# Metoda Dziel i Zwyciężaj.

IIUWr. II rok informatyki. *Opracował: Krzysztof Loryś*

## 8 Schemat ogólny.

Algorytmy skonstruowane metodą ”dziel i zwyciężaj” składają się z trzech zasadniczych kroków:

1. transformacja danych $x$ na dane $x_1, \ldots, x_k$ o mniejszym rozmiarze;
2. rozwiązanie problemu dla danych $x_i$ $(i = 1, \ldots, k)$;
3. obliczenie rozwiązania dla danych $x$ na podstawie wyników otrzymanych w punkcie 2.

W naturalny sposób implementowane są jako procedury rekurencyjne:

**function** $DiZ(x)$
    **if** $x$ jest małe lub proste **then** **return** $AdHoc(x)$
    1. przekształć $x$ na $x_1, \ldots, x_k$ o mniejszym rozmiarze niż $x$
    2. **for** $i \leftarrow 1$ **to** $k$ **do** $y_i \leftarrow DiZ(x_i)$
    3. na podstawie $y_1 \ldots y_k$ oblicz rozwiązanie $y$ dla $x$
    **return** $y$

gdzie $AdHoc$ jest algorytmem używanym do rozwiązania problemu dla "łatwych"danych.

## 9 Ważne równanie rekurencyjne.

**Twierdzenie 1** *Niech $a, b, c \in \mathcal{N}$. Rozwiązaniem równania rekurencyjnego*

$$T(n) = \begin{cases} b & \text{dla } n = 1 \\ aT(n/c) + bn & \text{dla } n > 1 \end{cases}$$

*dla $n$ będących potęgą liczby $c$ jest*

$$T(n) = \begin{cases} \Theta(n) & \text{jeżeli } a < c, \\ \Theta(n \log n) & \text{jeżeli } a = c, \\ \Theta(n^{\log_c a}) & \text{jeżeli } a > c \end{cases}$$

Dowód. Niech: $n = c^k$, czyli $k = \log_c n$. Stosując metodę podstawiania otrzymujemy:

$$T(n) = a^k bn/c^k + a^{k-1}bn/c^{k-1} + \ldots + abn/c + bn = bn \sum_{i=0}^{k} \left(\frac{a}{c}\right)^i.$$

Rozważamy 3 przypadki:

1. $a < c$
   Wówczas $\frac{a}{c} < 1$, więc szereg $\sum_{i=0}^{k} \left(\frac{a}{c}\right)^i$ jest zbieżny do pewnego $m \in \mathcal{R}^+$. Stąd $T(n) = bmn$.

2. $a = c$
   Wówczas $\frac{a}{c} = 1$, więc $\sum_{i=0}^{k} \left(\frac{a}{c}\right)^i = k + 1 = \Theta(\log_c n)$. Stąd $T(n) = \Theta(n \log_c n)$.

3. $a > c$
   Wówczas $\frac{a}{c} > 1$, więc:

$$T(n) = bc^k \sum_{i=0}^{k} \left(\frac{a}{c}\right) = bc^k \frac{\left(\frac{a}{c}\right)^{k+1} - 1}{\left(\frac{a}{c}\right) - 1} =$$

$$b\frac{a^{k+1} - c^{k+1}}{a - c} = \frac{ba}{a-c} a^{\log_c n} - \frac{cb}{a-c} n = \frac{ba}{a-c} a^{\log_c n} - O(n).$$

   Ponieważ $n$ jest $O(a^{\log_c n}) = O(n^{\log_c a})$, więc $T(n) = \Theta(a^{\log_c n})$.

$\square$

INTERPRETACJA: Twierdzenie określa złożoność algorytmów opartych na strategii dziel i zwyciężaj, które:

- redukują problem dla danych o rozmiarze $n$ do rozwiązania tego problemu dla $a$ zestawów danych, każdy o rozmiarze $n/c$.
- wykonują kroki 1 i 3 schematu ogólnego w czasie liniowym.

# 10 Przykłady.

## 10.1 Sortowanie

Nasze przykłady rozpoczniemy od zaprezentowania dwóch strategii Dziel i Zwyciężaj dla problemu sortowania.

PROBLEM:

*Dane:* tablica $T[1..n]$ elementów z uporządkowanego liniowo uniwersum

*Zadanie:* uporządkować $T$

### 10.1.1 Strategia 1: Sortowanie przez scalanie.

Strategia ta oparta jest na tym, że dwa uporządkowane ciągi potrafimy szybko (w czasie liniowym) scalić w jeden ciąg. Aby posortować tablicę wystarczy więc podzielić ją na dwie części, niezależnie posortować każdą z części a następnie scalić je.

```
procedure mergesort(T[1..n])
   if n jest małe then insert(T)
   else
         X[1 .. ⌈n/2⌉] ← T[1 .. ⌈n/2⌉]
         Y[1 .. ⌊n/2⌋] ← T[1 + ⌈n/2⌉ .. n]
         mergesort(X); mergesort(Y)
         T ← merge(X, Y)
```

Czas działania algorytmu wyraża się równaniem $t(n) = t(\lceil n/2 \rceil) + t(\lfloor n/2 \rfloor) + \Theta(n)$, którego rozwiązaniem jest $t(n) = \Theta(n \log n)$. Jak pó"xniej pokażemy jest to asymptotycznie optymalny czas działania algorytmów sortowania.

Mankamentem tego algorytmu jest fakt wykorzystywania dodatkowych tablic (poza tablicą wejściową) podczas scalania ciągów. Niestety nie jest znany sposób usunięcia tej wady. Co prawda znane są metody "scalania w miejscu" w czasie liniowym, lecz są one bardzo skomplikowane, co sprawia, że stałe w funkcjach liniowych ograniczających czas działania są nieakceptowalnie wielkie.

Przy okazji prezentacji tego algorytmu chcemy zwrócić uwagę na niezwykle ważny, a często zaniedbywany, aspekt implementacji algorymów typu Dziel i Zwyciężaj: staranne dobranie progu na rozmiar danych, poniżej którego nie opłaca się stosować algorytmu rekurencyjnie. Przykładowo powyżej zastosowaliśmy dla małych danych algorytm *insert*. Może on wymagać czasu kwadratowego, ale jest bardzo prosty i łatwy w implementacji, dzięki czemu w praktyce jest on dla małych danych szybszy od rekurencyjnej implementacji sortowania przez scalanie. Teoretyczne wyliczenie wartości progu jest zwykle trudne i zawodne. Jego wartość zależy bowiem także od efektywności implementacji a nawet od typu maszyny, na którym program będzie wykonywany. Dlatego, jeśli zależy nam na optymalnym "dostrojeniu" programu (na przykład z tego powodu, że jest on bardzo często wykonywaną procedurą, ważącą na efektywności całego systemu), powinniśmy wyznaczyć ten próg poprzez starannie dobrane eksperymenty.

### 10.1.2 Strategia 2: Quicksort

Najistotniejszym krokiem algorytmu sortowania przez scalanie jest krok 3 (łączenie wyników podproblemów). Natomiast krok 1 jest trywialny i sprowadza się do wyliczenia indeksu środka tablicy (ze względów technicznych połączyliśmy go powyżej z kopiowaniem elementów do tablic roboczych).

W algorytmie *Quicksort* sytuacja jest odwrotna: istotnym krokiem jest krok 1. Polega on na podziale elementów tablicy na dwa ciągi, takie, że każdy element pierwszego z nich jest nie mniejszy od każdego elementu drugiego z nich. Jeśli teraz każdy z tych ciągów zostanie niezależnie posortowany, to krok 3 staje się zbyteczny.

```
procedure Quicksort(T[1..n])
   if n jest małe then insert(T)
   else
        wybierz element dzielący x
        (* niech k równa się liczbie elementów tablicy T nie większych od x*)
        przestaw elementy tablicy T tak, że ∀_{i≤k} T[i] ≤ x
        Quicksort(T[1..k]); Quicksort(T[(k+1)..n]);
```

Analizie złożoności algorytmu *Quicksort* poświęcimy oddzielny wykład. Wówczas omówimy także sposoby implementacji kroku 1.

## 10.2 Mnożenie bardzo dużych liczb.

PROBLEM:

*Dane:* liczby naturalne $a$ i $b$

komentarz: liczby $a$ i $b$ są długie.

*Wynik:* iloczyn $a \cdot b$

Dla prostoty opisu przyjmijmy, że obydwie liczby mają tę samą długość (równą $n = 2^k$). Narzucający się algorytm oparty na strategii Dziel i Zwyciężaj polega na podziale $n$-bitowych czynników na części $n/2$-bitowe, a następnie odpowiednim wymnożeniu tych części.

Niech $a = a_1 \cdot 2^s + a_0$ i $b = b_1 \cdot 2^s + b_0$, gdzie $s = n/2$; $0 \leq a_1, a_0, b_1, b_0 < 2^s$. Iloczyn $a \cdot b$ możemy teraz zapisać jako

$$ab = c_2 \cdot 2^{2s} + c_1 \cdot 2^s + c_0,$$

gdzie $c_2 = a_1 b_1$; $c_1 = a_0 b_1 + a_1 b_0$; $c_0 = a_0 b_0$.

Jak widać jedno mnożenie liczb $n$-bitowych można zredukować do czterech mnożeń liczb $n/2$-bitowych, dwóch mnożeń przez potęgę liczby 2 i trzech dodawań. Zarówno dodawania jak i mnożenia przez potęgę liczby 2 można wykonać w czasie liniowym. Taka redukcja nie prowadzi jednak do szybszego algorytmu - czas działania wyraża się wzorem $T(n) = 4T(n/2) + \Theta(n)$, którego rozwiązaniem jest $T(n) = \Theta(n^2)$. Aby uzyskać szybszy algorytm musimy potrafić obliczać współczynniki $c_2, c_1, c_0$ przy użyciu trzech mnożeń liczb $n/2$-bitowych. Uzyskujemy to przez zastąpienie dwóch mnożeń podczas obliczania $c_1$ jednym mnożeniem i dwoma odejmowaniami:

$$c_1 = (a_1 + a_0)(b_1 + b_0) - c_0 - c_2.$$

$multiply(a, b)$
  $n \leftarrow max(|a|, |b|)$  (* $|x|$ oznacza długość liczby $x$ *)
  **if** $n$ jest małe **then** pomnóż $a$ i $b$ klasycznym algorytmem
      **return** obliczony iloczyn
  $p \leftarrow \lfloor n/2 \rfloor$
  $a_1 \leftarrow \lfloor a/2^p \rfloor$; $a_0 \leftarrow a$ **mod** $2^p$
  $b_1 \leftarrow \lfloor b/2^p \rfloor$; $b_0 \leftarrow b$ **mod** $2^p$
  $z \leftarrow multiply(a_0, b_0)$
  $y \leftarrow multiply(a_1 + a_0, b_1 + b_0)$
  $x \leftarrow multiply(a_1, b_1)$;
  **return** $2^{2p}x + 2^p(y - x - z) + z$

**Fakt 2** *Złożoność czasowa powyższego algorytmu wynosi* $O(n^{\log 3})$.

DOWÓD: (Zakładamy, że $a$ i $b$ są liczbami $n$ - bitowymi i $n$ jest potęgą liczby 2.)
Wystarczy pokazać, że czas działania algorytmu wyraża się wzorem:

$$T(n) = \begin{cases} k & \text{dla } n = 1 \\ 3T(n/2) + \Theta(n) & \text{dla } n > 1 \end{cases}$$

Jedyną wątpliwość może budzić fakt, że wskutek przeniesienia liczby $a_1 + a_0$ i $b_1 + b_0$ mogą być $(n/2) + 1$-bitowe. W takiej sytuacji $a_1 + a_0$ zapisujemy w postaci $a'2^{n/2} + a''$, a $b_1 + b_0$ w postaci $b'2^{n/2} + b''$, gdzie $a'$ i $b'$ są bitami z pozycji $(n/2) + 1$ liczb $a$ i $b$, a $a''$ i $b''$ są złożone z pozostałych bitów. Obliczenie $y$ możemy teraz przedstawić jako:

$$(a_1 + a_0)(b_1 + b_0) = a'b'2^n + (a'b'' + a''b')2^{n/2} + a''b''$$

Jedynym składnikiem wymagającym rekurencyjnego wywołania jest $a''b''$ (obydwa czynniki są $n/2$-bitowe). Pozostałe mnożenia wykonujemy w czasie $O(n)$, ponieważ jednym z czynników jest pojedynczy bit ($a'$ lub $b'$) lub potęga liczby 2.
Tak więc przypadek, gdy podczas obliczania $y$ występuje przeniesienie przy dodawaniu, zwiększa złożoność jedynie o pewną stałą, nie zmieniając klasy złożoności algorytmu. □

### 10.2.1 Podział na więcej części

Pokażemy teraz, że powyższą metodę można uogólnić. Niech $k \in \mathcal{N}$ będzie dowolną stałą. Liczby $a$ i $b$ przedstawiamy jako

$$a = \sum_{i=0}^{k-1} a_i \cdot 2^{in/k} \qquad b = \sum_{i=0}^{k-1} b_i \cdot 2^{in/k}$$

gdzie wszystkie $a_i$ oraz $b_i$ są liczbami co najwyżej $n/k$-bitowymi. Naszym zadaniem jest policzenie liczb $c_0, c_1, \ldots, c_{2k}$ takich, że dla $j = 0, \ldots, 2k$:

$$c_j = \sum_{r=0}^{j} a_r b_{j-r}.$$

Niech liczby $w_1, \ldots, w_{2k+1}$ będą zdefiniowane w następujący sposób:

$$w_t = (\sum_{i=0}^{k-1} a_i t^i) \cdot (\sum_{i=0}^{k-1} b_i t^i).$$

Łatwo sprawdzić, że $w_t = \sum_{j=0}^{2k} c_j t^j$. Otrzymaliśmy więc układ $2k+1$ równań z $2k+1$ niewiadomymi $c_0, c_1, \ldots, c_{2k}$. Fakt ten możemy zapisać jako

$$U \cdot [c_0, c_1, \ldots, c_{2k-1}, c_{2k}]^T = [w_1, w_2, \ldots, w_{2k}, w_{2k+1}]^T,$$

gdzie elementy macierzy $U = u_{ij}$ są równe $u_{ij} = i^j$ (dla $i = 1, \ldots, 2k+1$ oraz $j = 0, \ldots, 2k$). Ponieważ $U$ jest macierzą nieosobliwą (jest macierzą Vandermonde'a), istnieje rozwiązanie powyższego układu. Jeśli $w_1, \ldots, w_{2k+1}$ potraktujemy jako wartości symboliczne, to rozwiązując ten układ wyrazimy $c_i$ jako kombinacje liniowe tych wartości. To stanowi podstawę dla następującego algorytmu:

1. Oblicz rekurencyjnie wartości $w_1, \ldots, w_{2k+1}$.
2. Oblicz wartości $c_0, \ldots, c_{2k}$.
3. **return** $\sum_{i=0}^{2k} c_i 2^{in/k}$.

**Fakt 3** *Powyższy algorytm działa w czasie* $\Theta(n^{\log_k(2k+1)})$.

Pomijamy formalny dowód tego faktu. Wynika on z tego, że w kroku 1 wywołujemy $2k+1$ razy rekurencyjnie funkcję dla danych o rozmiarze $n/k$ oraz z tego, że kroki 2 i 3 wykonują się w czasie liniowym.

Jakkolwiek zwiększając $k$ możemy z wykładnikiem nad $n$ dowolnie blisko przybliżyć się do 1, to jednak zauważmy, że otrzymane algorytmy mają znaczenie czysto teoretyczne. Stałe występujące w kombinacjach obliczanych w punkcie 2 już dla niewielkich wartości $k$ są bardzo duże i sprawiają, że algorytm działa szybciej od klasycznego mnożenia pisemnego dopiero dla danych o astronomicznie wielkich rozmiarach.

## 10.3 Równoczesne znajdowanie minimum i maksimum w zbiorze.

PROBLEM: Minmax
*Dane:* zbiór $S = \{a_1, a_2, \ldots, a_n\}$ [1]
*Wynik:* $min\{a_i \mid i = 1, \ldots, n\}$ $max\{a_i \mid i = 1, \ldots, n\}$

KOMENTARZ: Ograniczamy się do klasy algorytmów, które danych wejściowych używają jedynie w operacjach porównania. Interesują nas dwa zagadnienia:

- znalezienie algorytmu z tej klasy, który rozwiązuje problem Minmax, używając jak najmniejszej liczby porównań.
- dokładne wyznaczenie dolnej granicy na liczbę porównań.

[1] Termin ”zbiór” jest użyty tu w dość swobodnym znaczeniu. W zasadzie $S$ jest multizbiorem - elementy mogą się w nim powtarzać. O ile jednak nie będzie to "xródłem dwuznaczności, w przyszłości termin ”zbiór” będziemy stosować także w odniesieniu do multizbiorów.

Proste podejście do tego problemu polega na tym, by szukane liczby znale"xć niezależnie, np. najpierw minimum a potem maksimum. Takie rozwiązanie wymaga $2n-2$ porównań. Jego nieoptymalność wynika z tego, że algorytm podczas szukania maksimum nie wykorzystuje w żaden sposób informacji jakie nabył o elementach zbioru $S$ w czasie wyszukiwania minimum. W szczególności w trakcie szukania maksimum będą brały udział w porównaniach te elementy $S$-a, które podczas szukania minimum były porównywane z większymi od siebie elementami, a więc nie mogą być maksimum. To spostrzeżenie prowadzi do następującego algorytmu:

$MinMax1(S)$
  $S_m \leftarrow S_M \leftarrow \emptyset$
  **for** $i = 1$ **to** $n$ **div** 2 **do**
    porównaj $a_i$ z $a_{n-i+1}$; mniejszą z tych liczb wstaw do zbioru $S_m$, a większą - do zbioru $S_M$
  $m \leftarrow min\{a \mid a \in S_m\}$
  $M \leftarrow max\{a \mid a \in S_M\}$
  **if** $n$ parzyste **then return** $(m, M)$
    **else return** $(min(m, a_{\lceil n/2 \rceil}), max(M, a_{\lceil n/2 \rceil}))$

**Fakt 4** *Algorytm MinMax1 wykonuje* $\lceil \frac{3}{2}n - 2 \rceil$ *porównań na elementach zbioru* $S$.

Dowód. Niech $n = 2m$. W pętli **for** wykonywanych jest $m$ porównań, a w dwóch następnych wierszach po $m-1$ porównań. W sumie mamy $3m - 2 = \lceil \frac{3}{2}n - 2 \rceil$ porównań.

Gdy $n = 2m+1$, algorytm najpierw znajduje minimum i maksimum w zbiorze $S \setminus \{a_{\lceil n/2 \rceil}\}$. Zbiór ten ma $2m$ elementów, a więc liczba wykonanych porównań wynosi $3m-2$. Ostatnia instrukcja wymaga dwóch porównań, co w sumie daje $3m$ porównań. Jak łatwo sprawdzić $3m = 3(n-1)/2 = \lceil 3(n-1)/2 - 1/2 \rceil = \lceil \frac{3}{2}n - 2 \rceil$. □

Inne podejście polega na zastosowaniu strategii Dziel i Zwyciężaj: zbiór $S$ dzielimy na dwie części, w każdej części znajdujemy minimum i maksimum, jako wynik dajemy mniejsze z minimów i większe z maksimów. Poniższa, niestaranna implementacja tego pomysłu nie osiąga jednak liczby porównań algorytmu $MinMax1$ i powinna stanowić ostrzeżenie przed niefrasobliwym implementowaniem niedopracowanych algorytmów. Poprawienie tej implementacji pozostawiamy jako zadanie na ćwiczenia.

**Procedure** $MinMax2(S)$
  **if** $|S|=1$ **then** **return** $(a_1, a_1)$
  **else**
    **if** $|S|=2$ **then** **return** $(\max(a_1, a_2), \min(a_1, a_2))$
    **else**
      podziel $S$ na dwa równoliczne (z dokładnością do jednego elementu) podzbiory $S_1, S_2$
      $(max1, min1) \leftarrow MinMax2(S_1)$
      $(max2, min2) \leftarrow MinMax2(S_2)$
      **return** $(\max(max1, max2), \min(min1, min2))$

### 10.3.1 Granica dolna.

Algorytm *MinMax1* wyznacza górną granicę na złożoność problemu jednoczesnego znajdowania minimum i maksimum. Teraz pokażemy, że ta granica jest także granicą dolną, a więc dokładnie wyznaczymy złożoność problemu.

**Twierdzenie 2** *Każdy algorytm rozwiązujący powyższy problem (i używający elementów zbioru $S$ jedynie w porównaniach) wykonuje co najmniej $\lceil \frac{3}{2}n - 2 \rceil$ porównań.*

DOWÓD: Rozważmy następującą grę między algorytmem a złośliwym adwersarzem:

- Sytuacja początkowa: adwersarz twierdzi, że zna trudny dla algorytmu zbiór $S$, tj. taki, dla którego wskazanie przez algorytm minimum i maksimum będzie wymagało wykonania co najmniej $\lceil \frac{3}{2}n - 2 \rceil$ porównań. Algorytm nie zna $S$; wie tylko, że liczy on $n$ elementów.
- Cel gry
  - algorytmu: wskazanie indeksów elementów minimalnego i maksymalnego w zbiorze $S$ przy użyciu mniej niż $\lceil \frac{3}{2}n - 2 \rceil$ porównań;
  - adwersarza: zmuszenie algorytmu do zadania co najmniej $\lceil \frac{3}{2}n - 2 \rceil$ porównań.
- Ruchy
  - algorytmu: pytanie o porównanie dwóch elementów ze zbioru $S$;
  - adwersarza: odpowied"x na to pytanie.
- Koniec gry następuje, gdy algorytm wskaże minimum i maksimum w $S$. Wówczas adwersarz ujawnia zbiór $S$.

Tezę twierdzenia udowodnimy, jeśli pokażemy, że adwersarz zawsze, niezależnie od algorytmu, posiada strategię wygrywającą.

Strategia dla adwersarza:

- W trakcie gry adwersarz dzieli $S$ na 4 rozłączne zbiory:

  $A = \{i \mid a_i$ jeszcze nie był porównywany $\}$,
  $B = \{i \mid a_i$ wygrał już jakieś porównanie i nie przegrał żadnego $\}$,
  $C = \{i \mid a_i$ przegrał juz jakieś porównanie i nie wygrał żadnego$\}$,
  $D = \{i \mid a_i$ wygrał już jakieś porównanie i jakieś już przegrał $\}$.

  Początkowo oczywiście $|A| = n$ oraz $|B| = |C| = |D| = 0$.

- Adwersarz rozpoczyna grę z dowolnymi kandydatami na wartości elementów $a_i$. W trakcie gry będzie, w razie konieczności, modyfikował te wartości, tak by spełniony był warunek

  $$(*) \qquad \forall_{a\in A}\forall_{b\in B}\forall_{c\in C}\forall_{d\in D}\ \ b > d > c \text{ oraz } b > a > c.$$

  Zauważ, że wystarczy w tym celu zwiększać wartości elementów o indeksach z $B$ i zmniejszać wartości elementów o indeksach z $C$. Takie zmiany są bezpieczne dla adwersarza, ponieważ pozostawiają prawdziwymi jego odpowiedzi na dotychczasowe pytania.

**Fakt 5** *Powyższa strategia adwesarza jest zawsze wygrywająca.*

Dowód faktu: W trakcie gry wszystkie elementy przechodzą ze zbioru $A$ do $B$ lub $C$, a dopiero stąd do zbioru $D$. Ponadto dla danych spełnających (*):

- jedno porównanie może usunąć co najwyżej dwa elementy ze zbioru $A$,
- dodanie jednego elementu do zbioru $D$ wymaga jednego porównania,
- porównania, w których bierze udział element z $A$, nie zwiększają mocy zbioru $D$.

Dopóki $A$ jest niepusty lub któryś ze zbiorów $B$ lub $C$ zawiera więcej niż jeden element algorytm nie może udzielić poprawnej odpowiedzi. Na opróżnienie zbioru $A$ algorytm potrzebuje co najmniej $\lceil n/2 \rceil$ porównań. Następnych $n-2$ porównań potrzebnych jest na przesłanie wszystkich, poza dwoma, elementów do zbioru $D$. $\square$ (faktu i twierdzenia)

## 11 Dodatek

Poniższa tabela pokazuje w jaki sposób zmieniają się liczności zbiorów po wykonaniu różnych typów porównań (przy założeniu warunku (*)). Typ $XY$ oznacza, iż porównywany jest element zbioru $X$ z elementem zbioru $Y$. Mała litera $x$ oznacza element zbioru $X$.

| Typ porównania | $\|A\|$ | $\|B\|$ | $\|C\|$ | $\|D\|$ | warunek |
|---|---|---|---|---|---|
| AA | -2 | +1 | +1 | bz | |
| AB | -1 | bz | +1 | bz | $a < b$ |
| AC | -1 | +1 | bz | bz | $a > c$ |
| AD | -1 | +1 | bz | bz | $a > d$ |
| | -1 | bz | +1 | bz | $a < d$ |
| BB | bz | -1 | bz | +1 | |
| BC | bz | bz | bz | bz | $b > c$ |
| BD | bz | bz | bz | bz | $b > d$ |
| CC | bz | bz | -1 | +1 | |
| CD | bz | bz | bz | bz | $c < d$ |
| DD | bz | bz | bz | bz | |

# Metoda Dziel i Zwyciężaj (cd.)

IIUWr. II rok informatyki. *Opracował: Krzysztof Loryś*

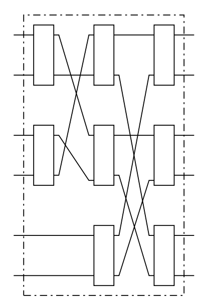
## 3.4 Sieci przełączników

*Przełącznikiem dwustanowym* nazywamy urządzenie o dwóch portach wejściowych i dwóch portach wyjściowych, które:

- w stanie 1 przesyła dane z wejścia $i$ na wyjście $i$ (dla $i = 0, 1$),
- w stanie 2 przesyła dane z wejścia $i$ na wyjście $(i + 1) \bmod 2$ (dla $i = 0, 1$).


Rysunek 1: *Przełącznik dwustanowy*

Łącząc ze sobą przełączniki otrzymujemy *sieci przełączników*, które poprzez różne ustawienia przełączników mogą realizować różne permutacje danych.



Rysunek 2: *Przykład sieci przełączników o 6 wejściach*

Problem:
*Dane:* liczba naturalna $n$.
*Zadanie:* skonstruować sieć $Perm_n$ przełączników realizującą wszystkie permutacje $n$ elementów.

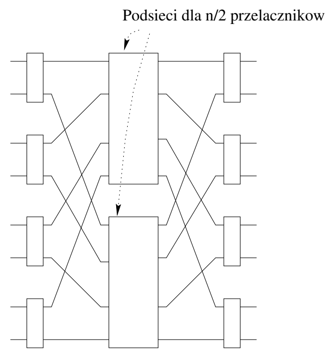
Kryteriami określającymi jakość sieci są:

- liczba przełączników;
- głębokość sieci, tj. długość maksymalnej drogi od portu wejściowego do portu wyjściowego. Długość ta mierzona jest liczbą przełączników znajdujących się na drodze od portu wejściowego sieci do portu wyjściowego (oczywiście połączenia między przełącznikami są jednokierunkowe).

### 3.4.1 Konstrukcja

Dla prostoty ograniczymy się do konstrukcji sieci dla $n$ będącego potęgą liczby 2.

Konstrukcja oparta jest za zasadzie Dziel i Zwyciężaj i sprowadza się do sprytnego rozesłania danych wejściowych do dwóch (zbudowanych rekurencyjnie) egzemplarzy sieci o $n/2$ wejściach, a następnie na umiejętnym połączeniu portów wyjściowych tych podsieci. Istota tej konstrukcji przedstawiona jest na poniższym rysunku.




Rysunek 3: *Konstrukcja sieci dla* $n = 8$

Porty wejściowe sieci połączone przełącznikiem w pierwszej warstwie oraz porty wyjściowe połączone przełącznikiem w ostatniej warstwie będziemy nazywać portami *sąsiednimi.*

**Fakt 1** *Tak skonstruowana sieć ma głębokość* $2\log n - 1$ *i zawiera* $\Theta(n \log n)$ *przełączników.*

DOWÓD: Głębokość sieci wyraża się równaniem

$$G(2^k) = \begin{cases} 1 & \text{dla } k = 1 \\ G(2^{k-1}) + 2 & \text{dla } k > 1 \end{cases}$$

a liczba przełączników - równaniem:

$$P(2^k) = \begin{cases} 1 & \text{dla } k = 1 \\ 2P(2^{k-1}) + 2^k & \text{dla } k > 1 \end{cases}$$

□

### 3.4.2 Poprawność konstrukcji

Niech $\pi$ będzie dowolną permutacją $n$-elementową. Pokażemy, że istnieje ustawienie przełączników sieci realizujące $\pi$, tj. takie, że dane z $i$-tego portu sieci zostaną przesłane na $\pi(i)$-ty port wyjściowy (dla $i = 1, \ldots, n$). Istnienie takiego ustawienia będzie konsekwencją istnienia odpowiedniego dwukolorowania wierzchołków w następującym grafie $G_\pi = (V, E)$.

- Zbiór $V = V_I \cup V_O \cup V_M$ składa się z:
  - $n$ wierzchołków (podzbiór $V_I$) odpowiadających portom wejściowym sieci;
  - $n$ wierzchołków (podzbiór $V_O$) odpowiadających przełącznikom z ostatniej warstwy sieci (po dwa wierzchołki na każdy przełącznik);
  - $n$ wierzchołków (podzbiór $V_M$) dodanych ze względów technicznych.
- Wszystkie wierzchołki z $V$ etykietujemy:
  - wierzchołek z $V_I$ odpowiadający $i$-temu portowi wejściowemu otrzymuje etykietę $i$;

– wierzchołki z $V_O$, z pary odpowiadającej $j$-temu przełącznikowi ostatniej warstwy, otrzymują etykiety $i$" i $k$", takie, że $2j-1=\pi(i)$ oraz $2j=\pi(k)$ (innymi słowy na porty wyjściowe $j$-tego przełącznika mają być wysłane wartości z $i$-tego oraz $k$-tego portu wejściowego sieci);
– wierzchołki z $V_M$ otrzymują w dowolny sposób różne etykiety ze zbioru $\{1', 2', \ldots, n'\}$.

- Zbiór $E = E_I \cup E_O \cup E_M$ składa się z:
  – $n/2$ krawędzi (podzbiór $E_I$) łączących wierzchołki o etykietach $2i-1$ i $2i$, a więc takie, które odpowiadają sąsiednim portom wejściowym;
  – $n/2$ krawędzi (podzbiór $E_O$) łączących wierzchołki odpowiadające temu samemu przełącznikowi ostatniej warstwy;
  – $2n$ krawędzi (podzbiór $E_M$) łączących wierzchołki o etykietach $i$ i $i'$ oraz wierzchołki o etykietach $i'$ i $i$".

**Fakt 2** *Graf $G_\pi$ jest sumą rozłącznych cykli parzystej długości.*

DOWÓD: Stopień każdego wierzchołka w $G_\pi$ jest równy 2, a więc $G$ jest sumą rozłącznych cykli. Są one parzystej długości, ponieważ, jak łatwo zauważyć, każdy cykl zawiera parzystą liczbę wierzchołków z $V_I$, parzystą liczbę wierzchołków z $V_O$, a więc także parzystą liczbę wierzchołków z $V_M$. □

Z faktu tego wprost wynika istnienie kolorowania wierzchołków $G_\pi$ dwoma kolorami (powiedzmy białym i czarnym). Kolorowanie to ma następujące własności:

- Wierzchołki odpowiadające sąsiednim portom (zarówno wejściowym jak i wyjściowym) otrzymują różne kolory.
- Wierzchołki o etykietach $i$ oraz $i$" otrzymują ten sam kolor (dla każdego $i = 1, \ldots, n$).

Stąd wnioskujemy istnienie ustawienia przełączników realizującego $\pi$:

- Przełączniki z pierwszej warstwy ustawiamy tak, by dane z portów białych (dokładniej: których odpowiadające wierzchołki otrzymały kolor biały) były przesłane do górnej podsieci $Perm_{n/2}$.
- Przełączniki w górnej podsieci $Perm_{n/2}$ ustawiamy tak, by permutowała swoje dane zgodnie z permutacją $\pi$. Dokładniej:
  Niech $K = k_1, \ldots, k_{n/2}$ będzie ciągiem etykiet białych wierzchołków z $V_I$ w kolejności ich występowania w ciągu $\{1, 2, \ldots, n\}$ a $L = l_1, \ldots, l_{n/2}$ ciągiem etykiet białych wierzchołków z $V_O$ w kolejności ich występowania w ciągu $\pi(1), \ldots, \pi(n)$. Niech $\pi_a : \{1, \ldots, n/2\} \to \{1, \ldots, n/2\}$ będzie permutacją taką, że $\pi_a(i) = j$ wtedy i tylko wtedy gdy $l_j = k_i^{"}$. Przełączniki ustawiamy tak, by podsieć realizowała permutację $\pi_a$. Takie ustawienie istnieje na mocy indukcji. Podobne rozważania przeprowadzamy dla dolnej podsieci $Perm_{n/2}$.
- Dla przelączników ostatniej warstwy stosujemy następującą regułę:
  Niech $i$" będzie etykietą białego wierzchołka z pary odpowiadającej $j$-temu przełącznikowi, a $k$" - etykietą czarnego wierzchołka z tej pary. Jeśli $i$ poprzedza $k$ w permutacji $\pi$ (tzn. $i = \pi(2j-1)$ oraz $k = \pi(2j)$) przełącznik ustawiamy na wprost, w przeciwnym razie ustawiamy na ukos.

## 3.5 Para najbliżej położonych punktów

PROBLEM:

*Dane:* Zbiór $P = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ współrzędnych punktów na płaszczyźnie.

*Zadanie:* Znaleźć dwa najbliżej położone względem siebie punkty w $P$, tj. znaleźć $i$,$j$ takie, że $d(x_i, y_i, x_j, y_j) = min\{d(x_k, y_k, x_l, y_l) \mid 1 \leq k < l \leq n\}$, gdzie $d(x_k, y_k, x_l, y_l) = \sqrt{(x_k - x_l)^2 + (y_k - y_l)^2}$.

Siłowe rozwiązanie, polegające na wyliczeniu i porównaniu odległości między każdą parą punktów, wymaga czasu $\Omega(n^2)$. Przedstawimy teraz strategię Dziel i Zwyciężaj, która daje algorytm działający w czasie $\Theta(n \log n)$.

TERMINOLOGIA: mówiąc o punkcie $r$ będziemy mieć na myśli punkt o współrzędnych $(x_r, y_r)$.

### 3.5.1 Strategia Dziel i Zwyciężaj

1. (a) Sortujemy punkty z $P$ według współrzędnych $x$ i zapamiętujemy je w tablicy $X$;

   (b) Sortujemy punkty z $P$ według współrzędnych $y$ i zapamiętujemy je w tablicy $Y$;

   (c) Znajdujemy prostą $l$ dzielącą $P$ na dwa równoliczne (z dokładnością do 1) podzbiory:

   - $P_L$- podzbiór punktów leżących na lewo od $l$,
   - $P_R$ - podzbiór punktów leżących na prawo od $l$.

   Punkty znajdujące się na prostej $l$ (o ile są takie) kwalifikujemy do tych podzbiorów w dowolny sposób.

2. { rekurencyjnie }
   $(i_1, j_1) \leftarrow$ para punktów z $P_L$ o najmniejszej odległości;
   $(i_2, j_2) \leftarrow$ para punktów z $P_R$ o najmniejszej odległości.

3. Niech $(i', j')$ będzie tą parą punktów znalezioną w kroku 2, dla której odległość (oznaczmy ją przez $d$) jest mniejsza.
   Sprawdzamy czy istnieje para punktów $(t, s)$ odległych o mniej niż $d$ takich, że $t \in P_L$ i $s \in P_R$. Jeśli istnieje, przekazujemy ją jako wynik procedury, w przeciwnym razie jako wynik przekazujemy parę $(i', j')$.

□

Wyjaśnienia wymaga sposób realizacji kroku 3. Oznaczmy przez $P_C$ zbiór tych punktów z $P$, które leżą w odległości nie większej niż $d$ od prostej $l$. Niech $Y'$ oznacza tablicę $Y$, z której usunięto wszystkie punkty spoza $P_C$. Korzystamy z następującego spostrzeżenia:

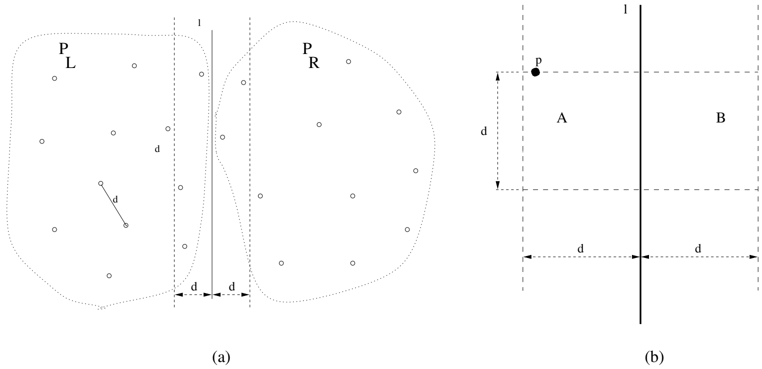
**Fakt 3** *Jeśli $(t, s)$ jest parą punktów odległych o mniej niż $d$ taką, że $t \in P_L$ i $s \in P_R$, to $t$ i $s$ należą do $P_C$. Ponadto w tablicy w $Y'$ pomiędzy $t$ a $s$ leży nie więcej niż 6 punktów.*

DOWÓD: Gdyby jeden z punktów leżał w odległości większej niż $d$ od prostej $l$, to odległość między nimi byłaby większa niż $d$. Oczywiste jest też, że współrzędne $y$-kowe tych punktów różnią się nie więcej niż o $d$. Tak więc punkty $t$ i $s$ leżą w prostokącie o wymiarach $d \times 2d$ jak pokazano na rysunku 4.

W części $A$ leżą tylko punkty z $P_L$. Ponieważ każde dwa z nich odległe są od siebie o co najmniej $d$, więc może ich tam znajdować się co najwyżej 4. Z analogicznego powodu w części $B$ może znajdować się nie więcej niż 4 punkty z $P_R$. Tak więc w całym prostokącie znajduje się nie więcej niż 8 punktów.

□

Krok 3 sprowadza się więc do utworzenia tablicy $Y'$, a następnie do obliczenia odległości każdego punktu z $Y'$ do co najwyżej siedmiu punktów następujących po nim w tej tablicy.




Rysunek 4: *(a) W kroku 3 pary $(t, s)$ należy szukać tylko w zaznaczonym pasie (b) Jeśli $p$ ma być jednym z punktów pary $(t, s)$, to drugi punkt musi znajdować się w kwadracie $A$. Ponadto wszystkie punkty z $Y'$ leżące między $t$ a $s$ muszą leżeć w $A$ lub w $B$.*

### 3.5.2 Koszt:

Krok 1:

- Sortowanie - $\Theta(n \log n)$.
- Znalezienie prostej $l$ i podział $P$ na podzbiory - koszt stały.

Krok 2: $2T(n/2)$

Krok 3:

- Utworzenie $Y'$ - $\Theta(n)$.
- Szukanie pary $(t, s)$ - $\Theta(n)$.

Stąd koszt całego algorytmu wyraża się równaniem $T(n) = 2T(n/2) + \Theta(n \log n)$, którego rozwiązaniem jest $\Theta(n \log^2 n)$. Koszt ten można zredukować do $\Theta(n \log n)$. Wystarczy zauważyć , że sortowanie punktów w każdym wywołaniu rekurencyjnym jest zbyteczne. Zbiór $P$ możemy przekazywać kolejnemu wywołaniu rekurencyjnemu jako tablice $X$ i $Y$. Na ich podstawie można w czasie liniowym utworzyć odpowiednie tablice dla zbiorów $P_L$ i $P_R$. Tak więc sortowanie wystarczy przeprowadzić jeden raz - przed pierwszym wywołaniem procedury rekurencyjnej.

Po takiej modyfikacji czas wykonania procedury rekurencyjnej wyraża się równaniem $T(n) = 2T(n/2) + \Theta(n)$, którego rozwiązaniem jest $\Theta(n \log n)$. Dodany do tego czas sortowania nie zwiększa rzędu funkcji.