Skrypt z Algorytmów i struktur danych

Zbiór mniej lub bardziej ciekawych algorytmów i struktur danych, jakie bywały omawiane na wykładzie (albo i nie).

praca zbiorowa pod redakcją
Krzysztofa Piecucha

Korzystać na własną odpowiedzialność.

Spis treści

1 Podstawy **5**
1.1 Złożoność obliczeniowa 6
1.2 Model obliczeń 7
1.3 Twierdzenie o rekursji uniwersalnej 8
1.3.1 Przykłady wykorzystania twierdzenia 9

2 Struktury danych **11**
2.1 Kopce binarne 12

3 Algorytmy **15**
3.1 Sortowanie bitoniczne 16
3.2 Algorytm rosyjskich wieśniaków 21
3.3 Algorytm macierzowy wyznaczania liczb Fibonacciego 24
3.4 Sortowanie topologiczne 26
3.5 Algorytmy sortowania 27
3.6 Minimalne drzewa rozpinające 28
3.6.1 Cut Property i Circle Property 28
3.6.2 Algorytm Prima 28
3.6.3 Algorytm Kruskala 28
3.6.4 Algorytm Borůvki 28
3.7 Algorytm Dijkstry 29
3.8 Algorytm szeregowania 30
3.9 Programowanie dynamiczne na drzewach 31
3.10 Mnożenie macierzy 32

Dodatek A Porównanie programów przedmiotu AiSD na różnych uczelniach **35**

Rozdział 1

Podstawy

1.1 Złożoność obliczeniowa

Todo, todo, todo...

1.2 Model obliczeń

Todo, todo, todo...

1.3 Twierdzenie o rekursji uniwersalnej

Twierdzenie o rekursji uniwersalnej pozwala nam ocenić złożoność obliczeniową algorytmu, który wywołuje się rekurencyjnie dla mniejszych danych.

Procedura T 1:

Input: n
if $(n < constant)$ **then**
  return
end
Stwórz a podproblemów wielkości n/b w czasie c(n)
for $i \leftarrow 1$ *to* a **do**
  T[n/b]
end
Połącz wyniki w czasie d(n)

Złożoność takiego algorytmu możemy zapisać zależnością rekurencyjną

$$T(n) = aT(\frac{n}{b}) + f(n)$$

gdzie $f(n) = c(n) + d(n)$

Twierdzenie 1. *Rekurencja* $T(n) = aT(\frac{n}{b}) + f(n)$ *może zostać rozwiązana*

- *jeżeli* $af(n/b) = cf(n)$ *gdzie* $c < 1$, *to* $T(n) = \Theta(f(n))$
- *jeżeli* $af(n/b) = cf(n)$ *gdzie* $c > 1$, *to* $T(n) = \Theta(n^{log_b a})$
- *jeżeli* $af(n/b) = f(n)$, *to* $T(n) = \Theta(f(n)log_b n)$

Jeżeli $f(n)$ jest większy o wielokrotność stałej od $af(b/n)$, wtedy suma jest malejącym ciągiem geometrycznym. Suma każdego ciągu geometrycznego jest wynikiem przemnożenia jej największego elementu przez stałą. W tym przypadku największym elementem jest $f(n)$.

Jeżeli $f(n)$ jest mniejszy o wielokrotność stałej od $af(b/n)$, wtedy suma jest rosnący ciągiem geometrycznym. Suma każdego ciągu geometrycznego jest wynikiem przemnożenia jej największego elementu przez stałą. W tym przypadku, jako, że $f(1) = \Theta(1)$ ostatni element sumy to ilość liści w drzewie rekursji $\Theta(a^{log_b n}) = \Theta(n^{log_b a})$

Tutaj wstawiłbym rysunek drzewka rekursji, ale nie umiem

Jeżeli $f(n) = af(b/n)$ to każdy element sumy jest równy $f(n)$.

1.3.1 Przykłady wykorzystania twierdzenia

- Mergesort: T(n) = 2 * T(n/2) + n log(n)

  $a * f(n/b) = 2 * n/2 = n = f(n)$, więc $T(n) = \Theta(nlogn)$

- Mergesort z dziwnym mergem(przykład z próbnego egzaminu): T(n) = 2 * T(n/2) + n log(n)

  $a * f(n/b) = 2 * n/2 * log(n/2) = n * log(n/2) = n * (log(n) - 1)$, nie możemy zastosować tutaj prostej równości, jednak zauważmy, iż przy odpowiednio dużym n wartość ta jest bliska $f(n)$; tj. $\lim_{x \to \infty} \frac{nlog(n)}{n(log(n)-1)} = 1$, więc $T(n) = n * log(n) * log(n) = n * log^2(n)$

- Mergesort z mergem w czasie stałym(przykład z próbnego egzaminu): T(n) = 2 * T(n/2) + 1

  $a * f(n/b) = 2$, więc nasze twierdzenie nie będzie miało zastosowania. Można jednak łatwo zauważyć, że $T(n) = \Theta(n)$ i udowodnić to indukcyjnie.

- Algorytm Karatsuby: T(n) = 3 * T(n/2) + n $a * f(n/b) = \frac{3}{2} * n$; $\frac{3}{2} > 1$, więc $T(n) = n^{log_2(3)}$.

Rozdział 2

Struktury danych

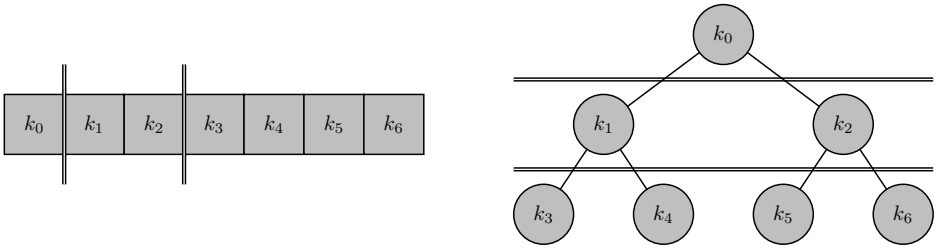
2.1 Kopce binarne

Chciałbym, aby skrypt był skryptem i żebyśmy tam pisali formalnie. Dlatego definicję kopca chciałbym mieć zapisaną formalnie (bez przypisów) i w znacznikach begin{definition} end{definition}. Teraz śmieszna rzecz jest taka, że kopiec trudno ładnie formalnie zdefiniować na drzewach. W sensie najpierw musielibyśmy powiedzieć co to jest poziom w drzewie, co to jest pełny poziom w drzewie a następnie .... w sumie to nawet ja nie wiem jak to ładnie zdefiniować na drzewach :P Więc zamiast mówić, że kopiec to drzewo które można reprezentować w tablicy, lepiej zdefiniować kopiec jako tablicę na którą możemy patrzeć jako na drzewo. Wtedy musimy zdefiniować co to jest lewy syn elementu w tablicy, prawy syn oraz ojciec. Na tej podstawie będzie łatwiej nam zdefiniować własność kopca jako, że dla każdego wierzchołka wartość elementu jest mniejsza od wartości elementów jego dzieci. Jeśli wolimy zamiast tego powiedzieć że ciąg elementów na ścieżce od liścia do korzenia tworzy ciąg malejący to musimy zdefiniować co to jest liść, korzeń i ścieżka. Co da się zrobić ale nie wiem czy jest to warte świeczki, gdyż to będzie bodajże jedyne miejsce w których użyjemy tych definicji).

Zamień element jest fajną funkcją, ale funkcje przesun w dol i przesun w gore są ważniejsze. Ponadto nie chcemy nazywać funkcje w języku polskim.

Kopiec binarny to struktura danych, która reprezentowana jest jako prawie pełne drzewo binarne[1] i na której zachowana jest własność kopca. Kopiec przechowuje klucze, które tworzą ciąg uporządkowany. W przypadku kopca typu *min* ścieżka prowadząca od dowolnego liścia do korzenia tworzy ciąg malejący.

Kopce można w prosty sposób reprezentować w tablicy jednowymiarowej –kolejne poziomy drzewa zapisywane są po sobie.



Rysunek 2.1: Reprezentacja kolejnych warstw kopca w tablicy jednowymiarowej.

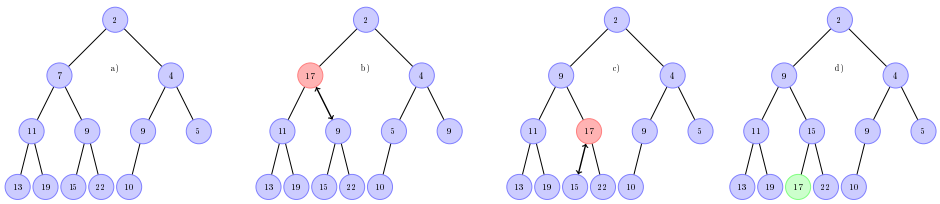
Warto zauważyć, że tak reprezentowane drzewo pozwala na łatwy dostęp do powiązanych węzłów. Synami węzła o indeksie i są węzły $2i+1$ oraz $2i+2$, natomiast jego ojcem jest $\lfloor \frac{i-1}{2} \rfloor$.

Kopiec powinien udostępniać trzy podstawowe funkcje: `zamien_element`, która podmienia wartość w konkretnym węźle kopca, `przesun_w_gore` oraz `przesun_w_dol`, które zamieniają odpowiednie elementy pilnując przy tym, aby własność kopca została zachowana.

Algorithm 2: Implementacja funkcji `zamien_element`

```
if k[i] < v then
    k[i] = v;
    przesun_w_dol(k, i);
end
else
    k[i] = v;
    przesun_w_gore(k, i);
end
```

[1] To znaczy wypełniony na wszystkich poziomach (poza, być może, ostatnim).



Rysunek 2.2: Przykład działania funkcji `zamien_element`. a) Oryginalny kopiec. b) Zmiana wartości w wyróżnionym węźle. c) Ponieważ nowa wartość jest większa od wartości swoich dzieci, należy wykonać wywołanie funkcji `przesn_w_dol`. d) Po zmianie własność kopca nie jest zachowana, dlatego należy ponownie wywołać funkcję `przesn_w_dol`. To przywraca kopcowi jego własność.

Rozdział 3

Algorytmy

3.1 Sortowanie bitoniczne

W tym rozdziale przedstawimy algorytm sortowania bitonicznego. Jest to algorytm działający w czasie $O(n \log^2 n)$ czyli gorszym niż inne, znane algorytmy sortujące takie jak sortowanie przez scalanie albo sortowanie szybkie. Zaletą sortowania bitonicznego jest to, że może zostać uruchomiony równolegle na wielu procesorach. Ponadto, dzięki temu, że algorytm zawsze porównuje te same elementy bez względu na dane wejściowe, istnieje prosta implementacja fizyczna tego algorytmu (np. w postaci tzw. sieci sortujących). Algorytm będzie zakładał, że rozmiar danych n jest potęgą dwójki. Gdyby tak nie było, moglibyśmy wypełnić tablicę do posortowania nieskończonościami, tak aby uzupełnić rozmiar danych do potęgi dwójki. Rozmiar danych zwiększyłby się wtedy nie więcej niż dwukrotnie, zatem złożoność asymptotyczna pozostałaby taka sama.

Sortowanie bitoniczne posługuje się tzw. ciągami bitonicznymi, które sobie teraz zdefiniujemy.

Definicja 1. ***Ciągiem bitonicznym właściwym*** *nazywamy każdy ciąg powstały przez sklejenie ciągu niemalejącego z ciągiem nierosnącym.*

Dla przykładu ciąg 2, 2, 5, 100, 72, 69, 42, 17 jest ciągiem bitonicznym, gdyż powstał przez sklejenie ciągu niemalejącego 2, 2, 5 oraz ciągu nierosnącego 100, 72, 69, 42, 17. Ciąg 1, 0, 1, 0 nie jest ciągiem bitonicznym, gdyż nie istnieją taki ciąg niemalejący i taki ciąg nierosnący, które w wyniku sklejenia dałyby podany ciąg.

Definicja 2. ***Ciągiem bitonicznym*** *nazywamy każdy ciąg powstały przez rotację cykliczną ciągu bitonicznego właściwego.*

Ciąg 69, 42, 17, 2, 2, 5, 100, 72 jest ciągiem bitonicznym, gdyż powstało przez rotację cykliczną ciągu bitonicznego właściwego 2, 2, 5, 100, 72, 69, 42, 17.

Istnieje prosty algorytm sprawdzający, czy ciąg jest bitoniczny. Należy znaleźć element największy oraz najmniejszy. Następnie od elementu najmniejszego należy przejść cyklicznie w prawo (tj. w sytuacji gdy natrafimy na koniec ciągu, wracamy do początku) aż napotkamy element największy. Elementy, które przeszliśmy w ten sposób powinny tworzyć ciąg niemalejący. Analogicznie, idziemy od elementy największego cyklicznie w prawo aż do elementu najmniejszego. Elementy, które odwiedziliśmy powinny tworzyć ciąg nierosnący. W sytuacji w której mamy wiele elementów najmniejszych (największych), powinny one ze sobą sąsiadować (w sensie cyklicznym) i nie ma znaczenia, który z nich wybierzemy. Dla przykładu w ciągu 69, 42, 17, 2, 2, 5, 100, 72 idąc od elementu najmniejszego do największego tworzymy ciąg 2, 2, 5, 100 i jest to ciąg niemalejący. Idąc od elementu największego do najmniejszego otrzymujemy ciąg 100, 72, 69, 42, 17, 2 i jest to ciąg nierosnący.

Jedyną procedurą, która będzie przestawiała elementy w tablicy, będzie procedura `bitonic_compare` (Algorytm 3). Jako dane wejściowe otrzymuje ona tablicę `A`, wielkość tablicy `n` oraz wartość logiczną `up`, która określa, czy ciąg będzie sortowany rosnąco czy malejąco. Procedura dzieli zadaną na wejściu tablicę na

Algorytm 3: Procedura `bitonic_compare`

```
Input: A, n, up
for i ← 1 to n do
    if (A[i] > A[i + n/2]) = up then
        A[i] ↔ A[i + n/2]
    end
end
```

dwie równe części. Następnie porównuje pierwszy element z pierwszej części z pierwszym elementem z drugiej części. Jeśli te elementy nie znajdują się w pożądanym porządku, to je przestawia. Następnie powtarza tą czynność z kolejnymi elementami.

Dla przykładu, jeśli procedurę uruchomimy z tablicą `A[] = 2, 8, 7, 1, 4, 3, 5, 6`, wartością `n = 8` oraz `up = true`, w wyniku otrzymamy tablicę `A[] = 2, 3, 5, 1, 4, 8, 7, 6`. W pierwszym kroku wartość 2 zostanie porównana z wartością 4. Ponieważ chcemy otrzymać porządek rosnący (wartość zmiennej `up` jest ustawiona na `true`), to zostawiamy tą parę w spokoju. W następnym kroku porównujemy wartość 8 z wartością 3. Te wartości są w złym porządku, dlatego algorytm zamienia je miejscami. Dalej porównujemy 7 z 5 i zamieniamy je miejscami i w końcu porównujemy 1 z 6 i te wartości zostawiamy w spokoju, gdyż są w dobrym porządku.

Procedura `bitonic_compare` ma bardzo ważną własność, którą teraz udowodnimy.

Twierdzenie 2. *Jeżeli elementy tablicy `A[0..n-1]` tworzą ciąg bitoniczny, to po zakończeniu procedury `bitonic_compare` elementy tablicy `A[0..n/2-1]` oraz tablicy `A[n/2..n-1]` będą tworzyły ciągi bitoniczne. Ponadto jeśli wartość zmiennej `up` jest ustawiona na `true` to każdy element tablicy `A[0..n/2-1]` będzie niewiększy od każdego elementu tablicy `A[n/2..n-1]`. W przeciwnym przypadku będzie większy.*

Weźmy dla przykładu ciąg bitoniczny 69, 42, 17, 2, 2, 5, 100, 72. Po przejściu procedury `bitonic_compare` z ustawioną zmienną `up` na wartość `true` otrzymamy ciąg 2, 5, 17, 2, 69, 42, 100, 72. Ciągi 2, 5, 17, 2 oraz 69, 42, 100, 72 są ciągami bitonicznymi. Ponadto każdy element ciągu 2, 5, 17, 2 jest niewiększy od każdego elementu ciągu 69, 42, 100, 72.

Przejdźmy do dowodu powyższego twierdzenia. Przyda nam się do tego poniższy lemat:

Lemat 1 (zasada zero-jeden)**.** *Twierdzenie 2 jest prawdziwe dla dowolnych tablic wtedy i tylko wtedy, gdy jest prawdziwe dla tablic zero-jedynkowych.*

Dowód. Oczywiście - jeśli twierdzenie jest prawdziwe dla każdej tablicy to w szczególności jest prawdziwe dla tablic złożonych z zer i jedynek. Dowód w drugą stronę jest dużo ciekawszy.

Weźmy dowolną funkcję niemalejącą f. To znaczy funkcję $f : \mathbb{R} \to \mathbb{R}$ taką, że $\forall_{a,b\in\mathbb{R}}\ a \leqslant b \Rightarrow f(a) \leqslant f(b)$. Dla tablicy `T` przez $f(T)$ będziemy rozumieli tablicę powstałą przez zaaplikowanie funkcji f do każdego elementu tablicy `T`. Niech `A` oznacza tablicę wejściową do procedury `bitonic_compare` i niech `B` oznacza tablicę wyjściową. Udowodnimy, że karmiąc procedurę `bitonic_compare` tablicą $f(A)$ otrzymamy tablicę $f(B)$. W kroku i-tym procedura rozważa przestawienie elementów t_i oraz $t_{i+n/2}$. Jeśli $f(a_i) = f(a_{i+n/2})$ to nie ma znaczenia czy elementy zostaną przestawione. Z kolei jeśli $f(a_i) < f(a_{i+n/2})$ to $a_i < a_{i+n/2}$ zatem jeśli procedura przestawi elementy $f(a_i)$ oraz $f(a_{i+n/2})$ to również przestawi elementy a_i oraz $a_{i+n/2}$. Analogicznie gdy $f(a_i) > f(a_{i+n/2})$. Zatem istotnie: dla każdej funkcji niemalejącej f, procedura `bitonic_compare` otrzymując na wejściu tablicę $f(A)$ zwróci na wyjściu tablicę $f(B)$.

Wróćmy do dowodu lematu. Dowód niewprost. Załóżmy, że twierdzenie jest prawdziwe dla wszystkich tablic zero-jedynkowych i nie jest prawdziwe dla pewnej tablicy `T[0..n-1]`. Niech `S[0..n-1]` oznacza zawartość tablicy po zakończeniu procedury `bitonic_compare`. Jeśli twierdzenie nie jest prawdziwe, oznacza to, że albo któraś z tablic `S[0..n/2-1]`, `S[n/2..n-1]` nie jest bitoniczna albo, że elementy z jednej z nich nie są mniejsze od wszystkich elementów z drugiej tablicy. Rozważmy dwa przypadki.

Załóżmy, że tablica `S[0..n/2-1]` nie jest bitoniczna (przypadek kiedy druga z tablic nie jest bitoniczna, jest analogiczny). Załóżmy, że ciąg powstały przez przejście od najmniejszego elementu w tej tablicy do największego nie tworzy ciągu niemalejącego (przypadek gdy ciąg powstały przez przejście od największego elementu do najmniejszego nie tworzy ciągu nierosnącego jest analogiczny). Zatem istenieje w tablicy element `S[i]` większy od elementu `S[i+1]`. Rozważmy następują funkcję:

$$f(a) = \begin{cases} 1 & \text{jeśli } a \leqslant S[i] \\ 0 & \text{wpp.} \end{cases}$$

Zwróćmy uwagę, że w takiej sytuacji twierdzenie nie byłoby prawdziwe dla tablicy $f(T)$, zatem dla tablicy zero-jedynkowej. Gdyż ponownie - element $f(S[i]) = 1$ byłby większy od elementu $f(S[i+1]) = 0$.

Drugi przypadek. Załóżmy, że zmienna `up` ustawiona jest na `true` (przypadek drugi jest analogiczny). Załóżmy, że element `S[i]` jest mniejszy od elementu `S[j]`

gdzie $j < n/2$ oraz $i \geqslant n/2$. Rozważmy funkcję:

$$f(a) = \begin{cases} 1 & \text{jeśli } a \leqslant S[j] \\ 0 & \text{wpp.} \end{cases}$$

Wtedy twierdzenie nie byłoby prawdziwe dla tablicy $f(T)$ (zero-jedynkowej). Ponownie - element $f(S[i]) = 0$ byłby mniejszy od elementu $f(S[j]) = 1$. □

Do pełni szczęścia potrzebujemy udowodnić, że Twierdzenie 2 jest prawdziwe dla wszystkich ciągów zero-jedynkowych.

Lemat 2. *Twierdzenie 2 jest prawdziwe dla wszystkich ciągów zero-jedynkowych.*

Dowód. Zakładać będziemy, że zmienna `up` jest ustawiona na `true` (dowód dla sytuacji przeciwnej jest analogiczny). Istnieją cztery rodzaje bitonicznych ciągów zero-jedynkowych : 0^n, 0^k1^l, $0^k1^l0^m$, 1^n, 1^k0^l, $1^k0^l1^m$ z czego trzy ostatnie są symetryczne do trzech pierwszych (więc zostaną pominięte w dowodzie). Rozważmy wszystkie interesujące nas przypadki:

Ten dowód jest nudny.

- 0^n. Po wykonaniu procedury `bitonic_compare` otrzymamy $0^{n/2}$ oraz $0^{n/2}$. Oba ciągi są bitoniczne i każdy element z pierwszego ciągu jest niewiększy od każdego elementu z ciągu drugiego.
- 0^k1^l oraz $k < n/2$. Wtedy po wykonaniu procedury `bitonic_compare` otrzymamy ciągi $0^k1^{l-n/2}$ oraz $1^{n/2}$. Oba ciągi są bitoniczne i każdy element z pierwszego ciągu jest niewiększy od każdego elementu z ciągu drugiego.
- 0^k1^l oraz $k > n/2$. Otrzymamy ciągi $0^{n/2}$ oraz $0^{k-n/2}1^l$. Znowu - oba są bitoniczne i każdy element z pierwszego jest niewiększy od każdego z drugiego.
- $0^k1^l0^m$ oraz $k > n/2$. Wtedy otrzymujemy ciągi $0^{n/2}$ oraz $0^{k-n/2}1^l0^m$. Spełniają one tezę twierdzenia.
- $0^k1^l0^m$ oraz $m > n/2$. Ciągi, które otrzymamy wyglądają tak: $0^{n/2}$ oraz $0^k1^l0^{n/2-m}$. Są to ciągi, które nas cieszą.
- $0^k1^l0^m$ oraz $l > n/2$. Dostaniemy wtedy ciągi $0^k1^{l-n/2}0^m$ oraz $1^{n/2}$. Są to ciągi, które spełniają naszą tezę.
- $0^k1^l0^m$ oraz $k, l, m < n/2$. Ciągi, które uzyskamy to $0^{n/2}$ oraz $1^{n/2-m}0^{n/2-l}$ $1^{n/2-k}$. Spełniają one naszą tezę.

□

Na mocy Lematu 1 i 2 Twierdzenie 2 jest prawdziwe dla wszystkich tablic `T[0..n-1]`. Mając tak piękne twierdzenie, możemy napisać prosty algorytm sortujący ciągi bitoniczne (Algorytm 4).

Algorytm 4: Procedura `bitonic_merge`

Input: A - tablica bitoniczna, n, up
Output: A - tablica posortowana
if $n > 1$ **then**
    bitonic_compare(A[0..$n-1$], n, up)
    bitonic_merge(A[0..$n/2-1$], $n/2$, up)
    bitonic_merge(A[$n/2$..$n-1$], $n/2$, up)
end

Algorytm zaczyna od wywołania procedury `bitonic_compare`. Dzięki niej, wszystkie elementy mniejsze wrzucane są do pierwszej połowy tablicy, a elementy większe do drugiej połowy. Ponadto `bitonic_compare` gwarantuje, że obie podtablice pozostają bitoniczne (jakie to piękne!). Możemy zatem wykonać całą procedurę ponownie na obu podtablicach rekurencyjnie.

Złożoność algorytmu wyraża się wzorem rekurencyjnym $T(n) = 2 \cdot T(n/2) + O(n)$. Rozwiązując rekurencję otrzymujemy, że złożoność algorytmu to $O(n \log n)$.

Mamy algorytm sortujący ciągi bitoniczne. Jak uzyskać algorytm sortujący dowolne ciągi? Zrealizujemy to w najprostszy możliwy sposób! Posortujemy (rekurencyjnie) pierwszą połowę tablicy rosnąco, drugą połowę tablicy malejąco (dlatego potrzebna nam była zmienna `up`!) i uzyskamy w ten sposób ciąg bitoniczny. Teraz wystarczy już uruchomić algorytm sortujący ciągi bitoniczne i voilà.

Algorytm 5: Procedura `bitonic_sort`

Input: A, n, up
Output: A - tablica posortowana
if $n > 1$ **then**
    bitonic_sort(A[0..$n/2-1$], $n/2$, true)
    bitonic_sort(A[$n/2$..$n-1$], $n/2$, false)
    bitonic_merge(A[0..$n-1$], n, up)
end

Złożoność algorytmu wyraża się wzorem rekurencyjnym $T(n) = 2 \cdot T(n/2) + O(n \log n)$. Rozwiązaniem tej rekurencji jest $O(n \log^2 n)$.

3.2 Algorytm rosyjskich wieśniaków

Algorytm rosyjskich wieśniaków jest przypisywany sposobowi mnożenia liczb używanemu w XIX-wiecznej Rosji. Aktualnie jest on stosowany w niektórych układach mnożących.

W celu obliczenia $a \cdot b$ tworzymy tabelkę i liczby a i b zapisujemy w pierwszym jej wierszu. Kolumnę a wypełniamy następująco: w $i+1$ wierszu wpisujemy wartość z wiersza i podzieloną całkowicie przez 2. W kolumnie b kolejne wiersze tworzą ciąg geometryczny o ilorazie równym 2. Wypełnianie tabelki kończymy wtedy, gdy w kolumnie a otrzymamy wartość 1. Na koniec sumujemy wartości w kolumine b z tych wierszy dla których wartości w kolumine a są nieparzyste. Uzyskany wynik to $a \cdot b$.

W poniższym przykładzie obliczymy $42 \cdot 17$.

a	b
42	17
21	$17 \cdot 2 = 34$
10	$17 \cdot 2^2 = 68$
5	$17 \cdot 2^3 = 136$
2	$17 \cdot 2^4 = 272$
1	$17 \cdot 2^5 = 544$

Wartości a są nieparzyste w wierszach 2, 4 oraz 6. Zatem będziemy sumować wartości b z wierszy 2, 4 i 6.

Jak usuniecie enter po tabelce, to to zdanie nie dostanie w pdfie tabulatora. Tabulator powinien się znajdować przed akapitami. To zdanie jest częścią poprzedniego akapitu.

$$\begin{aligned} a \cdot b &= 17 \cdot 2 + 17 \cdot 2^3 + 17 \cdot 2^5 \\ &= 34 + 136 + 544 \\ &= 714 \end{aligned}$$

Faktycznie, otrzymaliśmy wynik poprawny. Spójrzmy raz jeszcze na na tę sumę:

$$\begin{aligned} a \cdot b &= 17 \cdot 2 + 17 \cdot 2^3 + 17 \cdot 2^5 \\ &= 17 \cdot (2^5 + 2^3 + 2) \\ &= 17 \cdot (1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0) \\ &= 17_{10} \cdot 101010_2 \\ &= 17 \cdot 42 = 714 \end{aligned}$$

Przypomnij sobie algorytm zamiany liczby w systemie dziesiętnym na system binarny. Okazuje się, że algorytm rosyjskich wieśniaków "po cichu" wylicza tę reprezentację a.

Algorytm 6: Algorytm rosyjskich wieśniaków

Input: a, b - liczby naturalne
Output: $wynik = a \cdot b$
$a' \leftarrow a$
$b' \leftarrow b$
$wynik \leftarrow 0$
while $a' > 0$ **do**
  if $a' \bmod 2 = 1$ **then**
    $wynik \leftarrow wynik + b'$
  end
  $a' \leftarrow a' \operatorname{div} 2$
  $b' \leftarrow b' \cdot 2$
end

W kolejnych paragrafach podamy algorytm i w celu jego udowodnienia sformułujmy *niezmiennik* oraz wykażemy jego prawdziwośc.

Twierdzenie 3. *Niech a'_i (kolejno: b'_i, $wynik_i$) będzie wartością zmiennej* `a'` *(*`b'`*,* `wynik`*) w $i-tej$ iteracji pętli* `while`*. Zachodzi następujący niezmiennik pętli:*

$$a'_i \cdot b'_i + wynik_i = a \cdot b \ .$$

Lemat 3. *Przed wejściem do pętli* `while` *niezmiennik jest prawdziwy.*

Dowód. Skoro przed przed wejściem do pętli mamy: $a'_0 = a$, $b'_0 = b$ oraz $wynik_0 = 0$, to oczywiście: $a'_0 \cdot b'_0 + wynik_0 = a \cdot b + 0 = a \cdot b$. □

Lemat 4. *Po $i-tym$ obrocie pętli niezmiennik jest spełniony.*

Dowód. Załóżmy, że niezmiennik zachodzi w $i-tej$ iteracji i sprawdźmy co dzieje się w $i+1$ iteracji. Rozważmy dwa przypadki.

Nie możesz czegoś takiego założyć :)

- a'_i parzyste. Instrukcja `if` się nie wykona, w $i+1$ iteracji $wynik_i$ pozostanie niezmieniony, a'_i zmniejszy się o połowę, a b'_i zwiększy dwukrotnie.

$$wynik_{i+1} = wynik_i$$
$$a'_{i+1} = a'_i \operatorname{div} 2 = \frac{a'_i}{2}$$
$$b'_{i+1} = b'_i \cdot 2$$

  W tym przypadku otrzymujemy:

$$a'_{i+1} \cdot b'_{i+1} + wynik_{i+1} = \frac{a'_i}{2} \cdot 2b'_i + wynik_i = a'_i \cdot b'_i + wynik_i = a \cdot b$$

- a'_i nieparzyste:

$$wynik_{i+1} = wynik_i + b'_i$$
$$a'_{i+1} = a'_i \text{ div } 2 = \frac{a'_i - 1}{2}$$
$$b'_{i+1} = b'_i \cdot 2$$

Ostatecznie otrzymujemy:

$$a'_{i+1} \cdot b'_{i+1} + wynik_{i+1} = \frac{a'_i - 1}{2} \cdot 2b'_i + wynik_i + b'_i = a'_i \cdot wynik_i + b'_i = a \cdot b$$

□

Lemat 5. *Po zakończeniu algorytmu wynik* $= a \cdot b$

Dowód. Wystarczy zauważyć, że tuż po wyjściu z pętli `while` wartość zmiennej a' wynosi 0. Podstawiając do niezmiennika okazuje się, że faktycznie algorytm rosyjskich wieśniaków liczy $a \cdot b$. □

Lemat 6. *Algorytm sie kończy.*

Dowód. Skoro $a_i \in \mathbb{N}$ oraz $\mathbb{N}$ jest dobrze uporządkowany, to połowiąc a_i po pewnej liczbie iteracji otrzymamy 0. □

Z powyższych lematów wynika, że niemiennik spełniony jest zarówno przed, w trakcie jak i po zakończeniu algorytmu. Algorytm rosyjskich wieśniaków jest poprawny.

Złożoność Z każdą iteracją połowimy a'. Biorąc pod uwagę kryterium jednorodne pozostałe instrukcje w pętli nic nie kosztują. Stąd złożoność to $O(\log a)$.

W kryterium logarytmicznym musimy uwzględnić czas dominującej instrukcji: dodawania $wynik \leftarrow wynik + b'$. W najgorszym przypadku zajmuje ono $O(\log ab)$. Zatem złożoność to $O(\log a \cdot \log ab)$.

3.3 Algorytm macierzowy wyznaczania liczb Fibonacciego

Algorytm szybkiego potęgowania będzie opisany w tym skrypcie, więc jak już będzie to będziemy chciali się odwołać do odpowiedniego rozdziału a nie do wikipedii.

W tym rozdziale opiszemy algorytm obliczania liczb Fibonacciego, który wykorzystuje szybkie potęgowanie[1]. Algorytm działa w czasie $O(\log n)$, co sprawia, że jest znacznie atrakcyjniejszy (gdy pytamy tylko o jedną liczbę) od algorytmu dynamicznego, który wymaga czasu $O(n)$.

Znajdźmy taką macierz M, która po wymnożeniu przez transponowany wektor wyrazów F_n i F_{n-1} da nam wektor, w którym otrzymamy wyrazy F_{n+1} oraz F_n. Łatwo sprawdzić, że dla ciągu Fibonacciego taka macierz ma postać:

$$M = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

Bo:

$$M \times \begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = \begin{bmatrix} F_{n+1} \\ F_n \end{bmatrix} \tag{3.1}$$

Wynika to wprost z definicji mnożenia macierzy oraz definicji ciągu Fibonacciego. Wykonajmy mnożenie z równania 3.1 n razy:

$$\underbrace{M \times (M \times (M \times \ldots (M}_{\text{n razy}} \times \begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix}) \ldots)) \tag{3.2}$$

Fakt 1. *Mnożenie macierzy jest łączne.*

Z Faktu 1. i wyrażenia 3.2. mamy:

$$M^n \times \begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix}$$

Pokażemy, że powyższa macierz ma zastosowanie w obliczaniu n-tej liczby Fibonacciego.

Lemat 7.

$$M^n \times \begin{bmatrix} F_1 \\ F_0 \end{bmatrix} = \begin{bmatrix} F_{n+1} \\ F_n \end{bmatrix}$$

Dowód. Przez indukcję.
Sprawdźmy dla $n = 1$. Mamy:

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^1 \times \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} F_2 \\ F_1 \end{bmatrix}$$

[1] `https://en.wikipedia.org/wiki/Exponentiation_by_squaring`

Rozważmy $n+1$ zakładając poprawność dla n.

$$\begin{bmatrix}1 & 1\\ 1 & 0\end{bmatrix}^{n+1} \times \begin{bmatrix}1\\ 0\end{bmatrix} = \begin{bmatrix}1 & 1\\ 1 & 0\end{bmatrix} \times \begin{bmatrix}1 & 1\\ 1 & 0\end{bmatrix}^{n} \times \begin{bmatrix}1\\ 0\end{bmatrix} = \begin{bmatrix}1 & 1\\ 1 & 0\end{bmatrix} \times \begin{bmatrix}F_{n+1}\\ F_n\end{bmatrix} \overset{3.1}{=} \begin{bmatrix}F_{n+2}\\ F_{n+1}\end{bmatrix}$$

□

Algorytm 7: Procedura `get_fibonacci`

Input: n
Output: n-ta liczba Fibonacciego
$M \leftarrow \begin{bmatrix}1 & 1\\ 1 & 0\end{bmatrix}$
$M' \leftarrow$ `exp_by_squaring(M, n - 1)`
$M'' \leftarrow M' \times \begin{bmatrix}1\\ 0\end{bmatrix}$
return $M''_{1,1}$

Mimo że powyższy algorytm działa w czasie $O(\log n)$, warto mieć na uwadze fakt, że liczby Fibonacciego rosną wykładniczo. W praktyce oznacza to pracę na liczbach przekraczających długość słowa maszynowego.

Zaprezentowaną metodę można uogólnić na dowolne ciągi, które zdefiniowane są przez liniową kombinację skończonej liczby poprzednich elementów. Wystarczy znaleźć odpowiednią macierz M; dla ciągów postaci:

$$G_{n+1} = a_n G_n + a_{n-1} G_{n-1} + ... + a_{n-k} G_{n-k}$$

wygląda ona następująco:

$$M = \begin{bmatrix} a_n & a_{n-1} & a_{n-2} & \dots & a_{n-k+1} & a_{n-k} \\ 1 & 0 & 0 & \dots & 0 & 0 \\ 0 & 1 & 0 & \dots & 0 & 0 \\ 0 & 0 & \ddots & & & \\ \vdots & & & \ddots & & \\ 0 & 0 & 0 & \dots & 1 & 0 \end{bmatrix}$$

Dowód tej konstrukcji pozostawiamy czytelnikowi jako ćwiczenie.

3.4 Sortowanie topologiczne

Rysunek 3.1: Przykładowy graf z ubraniami dla bramkarza hokejowego. Krawędź między wierzchołkami a oraz b istnieje wtedy i tylko wtedy, gdy gracz musi ubrać a zanim ubierze b. Pytanie o to w jakiej kolejności bramkarz powinien się ubierać, jest pytaniem o posortowanie topologiczne tego grafu.

3.5 Algorytmy sortowania

W tym rozdziale zapoznamy się z algorytmem sortowania przez scalanie (ang. *merge sort*). Wykorzystuje on metodę "dziel i zwyciężaj" - problem jest dzielony na kilka mniejszych podproblemów podobnych do początkowego problemu, problemy te są rozwiązywane rekurencyjnie, a następnie rozwiązania otrzymane dla podproblemów scala się, uzyskując rozwiązanie całego zadania.

Idea. Algorytm sortujący dzieli porządkowany n-elementowy zbiór na kolejne połowy, aż do uzyskania n jednoelementowych zbiorów - każdy taki zbiór jest już posortowany. Uzyskane w ten sposób części zbioru sortuje rekurencyjnie - posortowane części łączy ze sobą za pomocą scalania tak, aby wynikowy zbiór był posortowany.

Scalanie. Podstawową operacją algorytmu jest scalanie dwóch uporządkowanych zbiorów w jeden uporządkowany zbiór. W celu wykonania scalania skorzystamy z pomocniczej procedury `merge`(A, p, q, r), gdzie A jest tablicą, a p, q, r są indeksami takimi, że $p \leqslant q < r$. W procedurze zakłada się, że tablice $A[p..q]$ oraz $A[q+1..r]$ (dwie przyległe połówki zbioru, który został przez ten algorytm podzielony) są posortowane. Procedura `merge` scala te tablice w jedną posortowaną tablicę $A[p..r]$. Ogólna zasada działania jest następująca:

1. Przygotuj pusty zbiór tymczasowy.

2. Dopóki żaden ze scalanych zbiorów nie wyczerpał elementów, porównuj ze sobą pierwsze elementy każdego z nich i w zbiorze tymczasowym umieszczaj mniejszy z elementów.

3. W zbiorze tymczasowym umieść zawartość tego scalanego zbioru, który zawiera niewykorzystane jeszcze elementy.

4. Zawartość zbioru tymczasowego przepisz do zbioru wynikowego i zakończ algorytm.

Zapis algorytmu scalania dwóch list w pseudokodzie podano niżej.

Scalanie wymaga $O(n+m)$ operacji porównań elementów i wstawienia ich do tablicy wynikowej.

Sortowanie. Algorytm sortowania przez scalanie jest algorytmem rekurencyjnym. Wywołuje się go z zadanymi wartościami indeksów wskazujących na początek i koniec sortowanego zbioru, zatem początkowo indeksy obejmują cały zbiór. Algorytm wyznacza indeks elementu połowiącego przedział, a następnie sprawdza, czy połówki zbioru zawierają więcej niż jeden element. Jeśli tak, to rekurencyjnie sortuje je tym samym algorytmem. Po posortowaniu obu połówek zbioru scalamy je za pomocą opisanej wcześniej procedury scalania podzbiorów uporządkowanych i kończymy algorytm. Zbiór jest posortowany.

Algorytm 8: Procedura `merge`

Input: tablica A, liczby p, q, r
Output: posortowana tablica $A[p..r]$
$C \leftarrow$ pusta tablica
$i \leftarrow p$, $j \leftarrow q+1$, $k \leftarrow 0$
while $i \leqslant q$ *oraz* $j \leqslant r$ **do**
  if $A[i] \leqslant A[j]$ **then**
    $C[k] \leftarrow A[i]$, $i \leftarrow i+1$
  else
    $C[k] \leftarrow A[j]$, $j \leftarrow j+1$
  end
  $k \leftarrow k+1$
end
while $i \leqslant q$ **do**
  $C[k] \leftarrow A[i]$, $i \leftarrow i+1$, $k \leftarrow k+1$
end
while $j \leqslant r$ **do**
  $C[k] \leftarrow A[j]$, $j \leftarrow j+1$, $k \leftarrow k+1$
end

Złożoność. Chociaż algorytm sortowania przez scalanie działa poprawnie nawet wówczas, gdy n jest nieparzyste, dla uproszczenia analizy załóżmy, że n jest potęgą dwójki. Dzielimy wtedy problem na podproblemy rozmiaru dokładnie $\frac{n}{2}$. Rekurencję określającą czas $T(n)$ sortowania przez scalanie otrzymujemy, jak następuje.

Sortowanie przez scalanie jednego elementu wykonuje się w czasie stałym. Jeśli $n > 1$, to czas działania zależy od trzech etapów:
Dziel: podczas tego etapu znajdujemy środek przedziału, co zajmuje czas stały, zatem $D(n) = \theta(1)$.
Zwyciężaj: rozwiązujemy rekurencyjnie dwa podproblemy, każdy rozmiaru $\frac{n}{2}$, co daje czas działania $2T(\frac{n}{2})$.
Połącz: procedura `merge`, jak wspomniano wyżej, działa w czasie liniowym, a więc $P(n) = \theta(n)$.

Funkcje $D(n)$ i $P(n)$ dają po zsumowaniu funkcję rzędu $\theta(n)$. Dodając do tego $2T(\frac{n}{2})$ z etapu "zwyciężaj", otrzymujemy następującą rekurencję dla $T(n)$:

$$T(n) = \begin{cases} \theta(1) & n = 1 \\ 2T(\frac{n}{2}) + \theta(n) & n > 1 \end{cases}$$

Algorytm 9: Procedura `merge sort`

Input: tablica A, liczby p, r
Output: posortowana tablica $A[p..r]$
$q \leftarrow 0$
if $p < r$ **then**
  $q \leftarrow \lfloor \frac{p+r}{2} \rfloor$
  `merge sort`(A, p, q)
  `merge sort`$(A, q+1, r)$
  `merge`(A, p, q, r)
end

Poniższy przykład ilustruje zasadę działania sortowania przez scalanie:

<tu obrazek, ale nie umiem w obrazki, na dniach ogarnę>

Podsumowanie. Sortowanie przez scalanie należy do algorytmów szybkich, posiada klasę złożoności równą $\theta(n \log n)$. Jest oparty na metodzie dziel i zwyciężaj, która powoduje podział dużego problemu na mniejsze, łatwo rozwiązywane podproblemy. Sortowanie nie odbywa się w miejscu, potrzebujemy dodatkowej struktury. Algorytm jest stabilny.

3.5.1 Quick sort

In progress

3.6 Minimalne drzewa rozpinające

Todo, todo, todo...

3.6.1 Cut Property i Circle Property

3.6.2 Algorytm Prima

3.6.3 Algorytm Kruskala

3.6.4 Algorytm Borůvki

3.7 Algorytm Dijkstry

Todo, todo, todo...

3.8 Algorytm szeregowania

Todo, todo, todo...

3.9 Programowanie dynamiczne na drzewach

Todo, todo, todo...

3.10 Mnożenie macierzy

Z mnożeniem macierzy mieliście już prawdopodobnie do czynienia na Algebrze. Mając dane dwie macierze nad ciałem liczb rzeczywistych A (o rozmiarze $n \times m$) oraz B (o rozmiarze $m \times p$), chcemy policzyć ich iloczyn:

$$A \cdot B = C$$

gdzie elementy macierzy C (o rozmiarze $n \times p$) zadane są wzorem:

$$c_{i,j} = \sum_{r=1}^{m} a_{i,r} \cdot b_{r,j}$$

Przykładowo:

$$\begin{bmatrix} 1 & 0 & 2 \\ 0 & 3 & 1 \end{bmatrix} \cdot \begin{bmatrix} 3 & 1 \\ 2 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} (1 \cdot 3 + 0 \cdot 2 + 2 \cdot 1) & (1 \cdot 1 + 0 \cdot 1 + 2 \cdot 0) \\ (0 \cdot 3 + 3 \cdot 2 + 1 \cdot 1) & (0 \cdot 1 + 3 \cdot 1 + 1 \cdot 0) \end{bmatrix} = \begin{bmatrix} 5 & 1 \\ 7 & 3 \end{bmatrix}$$

Korzystając prosto z definicji możemy napisać następujący algorytm mnożenia dwóch macierzy:

Algorytm 10: Naiwny algorytm mnożenia macierzy

```
Input: A, B - macierze o rozmiarach n × m oraz m × p
Output: C = A · B
for i ← 1 to n do
    for j ← 1 to p do
        C[i][j] ← 0
        for r ← 1 to m do
            C[i][j] ← C[i][j] + A[i][r] · B[r][j]
        end
    end
end
```

Powyższy algorytm działa w czasie $O(n^3)$. Korzystając ze sprytnej sztuczki, jesteśmy w stanie zmniejszyć złożoność naszego algorytmu.

Zacznijmy od założenia, że rozmiar macierzy jest postaci $2^k \times 2^k$. Jeśli macierze nie są takiej postaci, to możemy uzupełnić brakujące wiersze i kolumny zerami. Następnie podzielmy macierze na cztery równe części:

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} \quad B = \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix} \quad C = \begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix}$$

Każda z części jest rozmiaru $2^{k-1} \times 2^{k-1}$. Ponadto wzór na każdą część macierzy C wyraża się wzorem:

$$C_{i,j} = A_{i,1} \cdot B_{1,j} + A_{i,2} \cdot B_{2,j}$$

Czy wzór ten umożliwia nam ułożenie efektywnego algorytmu mnożenia macierzy? Nie. W algorytmie mamy do policzenia 4 podmacierze macierzy C. Każda podmacierz wymaga 2 mnożeń oraz jednego dodawania. Dodawanie macierzy możemy w prosty sposób zrealizować w czasie $O(n^2)$. Mnożenie podmacierzy możemy wykonać rekurencyjnie. Taki algorytm będzie działał w czasie $T(n) = 8 \cdot T(n/2) + O(n^2)$ czyli $O(n^3)$. Osiągnęliśmy tą samą złożoność czasową jak w przypadku algorytmu liczącego iloczyn wprost z definicji.

Algorytm Strassena osięga lepszą złożoność asymptotyczną przez pozbycie się jednego z mnożeń. Algorytm ten liczy następujące macierze:

$$
\begin{aligned}
M_1 &= (A_{1,1} + A_{2,2}) \cdot (B_{1,1} + B_{2,2}) \\
M_2 &= (A_{2,1} + A_{2,2}) \cdot B_{1,1} \\
M_3 &= A_{1,1} \cdot (B_{1,2} - B_{2,2}) \\
M_4 &= A_{2,2} \cdot (B_{2,1} - B_{1,1}) \\
M_5 &= (A_{1,1} + A_{1,2}) \cdot B_{2,2} \\
M_6 &= (A_{2,1} - A_{1,1}) \cdot (B_{1,1} + B_{1,2}) \\
M_7 &= (A_{1,2} - A_{2,2}) \cdot (B_{2,1} + B_{2,2})
\end{aligned}
$$

Do policzenia każdej z tych macierzy potrzebujemy jednego mnożenia i co najwyżej dwóch dodawań/odejmowań. Podmacierze macierzy C możemy policzyć teraz w następujący sposób:

$$
\begin{aligned}
C_{1,1} &= M_1 + M_4 - M_5 + M_7 \\
C_{1,2} &= M_3 + M_5 \\
C_{2,1} &= M_2 + M_4 \\
C_{2,2} &= M_1 - M_2 + M_3 + M_6
\end{aligned}
$$

Wykonując proste przekształcenia arytmetyczne, możemy dowieść poprawności powyższych równań.

Używając powyższych wzorów, możemy skonstruować algorytm rekurencyjny. Będzie on dzielić macierze A oraz B o rozmiarze $2^k \times 2^k$ na cztery równe części. Następnie policzy on macierze M_i. Tam, gdzie będzie musiał dodawać/odejmować użyje on algorytmu działającego w czasie $O(n^2)$. Tam, gdzie będzie musiał mnożyć - wywoła się on rekurencyjnie. Na podstawie macierzy M_i policzy macierz C. Ponieważ wykona dokładnie 7 mnożeń oraz stałą ilość dodawań, jego złożoność obliczeniowa będzie wyrażała się wzorem rekurencyjnym $T(n) = 7 \cdot T(n/2) + O(n^2)$. Korzystając z twierdzenia o rekurencji uniwersalnej otrzymujemy złożoność $O(n^{\log_2 7})$ czyli około $O(n^{2.81})$.

Dodatek A

Porównanie programów przedmiotu AiSD na różnych uczelniach

	UWr	UW	UJ	MIT	Oxford
Stosy, kolejki, listy		✓			
Dziel i zwyciężaj	✓				
Programowanie Dynamiczne	✓	✓	✓	✓	
Metoda Zachłanna	✓	✓	✓		
Koszt zamortyzowany	✓	✓			✓
NP-zupełność	✓	✓		✓	
PRAM / NC	✓				
Sortowanie	✓	✓			
Selekcja	✓	✓			
Słowniki	✓	✓	✓		✓
Kolejki priorytetowe	✓	✓			
Hashowanie	✓	✓			
Zbiory rozłączne	✓				
Algorytmy grafowe	✓	✓	✓	✓	✓
Algorytmy tekstowe	✓	✓			
Geometria obliczeniowa	✓				
FFT	✓				✓
Algorytm Karatsuby	✓			✓	
Metoda Newtona				✓	
Algorytmy randomizowane	✓				✓
Programowanie liniowe					✓
Algorytmy aproksymacyjne	✓				✓
Sieci komparatorów	✓				
Obwody logiczne	✓				