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

Rozdział 1

Podstawy

1.1 Złożoność obliczeniowa

Todo, todo, todo...

1.2 Model obliczeń

Todo, todo, todo...

Rozdział 2

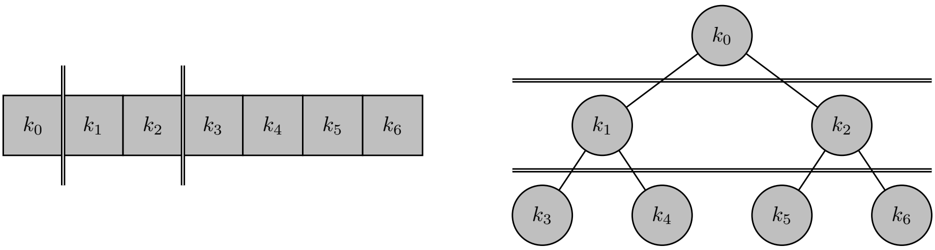
Struktury danych

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).

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 $\left\lfloor \frac{i-1}{2} \right\rfloor$.

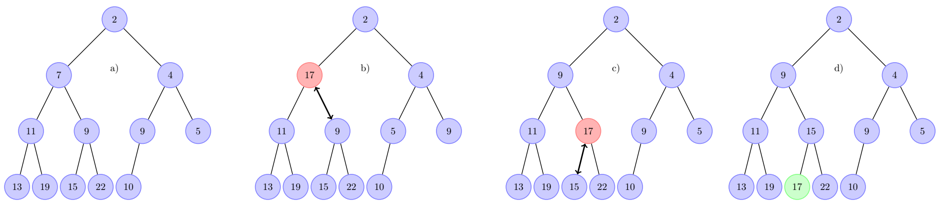
Kopiec powinien udostępniać trzy podstawowe funkcje: `zamien_element`, która podmienia wartość w konkretnym węźle kopca, `przesun_w_gore` oraz `przesun_w_dol`, które zamieniają odpowiednie elementy pilnując przy tym, aby własność kopca została zachowana.

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

Algorithm 1: 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 2). 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 2: Procedura `bitonic_compare`

Input: A, n, up
for $i \leftarrow 1$ *to* n **do**
  if $(A[i] > A[i + n/2]) = up$ **then**
    $A[i] \leftrightarrow A[i + n/2]$;
  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 1. *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 1 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 1 jest prawdziwe dla wszystkich ciągów zero-jedynkowych.

Lemat 2. *Twierdzenie 1 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 1 jest prawdziwe dla wszystkich tablic `T[0..n-1]`. Mając tak piękne twierdzenie, możemy napisać prosty algorytm sortujący ciągi bitoniczne (Algorytm 3).

Algorytm 3: 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 4: 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. Mnożone liczby, a i b zapisujemy w obok siebie. Tworzą one nagłówek dwukolumnowej tabeli, w wierszach której wpisujemy się $\lfloor \frac{a'}{2} \rfloor$ i $2 \cdot b'$, gdzie a' i b' to wartości z wiersza powyżej. Czynności te wykonujemy, aż uzyskamy 1 w kolumine pod a. Następnie sumujemy wartości w kolumine pod b z tych wierszy dla których wartości w kolumine pod a są nieparzyste. Uzyskany wynik to $a \cdot b$.

Zawile napisane. Dodać przykład. Napisać, że to analog mnożenia pisemnego w systemie binarnym.

Algorytm 5: 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

Niech a'_i (kolejno: b'_i, $wynik_i$) będzie wartością zmiennych `a'` (`b'`, `wynik`) w $i - tej$ iteracji pętli `while`. Udowodnimy następujący niezmiennik:

Jakiś wstęp, do czego służy algorytm, przykład działania, analogia między algorytmem a algorytmem szybkiego potęgowania. Dowód poprawności jako jakieś twierdzenie / lemat. Popracować nad składaniem latexa.

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

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

- 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:

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

Ostatecznie otrzymujemy:

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

Teraz 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 $a \cdot b$.

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

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 od algorytmu dynamicznego, który wymaga czasu $O(n)$.

W niektórych przypadkach lepiej używać algorytmu dynamicznego (w jakich?) Algorytm szybkiego potęgowania będzie opisany w tym skrypcie, więc jak już będzie to będziemy chciali się odwołać do odpowiedniego rozdziału a nie do wikipedii.

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:

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

Jeśli nie będziesz korzystać z danego wzoru w rozdziale, to lepiej pominąć jego numerek. Robi się to przez dodanie symbolu * do equation

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.

Nawiasy okrągłe są zbyt małe. Zastąpić je należy \left(oraz \right).

Obserwacja 1. *Zauważmy, że możemy M przemnożyć przez macierz otrzymaną w 3.1. Otrzymamy wtedy macierz postaci:*

$$M \times \left(M \times \begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} \right) \tag{3.2}$$

To nie są obserwacje. Obserwacje by były, gdybyś napisał czemu te wzory są równe.

Obserwacja 2. *A gdy zrobimy to n razy...*

$$M \times (M \times (M \times ... (M \times \begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix}) ...)) \tag{3.3}$$

Fakt 1. *Mnożenie macierzy jest łączne.*

Z Faktu 1. i Obserwacji 2. mamy:

Jeśli nie będziesz wstawiał niepotrzebnych enterów to latex nie będzie tak brzydku tabulował tych wierszy.

$$M^n \times \begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} \tag{3.4}$$

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

Lemat 3.

$$M^n \times \begin{bmatrix} F_1 \\ F_0 \end{bmatrix} = \begin{bmatrix} F_{n+1} \\ F_n \end{bmatrix} \tag{3.5}$$

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

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

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^1 \times \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} F_2 \\ F_1 \end{bmatrix} \tag{3.6}$$

Rozważmy $n + 1$ zakładając poprawność dla n.

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n+1} \times \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n \times \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} F_{n+1} \\ F_n \end{bmatrix} \stackrel{3.1}{=} \begin{bmatrix} F_{n+2} \\ F_{n+1} \end{bmatrix} \tag{3.7}$$

□

Algorytm 6: Procedura `get_fibonacci`

Input: n
Output: $n + 1$-sza liczba Fibonacciego
$M \leftarrow \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$
$M' \leftarrow$ `exp_by_squaring(M, n)`
return *`pierwszy element wektora`*$(M' \times \begin{bmatrix} 1 \\ 0 \end{bmatrix})$

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.

Chcemy do konstrukcji dodać jeszcze wielomiany (aka rozwiązać zadanie z listy?)

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

$$M = \begin{bmatrix} a_n & a_{n-1} & a_{n-2} & \dots & a_{n-k} & a_{n-k} \\ 1 & 0 & 0 & \dots & 0 & 0 \\ 0 & 1 & 0 & \dots & 0 & 0 \\ 0 & 0 & \ddots & & & \\ \vdots & & & \ddots & & \\ 0 & 0 & 0 & \dots & 1 & 0 \end{bmatrix} \tag{3.8}$$

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

Todo, todo, todo...

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...

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	✓				