do początkowo pustego kontenera `vector` pewną liczbę elementów. Zakładając, że jest to realizowane tak, jak opisano wyżej udowodnij, że dodanie pojedynczego elementu zajmuje zamortyzowany czas stały.

Aby przejrzeć wszystkie elementy kontenera `vector` można napisać pętlę:

```
for (int i = 0; i < (int) v.size(); i++) { ... }
```

`v.size()` zwraca wynik typu `unsigned int` dlatego rzutowanie na `int` jest istotne, jeżeli nie chcemy, by kompilator męczył nas ostrzeżeniami. Można by też zadeklarować licznik jako zmienną typu `unsigned`

Powiedzmy jeszcze o ważnej metodzie `swap` dostępnej dla kontenerów biblioteki STL. Jak nie trudno zgadnąć, zamienia ona ze sobą kontenery przechowywane w dwóch zmiennych. Co ciekawe, jest to wykonywane w czasie stałym niezależnie od wielkości struktury (tak na prawdę podmieniane są tylko pewne wskaźniki).

Kontenerem podobnym do `vector`, który czasami się przydaje, jest `deque` — połączenie kolejki i tablicy. Do operacji wykonywanych na kontenerze `vector` w czasie zamortyzowanym stałym dodaje jeszcze `push_front` i `pop_front`, czyli odpowiednio dołożenie i zdjęcie elementu z początku. Niestety w praktyce ta struktura nie jest tak wydajna jak `vector`, dlatego też należy jej używać tylko jeżeli jest to absolutnie konieczne.

## 2.2 Iteratory

Bardzo ważnym i często używanym narzędziem biblioteki STL są *iteratory*. Możemy na nie patrzeć jako na specjalne wskaźniki, które potrafią przechodzić po elementach kontenerów. Składnia jest zresztą taka sama jak w przypadku wskaźników: `*it` oznacza element, na który wskazuje iterator `it`, `++it` służy do zwiększenie iteratora, czyli przejście do kolejnego elementu kontenera, a `--it` do cofania się[1]. Kolejność, w jakiej iterator przegląda elementy kontenera zależy od rodzaju kontenera.

Aby zadeklarować iterator dla odpowiedniego typu kontenera piszemy `kontener::iterator it`, (np. `vector<int>::iterator it`). Zachowanie iteratorów dla różnych kontenerów może się mocno od siebie różnić, stąd konieczność wykorzystania iteratora z odpowiedniej przestrzeni nazw (jak np. `vector<int>::`).

### Przeglądanie elementów

Jeżeli chcielibyśmy „przeiterować" wszystkie elementy kontenera przydadzą się nam funkcje `begin()` i `end()`, które zwracają pierwszy i ostatni iterator w kontenerze. **Uwaga:** Ostatni iterator nie wskazuje na żaden obiekt przechowywany w kontenerze, dlatego nigdy nie należy się odwoływać do `*(v.end())`.

Możemy teraz w inny sposób napisać pętlę przeglądającą elementy `v`:

```
for (vector<int>::iterator it = v.begin(); it != v.end(); ++it) { ... }
```

Pętle tego typu wykorzystuje się często, a niestety wymagają one od nas dość dużo pisania. Aby zaoszczędzić sobie czas można próbować napisać makro, które dostając tylko nazwę kontenera i iteratora tworzyłoby odpowiednią pętlę. Problemem jest to, że na podstawie kontenera zwykle nie można uzyskać typu potrzebnego iteratora. Na szczęście w kompilatorze `gcc`, który jest wykorzystywany na większości zawodów programistycznych, udostępniona jest funkcja `__typeof()`, dzięki której możemy napisać następujące makro:

```
#define FOREACH(i,t) for(__typeof((t).begin()) i=(t).begin(); i!=(t).end(); ++i)
```

Później w programie, jeżeli chcemy pętlę przeglądającą wszystkie elementy kontenera `v`, wystarczy napisać: `FOREACH(i,v)`.

### Iteratory w metodach

Iteratory są często argumentami i wynikami różnych metod. Przyjrzymy się takiej sytuacji na przykładzie kontenera `list`, czyli listy dwukierunkowej. Jej główną przewagą w stosunku do `vector` czy też `deque` jest to, że operacje wstawiania i usuwania pojedynczego elementu ze środka działają w czasie stałym (do jej elementów nie możemy jednak odwoływać się jak do tablicy).

Do wstawiania służy metoda `l.insert(it, x)`, która umieszcza element o wartości `x` w liście `l` przed iteratorem `it`. Zwracana wartość to iterator do właśnie wstawionego elementu.

[1] Istnieją też wersje operatorów ++ oraz -- stawiane za argumentem, ale są one mniej wydajne.

W kontenerze `list` dostępna jest metoda `size`. Należy jej używać z rozwagą, gdyż nie jest ustalone, w jakim czasie ona działa. Może się okazać, że obliczenie rozmiaru listy będzie wymagało przejrzenia wszystkich jej elementów, co będzie bardzo kosztowne. Dlatego gdzie tylko można należy raczej używać metody `l.empty()`, która sprawdza czy lista `l` jest pusta (zamiast sprawdzać warunek `l.size() > 0`).

## 2.3 Pary

Przy pisaniu programów bardzo często okazuje się, że chcemy przechowywać informacje zawarte w kilku zmiennych w jednym miejscu. Najbardziej naturalnym rozwiązaniem wydaje się zdefiniowanie klasy, w której będą ukryte owe zmienne. Często jednak wygodniej jest użyć par.

- Deklaracja pary złożonej ze zmiennych typów `T1` i `T2`: `pair<T1, T2> p`.
- Pierwszy i drugi element pary: `p.first` i `p.second`.
- Utworzenie nowej pary: `make_pair(a,b)`

Pary są bardzo wygodnie, między innymi dlatego, że można je ze sobą porównywać za pomocą standardowych operatorów. Operator < wyznacza porządek leksykograficzny na parach: to znaczy, że najpierw porównujemy pierwsze pozycje z pary. Jeżeli są różne, to mamy już wyznaczony porządek dwóch par. Jeżeli są równe, to porównujmy drugie pozycje par.

Pary okazują się na tyle wygodne, że niekiedy używa się ich do przechowywania więcej niż dwóch zmiennych. Na przykład $4$ obiekty można zgrabnie „upakować" do pary złożonej z dwóch par.

Aby sprawniej korzystać z par często stosuje się dyrektywy `define`. Zwykle używa się skrótu `MP` zamiast `make_pair`.
Dwa popularne sposoby skracania `first`, `second` to `F`, `S` oraz `ST`, `ND`.

## 2.4 Kontener set

Kontener `set` służy do przechowywania informacji o zbiorach uporządkowanych elementów. Elementy w zbiorze nie mogą się powtarzać, co jest zgodne z matematycznym pojęciem zbioru. Kontener ten jest zaimplementowany jako drzewo czerwono-czarne (jedno ze zrównoważonych drzew poszukiwań). Dzięki temu, poniższe operacje wstawiania, usuwania i szukania elementu mają złożoność logarytmiczną.

- `s.insert(x)` próbuje wstawić element `x` do zbioru `s`. Zwraca parę złożoną z iteratora wskazującego na element równy `x` oraz wartości logicznej mówiącej, czy mu się udało. Wstawienie do zbioru nie udaje się, gdy istnieje już w nim element równy `x`.
- `s.erase(it)` usuwa element wskazywany przez iterator `it`. Bardzo ważną cechą tej funkcji jest to, że wszystkie iteratory nie wskazujące na usuwany element pozostają poprawne, dalej można się nimi posługiwać.
  Element o wartości `x` można usuwać za pomocą funkcji: `s.erase(x)`.
- `s.find(x)` służy do znajdowania elementu w zbiorze, zwraca iterator do elementu o wartości `x` (jeżeli taki iterator nie istnieje to zwraca jedyny iterator, który na nic nie wskazuje, czyli `s.end()`).
  Jeżeli chcemy tylko przekonać się czy dany element występuje w zbiorze możemy sprawdzić warunek `s.count(x) > 0`.

Podobnym kontenerem jest `multiset`, który jednakże pozwala na przechowywanie kilku równych elementów.

## Uporządkowane elementy w zbiorze

Elementy przechowywane w kontenerach `set` i `mutliset` są uporządkowane (zwykle zgodnie z porządkiem wyznaczanym przez operator <). Dwa przydatne fakty z tego wynikające:

- `s.begin()` to iterator do najmniejszego elementu
- Przeglądając kolejne elementy za pomocą iteratora (na przykład za pomocą makra `FOREACH`) będziemy widzieli elementy w kolejności niemalejącej (a w zbiorze nawet ściśle rosnącej).

**Ćwiczenie 2.** W algorytmie Dijkstry potrzebny jest kopiec przechowujący dla każdego wierzchołka jego odległość od wierzchołka startowego i obsługujący następujące operacje:

- zmień wartość przechowywaną dla pewnego wierzchołka (to może wymagać dodania tego wierzchołka, jeżeli wcześniej nie było go w kopcu)
- znajdź jeden z wierzchołków o najmniejszej odległości, zwróć tę odległość i usuń go z kopca.

Jak można prosto zrealizować powyższe operacje przy użyciu zbioru z biblioteki STL? Proponujemy wykorzystać `set<pair<int, int> >`.

W omawianych kontenerach są jeszcze dwie ważne metody:

- `s.lower_bound(x)` (ang. *dolne ograniczenie*) zwraca iterator do pierwszego elementu, którego wartość jest nie mniejsza niż `x`.
- `s.upper_bound(x)` (ang. *górne ograniczenie*) zwraca iterator do pierwszego elementu, którego wartość jest większa niż `x`.

W obu przypadkach gdy nie istnieje taki element zwracany jest iterator `s.end()`.

Para funkcji o tych samych nazwach wystąpi jeszcze przy omawianiu algorytmów z biblioteki STL, dlatego dobrze jest zrozumieć, dlaczego akurat takie są wartości przez nie zwracane. Pomoże w tym poniższe ćwiczenie:

**Ćwiczenie 3.** Masz dany `multiset<int> ms`. Napisz pętlę, która przechodzi po iteratorach, którym odpowiadają wartości równe $1729$.

## Definiowanie własnego porządku

Czasami chcemy by porządek elementów w zbiorze był inny niż standardowy. W tym celu powinniśmy zadeklarować specjalną klasę, która będzie miała tylko jedną metodę (a dokładniej operator `()`), porównującą dwa obiekty określonego typu. Metoda ta musi zachowywać się tak, jak <, a więc zwracać wartość `true` wtedy i tylko wtedy, gdy pierwszy argument jest mniejszy od drugiego.

Poniżej prezentujemy przykładową klasę, która umożliwia porównywanie elementów typu `vector` względem ich długości:

```
struct por_vec {
    bool operator() (const vector<int>& a, const vector<int>& b){
        return a.size() < b.size();
    }
};
```

Zwróćmy uwagę na jeden z argumentów operatora: `const vector<int>& a`. Argumentem jest referencja do obiektu `a`, dzięki czemu unikamy kopiowania. Co więcej, jest ona stała, gdyż chcemy upewnić się, że porównywanie dwóch elementów nie zmieni żadnego z nich.

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI
Warszawska Wyższa Szkoła Informatyki
UNIA EUROPEJSKA
EUROPEJSKI FUNDUSZ SPOŁECZNY
Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego

Teraz możemy zadeklarować set złożony z vector<int> uporządkowanych po liczbie elementów:

```
set<vector<int>, por_vec> zb;
```

Własną klasę porównującą możemy też wykorzystać, gdy chcemy przechowywać w zbiorze elementy, które nie mają zdefiniowanego operatora <.

## 2.5 map

map<T1, T2> to *słownik* przechowujący „tłumaczenia" obiektów typu T1 na te typu T2. Obiekty typu T1 nazywamy *kluczami*, a typu T2 — *wartościami*. W słowniku do jednego klucza może być przyporządkowana tylko jedna wartość.

Przykładem słownika może być map<string, int> kalendarz przyporządkowujący nazwie miesiąca jego numer (np. „czerwiec" — $6$). Wygodna składnia pozwala nam korzystać ze słownika jak z tablicy np.: kalendarz["marzec"] = 3;.

Poza tym map<T1, T2> m przypomina pod względem używania set<pair<T1, T2> >. Iterator wskazuje na parę klucza i odpowiadającej mu wartości. Wstawianie elementu może się odbywać za pomocą funkcji m.insert(make_pair(t1,t2)) (lub też za pomocą składni tablicowej). Wyszukiwanie i usuwanie są analogiczne jak w przypadku zbioru z tą różnicą, że jako argument podaje się tylko klucz.

Jeżeli za pomocą składni tablicowej odwołamy się do klucza $k$, który nie istnieje w map<T1,T2> to zostanie wstawiona para make_pair(k, T2()).
T2() to bezparametrowy konstruktor obiektów T2, który zwykle reprezentuje domyślną wartość (na przykład $0$, vector pusty).

## 2.6 Sortowanie i wyszukiwanie binarne

Zaczynając od sortowania, zajmiemy się teraz ciekawszymi algorytmami dostępnymi w bibliotece STL. Argumentami funkcji sort(p, k) są dwa iteratory o dostępie swobodnym (np. vector<T>::iterator, ale też int* dzięki czemu możemy stosować tę funkcję do zwykłej tablicy). Ustawia ona w kolejności niemalejącej elementy od tego wskazywanego przez p do ostatniego przed k. Zatem aby posortować vector<int> v należy napisać: sort(v.begin(), v.end()).

Jeżeli chcemy sortować elementy, dla których nie ma zdefiniowanego operatora <, lub też chcemy je ułożyć w innej kolejności to należy dodać funkcję porównującą. Poniżej przykład funkcji do porównywania elementów typu vector ze względu na ich długość. Zwróć uwagę na jej podobieństwo do operatora porównującego wykorzystywanego przez kontener set.

```
bool fpor_vec (const vector<int>& a, const vector<int>& b) {
    return a.size() < b.size();
}
```

Aby skorzystać z tej funkcji należy podać ją jako trzeci argument funkcji sort. Kontynuując nasz przykład, żeby posortować tablicę vector<int> t[10] napiszemy sort(t,t+10,fpor_vec).

Czasami chcemy, żeby po sortowaniu elementy, które są równe zgodnie z obowiązującym sposobem porównywania, pozostały względem siebie w niezmienionym porządku. Na przykład gdy w powyższej tablicy t były dwa kontenery vector: a i b o trzech elementach to jeżeli początkowo a był przed b, to powinno tak zostać po posortowaniu całej tablicy. Sortowanie takie nazywamy *stabilnym*. Jest ono zaimplementowane w bibliotece STL w funkcji stable_sort.

**Ćwiczenie 4.** Jaki będzie wynik sortowania z wykorzystaniem poniższej funkcji porównującej?
bool por(const int & a, const int & b) { return a >= b; }
Czy na pewno jest ona dobrze napisana?

### Wyszukiwanie binarne

W posortowanej tablicy (lub kontenerze `vector`) możemy oczywiście wyszukiwać binarnie. Do szukania wartości `x` w przedziale iteratorów (z dostępem swobodnym) od `p` do `k` służą funkcje:

- `binary_search(p, k, x)` — zwraca tylko wartość logiczną oznaczającą, czy `x` występuje w zadanym przedziale
- `lower_bound(p, k, x)` i `upper_bound(p, k, x)` — podobnie jak w przypadku kontenera `set` zwracają one iterator do pierwszego elementu odpowiednio nie mniejszego niż `x` i większego niż `x` (lub też `k` jeżeli taki element nie istnieje).

## 2.7 Permutacje

W rozwiązaniach o dużej złożoności czasowej, które sprawdzają wszystkie możliwości, zdarza się konieczność generowania kolejnych permutacji zbioru. Okazuje się, że także do tego istnieje w bibliotece STL gotowa funkcja: `bool next_permutation(p, k)`. Wywołanie jej na przedziale iteratorów o dostępie swobodnym powoduje zmienienie permutacji tworzonej przez elementy `*p`, `*(p+1)`, ..., `*(k-1)` na następną w kolejności leksykograficznej i zwróceniu `true`. Jeżeli taka permutacja nie istnieje (tzn. aktualna jest największą w kolejności leksykograficznej) to zwracana jest wartość `false`, a permutacja jest zmieniana na pierwszą w kolejności leksykograficznej. Pojedyncze wywołanie tej funkcji zajmuje co najwyżej czas liniowy w stosunku do długości permutowanego przedziału.

Poniższy kod powoduje wypisanie wszystkich permutacji zbioru $1, 2, \ldots, n$:

```
vector<int> v(n); // konstruowany jest vector o n elementach
for (int i = 0; i < n; i++)
    v[i] = i+1;
do {
    for (int i = 0; i < n; i++)
        printf("%d ", v[i]);
    printf("\n");
} while (next_permutation(v.begin(), v.end()));
```

Istnieje też funkcja `prev_permutation`, która zmienia permutację na poprzednią w porządku leksykograficznym.

# 3 Haszowanie

## 3.1 Czym jest haszowanie?

O haszowaniu można myśleć jak o nadawaniu nowych nazw obiektom z pewnego ustalonego zbioru. Celem haszowania jest najczęściej zmniejszenie puli możliwych nazw (**uniwersum**) do odpowiednio małego rozmiaru. Przykładowo, jeśli rozważymy zbiór ludzi mieszkających w Polsce i będziemy identyfikować ich po imieniu i nazwisku to możliwych par <imię, nazwisko> może być praktycznie dowolnie wiele. Jeśli jednak zechcemy uprościć sobie sytuację i identyfikować ludzi po ich inicjałach to wszystkich możliwych inicjałów jest już tylko $32^2 = 1024$ (przy założeniu, że alfabet liczy $32$ litery). Inicjały są w tym przypadku przykładem **hasza** (inaczej **identyfikatora** lub **kodu**), czyli właśnie nowej nazwy obiektu. Innym, mniej naturalnym przykładem haszowania może być identyfikowanie ludzi po sumie kodów ASCII liter w ich imieniu i nazwisku modulo $1000$. W tym wypadku możliwych identyfikatorów jest tyle co reszt, czyli $1000$.

Kiedy zmniejszenie uniwersum nazw może być korzystne? Wyobraźmy sobie, że chcemy przechowywać dla każdego człowieka z pewnego zbioru jakieś dane, na przykład datę urodzenia. Jeśli nawet przyjmiemy, że imiona i nazwiska mają ograniczone długości to wciąż nie będziemy w stanie zaalokować osobnej komórki pamięci dla każdej możliwej kombinacji <imię,nazwisko>. Z drugiej strony,

KAPITAŁ LUDZKI NARODOWA STRATEGIA SPÓJNOŚCI
Warszawska Wyższa Szkoła Informatyki
UNIA EUROPEJSKA EUROPEJSKI FUNDUSZ SPOŁECZNY
Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego

przechowując zbiór ludzi w słowniku będącym drzewem zrównoważonym, skazujemy się na złożoność co najmniej liniowo-logarytmiczną ze względu na wielkość zbioru. Ponadto, porównywanie nazwisk może okazać się bardzo kosztowne w przypadku gdy będą one długie.

Korzystając z haszowania, moglibyśmy utworzyć tablicę o rozmiarze odpowiadającym rozmiarowi uniwersum haszy, po czym dla każdego człowieka ze zbioru przechowywać jego dane w tej tablicy pod indeksem, takim jak hasz z imienia i nazwiska. Odczyt i zapis danych mógłby być zrealizowany w czasie stałym.

Oczywistym problemem w powyższym rozwiązaniu są tzw. **kolizje**, czyli przypadki, w których dwie, różnie nazywające się osoby otrzymają takie same hasze. W takiej sytuacji nasz algorytm będzie przechowywał dane dwóch różnych osób w tym samym miejscu pamięci, co na pewno doprowadzi do błędu. Istnieje wiele różnych metod rozwiązywania problemu kolizji, jedną z nich prezentujemy poniżej.

## 3.2 Tablica haszująca

Tablica haszująca rozmiaru $M$ składa się z tablicy (początkowo pustych) list jednokierunkowych, przechowujących obiekty z $U$ oraz *funkcji haszującej* $h\colon U \to \{0, 1, \ldots, M-1\}$. Funkcja ta, wcześniej nieformalnie nazywana „zmianą nazwy", generuje dla obiektów ze zbioru $U$ hasze (kody) będące liczbami naturalnymi z przedziału $[0, M-1]$.

Zależy nam na tym, aby hasze należały właśnie do takiego przedziału gdyż odpowiadają one wtedy komórkom tablicy.

Tablica haszująca $T$ udostępnia następujące operacje:

```
void insert(x) {
    wstaw obiekt x na początek listy T[h(x)];
}

void delete(x) {
    usuń obiekt x z listy T[h(x)];
}

bool find(x) {
    if(obiekt x jest na liście T[h(x)])
        return true;
    else
        return false;
}
```

Dzięki temu, że z każdym haszem jest związane nie jedno pole tablicy a lista, kolizje haszy nie powodują utraty danych. Jeśli mamy pecha i okaże się, że wszystkie obiekty umieszczone w tablicy mają taki sam hasz, złożoność powyższych operacji będzie liniowa względem ilości przechowywanych elementów. Jednakże przy dobrym wyborze funkcji haszującej oraz założeniu, że liczba elementów w tablicy jest rzędu $O(M)$ możemy założyć, że średni czas operacji na tablicy haszującej jest stały.

Rozwiązywanie problemu kolizji haszy za pomocą list nosi nazwę **metody łańcuchowej**. Istnieje wiele innych, bardziej efektywnych metod radzenia sobie z kolizjami. Można o nich przeczytać na przykład w [2].

**Ćwiczenie 5.** Zaproponuj prostą funkcję haszującą, przypisującą ciągom znaków liczby całkowite z przedziału $[0 \ldots 10000]$. Zaimplementuj tablicę haszującą, która wykorzystuje Twoją funkcję. Porównaj szybkość jej działania na losowych danych testowych z drzewem zrównoważonym (np. kontenerem *set* z STL).

## 3.3 Haszowanie słów

Technika haszowania znajduje zastosowanie w wielu algorytmach tekstowych. Ustalmy, dla uproszczenia, że rozważamy słowa złożone z małych liter alfabetu angielskiego oraz, że kolejne litery alfabetu utożsamiamy z kolejnymi liczbami naturalnymi (licząc od zera), to znaczy **a** = 0, **b** = 1, ..., **z** = 25. Hasze dla słów będziemy obliczać w następujący sposób:

$$h(a_0a_1\ldots a_n) = (a_0 + a_1x + a_2x^2 + \ldots + a_nx^n) \mod p$$

gdzie $p$ jest dużą liczbą pierwszą a $x$ dowolną ustaloną liczbą całkowitą większą od rozmiaru alfabetu, ale mniejszą od $p$. Wartości tej funkcji haszującej możemy łatwo obliczać dla kolejnych, coraz to dłuższych, sufiksów słowa, stosując zależność:

$$h(a_ia_{i+1}\ldots a_n) = (a_i + x\,h(a_{i+1}a_{i+2}\ldots a_n)) \mod p$$

### Wyszukiwanie wzorca w tekście

Zastanówmy się jak za pomocą powyższej metody haszowania zrealizować wyszukiwanie wzorca w tekście. Przyjmijmy następujące oznaczenia:

- $m$-literowy wzorzec to $W = w_1w_2\ldots w_m$
- $n$-literowy tekst to $T = t_1t_2\ldots t_n$
- hasz podsłowa $t_i\ldots t_{i+m-1}$ (długości $m$) oznaczmy przez $h_i$

Potrzebujemy szybko obliczyć hasze dla wszystkich $m$-literowych podsłów tekstu i porównać je z haszem wzorca. Mając obliczony hasz $h_{i+1}$ dla podsłowa $t_{i+1}\ldots t_{i+m}$ możemy w czasie stałym znaleźć hasz podsłowa zaczynającego się na pozycji $t_i$ za pomocą łatwego wzoru:

$$h_i = (t_i + x\,h_{i+1} - t_{i+m}x^m) \mod p$$

Warto na początku stablicować sobie wartości potęg $x$ modulo $p$.

Mając obliczone wszystkie hasze $h_i$ porównujemy je kolejno z haszem wzorca $h(W)$. Jeśli dla pewnego $i$ zachodzi $h_i \neq h(W)$ to **na pewno** na pozycji $i$ nie ma wystąpienia wzorca $W$. Jeśli natomiast zachodzi $h_i = h(W)$ to **z dużym prawdopodobieństwem** na pozycji $i$ jest wystąpienie wzorca $W$.

Oczywiście w przypadku jednowymiarowym takie zastosowanie haszowania nie jest zazwyczaj używane gdyż mamy do dyspozycji szybki i łatwy do zaimplementowania algorytm KMP. Jednakże w przypadku dwuwymiarowym haszowanie może okazać się interesującą alternatywą dla dość złożonego algorytmu Bakera (problem wyszukiwania dwuwymiarowego wzorca w tekście został omówiony w kursie "Struktury danych i ich zastosowania").

**Ćwiczenie 6.** Opracuj szczegóły algorytmu wyszukiwania dwuwymiarowego wzorca w tekście, opartego na haszowaniu słów.

**Ćwiczenie 7.** Spróbuj rozwiązać zadanie *Pociągi* z XV Olimpiady Informatycznej stosując haszowanie (zadanie jest dostępne w serwisie [6]).

# 4 Algorytm Karpa-Millera-Rosenberga

Algorytm Karpa-Millera-Rosenberga (w skrócie KMR) jest algorytmem wyszukiwania wzorca w tekście. Dzięki swojej interesującej budowie, znajduje on także zastosowanie w wielu innych problemach tekstowych. KMR opiera się na utworzeniu dla słowa pewnej struktury zwanej **słownikiem podsłów bazowych**. Struktura ta ma rozmiar $\Theta(n \log n)$ (gdzie $n$ jest długością słowa) i może być zbudowana w takim samym czasie.

Jak zwykle w analizie algorytmów tekstowych zakładać będziemy, że alfabet ma stały rozmiar.

## 4.1 Słownik podsłów bazowych

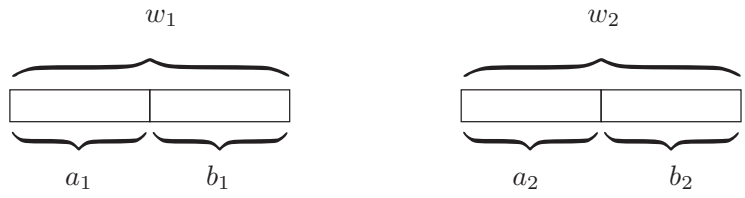
Konstrukcja słownika podsłów bazowych polega na obliczeniu haszy dla wszystkich podsłów wyjściowego słowa o długości $2^p$ dla $p = 0, 1, \ldots, \lfloor \log p \rfloor$. Tym razem jednak nie możemy użyć haszowania, takiego jak w rozdziale 3, gdyż będziemy wymagali od haszy nieco mocniejszych własności. Chcemy bowiem nie dopuścić do sytuacji, w której różne słowa tej samej długości otrzymają te same hasze.

Przyjmijmy, że budujemy strukturę słownika podsłów bazowych dla słowa $S = s_0 s_1 \ldots s_{n-1}$. Przez $id_p[t]$ oznaczmy hasz podsłowa długości $2^p$ zaczynającego się na pozycji $t$ (jeśli takie słowo nie istnieje to przyjmujemy, że $id_p[t] = \infty$). Tablice $id_p[]$ będziemy konstruować dla kolejnych $p$ zaczynając od $p = 0$. Tablica $id_0[]$ reprezentuje słowa długości 1 czyli pojedyncze litery. Doskonałym haszem dla pojedynczej litery jest jej pozycja w alfabecie (tzn. **a** $= 0$, **b** $= 1$, etc.). A zatem $id_0[t] = s_t$. Takie identyfikatory spełniają oczywiście założenie unikalności.

Zastanówmy się teraz, jak obliczyć tablicę $id_{p+1}$ mając obliczoną tablicę $id_p$. Korzystać będziemy z następującego, oczywistego faktu:

> **Fakt 1.** Słowa $s[i \ldots i + 2^{p+1} - 1]$ i $s[j \ldots j + 2^{p+1} - 1]$ są równe wtedy i tylko wtedy, gdy są równe słowa $s[i \ldots i + 2^p - 1]$ i $s[j \ldots j + 2^p - 1]$ oraz są równe słowa $s[i + 2^p \ldots i + 2^{p+1} - 1]$ i $s[j + 2^p \ldots j + 2^{p+1} - 1]$.

Innymi słowy, dwa słowa są równe gdy ich lewe połowy są równe oraz ich prawe połowy są równe. Ilustruje to poniższy rysunek:




$$w_1 = w_2 \Leftrightarrow a_1 = a_2 \wedge b_1 = b_2$$

Na podstawie Faktu 1 można zauważyć, że dobrym haszem $id_{p+1}[t]$ jest para $(id_p[t], id_p[t + 2^p])$. My jednak nie chcemy przechowywać par tylko zamienić je na nowe identyfikatory w postaci kolejnych liczb naturalnych. Aby to osiągnąć, musimy posortować leksykograficznie wszystkie takie pary identyfikatorów a następnie, przeglądając je od najmniejszych, przydzielać nowe hasze w postaci kolejnych liczb naturalnych. Oczywiście takie same pary muszą otrzymać ten sam hasz.

Warto w tym miejscu zaznaczyć, że ważne jest aby na każdym etapie sortować pary identyfikatorów w kolejności niemalejącej. Jak wkrótce zobaczymy, będzie to miało kluczowe znaczenie przy porównywaniu leksykograficznym podsłów. Zauważmy, że hasze dla każdej długości słów są liczbami z przedziału $[0, max(n, |\Sigma|)]$, przez co możemy sortowanie par zaimplementować w czasie liniowym np. używając sortowania kubełkowego.

Oto (nieco okrojona) implementacja funkcji `build_KMR(s)` konstruującej słownik podsłów bazowych dla słowa $s$ i zapisującej go w tablicy `ids[]`.

```
/* Para identyfikatorów, których zbiór będziemy sortowali.
   Aby móc zapisać wynikowy hasz musimy pamiętać też indeks
   słowa, któremu odpowiada dana para */

struct Pair {
    int p1, p2, index;
};

/* tablica (dwuwymiarowa) identyfikatorów */
vector<vector<int> > ids;

void build_KMR(string &s){
    vector<int> curr_id(s.size());
    vector<Pair> pairs;

    /* hasze słów długości 1 to kody symboli alfabetu */
    for(int i = 0; i < s.size(); ++i)
        curr_id[i] = (int) s[i];
    ids.push_back(curr_id);

    int p = 2;
    while(p < s.size()){   /* dla kolejnych potęg dwójki p */
        curr_id.resize(s.size() - p + 1);
        pairs.resize(s.size() - p + 1);
        for(int i = 0; i <= s.size() - p; ++i){
            pairs[i].p1 = ids.back()[i];
            pairs[i].p2 = ids.back()[i + p/2];
            pairs[i].index = i;
        }

        /* posortuj kubełkowo pary identyfikatorów */
        bucket_sort(pairs);

        /* przydziel nowe identyfikatory */
        int new_id = 0;
        for(int i = 0; i < pairs.size(); ++i){
            if(i > 0 && pairs[i-1] != pairs[i])
                ++new_id;
            curr_id[pairs[i].index] = new_id;
        }
        ids.push_back(curr_id);
        p *= 2;
    }
}
```

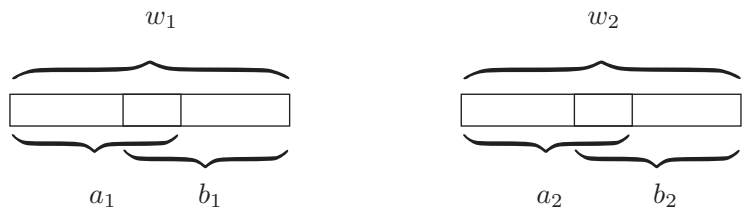
W powyższym fragmencie kodu pominięty został algorytm sortowania kubełkowego.

**Ćwiczenie 8.** Jak zmieni się złożoność funkcji `build_KMR()` gdy zamiast sortowania kubełkowego zastosujemy sortowanie przez scalanie? Zaimplementuj obie wersje algorytmu i porównaj czasy ich działania dla długich słów (w drugiej wersji możesz wykorzystać funkcję `sort()` z STL).

## 4.2 Porównywanie podsłów

Mając zbudowany słownik podsłów bazowych możemy w czasie stałym sprawdzić, czy dwa podsłowa są równe. Oczywiście jest sens sprawdzać równość tylko słów tej samej długości. Jeśli długość porównywanych słów jest potęgą dwójki, sprawdzenie równości sprowadza się do porównania haszy z odpowiedniej tablicy. Co, jeśli tak nie jest? Można skorzystać z prostej obserwacji, podobnej do Faktu 1:

**Fakt 2.** Niech dane będą słowa $s_1$ i $s_2$ długości $n$. Niech $p$ będzie największą liczbą całkowitą, taką, że $2^p \leqslant n$. Słowa $s_1$ i $s_2$ są równe wtedy i tylko wtedy, gdy są równe słowa $s_1[0 \ldots 2^p - 1]$ i $s_2[0 \ldots 2^p - 1]$ oraz są równe słowa $s_1[n - 2^p \ldots n - 1]$ i $s_2[n - 2^p \ldots n - 1]$.




$$w_1 = w_2 \Leftrightarrow a_1 = a_2 \wedge b_1 = b_2$$

Korzystając z Faktu 2 wystarczy porównać najdłuższe prefiksy oraz najdłuższe sufiksy, których długości są potęgami dwójki. Jeśli założymy, że na początku stablicowaliśmy dla każdej możliwej długości podsłowa największą potęgę dwójki nie większą niż ta długość, porównywanie podsłów można zrealizować w czasie stałym.

Jeśli przy konstrukcji słownika podsłów bazowych, za każdym razem sortowaliśmy pary identyfikatorów niemalejąco, to możemy w analogiczny sposób zrealizować w czasie stałym porównywanie leksykograficzne podsłów. W terminach oznaczeń na powyższej ilustracji oznacza to, że wystarczy porównać leksykograficznie parę $(a_1, b_1)$ z parą $(a_2, b_2)$. Co zrobić w przypadku słów różnej długości? Najpierw porównać krótsze słowo z prefiksem dłuższego o tej samej długości. W przypadku, gdy te okażą się równe, mniejsze leksykograficzne będzie słowo krótsze.

## 4.3 Najdłuższe powtarzające się podsłowo

Zastanówmy się najpierw, jak sprawdzić, czy w słowie istnieją dwa równe podsłowa ustalonej długości. Każdemu takiemu podsłowu odpowiada para identyfikatorów dwóch krótszych podsłów (jak na ilustracji do Faktu 2). Możemy wszystkie takie pary posortować kubełkowo w czasie liniowym a następnie sprawdzić czy nie ma dwóch takich samych. Jeśli dodatkowo, obok pary zapamiętamy indeks w tablicy, będziemy mogli takie powtarzające się podsłowo łatwo odtworzyć.

Jeśli w słowie istnieją dwa równe podsłowa długości $d$, to istnieją również dwa równe podsłowa każdej długości mniejszej od $d$ (są nimi na przykład prefiksy podsłów długości $d$). Wykorzystując tę obserwację możemy znaleźć najdłuższe powtarzające się podsłowo w czasie $O(n \log n)$ stosując wyszukiwanie binarne po jego długości.

**Ćwiczenie 9.** Jak znaleźć najdłuższe podsłowo, które występuje w tekście **dokładnie** $k$ razy?

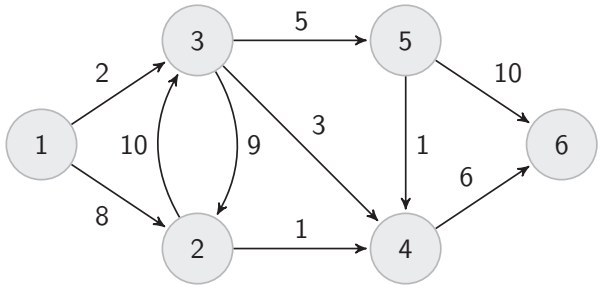
## 4.4 Najdłuższe wspólne podsłowo

Algorytm znajdowania najdłuższego wspólnego podsłowa dwóch słów $s_1$ i $s_2$ jest analogiczny do algorytmu szukającego powtarzającego się podsłowa. Wystarczy zbudować słownik podsłów bazowych dla słowa $s_1 \# s_2$ (gdzie „#" jest symbolem nie występującym w alfabecie), a następnie znaleźć najdłuższe powtarzające się podsłowo. Aby zagwarantować, że znalezione podsłowo występuje zarówno w $s_1$ jak i w $s_2$ (a nie na przykład dwukrotnie w $s_1$) musimy przy sortowaniu par identyfikatorów pamiętać dodatkowo, z którego słowa pochodzi dana para i akceptować tylko powtórzenie pary z różnych słów.

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI
Warszawska Wyższa Szkoła Informatyki
UNIA EUROPEJSKA
EUROPEJSKI FUNDUSZ SPOŁECZNY
Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego

# 5 Maksymalny przepływ

Wyobraźmy sobie, że planujemy sieć rurociągów, która ma za zadanie transportować ropę ze złoża do rafinerii. Mamy już mapę z zaznaczonymi rurami, o każdej z nich wiemy, jaka jest jej maksymalna przepustowość (ilość przepływającej ropy w jednostce czasu).




Przykładowa sieć rurociągów z przepustowościami

Zastanawiamy się teraz, jaka jest przepustowość całej sieci, czyli ile ropy może przez nią przepłynąć od złoża (oznaczonego numerem $1$) do rafinerii (numer $6$). Rozważamy właśnie problem poszukiwania maksymalnego przepływu, który jest tematem tego rozdziału.

## Formalne ujęcie problemu

Mamy dany graf skierowany $G$, z wyróżnionymi dwoma wierzchołkami: *źródłem* $s$ i *ujściem* $t$. Każda krawędź w grafie ma określoną *przepustowość*: dla krawędzi między wierzchołkami $u$ i $v$ oznaczamy ją przez $c(u, v)$. Wymaga to założenia, że między dwoma wierzchołkami, w jedną stronę, istnieje tylko jedna krawędź. W celu uproszczenia dalszego opisu możemy przyjąć też, że jeżeli w grafie istnieje krawędź $(u, v)$ to jest także krawędź $(v, u)$. Aby dany graf zaczął spełniać to ograniczenie wystarczy dodać doń krawędzie „powrotne" z przepustowością równą $0$.

Graf zadany w ten sposób będziemy nazywali *siecią*, w której będziemy szukali *przepływu* $f$. Jest to funkcja, która dla dwóch wierzchołków $u$ i $v$ sieci $G$ mówi nam, ile jednostek przemieszcza się bezpośrednio między $u$ i $v$. Oznacza to, że zawsze $f(u, v) \leqslant c(u, v)$: przepływ przez krawędź między dwoma wierzchołkami nie może być większy niż jej przepustowość. Ustalmy też, że $f(u, v) = -f(v, u)$, czyli z dodatnim przepływem w jedną stronę jest związany ujemny w drugą.

Przez przepływ wchodzący do wierzchołka $u$ rozumiemy sumę dodatnich elementów postaci $f(v, u)$, dla dowolnego wierzchołka $v$. Natomiast przepływ wychodzący z $u$ to suma dodatnich elementów postaci $f(u, v)$ dla dowolnego wierzchołka $v$. Chcemy aby przesyłane jednostki nie akumulowały się w wierzchołkach, czyli aby dla każdego wierzchołka przepływ wchodzący był równy przepływowi wychodzącemu. Jedynie źródło i ujście mogą nie spełniać tego warunku. Zgodnie z nazwą, w źródle przepływ wychodzący może być większy niż przepływ wchodzący, natomiast w ujściu odwrotnie: przepływ wchodzący może być większy od wychodzącego.

Można udowodnić, że różnica przepływu wychodzącego i wchodzącego do źródła, jest równa różnicy przepływu wchodzącego i wychodzącego z ujścia. Wielkość tę nazywamy *wartością przepływu* i oznaczamy $W(f)$. *Maksymalny przepływ* to ten o największej wartości przepływu.

Krawędź $(u, v)$ będziemy nazywali *użyteczną*, jeżeli $f(u, v) < c(u, v)$. Po krawędzi użytecznej możemy zawsze przesłać jakiś dodatkowy przepływ. Zauważmy, że krawędź $(u, v)$ może być użyteczna, nawet jeśli ma przepustowość równą $0$. Dzieje się tak gdy przepływ $f(u, v)$ jest ujemny, czyli gdy z $v$ do $u$ są przesyłane jakieś jednostki ($-f(u, v) = f(v, u) > 0$).

**Ćwiczenie 10.** Przy opisie problemu przyjęliśmy założenie, że w grafie nie ma wielokrotnych krawędzi między żadną parą wierzchołków. Bardzo prosto jest zmienić dowolny graf z przepustowościami do takiej postaci: wystarczy krawędzie wielokrotne zastąpić jedną, której przepustowość będzie sumą przepustowości tamtych.
Czy można na podstawie przepływu znalezionego w tak zmodyfikowanym grafie skonstruować przepływ w oryginalnym grafie?

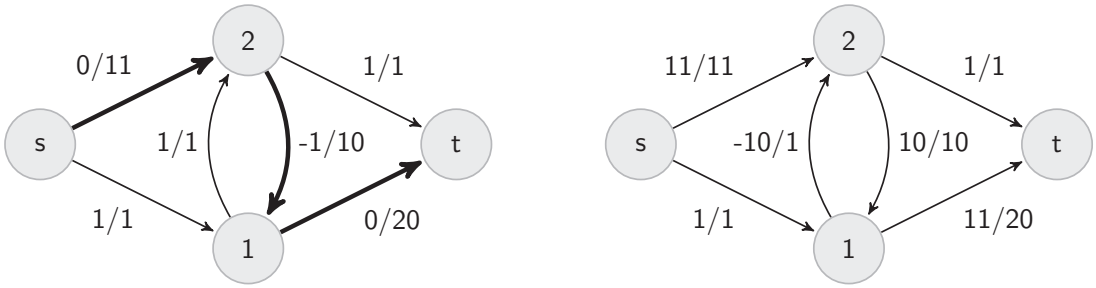
**Ćwiczenie 11.** Mamy daną sieć w której znajduje się kilka źródeł i kilka ujść. Jak należy zmienić tę sieć, aby miała po jednym źródle i ujściu, ale przepływy w zmodyfikowanej sieci dały się łatwo przełożyć na te w oryginalnej.

## 5.1 Metoda Forda-Fulkersona

Metoda Forda-Fulkersona jest podstawą wielu algorytmów rozwiązujących problem maksymalnego przepływu. Polega ona na konstruowaniu kolejnych przepływów aż do osiągnięcia przepływu maksymalnego.

Zaczynamy od pustego przepływu ($f(u,v) = 0$ dla dowolnych par wierzchołków). Następnie szukamy *ścieżki powiększającej*, czyli ścieżki od źródła do ujścia złożonej tylko z użytecznych krawędzi. Jeżeli takiej ścieżki nie ma, to mamy już maksymalny przepływ. Jeżeli zaś znaleźliśmy ścieżkę złożoną z wierzchołków $s = v_1, v_2, \ldots, v_n = t$, to możemy powiększyć przepływ. Na każdej z krawędzi od $v_i$ do $v_{i+1}$ możemy przesłać co najwyżej $c(v_i, v_{i+1}) - f(v_i, v_{i+1})$, zatem możemy powiększyć przepływ na każdej krawędzi ze ścieżki o najmniejszą z tych wartości. Dla krawędzi $(u,v)$ wielkość $c(u,v) - f(u,v)$ nazywa się niekiedy *przepustowością residualną*.

Na poniższym przykładzie zostało zilustrowana jedna faza metody Forda-Fulkersona. Na tym i na następnych obrazkach etykietki nad krawędzią od $u$ do $v$ będą postaci $f(u,v)/c(u,v)$. Aby nie komplikować niepotrzebnie diagramu część krawędzi o zerowej przepustowości zostanie pominięta.

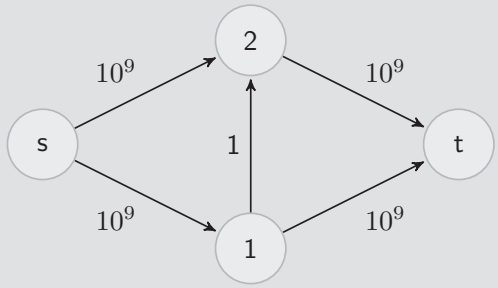



Wyszukanie ścieżki powiększającej i poprawienie przepływu

Nietrudno uwierzyć w to, że metoda ta zwróci poprawną odpowiedź. Formalnie dowiedziemy tego trochę później, w twierdzeniu o maksymalnym przepływie i minimalnym przekroju.

Oszacujmy teraz czas działania metody Forda-Fulkersona. Każdą fazę (znalezienie ścieżki powiększającej i poprawienie przepływu) można zrealizować za pomocą pojedynczego przeszukiwania grafu (wszerz lub w głąb). W każdej fazie zwiększamy przepływ o co najmniej $1$, zatem maksymalna liczba faz jest równa wartości maksymalnego przepływu. Jeżeli liczbę krawędzi grafu oznaczymy przez $m$, to omawiana metoda działa w czasie $O(W(f_{\max})\, m)$.

**Ćwiczenie 12.** W poniższym grafie znajdź maksymalny przepływ.
Załóżmy, że znajdujemy go wykorzystując metodę Forda-Fulkersona. Jaka jest minimalna i maksymalna możliwa liczba faz, które wykonujemy?




Graf z podanymi przepustowościami krawędzi

## 5.2 Algorytm Edmondsa-Karpa

Ostatnie ćwiczenie pokazuje, że metoda Forda-Fulkersona może działać wolno nawet dla bardzo małych grafów, gdyż liczba faz może okazać się bardzo duża. Aby poprawić tę sytuację w algorytmie Edmondsa-Karpa (który jest ukonkretnieniem tej metody) poszukujemy zawsze możliwe najkrótszej (tzn. o najmniejszej liczbie krawędzi) ścieżki powiększającej. Możemy tego prosto dokonać za pomocą przeszukiwania wszerz. Okazuje się, że po dodaniu tego ograniczenia uzyskujemy algorytm wydajny niezależnie od wielkości maksymalnego przepływu: liczba faz w sieci o $n$ wierzchołkach i $m$ krawędziach wynosi co najwyżej $nm$.

Aby udowodnić powyższe ograniczenie skorzystamy z następującego lematu:

**Lemat.**
Odległością wierzchołków nazwiemy długość najkrótszej ścieżki między nimi złożonej tylko z krawędzi użytecznych. W kolejnych iteracjach algorytmu Edmondsa-Karpa odległość dowolnego wierzchołka od źródła nie maleje.

Dowód tego lematu jest trochę techniczny, nie będziemy go tu przedstawiać. Zainteresowanych odsyłam do „Wprowadzenia do algorytmów" [2]. Przystąpmy teraz do dowodu oszacowania.

W każdej ścieżce powiększającej istnieją dwa kolejne wierzchołki $v_i$ i $v_{i+1}$, dla których przepustowość residualna $c(v_i, v_{i+1}) - f(v_i, v_{i+1})$ jest najmniejsza na tej ścieżce. Oznacza to, że po powiększeniu przepływu wzdłuż tej ścieżki nowy przepływ między $v_i$ i $v_{i+1}$ wyniesie $f(v_i, v_{i+1}) + (c(v_i, v_{i+1}) - f(v_i, v_{i+1})) = c(v_i, v_{i+1})$, a więc krawędź od $v_i$ do $v_{i+1}$ przestanie być użyteczna. Jedyną sytuacją, po której znowu stanie się użyteczna, jest przesłanie przepływu od $v_{i+1}$ do $v_i$.

Weźmy teraz dowolną krawędź od $u$ do $v$ i obliczmy ile razy w czasie wykonania algorytmu Edmondsa-Karpa może być ona krawędzią o najmniejszej przepustowości residualnej na ścieżce powiększającej. Przez $d$ oznaczmy odległość $u$ od źródła tuż przed jakaś fazą algorytmu, w której ustalona krawędź będzie tą o najmniejszej przepustowości residualnej. Ponieważ ścieżka powiększająca jest najkrótszą w grafie użytecznych krawędzi, to odległość $v$ od źródła wynosi $d+1$. Zgodnie z tym, co powiedzieliśmy wcześniej, po tej fazie krawędź $(u, v)$ przestanie być użyteczna. Oznacza to, że nie będzie też w żadnej ścieżce powiększającej, dopóki nie prześlemy przepływu z $v$ do $u$. W takim zaś momencie odległość od źródła do $v$ wyniesie co najmniej $d + 1$ (bo odległości nie maleją, zgodnie z lematem), a $v$ będzie na najkrótszej ścieżce ze źródła do $u$, zatem odległość $u$ wyniesie nie mniej niż $d + 2$.

Podsumowując powyższy wywód: pomiędzy dwoma fazami, w których ustalona krawędź od $u$ do $v$ będzie tą o najmniejszej przepustowości residualnej, odległość od źródła do $u$ musi się zwiększyć co najmniej o $2$. Jako, że największa możliwa odległość to $n$, każda krawędź może być tą o najmniejszej przepustowości residualnej co najwyżej $\frac{n}{2}$ razy.

Wszystkich krawędzi jest $m$, a w każdej fazie musi być jakaś krawędź o najmniejszej przepustowości residualnej, zatem rzeczywiście wszystkich faz będzie nie więcej niż $\frac{n}{2}m$, czyli $O(nm)$.

Powyższe ograniczenie na liczbę faz pozwala podać oszacowanie złożoności algorytmu Edmondsa-Karpa: wynosi ono $O(nm^2)$ (co najwyżej $nm$ faz, każdą wykonujemy w czasie $m$). Istnieją szybsze sposoby wyszukiwania maksymalnego przepływu, oparte o metodę Forda-Fulkersona: algorytm Dinica oraz algorytm Malhotry, Kumara i Maheshwariego. Są one jednak dość skomplikowane, także w implementacji. Zainteresowani mogą o nich poczytać w książkach „Kombinatoryka dla programistów" [5] oraz „Algorytmy optymalizacji dyskretnej" [8].

**Ćwiczenie 13.** Zaimplementuj algorytm Edmondsa-Karpa.
*Wskazówka:* Złożoność algorytmu powoduje, że zwykle dane nie są zbyt duże i można sobie pozwolić na posłużenie się reprezentacją grafu w postaci macierzy sąsiedztwa, a także trzymanie aktualnych wartości przepływu w tablicy kwadratowej.

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI
Warszawska Wyższa Szkoła Informatyki
UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY
Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego

## 5.3 Twierdzenie o maksymalnym przepływie i minimalnym przekroju

Podział wierzchołków sieci $G$ o źródle $s$ i ujściu $t$ na dwa zbiory $S$ i $T$ takie, że $s \in S$ a $t \in T$ nazywamy *przekrojem* grafu G. Wówczas *przepustowością przekroju* nazywamy sumę wszystkich przepustowości krawędzi $(u, v)$, które prowadzą z $S$ do $T$ (czyli $u \in S$ i $t \in T$). Oznaczamy ją przez $c(S, T)$. Natomiast sumę przepływów przez krawędzie prowadzące z $S$ do $T$ nazywamy *przepływem netto* przez przekrój i oznaczamy $f(S, T)$. Nietrudno zauważyć, że przepływ netto przez przekrój jest nie większy niż jego przepustowość: $f(S, T) \leqslant c(S, T)$.

Co więcej, przepływ netto przez dowolny przekrój jest równy wartości przepływu. Gdy przekrój składa się z dwóch zbiorów, w którym jednym jest tylko źródło, a w drugim wszystkie pozostałe wierzchołki to po prostu definicja przepływu netto i wartości przepływu pokrywają się. Aby udowodnić ten fakt dla dowolnego podziału wystarczy zauważyć, że przekroje $S$ i $T$ oraz $S \cup \{x\}$ i $T \setminus \{x\}$ mają taką samą wartość przepływu netto (o ile $x \neq t$). Jest tak gdyż różnica tych dwóch przepływów netto: $f(S, T) - f(S \cup \{x\}, T \setminus \{x\})$ równa jest sumie elementów postaci $f(s_1, x)$ dla $s_1 \in S$ odjąć suma $f(x, t_1)$ dla $t_1 \in T$. W związku z tym, że $-f(x, t_1) = f(t_1, x)$ otrzymujemy, że szukana różnica to po prostu suma $f(y, x)$ dla wszystkich wierzchołków $y$, czyli różnica wchodzącego i wychodzącego z $x$ przepływu. Ta zaś wartość jest równa $0$.

Sformułujmy teraz i udowodnimy bardzo ważne twierdzenie:

**Twierdzenie o maksymalnym przepływie i minimalnym przekroju.**
W danej sieci $G$ jest określony przepływ $f$. Wówczas następujące warunki są równoważne:

1. przepływ jest maksymalny
2. nie istnieje ścieżka powiększająca
3. istnieje przekrój, którego przepustowość jest równa wielkości przepływu

Dowód składa się z trzech implikacji:

$1 \Rightarrow 2$: dowód nie wprost jest trywialny. Jeżeli istniałaby ścieżka powiększająca to moglibyśmy wzdłuż niej zwiększyć przepływ $f$ co oznacza, że nie był on maksymalny. Sprzeczność.

$2 \Rightarrow 3$: Niech $S$ będzie zbiorem tych wierzchołków, które można odwiedzić ze źródła idąc tylko po użytecznych krawędziach, a $T$ pozostałym zbiorem wierzchołków. Ponieważ nie ma ścieżki powiększającej, to źródło $t \in T$, a zatem $S$ i $T$ jest pewnym przekrojem grafu.

Zgodnie z definicją, między $S$ i $T$ nie istnieje żadna użyteczna krawędź, a zatem dla dowolnych $s_1 \in S$, $t_1 \in T$ połączonych krawędzią mamy $f(s_1, t_1) = c(s_1, t_1)$. Gdy zsumujemy te równości dla wszystkich możliwych $s_1$ i $t_1$ po lewej stronie otrzymamy przepływ netto przez przekrój $S$ i $T$ (a więc $W(f)$ zgodnie ze wcześniejszym spostrzeżeniem), a po prawej przepustowość przekroju.

$3 \Rightarrow 1$: Dla dowolnego przekroju $S$ i $T$ mamy, że $f(S, T) \leqslant c(S, T)$, czyli przepustowość dowolnego przekroju stanowi górne ograniczenie na wielkość przepływu przez ten przekrój (a zatem też na wielkość przepływu w sieci). Jeżeli zatem istnieje przekrój, dla którego mamy równość między przepływem netto, a przekrojem to tym samym nas przepływ osiąga największą możliwą wartość.

Właśnie zakończyliśmy dowód twierdzenia. Przy okazji pokazaliśmy też formalnie poprawność metody Forda-Fulkersona dla sieci o całkowitoliczbowych przepustowościach: algorytm kończymy, gdy nie ma już żadnej ścieżki powiększającej, wtedy też uzyskany przepływ jest maksymalny. Gdy przepustowości są całkowitoliczbowe, w każdej fazie zwiększamy wartość przepływu o co najmniej $1$. Zatem metoda Forda-Fulkersona zawsze się zakończy: wykonamy tylko skończenie wiele faz, gdyż skończona jest wielkość minimalnego przekroju, który ogranicza z góry wielkość przepływu.

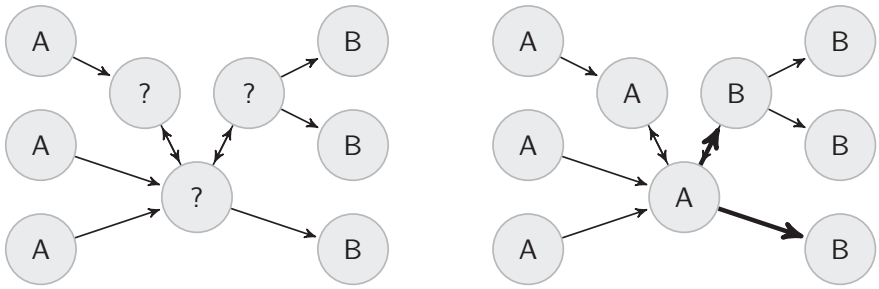
**Ćwiczenie 14.** Wiemy już, że wielkość maksymalnego przepływu jest równa minimalnej przepustowości przekroju. Jak możemy wykorzystać metodę Forda-Fulkersona do znalezienia przekroju, który ma minimalną przepustowość?

## 5.4 Przekroje w zadaniach

Okazuje się, że udowodnione właśnie twierdzenie ma znaczenie nie tylko teoretyczne. Na konkursach zdarzają się trudne zadania, których rozwiązanie polega właśnie na wskazaniu minimalnego przekroju. Przyjrzyjmy się następującemu przykładowi:

> W pewnym regionie mieszkają dwa wrogo nastawione do siebie plemiona: Arbuzanie i Bananici. W regionie tym znajduje się wiele wiosek, każda z nich jest zamieszkana przez mieszkańców jednego plemienia. Obecnie planowane jest zasiedlenie kilku kolejnych wiosek.
> Wiesz, które wioski są zamieszkane przez Arbuzan, które przez Bananitów, a które czekają na zasiedlenie. Mając dany graf opisujący drogi między wioskami wskaż takie zasiedlenie, które zminimalizuje liczbę dróg łączących wioski zamieszkane przez różne plemiona.

Gdy z góry wiemy, w jaki sposób należy rozwiązać zadanie, nie wydaje się ono bardzo trudne. Należy zbudować sieć, gdzie źródłami będą wioski zamieszkane przez Arbuzan, a ujściami te bananickie.

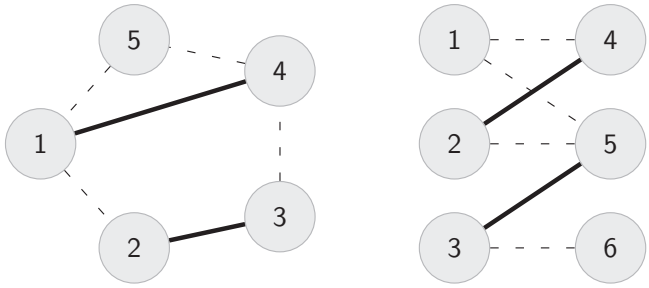



Sieć reprezentująca wioski oraz poprawne rozwiązanie
(pogrubione są krawędzie liczone w przepustowości przekroju)

Każdy przekrój $S$ i $T$ w takim grafie będzie stanowił w istocie podział wiosek na arbuzańskie i bananickie (oczywiście wszystkie źródła będą w $S$, a wszystkie ujścia w $T$). Przepustowością przekroju będzie liczba dróg łączących wioski zamieszkałe przez Arbuzan i Bananitów. Liczba takich dróg ma być jak najmniejsza, zatem poszukiwany podział jest wyznaczany przez minimalny przekrój.

# 6 Skojarzenia

*Skojarzeniem* w grafie nazywamy taki podzbiór jego krawędzi, że żadne dwie nie stykają się w wierzchołku. Innymi słowy z żadnego wierzchołka nie wychodzą dwie krawędzie ze skojarzenia.




Skojarzenia w grafie i grafie dwudzielnym

Graf $G$ nazywamy *dwudzielnym*, gdy jego wierzchołki można podzielić na dwa zbiory: $X$ i $Y$ w taki sposób, że nie ma krawędzi pomiędzy wierzchołkami z jednego zbioru (a więc każda krawędź jest postaci $(x, y)$ gdzie $x \in X$ oraz $y \in Y$). Umówmy się na potrzeby tego rozdziału, że zbiory $X$ i $Y$ zawsze będą oznaczały opisany wyżej podział wierzchołków grafu $G$.

W ramach zajęć będziemy się zajmować wyłącznie skojarzeniami w grafach dwudzielnych. Mają one następującą naturalną interpretację:

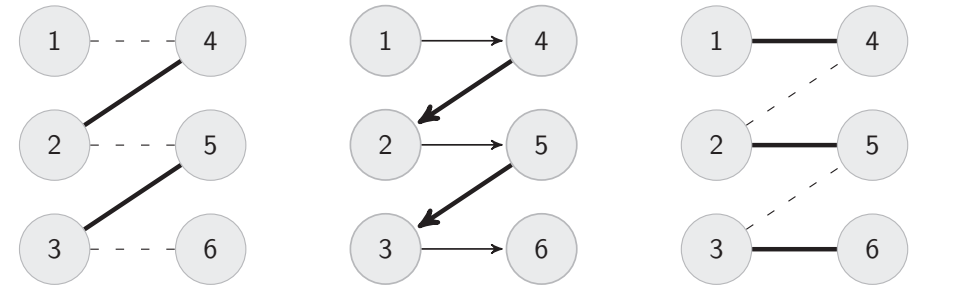
$X$ i $Y$ to zbiory mężczyzn i kobiet. Krawędziami łączymy osoby, które są sobą zainteresowane w sensie matrymonialnym. Oczywiście nie możemy żądać, by jakikolwiek mężczyzna był zainteresowany tylko jedną kobietą, lub też jakaś kobieta tylko jednym mężczyzną! Przez skojarzenie rozumiemy wskazanie tych par, które zostaną małżeństwami: tu już obowiązuje monogamia!

Krawędzie które należą do skojarzenia nazywamy *skojarzonymi*, pozostałe zaś *wolnymi*. Podobnie *wolne* będą te wierzchołki, które nie są incydentne z żadną krawędzią skojarzenia.

## 6.1 Maksymalne skojarzenia

Jak więc znaleźć maksymalne (tj. o największej liczbie krawędzi) skojarzenie w grafie dwudzielnym? Podobnie jak w przypadku maksymalnego przepływu będziemy starali się w kolejnych krokach zwiększać skojarzenie dopóki będzie to możliwe.

Przez *ścieżkę powiększającą* będziemy teraz rozumieli ścieżkę pomiędzy dwoma wierzchołkami wolnymi, której krawędzie będą na przemian wolne i zajęte. Znalezienie w grafie takiej ścieżki oznacza, że możemy powiększyć skojarzenie:




Znalezienie ścieżki powiększającej i zwiększanie skojarzenia

Otrzymaliśmy zatem bardzo prosty algorytm: w każdej fazie szukamy ścieżki powiększającej. Jeżeli ona istnieje, to zwiększamy skojarzenie. Jeżeli nie, to kończymy.

Aby udowodnić, że algorytm jest poprawny potrzebny jest nam następujący fakt:

**Twierdzenie Berge'a.**
W grafie nie ma ścieżki powiększającej wtedy i tylko wtedy, gdy uzyskane skojarzenie jest maksymalne.

**Ćwiczenie 15.** Udowodnij powyższe twierdzenie. Możesz ograniczyć się tylko do grafów dwudzielnych, choć twierdzenie jest prawdziwe dla dowolnych.

Jaka jest złożoność zaproponowanego algorytmu dla grafu o $n$ wierzchołkach i $m$ krawędziach? W takim przypadku wielkość skojarzenia jest na pewno nie większa niż $n$. W każdej fazie, po znalezieniu nowej ścieżki powiększającej wielkość skojarzenia zwiększa się o $1$, zatem będzie co najwyżej $n$ faz. Co więcej, do znalezienia jednej ścieżki wystarczy przeszukiwanie grafu w głąb, które zajmuje czas liniowy względem liczby krawędzi. Podsumowując złożoność czasowa tego algorytmu to $O(nm)$.

Przyjrzymy się teraz bardzo prostej implementacji tego algorytmu dla grafu dwudzielnego $G$. W zbiorach $X$ i $Y$ wierzchołki są w ponumerowane od $0$ do $nx-1$ i od $0$ do $ny-1$ odpowiednio.

W implementacji korzystamy z następujących globalnych zmiennych:

- Krawędzie grafu incydentne z wierzchołkiem $i$-tym z $X$ przechowywane są w $i$-tym wektorze tablicy `vector<int> gr[N]`.
- W $i$-tej komórce tablic `int skojx[N], skojy[N]` przechowujemy numer wierzchołka aktualnie skojarzonego z $i$-tym z odpowiedniego zbioru ($X$ lub $Y$). Wartość $-1$ oznacza wierzchołek wolny.
- Do wykonywania algorytmu DFS będziemy wykorzystywali tablicę `bool odw[N]`.

Potrzebujemy funkcji, która będzie próbowała znaleźć ścieżkę powiększającą z danego wierzchołka, a jeżeli się uda to uaktualni skojarzenia. Oto ona:

```
bool skojarz (int a)
{
    if (odw[a])
        return false;
    odw[a] = true;
    FOREACH (it, gr[a])
        if (skojy[*it] == -1 || skojarz (skojy[*it]))
        /* sąsiad *it aktualnego wierzchołka jest wolny, lub też można
           znaleźć ścieżkę powiększającą zaczynającą się od wierzchołka,
           z którym *it jest skojarzony
         */
        {
            skojx[a] = *it;
            skojy[*it] = a;
            return true;
        }
    return false;
}
```

Następnie wystarczy uruchomić tę procedurę dla kolejnych wierzchołków ze zbioru $X$.

```
int wyn = 0;
for (int i = 0; i < nx; i++)
{
    for (int j = 0; j < nx; j++)
        odw[j] = false;
    if (skojarz(i))
        wyn++;
}
```

Aby się przekonać o poprawności tego rozwiązania, trzeba zauważyć dwa fakty. Po pierwsze, jeżeli istnieje ścieżka powiększająca z danego wierzchołka (a tablica `odw` jest cała ustawiona na `false`), to funkcja `skojarz` na pewno ją znajdzie. Po drugie, gdy wierzchołek raz stanie się skojarzony, to nie zmieni tego żadne wywołanie funkcji `skojarz` (choć oczywiście może się zmienić wierzchołek, z którym jest skojarzony). Zatem rzeczywiście możemy próbować znajdować ścieżki powiększające w dowolnej kolejności, na przykład od zaczynających się w $0$ do zaczynających się w $nx - 1$.

W wielu książkach problem maksymalnego skojarzenia w grafie dwudzielnym podaje się jako szczególny przypadek problemu maksymalnego przepływu. Tutaj przedstawiliśmy jednak bezpośrednie rozwiązanie, by podkreślić jego niezwykłą prostotę.

## 6.2 Szybsze rozwiązania

Czy istnieje szybszy sposób znajdowania skojarzenia w grafie dwudzielnym? Odpowiedź jest twierdząca — algorytm Hopcrofta-Karpa rozwiązuje ten problem i działa ze złożonością $O(m\sqrt{n})$. Jest to jednak dość złożony algorytm, dlatego nie będziemy go tutaj omawiać. Zainteresowanych ponownie odsyłamy do znakomitej książki Witolda Lipskiego [5].

Przedstawimy natomiast lekko zmodyfikowaną wersję poprzedniego algorytmu. Wprawdzie jego złożoność to nadal $O(nm)$, ale w praktyce działa bardzo szybko, często porównywalnie do algorytmu Hopcrofta-Karpa.

Pomysł jest bardzo prosty: zamiast za każdym razem czyścić tablicę odwiedzonych wierzchołków (`odw`) spróbujmy w każdej fazie znajdować od razu kilka rozłącznych ścieżek powiększających.

Implementacja wykorzystuje zdefiniowaną wcześniej funkcję `skojarz` i jest bardzo prosta:

KAPITAŁ LUDZKI NARODOWA STRATEGIA SPÓJNOŚCI
Warszawska Wyższa Szkoła Informatyki
UNIA EUROPEJSKA EUROPEJSKI FUNDUSZ SPOŁECZNY
Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego

```
int wyn = 0;
bool zmiana;
do {
    zmiana = false;
    for (int i = 0; i < nx; i++)
        odw[i] = false;
    for (int i = 0; i < nx; i++)
        if (skojx[i]==-1 && skojarz(i)){
            zmiana = true;
            wyn ++;
        }
} while(zmiana);
```

**Ćwiczenie 16.** Udowodnij, że jeżeli wielkość maksymalnego skojarzenia w grafie jest równa $M$, to w pierwszym obrocie pętli do `while` algorytm znajdzie skojarzenie o liczności co najmniej $\lfloor \frac{M}{2} \rfloor$.

Niestety nie istnieje formalny dowód tego, że algorytm ten wykonuje małą liczbę faz — istnieją złośliwe przypadki, w których może on działać nieefektywnie. W praktyce sprawdza się jednak bardzo dobrze. Aby uchronić się przed owymi złośliwymi przypadkami można próbować na początku w losowej kolejności poustawiać wierzchołki w listach sąsiedztwa.

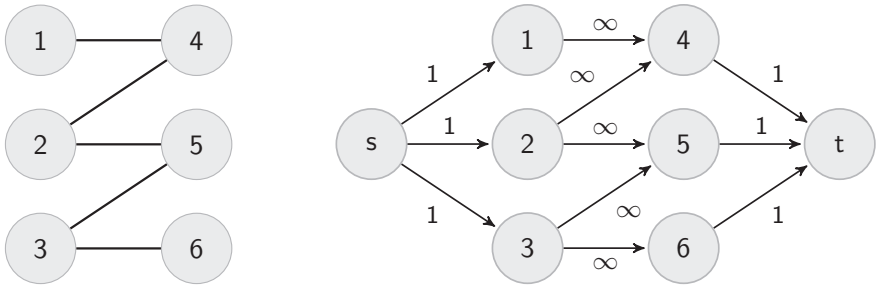
## 6.3 Twierdzenie Königa

Niekiedy wykorzystuje się maksymalne skojarzenie do obliczania innych wartości dla grafu. W tym rozdziale zajmiemy się twierdzeniu Königa, które jest przykładem takiej sytuacji.

Zaczniemy od definicji: *pokryciem wierzchołkowym* grafu $G$ nazywamy taki podzbiór $P$ jego wierzchołków, że każda krawędź z $G$ jest incydentna z wierzchołkiem z $P$. Okazuje się, że zachodzi:

**Twierdzenie Königa.**
W grafie dwudzielnym liczności maksymalnego skojarzenia i minimalnego pokrycia wierzchołkowego są sobie równe.

Dowód tego faktu będzie przeprowadzony na podstawie twierdzenia o maksymalnym przepływie i minimalnym przekroju. Aby z niego skorzystać musimy jednak mieć sieć zamiast grafu dwudzielnego. Istnieje prosty sposób uzyskania odpowiedniej sieci.




Graf dwudzielny i odpowiadająca jemu sieć

Dany jest graf dwudzielny $D$, o dwóch zbiorach wierzchołków $X$ i $Y$ (między którymi przebiegają wszystkie krawędzie grafu). Dodajemy do niego dwa wierzchołki: źródło $s$ i ujście $t$. Źródło łączymy z wierzchołkami zbioru $X$ za pomocą krawędzi o przepustowości 1, podobnie wierzchołki zbioru $Y$ łączymy z ujściem. W końcu wszystkie krawędzie grafu $D$ zamieniamy na skierowane krawędzie od wierzchołków z $X$ do tych z $Y$, nadajemy im też bardzo dużą przepustowość — na przykład większą niż liczba wszystkich wierzchołków $D$.

Aby spełnić zadane przez nas wcześniej wymagania dotyczące sieci, należy jeszcze dodać dla każdej krawędzi w sieci krawędź powrotną o przepustowości 0. Uzyskaną w ten sposób sieć oznaczmy przez $G$.

Pokażemy teraz, że maksymalne skojarzenie w grafie $D$ jest równe maksymalnemu przepływowi w sieci $G$. Wynika to z tego, że dowolne skojarzenie w $D$ wyznacza przepływ w $G$ i odwrotnie:

- Każda krawędź $(x, y)$ w pewnym skojarzeniu w grafie $D$ wyznacza nam ścieżkę $s \to x \to y \to t$ w $G$, po której możemy przesłać jedną jednostkę przepływu. Krawędzie ze skojarzenia nie są incydentne ze sobą, a więc uzyskane ścieżki nie mają punktów wspólnych (poza źródłem i ujściem). Oznacza to, że możemy puścić przepływ przez wszystkie ścieżki naraz, uzyskując przepływ o wartości równej wielkości skojarzenia w $D$.
- Mając dany przepływ w sieci $G$ wybierzmy te krawędzie ze zbioru $X$ do $Y$, które mają niezerowy przepływ (mają one zawsze przepływ równy $1$). Zauważmy, że co najwyżej jedna wybrana krawędź może być incydentna z pewnym wierzchołkiem $x \in X$ - gdyby incydentnych krawędzi było więcej to przepływ przechodzący przez $x$ byłby nie mniejszy niż $2$, a tymczasem krawędź $(s, x)$ o przepustowości $1$ to jedna krawędź o niezerowej przepustowości wchodząca do $x$. Podobnie dowiedziemy, że co najwyżej jedna wybrana krawędź jest incydenta z dowolnym wierzchołkiem z $Y$. Oznacza to, że odpowiedniki wybranych przez nas krawędzi stanowią skojarzenie w $D$ o liczności równej wartości rozważanego przepływu w $G$.

Pokażemy teraz równość liczebności minimalnego pokrycia wierzchołkowego w $D$ i przepustowości minimalnego przekroju w $G$. Podobnie jak przed chwilą pokażemy pewnego rodzaju odpowiedniość między tymi obiektami:

- Mamy dany graf $D$ z pokryciem złożonym z wierzchołków zbioru $P$. Wskażemy teraz taki podział wierzchołków sieci $G$ na zbiory $S$ i $T$, że wyznaczany przez nie przekrój będzie miał przepustowość równą $|P|$. Do zbioru $S$ należeć będzie źródło $s$, zbiór $X \backslash P$ (incydentne z $s$, ale nie wybrane do pokrycia) oraz zbiór $Y \cap P$ (wierzchołki incydentne z $t$, które zostały wybrane do pokrycia). Pozostałe wierzchołki wpadają do $T$.

  Policzymy teraz przepustowość tego przekroju, czyli sumę przepustowości krawędzi prowadzących ze zbioru $S$ do $T$. Po pierwsze zauważmy, że do tej sumy nie zostanie nigdy wliczona przepustowość krawędzi $(x, y)$ dla $x \in X$ i $y \in Y$. Można by ją wliczyć tylko wtedy, gdy $x \in S$ (czyli $x \notin P$) oraz $y \in T$ (czyli $y \notin P$), a zatem istniałaby krawędź, której oba wierzchołki nie byłyby w pokryciu $P$, co stoi w sprzeczności z definicją pokrycia.

  Zatem krawędzie liczone do przepustowości przekroju będą postaci $(s, a)$ dla $a \in X \cap P \subset T$ oraz $(b, t)$ dla $b \in Y \cap P \subset S$. Widać wyraźnie, że jest ich dokładnie $|P|$, a każda z nich ma przepustowość $1$, zatem wskazany przekrój rzeczywiście ma przepustowość $|P|$.

- Teraz mając dany przekrój $(S, T)$ w sieci $G$ zbudujemy pokrycie grafu $D$. Rozpatrujemy wszystkie krawędzie prowadzące z $S$ do $T$. Dla każdej krawędzi postaci $(s, x)$, gdzie $x \in X \cap T$ dodajemy wierzchołek $x$ do pokrycia. Podobnie dla krawędzi $(y, t)$, gdzie $y \in Y \cap S$ dodajemy wierzchołek $y$. Natomiast dla każdej krawędzi $(x, y)$, gdzie $x \in X \cap S$ i $y \in Y \cap T$ dodajemy zarówno $x$ jak i $y$.

  Nietrudno zauważyć, że tak wskazany zbiór rzeczywiście jest pokryciem: weźmy dowolną krawędź $(x, y)$ w $D$, która wyznacza ścieżkę $s \to x \to y \to t$ w sieci $G$. Ścieżka ta zaczyna się w zbiorze $S$, a kończy w $T$, więc któraś z krawędzi $(s, x), (x, y), (y, t)$ będzie „graniczna" między tymi zbiorami, a zatem $x$, lub $y$, lub nawet oba wierzchołki zostaną dodane do pokrycia. Dzięki temu krawędź $(x, y)$ grafu $D$ będzie incydentna z którymś wierzchołkiem pokrycia.

  Zauważmy teraz, że wprawdzie liczność pokrycia może się różnić od przepustowości przekroju, ale tylko w przypadku gdy do przekroju wliczymy krawędź prowadzącą z $X$ do $Y$. Każda taka krawędź ma przepustowość większą niż $|X \cup Y|$, więc przepustowość takiego przekroju jest z pewnością większa niż przepustowość przekroju $(\{s\}, X \cup Y \cup \{t\})$. Wynika z tego, że jeżeli interesują nas minimalne przekroje (a ostatecznie pokrycia) to nie musimy się zajmować przekrojami, do których wliczają się krawędzie prowadzące z $X$ do $Y$.

Udowodniliśmy właśnie, że liczność minimalnego pokrycia w grafie $D$ jest równa wielkości minimalnego przekroju w sieci $G$. Przed chwilą uzyskaliśmy też równość maksymalnego przepływu w $G$ i maksymalnego skojarzenia w $D$. Twierdzenie o maksymalnym przepływie i minimalnym przekroju mówi nam

o równości tych dwóch wartości w dowolnej sieci. Zaaplikowanie tego twierdzenia do sieci $G$ kończy dowód twierdzenia Königa.

**Ćwiczenie 17.** Wykorzystując twierdzenie Königa, rozwiąż następujące zadanie:
Mamy punkty na płaszczyźnie. Ile minimalnie trzeba poprowadzić prostych pionowych lub poziomych, żeby każdy punkt leżał na którejś z nich?

# 7 Przecinanie się obiektów geometrycznych na płaszczyźnie

Niezależnie, jakim problemem geometrycznym się zajmujemy, mając pewne obiekty na płaszczyźnie, mogą nas interesować pewne szczególne punkty. Naturalnie, podstawowym rodzajem punktów szczególnych są przecięcia obiektów, tzn. miejsca gdzie co najmniej dwa obiekty się spotykają. Zajmijmy się więc sposobami znajdowania takich punktów.

## 7.1 Przecięcie dwóch prostych

Proste to najpopularniejsze i najprostsze z obiektów, których przecięcie może nas interesować. Przecięcie dwóch prostych sprowadza się do rozwiązania układu dwóch równań, złożonego z równań opisujących te proste. Jeśli przyjmiemy, że proste to $A_1x + B_1y + C_1 = 0$ oraz $A_2x + B_2y + C_2 = 0$, to:

- jeśli $A_1B_2 = A_2B_1$ oraz $A_1C_2 = A_2C_1$, to te dwie proste są jednakowe, a więc ich przecięcie jest im równe
- jeśli $A_1B_2 = A_2B_1$ ale $A_1C_2 \neq A_2C_1$, to te dwie proste są równoległe, ale się nie pokrywają i w takim wypadku nie ma punktów wspólnych
- jeśli $A_1B_2 \neq A_2B_1$, to jest dokładnie jeden punkt przecięcia i jest nim

$$\left(\frac{C_1B_2 - C_2B_1}{A_1B_2 - A_2B_1}, \frac{C_1A_2 - C_2A_1}{B_1A_2 - B_2A_1}\right).$$

## 7.2 Przecięcie dwóch odcinków

Odcinek jest fragmentem prostej ograniczonym z obu stron, naturalnym więc pomysłem na znalezienie przecięcia dwóch odcinków jest przecięcie dwóch prostych, które te odcinki zawierają. Nie jest to jednakże pełne rozwiązanie. Może się bowiem zdarzyć, że odcinki nie przecinają się w ogóle, a ich proste — tak.

Niech jeden z odcinków ma końce $(x_1, y_1)$ i $(x_2, y_2)$, a drugi: $(p_1, q_1)$ i $(p_2, q_2)$. W takim razie, ich proste są opisane równaniami $A_1x + B_1y + C_1 = 0$ oraz $A_2x + B_2y + C_2 = 0$, gdzie:

$$\begin{aligned} A_1 &= y_1 - y_2 & \qquad A_2 &= q_1 - q_2 \\ B_1 &= x_2 - x_1 & \qquad B_2 &= p_2 - p_1 \\ C_1 &= x_1y_2 - x_2y_1 & \qquad C_2 &= p_1q_2 - p_2q_1 \end{aligned}$$

Postępujemy teraz tak, jakbyśmy przecinali te dwie proste, przy czym:

- gdy są tą samą prostą, to ich przecięcie jest puste, jednym punktem lub odcinkiem; aby to sprawdzić przyporządkowujemy punktom współrzędne na tej prostej tak, że odcinki mają końce $a_1 \leqslant a_2$ oraz $b_1 \leqslant b_2$, odpowiednio, a następnie rozpatrujemy odcinek

$$[\max(a_1, b_1), \min(a_2, b_2)].$$

- gdy są równoległe ale różne, to odcinki się nie przecinają
- gdy nie są równoległe, to proste przecinają się w jednym punkcie a odcinki albo przecinają się w tym właśnie punkcie, albo wcale — aby to sprawdzić, pytamy czy tenże punkt należy do każdego z odcinków.

**Ćwiczenie 18.** Jak zamienić współrzędne punktu płaszczyzny $(x_1, y_1)$ leżącego na prostej $Ax + By + C = 0$ na jedną współrzędną opisującą położenie na tej prostej?

## 7.3 Przecięcie prostej i okręgu

Tym razem załóżmy, że prosta ma równanie $y = px + q$, a okrąg: $(x-a)^2 + (y-b)^2 = r^2$. Wtedy mamy:

$$(x-a)^2 + (px+q-b)^2 = r^2$$

$$x^2(1+p^2) + x(2pq - 2pb - 2a) + \left(a^2 + (q-b)^2 - r^2\right) = 0$$

Takie równanie kwadratowe umiemy już rozwiązać. Mamy $\Delta = 4r^2(1+p^2) - 4(ap+q-b)^2$, i w zależności od tego, czy $\Delta$ jest ujemna, równa zero czy dodatnia, mamy zero, jeden lub dwa punkty przecięcia.

**Ćwiczenie 19.** Co należy zrobić, gdy prosta jest pionowa (o równaniu $x = c$)?

## 7.4 Przecięcie dwóch okręgów

Układ równań kwadratowych może nie być prosty do rozwiązania. W przypadku układu

$$(x-a_1)^2 + (y-b_1)^2 = r_1^2$$

$$(x-a_2)^2 + (y-b_2)^2 = r_2^2$$

jest o tyle łatwiej, że po odjęciu tych równań stronami otrzymujemy

$$(2a_1 - 2a_2)x + (2b_1 - 2b_2)y + (a_2^2 - a_1^2 + b_2^2 - b_1^2 + r_1^2 - r_2^2) = 0$$

co jest równaniem prostej, do której będą należały wszystkie punkty przecięcia. Wystarczy więc przeciąć tę właśnie prostą z dowolnym z tych dwóch okręgów i otrzymane punkty będą właśnie punktami przecięcia okręgów. Jedyna wątpliwość nastaje, gdy $a_1 = a_2$ oraz $b_1 = b_2$, bo wtedy otrzymane równanie nie opisuje prostej.

**Ćwiczenie 20.** Czym jest przecięcie okręgów w takim wypadku?

# 8 Położenie punktu względem obiektów płaszczyzny

## 8.1 Względem prostej

Tym razem niech prosta będzie opisana przez dwa różne jej punkty, $(x_1, y_1)$ i $(x_2, y_2)$. Punkt $(p, q)$ leży po którejś ze stron tej prostej, w zależności od znaku iloczynu wektorowego wektorów o początkach w $(x_1, y_1)$ i końcach w $(x_2, y_2)$ i $(p, q)$. W szczególności, gdy iloczyn ten jest zerem, punkt leży na prostej.

**Ćwiczenie 21.** Mając daną prostą o równaniu $Ax + By + C = 0$ oraz dwa punkty $(p_1, q_1)$ i $(p_2, q_2)$, sprawdź czy leżą one po dwóch różnych stronach tej prostej.

## 8.2 Względem okręgu

Punkt $(x_1, y_1)$ leży względem okręgu o promieniu $r$ i środku w $(a, b)$:

- wewnątrz okręgu, gdy $(x_1 - a)^2 + (y_1 - b)^2 < r^2$
- na okręgu, gdy $(x_1 - a)^2 + (y_1 - b)^2 = r^2$
- na zewnątrz okręgu, gdy $(x_1 - a)^2 + (y_1 - b)^2 > r^2$

KAPITAŁ LUDZKI NARODOWA STRATEGIA SPÓJNOŚCI
WARSZAWSKA WYŻSZA SZKOŁA INFORMATYKI
UNIA EUROPEJSKA EUROPEJSKI FUNDUSZ SPOŁECZNY
Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego

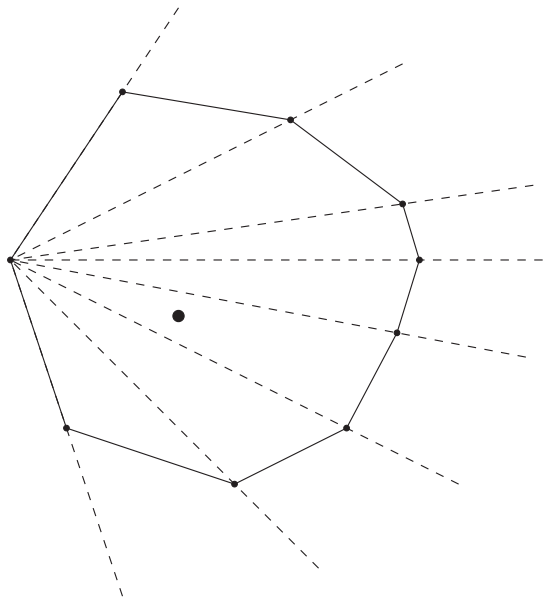
## 8.3 Względem wielokąta

Najpopularniejszą metodą sprawdzania, czy punkt leży wewnątrz wielokąta jest wzięcie półprostej o początku w tym punkcie. Jeśli półprosta ta przecina wielokąt parzystą liczbę razy, to punkt leży na zewnątrz, a jeśli nieparzystą — wewnątrz. O ile sama metoda jest prosta, należy uważać przy jej implementacji. Problemów nastręczają przede wszystkim wierzchołki wielokąta leżące na naszej półprostej. Najlepiej więc wybrać taką półprostą, żeby po prostu wierzchołków wielokąta na niej nie było. Co więcej, półprostą można traktować jako odcinek, którego drugi koniec jest po prostu bardzo daleko. Przykładowo, jeśli interesuje nas punkt $(x, y)$ to często dobrym pomysłem jest wziąć za „półprostą" odcinek $[(x, y), (x + 1, y + M)]$. Dla odpowiednio dużego $M$, taki odcinek przecina tyle samo, co półprosta go zawierająca, a także nie zawiera żadnych wierzchołków wielokąta.

Takie sprawdzenie wymaga liniowej względem ilości boków wielokąta liczby operacji.

### Względem wielokąta wypukłego

Naturalnie, skoro potrafimy rozwiązać ten problem dla dowolnego wielokąta, potrafimy także zrobić to dla wielokąta wypukłego. W tym szczególnym przypadku możliwe jest jednak znacznie efektywniejsze rozwiązanie. Niech będzie wyróżniony jeden z wierzchołków wielokąta, $A$, a pozostałe ułożone w kolejności zgodnej z ruchem wskazówek zegara. Poprowadźmy z $A$ wszystkie przekątne, tak aby podzielić wielokąt na trójkąty. Zauważmy, że możemy wyszukać binarnie, pomiędzy którymi dwoma prostymi (zawierającymi przekątne) znajduje się interesujący nas punkt, używając iloczynu wektorowego do sprawdzenia po której stronie pewnej przekątnej się on znajduje. Na koniec trzeba już tylko sprawdzić, czy zawiera się w pewnym trójkącie co można zrobić w czasie stałym (choćby metodą z półprostą i obliczeniem parzystości liczby przecięć).



Całość działa w czasie $O(\log n)$, gdzie $n$ jest ilością boków wielokąta, przy założeniu, że wierzchołki wielokąta podane są po kolei.

## 8.4 Znajdowanie najdalszych punktów w danym zbiorze

Mamy dany zbiór $n$ punktów na płaszczyźnie i chcemy znaleźć takie dwa z nich, żeby odległość pomiędzy nimi była jak największa.

Zauważmy najpierw, że oba te punkty będą należały do wypukłej otoczki całego zbioru. Gdyby bowiem któryś koniec nie należał do otoczki, moglibyśmy go zamienić na pewien punkt otoczki tak, żeby nadal odległość była maksymalna (albo wręcz większa, co znaczyłoby, że dotąd nie mieliśmy dobrego rozwiązania).

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI
Warszawska Wyższa Szkoła Informatyki
UNIA EUROPEJSKA
EUROPEJSKI FUNDUSZ SPOŁECZNY
Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego

Zacznijmy więc od znalezienia wypukłej otoczki dla tego zbioru. Następnie, ustalmy punkt $a$, jeden z wierzchołków otoczki, i poszukajmy dla niego najdalszego wierzchołka, $b$. Będziemy teraz przesuwać wskaźnik $a$ zgodnie z ruchem wskazówek zegara na kolejne wierzchołki otoczki. Wskaźnik $b$ (aktualnie najdalszy punkt od $a$) także będzie się przesuwał i także zgodnie z ruchem wskazówek zegara, a kiedy $a$ zatoczy pełen obrót po otoczce — $b$ uczyni to samo, co oznacza że łączny czas takiego przeszukiwania będzie liniowy. Zauważmy, że w każdym momencie, aby znaleźć dla danego $a$ optymalne $b$, przesuwamy $b$ tak długo zgodnie z ruchem wskazówek zegara, jak długo powiększa to odległość. Taka metoda działa tylko dlatego, że otoczka jest wielokątem wypukłym.

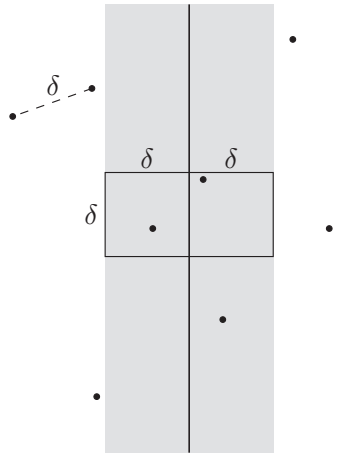
Całe rozwiązanie działa w czasie $O(n \log n)$, który jest potrzebny na znalezienie otoczki wypukłej.

## 8.5 Znajdowanie najbliższych punktów w danym zbiorze

Tym razem w danym zbiorze $n$ punktów chcemy znaleźć takie dwa, które leżą najbliżej siebie. Problem znajdowania najbliższych punktów wydaje się podobny do poprzedniego, jednak jego rozwiązanie jest zupełnie inne.

Użyjemy metody „dziel i zwyciężaj". Weźmy taką pionową prostą na płaszczyźnie, która dzieli nasz zbiór na pół, tzn. połowa punktów leży po lewej, a połowa po prawej stronie (blisko połowa, gdyby było ich nieparzyście wiele). Punkty które leżą dokładnie na prostej przydzielamy do stron wedle uznania. Rozwiązujemy każdy z dwóch mniejszych podproblemów niezależnie, a następnie przyjmujemy $\delta$ jako mniejszy z dwóch uzyskanych wyników.

Wynikiem dla całego zbioru jest albo $\delta$, albo też jakaś mniejsza liczba, jeśli istnieją punkt z lewego i punkt z prawego zbioru, które leżą od siebie w odległości mniejszej niż $\delta$. Zauważmy ponadto, że takie punkty muszą leżeć w pionowym pasie szerokości $2\delta$ którego środkiem jest wybrana wcześniej prosta.




Blisko położone punkty z różnych stron prostej

Co więcej, takie dwa punkty muszą leżeć w pewnym prostokącie — wycinku tegoż pasa o wysokości $\delta$. W takim prostokącie może być jednak co najwyżej 8 punktów, w każdym z kwadratów po 4, gdyż jak wiemy, wyniki dla lewej i prawej części zbioru osobno są równe co najmniej $\delta$. Stąd, wystarczy posortować po współrzędnej $y$ punkty leżące w pasie szerokości $2\delta$ wokół naszej prostej, a następnie sprawdzić odległości z każdego z nich do siedmiu kolejnych. W ten sposób uzyskamy wynik dla całego zbioru. Widzimy, że naiwna implementacja działa w czasie $O(n \log^2 n)$ (jest to rozwiązanie równości $T(n) = 2T(\frac{n}{2}) + O(n \log n)$).

**Ćwiczenie 22.** Jak zaimplementować powyższe rozwiązanie aby działało w czasie $O(n \log n)$?
**Podpowiedź:** głównym kosztem każdej fazy jest sortowanie punktów pasa po współrzędnej $y$. Spróbuj wyeliminować ten składnik sortując punkty tylko raz na początku.

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI
Warszawska Wyższa Szkoła Informatyki
UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY
Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego

## Literatura


1. Banachowski L., Diks K., Rytter W., *Algorytmy i struktury danych*, WNT, Warszawa 2003
2. Cormen T.H., Leiserson C.E., Rivest R.L., Stein C., *Wprowadzenie do algorytmów*, WNT, Warszawa 2004
3. Diks K., Malinowski A., Rytter W. Waleń T., *Algorytmy i struktury danych*, `http://wazniak.mimuw.edu.pl/index.php?title=Algorytmy_i_struktury_danych`
4. Graham R.L., Knuth D.E., Patashnik O., *Matematyka konkretna*, PWN, Warszawa 2002
5. Lipski W., *Kombinatoryka dla programistów*, WNT, Warszawa 2004
6. *Młodzieżowa Akademia Informatyczna*, `http://main.edu.pl/`
7. *Standard Template Library Programmer's Guide*, `http://www.sgi.com/tech/stl/`
8. Sysło M.M., Deo N., Kowalik J.S, *Algorytmy optymalizacji dyskretnej*, PWN, Warszawa 1999