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

Zrobione

1.1 Twierdzenie o rekurencji uniwersalnej

Popularną metodą rozwiązywania zadań jest metoda Dziel i Zwyciężaj. Polega ona na podzieleniu problemu na mniejsze, rozwiązaniu ich w sposób rekurencyjny, a następnie na scaleniu wyniku w jeden. Schemat tej metody jest przedstawiony jako Schemat 1.

Schemat 1: Procedura `Dziel_i_zwyciezaj`

if $n \leqslant 1$ **then**
  rozwiąż trywialny przypadek
end
Stwórz a podproblemów wielkości n/b w czasie $D(n)$
for $i \leftarrow 1$ *to* a **do**
  wykonaj procedurę `Dziel_i_zwyciezaj` rekurencyjnie dla i-tego podproblemu
end
Połącz wyniki w czasie $P(n)$

Złożoność takiego algorytmu możemy zapisać zależnością rekurencyjną $T(n) = aT(n/b) + \Theta(n^k \log^p n)$ przy czym $P(n) + D(n) \in \Theta(n^k \log^p n)$. Jednakże zależność rekurencyjna na czas działania algorytmu nie zawsze nas satysfakcjonuje. Zazwyczaj chcielibyćmy uzyskać wzór zwarty. Do tego celu służy poniższe twierdzenie, znane Twierdzeniem o rekurencji uniwersalnej.

Twierdzenie 1. *Niech* $T(n) = aT(n/b) + \Theta(n^k \log^p n)$ *oraz* $a \geqslant 1$, $b > 1$, $k \geqslant 0$ *oraz* p *liczby rzeczywiste. Wtedy*

1. *jeżeli* $a > b^k$, *to* $T(n) \in \Theta(n^{\log_b a})$
2. *jeżeli* $a = b^k$ *oraz*
   (a) $p > -1$ *to* $T(n) \in \Theta(n^{\log_b a} \log^{p+1} n)$
   (b) $p = -1$ *to* $T(n) \in \Theta(n^{\log_b a} \log \log n)$
   (c) $p < -1$ *to* $T(n) \in \Theta(n^{\log_b a})$
3. *jeżeli* $a < b^k$ *oraz*
   (a) $p \geqslant 0$ *to* $T(n) \in \Theta(n^k \log^p n)$
   (b) $p < 0$ *to* $T(n) \in O(n^k)$

Dowód. TODO TODO TODO. □

1.1.1 Przykłady wykorzystania twierdzenia

W rozdziale 2.6 zapoznamy się z algorytmem sortowania przez scalanie. Jego złożoność określona jest wzorem rekurencyjnym $T(n) = 2 \cdot T(n/2) + \Theta(n)$. W tym przypadku $a = 2$, $b = 2$, $k = 1$ oraz $p = 0$. Ponieważ $a = b^k$ sprawdzamy dodatkowo, że $p > -1$ i otrzymujemy, że w naszym przypadku powinniśmy skorzystać z pkt 2a. Otrzymujemy, że złożoność naszego algorytmu jest $\Theta(n^{\log_b a} \log^{p+1} n)$ czyli $\Theta(n \log n)$.

Nie podoba mi się to, że to jest osobny podrozdział. Nie wiem czy to powinnien być osobny akapit, tabelka czy może coś jeszcze innego.

W rodziale 1.2 opisany został algorytm sortowania bitonicznego. Jego złożoność określona jest wzorem $T(n) = 2 \cdot T(n/2) + \Theta(n \log n)$. W tym przypadeku $a = 2$, $b = 2$, $k = 1$ i $p = 1$. Ponieważ $a = b^k$ i $p > -1$ korzystamy z punktu 2a. Otrzymujemy złożoność $\Theta(n \log^2 n)$.

Naprawić referencję.

W rozdziale **??** opisany jest algorytm Karatsuby. Jego złożoność opisana jest rekurencyjnym wzorem $3T(n/2) + \Theta(n)$. W tym przypadku $a = 3$, $b = 2$, $k = 1$ i $p = 0$. Ponieważ $a > b^k$ korzystamy z punktu 1. Otrzymujemy złożoność $\Theta(n^{\log_2 3})$ czyli około $\Theta(n^{1.585})$.

Wyszukiwanie binarne to algorytm klasy Dziel i Zwyciężaj wyszukujący element w posortowanym ciągu. Ma złożoność opisaną rekurencyjnym wzorem $T(n) = T(n/2) + \Theta(1)$. Mamy więc $a = 1$, $b = 2$, $k = p = 0$. Ponieważ $a = b^k$ i $p > -1$ korzystamy z punktu 2a. Otrzymujemy złożoność $\Theta(\log n)$.

Algorytm Strassena jest opisany w rodziale 1.4. Jego złożoność opisana jest rekurencyjnym wzorem $T(n) = 7 \cdot T(n/2) + \Theta(n^2)$. W tym przypadku mamy $a = 7$, $b = 2$, $k = 2$, $p = 0$. Jako, że $a > b^k$ wykorzystamy punkt 1. Otrzymujemy złożoność $\Theta(n^{\log_2 7})$ czyli około $\Theta(n^{2.807})$

1.2 Sortowanie bitoniczne

W tym rozdziale przedstawimy algorytm sortowania bitonicznego. Jest to algorytm działający w czasie $\Theta(n \log^2 n)$ 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 właściwym, 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 właściwym, 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 powstał 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[0..n-1], up
for i ← 0 to n − 1 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 niemniejszy.*

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. 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 nie wprost. 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 element pierwszej z nich jest większy od któregoś elementu 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 następującą 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 $f(S[i+1]) = 0$.

1	2	3	5	4	6	7	1	`S[]`
0	0	0	1	0	1	1	0	$f(S)$
			i	$i+1$				

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 $f(S[j]) = 1$.

1	2	100	2	102	99	103	107	`S[]`
0	0	1	0	1	0	1	1	$f(S)$
		j			i			

□

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). Istnieje sześć rodzai 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 $0^k1^l0^{m-n/2}$. 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 Lematów 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 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 $T(n) = 2 \cdot T(n/2) + \Theta(n)$. Rozwiązując rekurencję otrzymujemy, że złożoność algorytmu to $\Theta(n \log n)$.

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.

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

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

1.3 Algorytm macierzowy wyznaczania liczb Fibonacciego

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

$$F_n = \begin{cases} n, & \text{jeśli } n \leqslant 1 \\ 0, & \text{wpp.} \end{cases}$$

Teraz, 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:

$$M \times \begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = \begin{bmatrix} F_n + F_{n-1} \\ F_n \end{bmatrix} = \begin{bmatrix} F_{n+1} \\ F_n \end{bmatrix} \tag{1.1}$$

Wynika to wprost z definicji mnożenia macierzy oraz definicji ciągu Fibonacciego. Wykonajmy mnożenie z równania 1.1 n razy:

$$\underbrace{M \times \left(M \times \left(M \times ... \left(M \times \right.\right.\right.}_{\text{n razy}} \left.\left.\left. \begin{bmatrix} F_1 \\ F_0 \end{bmatrix} \right) ... \right) \right)$$

Z faktu, że mnożenie macierzy jest łączne oraz powyższego wyrażenia otrzymujemy:

$$M^n \times \begin{bmatrix} F_1 \\ F_0 \end{bmatrix}$$

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

Lemat 3.

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

Dowód przez indukcję. Sprawdźmy dla $n = 0$. Mamy:

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^0 \times \begin{bmatrix} 1 \\ 0 \end{bmatrix} = I \times \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} F_1 \\ F_0 \end{bmatrix}$$

Rozważmy $n+1$ zakładając poprawność dla n.

$$\begin{bmatrix}1 & 1\\ 1 & 0\end{bmatrix}^{n+1} \times \begin{bmatrix}1\\ 0\end{bmatrix} = \begin{bmatrix}1 & 1\\ 1 & 0\end{bmatrix} \times \begin{bmatrix}1 & 1\\ 1 & 0\end{bmatrix}^{n} \times \begin{bmatrix}1\\ 0\end{bmatrix} \overset{teza}{=} \begin{bmatrix}1 & 1\\ 1 & 0\end{bmatrix} \times \begin{bmatrix}F_{n+1}\\ F_n\end{bmatrix} \overset{1.1}{=} \begin{bmatrix}F_{n+2}\\ F_{n+1}\end{bmatrix}$$

□

Algorytm 5: 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 $\Theta(\log n)$, 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:

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

Dowód tej konstrukcji pozostawiamy Czytelnikowi jako ćwiczenie.

1.4 Algorytm Strassena

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 6: 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 $\Theta(n^3)$. 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 $\Theta(n^2)$. Mnożenie podmacierzy możemy wykonać rekurencyjnie. Taki algorytm będzie działał w czasie $T(n) = 8 \cdot T(n/2) + \Theta(n^2)$ czyli $\Theta(n^3)$. 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 $\Theta(n^2)$. 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 $T(n) = 7 \cdot T(n/2) + \Theta(n^2)$. Korzystając z twierdzenia o rekurencji uniwersalnej otrzymujemy złożoność $\Theta(n^{\log_2 7})$ czyli około $\Theta(n^{2.81})$.

1.5 Model afinicznych drzew decyzyjnych

Zdefiniujmy następujący problem (ang. element uniqness). Mając daną tablicę `T[0..n-1]` liczb rzeczywistych, odpowiedzieć na pytanie czy istnieją w tablicy dwa elementy, które są sobie równe. Pierwsze rozwiązanie jakie przychodzi wielu ludziom do głowy, to posortować tablicę `T` a następnie sprawdzić sąsiednie elementy. Algorytm ten rozwiązuje nasz problem w czasie $\Theta(n \log n)$. Pytanie - czy da się szybciej? W niniejszym rozdziale udowodnimy, że w modelu afinicznych drzew decyzyjnych problemu nie da się rozwiązać lepiej.

W modelu afinicznych drzew decyzyjnych, w każdym zapytaniu możemy wybrać sobie $n+1$ liczb: c, a_0, a_1, $\ldots$, a_{n-1}, a następnie zapytać czy

$$c + \sum_{i=0}^{n-1} a_i t_i \geqslant 0$$

gdzie t_i to elementy tablicy `T`. Gdybyśmy użyli terminologii algebraicznej, to powiedzielibyśmy, że t jest punktem w przestrzeni $\mathbb{R}^n$, lewa strona powyższej nierówności to przekształcenie afiniczne, a zbiór wszystkich punktów z $\mathbb{R}^n$, które spełniają tą nierówność to półprzestrzeń afiniczna. Jeśli na Algebrze nie wyrobiliście sobie jeszcze intuicji, to w $\mathbb{R}^2$ półprzestrzeń afiniczną otrzymujemy przez narysowanie dowolnej prostej i wzięcie wszystkich elementów z jednej ze stron. Podobnie w $\mathbb{R}^3$ półprzestrzeń afiniczną otrzymujemy poprzez narysowanie dowolnej płaszczyzny, a następnie wzięcia wszystkich elementów z jednej ze stron. W wyższych wymiarach wygląda to analogicznie.

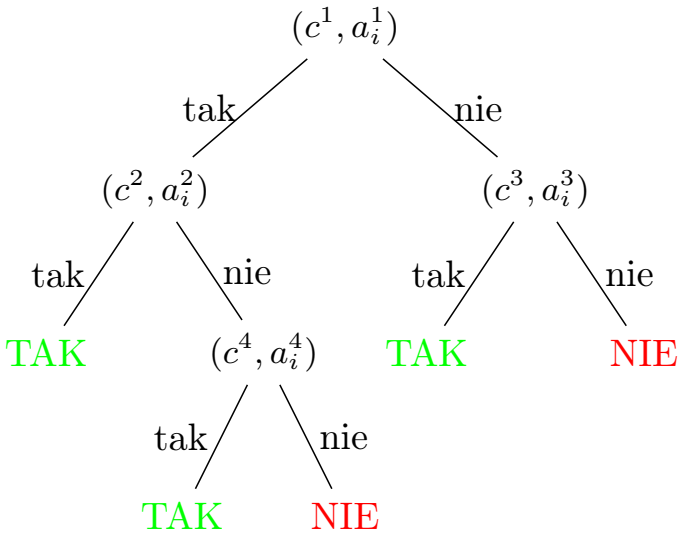
Algorytm używający tego typu porównań można zapisać za pomocą drzewa (rys. 1.5). Zaczynamy z korzenia tego drzewa. W każdym wierzchołku wewnętrznym zadajemy zapytanie. W zależności od tego czy odpowiedź na pytanie była pozytywna czy negatywna, idziemy w drzewie w lewo lub w prawo. Gdy dojdziemy do liścia w drzewie otrzymujemy nasze rozwiązanie (tak lub nie). Takie drzewo będziemy nazywać afinicznym drzewem decyzyjnym.

Aby było nam łatwiej zdefiniować główny lemat naszego rozdziału, zdefiniujemy sobie dwa pojęcia.

Definicja 3. *Mówimy, że punkt $t \in \mathbb{R}^n$* ***osiąga*** *liść l w afinicznym drzewie decyzyjnym, jeśli algorytm uruchomiony dla punktu t dochodzi do liścia l.*

Definicja 4. *Mówimy, że podzbiór $C \subseteq \mathbb{R}^n$ jest* ***zbiorem wypukłym****, jeśli dla dowolnych punktów $u, v \in C$ oraz dowolnej liczby rzeczywistej $0 \leqslant \alpha \leqslant 1$ punkt $\alpha \cdot u + (1-\alpha)v$ także należy do C.*

Pierwsza z definicji pozwala nam mówić o elementach, które trafiają do tego samego liścia, a druga to sformalizowane pojęcie wypukłości znane z liceum. Uzbrojeni w nowe definicje, możemy przejść do obiecanego lematu:



Rysunek 1.1: Przykład afinicznego drzewa decyzyjnego. W wierzchołkach wewnętrznych mamy zapytanie (c^j, a_i^j). W zależności od tego czy $c^k + \sum_{i=0}^{n-1} a_i^k t_i \geqslant 0$ czy też nie, idziemy odpowiednio w lewo lub w prawo. Liście zawierają odpowiedź naszego algorytmu.

Lemat 4. *Zbiór punktów osiągających liść l w afinicznym drzewie decyzyjnym, jest zbiorem wypukłym.*

Dowód. Weźmy dowolne afiniczne drzewo decyzyjne i wybierzmy w nim dowolny liść l. Dobrze wiemy, że istnieje dokładnie jedna ścieżka prosta z korzenia do tego liścia. Weźmy dowolny wierzchołek wewnętrzny w na tej ścieżce i dowolne punkty u oraz v, które osiągają liść l. W końcu weźmy dowolną liczbę rzeczywistą $0 \leqslant \alpha \leqslant 1$. Załóżmy ponadto, że ścieżka z korzenia do liścia l w wierzchołku w skręca w lewo (zatem zapytanie zadane w wierzchołku w punkty u oraz v otrzymały odpowiedź twierdzącą). Przypadek przeciwny jest analogiczny. Wiemy zatem, że

$$c^w + \sum_{i=0}^{n-1} a_i^w u_i \geqslant 0$$

oraz, że

$$c^w + \sum_{i=0}^{n-1} a_i^w v_i \geqslant 0$$

Ponieważ $\alpha \geqslant 0$ możemy przemnożyć pierwsze równanie przez α:

$$\alpha c^w + \sum_{i=0}^{n-1} a_i^w \alpha u_i \geqslant 0$$

a ponieważ $1 - \alpha \geqslant 0$ możemy drugie równanie przemnożmyć przez $1 - \alpha$:

$$(1-\alpha) c^w + \sum_{i=0}^{n-1} a_i^w (1-\alpha) v_i \geqslant 0$$

Teraz sumując oba równania otrzymujemy:

$$c^w + \sum_{i=0}^{n-1} a_i^w (\alpha u_i + (1-\alpha) v_i) \geqslant 0$$

Zatem punkt $\alpha \cdot u + (1-\alpha)v$ również w wierzchołku w skręci w tą samą stronę co punkty u oraz v. Ponieważ wybraliśmy dowolny wierzchołek w, to punkt $\alpha \cdot u + (1-\alpha)v$ osiągnie liść l. Stąd zbiór wszystkich punktów, które osiągają liść l w afinicznym drzewie decyzyjnym, jest zbiorem wypukłym. □

Wyobraźmy sobie, że płaszczyznę $\mathbb{R}^2$ kładziemy półpłaszczyzny. Wtedy przecięcie dowolnej liczby półpłaszczyzn jest zbiorem wypukłym. Podobnie jeśli w przestrzeni $\mathbb{R}^3$ wyznaczymy sobie półprzestrzenie, to ich przecięcie będzie tworzyło zbiór wypukły. Lemat 4 mówi, że tak samo się dzieje w każdej przestrzeni $\mathbb{R}^n$.

Ten lemat za chwilę okaże się dla nas kluczowy, gdyż za jego pomocą udowodnimy, że jeśli afiniczne drzewo decyzyjne poprawnie rozwiązuje problem element uniqness to musi posiadać conajmniej $n!$ liści. Oznacza to, że wysokość takiego drzewa musi wynosić conajmniej $(n \log n)$.

Lemat 5. *Niech $\{r_0, r_1, \ldots, r_{n-1}\}$ będzie n elementowym zbiorem liczb rzeczywistych i niech $(a_0, a_1, \ldots, a_{n-1})$ oraz $(b_0, b_1, \ldots, b_{n-1})$ będą dwoma różnymi permutacjami liczb z tego zbioru. W każdym afinicznym drzewie decyzyjnym poprawnie rozwiązującym problem element uniqness, punkty a oraz b osiągają różne liście w drzewie.*

Dowód. Dowód niewprost. Załóżmy, że a oraz b osiągają ten sam liść w drzewie. Liść ten musi odpowiadać przecząco na zadany problem, gdyż ani a ani b nie zawierają dwóch tych samych elementów. Ponieważ a oraz b składają się z tych samych liczb rzeczywistych i różnie się od siebie permutacją, to muszą istnieć takie indeksy i oraz j, że $a_i > a_j$ oraz $b_i < b_j$. Weźmy następującą wartość α:

$$\alpha = \frac{b_j - b_i}{(a_i - a_j) + (b_j - b_i)}$$

Wykonując proste przekształcenia arytmetyczne, możemy przekonać się, że $0 < \alpha < 1$. Oznacza to, na mocy lematu 4, że punkt $\alpha a + (1-\alpha)b$ również osiąga ten sam liść co punkty a i b. Ponieważ jednak zachodzi:

$$\alpha a_i + (1-\alpha)b_i = \alpha a_j + (1-\alpha)b_j$$

(o czym można się przekonać wykonując proste przekształcenia arytmetyczne), odpowiedź algorytmu dla tego punktu powinna być twierdząca. Zatem afiniczne drzewo decyzyjne dla tego punktu zwraca złą odpowiedź. Sprzeczność z założeniem, że drzewo rozwiązywało problem poprawnie. □

Weźmy dowolny n elementowy zbiór liczb rzeczywistych. Na mocy Lematu 5 każda permutacja tych liczb musi osiągać inny liść w afinicznym drzewie decyzyjnym poprawnie rozwiązującym problem element uniqness. Oznacza to, że liczba liści w takim drzewie musi wynosić przynajmniej $n!$. Zatem wysokość takiego drzewa musi wynosić conajmniej $\Omega(n \log n)$.

1.6 Problem plecakowy

Wyobraźmy sobie, że jesteśmy złodziejem. Pod przykryciem nocy, udało nam się dotrzeć w ustalone miejsce. Wszystko idzie wyjątkowo gładko. Wpisujemy kod, który otrzymaliśmy od sprzątaczki i jesteśmy już w środku. Wtem okazuje się, że zapomnieliśmy listy przedmiotów, które mieliśmy ukraść. Próbujemy zachować zimną krew i zmaksymalizować nasz zysk. Możemy ukraść przedmioty różnego rodzaju. Dokładniej rzecz ujmując - mamy M różnych typów przedmiotów. Każdy typ przedmiotu ma swoją wielkość $w_i > 0$ i swoją cenę u pasera $v_i > 0$. Wiemy, że w naszym plecaku zmieszczą się przedmioty o łącznej wielkości nie przekraczającej W. Które przedmioty mamy wybrać (i w jakiej ilości), aby zmaksymalizować nasz zysk i żeby nie przepakować naszego plecaka? Bardziej formalnie: w jaki sposób wybrać zmienne x_i tak aby zysk $\sum_{i=1}^{M} x_i \cdot v_i$ był jak największy oraz aby był zachowany warunek $\sum_{i=1}^{M} x_i \cdot w_i \leqslant W$. W zależności od tego do jakiego sklepu się włamaliśmy, rozróżniamy następujące rodzaje problemu plecakowego:

- Galeria sztuki. W tym przypadku mamy dodatkowe ograniczenie: $x_i \in \{0, 1\}$. Każde dzieło sztuki jest unikalne i nie możemy z galerii wynieść dwóch takich samych waz ani obrazów. Wersję tą nazywamy czasem dyskretnym $0/1$ problemem plecakowym.

- Sklep RTV. Tym razem ukraść możemy dwa takie same telewizory. Ale wciąż ilość przedmiotu danego typu jest ograniczona: $x_i \in \mathbb{N}$, $x_i \leqslant c_i$. Problem ten nazywamy czasem ograniczonym, dyskretnym problemem plecakowym.

- Cyberprzestrzeń. W tym przypadku możemy ukraść dowolną ilość przedmiotu danego typu (np. klucza aktywacji). Wtedy $x_i \in \mathbb{N}$. Problem ten można znaleźć w literaturze pod hasłem: nieograniczony, dyskretny problem plecakowy.

- Laboratorium chemiczne. W tym przypadku, ilość chemikaliów, które kradniemy, jest ograniczona ($0 \leqslant x_i \leqslant c_i$). Ponieważ chemikalia możemy przelewać, ilość chemikaliów nie musi wyrażać się liczbą naturalną ($x_i \in \mathbb{R}$). Jest to ciągły problem plecakowy.

Najprostszym pomysłem jaki wpada do głowy, jest policzenie dla każdego przedmiotu stosunku wartości do wielkości (v_i/w_i). Następnie wzięcie w pierwszej kolejności przedmiotów o większej wartości tego ilorazu. Taki zachłanny algorytm sprawdza się w przypadku ciągłego problemu plecakowego (TODO: dowód). Złożoność tego algorytmu to $\Theta(n \log n)$, gdyż potrzebujemy posortować tablicę ilorazów.

Nie jest jednak poprawnym algorytmem dla wersji dyskretnych. Kontrprzykładem będzie sytuacja, w której plecak ma wielkość $W = 10$ i do dyspozycji

mamy $M = 3$ przedmioty: $v_1 = 9$, $v_2 = v_3 = 5$, $w_1 = 6$, $w_2 = w_3 = 5$. W każdym z wariantów problemu postępując w sposób zachłanny, wybierzemy w pierwszym kroku przedmiot 1. Otrzymamy w ten sposób plecak o wartości 9, do którego nie jesteśmy w stanie już wstawić kolejnego przedmiotu. Wybierając przedmioty 2 i 3 otrzymując plecak o wartości 10. Zatem wybranie przedmiotu o największym stosunku wartości do wielkości było w takiej sytuacji błędem. Poprawnym rozwiązaniem wersji dyskretnych okazuje się podejście dynamiczne.

Zacznijmy od najprostszej wersji - nieograniczonej. Użyjemy tutaj tablicy `T[0..W]`. W komórce `T[w]` będziemy pamiętać optymalne rozwiązanie dla plecaka wielkości w. Rozwiązanie naszego problemu będzie znajdowało się w komórce `T[W]`. Wartości poszczególnych komórek liczymy według wzoru:

$$T[w] = \begin{cases} 0, & \text{jeśli } w = 0 \\ max_{w_i \leqslant w}\{v_i + T[w - w_i]\}, & \text{wpp} \end{cases}$$

Gdy plecak jest rozmiaru 0 nie możemy zapakować do niego żadnego przedmiotu. Jeśli mamy dostępne miejsce - patrzymy na wszystkie przedmioty, które mieszczą się do plecaka i obliczamy potencjalny zysk przy wzięciu każdego z nich. Z otrzymanych wartości bierzemy maksimum. Do policzenia mamy $\Theta(W)$ komórek. Do policzenia każdej z nich potrzebujemy $O(M)$ operacji w najgorszym przypadku. Złożoność całego algorytmu można zatem ograniczyć przez $O(M \cdot W)$.

Przejdźmy teraz do nieco trudniejszego przypadku - wersji $0/1$. W tym wariancie problemu, będziemy potrzebowali tablicy dwuwymiarowej `T[0..W][0..M]`. W komórce `T[w][m]` będziemy trzymać optymalne rozwiązanie w sytuacji w którym ograniczamy się do plecaka rozmiaru w oraz rozważamy tylko m pierwszych typów przedmiotów.

$$T[w][m] = \begin{cases} 0, & \text{jeśli } w = 0 \text{ lub } m = 0 \\ T[w][m-1], & \text{jeśli } w_m > w \\ max\{v_m + T[w - w_m][m-1], T[w][m-1]\}, & \text{wpp} \end{cases}$$

Pierwsza linijka wzoru to przypadek bazowy rekurencji - wtedy gdy plecak jest pusty lub nie mamy do wyboru żadnego przedmiotu. Teraz gdy wiemy, że nie jesteśmy w przypadku bazowym, bierzemy do ręki przedmiot typu m i kontemplujemy nad tym czy wziąć ten przedmiot czy nie. Linijka druga wzoru rozwiewa nasz problem w momencie w którym przedmiot ten i tak nie zmieściłby się nam do plecaka (nie bierzemy go w takiej sytuacji; duh...). Jeśli jednak by się zmieścił to liczymy potencjalną wartość plecaka wraz z tym przedmiotem oraz bez tego przedmiotu. Jako wynik bierzemy maksimum - o czym mówi linijka trzecia. Tym razem do policzeniania mamy $\Theta(W \cdot M)$ komórek. Na każdą komórkę potrzebujemy jednak tylko $\Theta(1)$ operacji. Zatem złożoność całego algorytmu wynosi $\Theta(W \cdot M)$.

Ostatnim przypadkiem jest wersja ograniczona. Tak samo jak w przypadku wersji 0/1 będziemy używać tablicy `T[0..W][0..M]`. Dokładnie tak samo jak w poprzednim przypadku w komórce `T[w][m]` będziemy trzymać optymalne rozwiązanie w sytuacji w którym ograniczamy się do plecaka rozmiaru w oraz rozważamy tylko m pierwszych typów przedmiotów. Również wzór rekurencyjny będzie wyglądał bardzo podobnie.

$$T[w][m] = \begin{cases} 0, & \text{jeśli } w = 0 \text{ lub } m = 0 \\ max_{0 \leqslant c \leqslant c_m, c \cdot w_m \leqslant w}\{c \cdot v_m + T[w - c \cdot w_m][m-1]\}, & \text{wpp} \end{cases}$$

Tu i we wzorze wyżej wyjechałem na margines :C

Różnica polega na tym, że gdy rozważamy przedmiot typu m to rozważamy każdą jego dostępną ilość, która mieści się w plecaku. Dla każdej ilości liczymy potencjalną wartość plecaka w którym wzięliśmy daną ilość przedmiotów tego typu. Z wszystkich wartości bierzemy maksimum jako wynik. Do policzenia zostaje nam $\Theta(W \cdot M)$ komórek. Każda komórka `T[..][m]` wymaga $O(c_m)$ operacji. Zakładając, że dla każdego typu przedmiotów zachodzi $c_m > 0$, cały algorytm działać będzie w czasie $O(W \cdot \sum_{m \leqslant M} c_m)$.

Rozdział 2

Under construction

2.1 Złożoność obliczeniowa

Todo, todo, todo...

2.2 Model obliczeń

Todo, todo, todo...

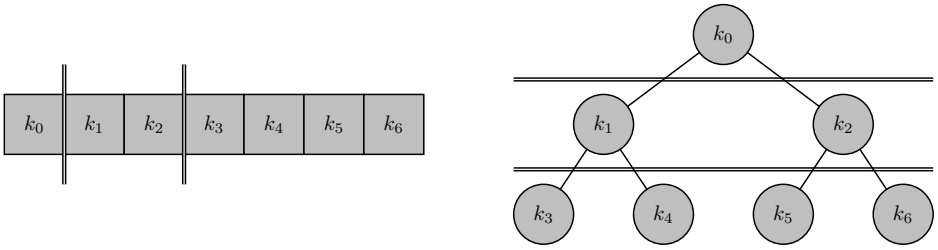
2.3 Kopce binarne

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.

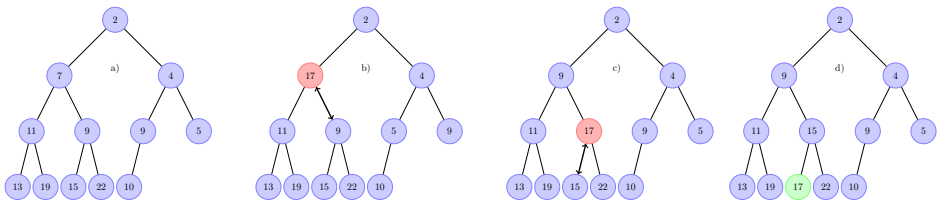
Warto zauważyć, że tak reprezentowane drzewo pozwala na łatwy dostęp do powiązanych węzłów. Synami węzła o indeksie i są węzły $2i + 1$ oraz $2i + 2$, natomiast jego ojcem jest $\lfloor \frac{i-1}{2} \rfloor$.

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 7: 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.

2.4 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	$17 \cdot 2^5 = 544$

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.

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

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

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 8: 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' mod $2 = 1$ **then**
    $wynik \leftarrow wynik + b'$
  end
  $a' \leftarrow a'$ 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 6. *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 7. *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.

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

  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 8. *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 9. *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 $O(\log a \cdot \log ab)$.

2.5 Sortowanie topologiczne

Rysunek 2.3: 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.

2.6 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 9: 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 10: 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.

2.6.1 Quick sort

In progress

2.7 Minimalne drzewa rozpinające

Todo, todo, todo...

2.7.1 Cut Property i Circle Property

2.7.2 Algorytm Prima

2.7.3 Algorytm Kruskala

2.7.4 Algorytm Borůvki

2.8 Algorytm Dijkstry

Todo, todo, todo...

2.9 Algorytm szeregowania

Todo, todo, todo...

2.10 Programowanie dynamiczne na drzewach

Todo, todo, todo...

2.11 Problemy NP

Na wstępie chcielibyśmy zaznaczyć że tematyka NP-zupełności nie będzie poruszana dogłębnie. Nie będziemy np. wprowadzać definicji maszyny Turinga(lub innego modelu) oraz przeprowadzać skomplikowanych dowodów, gdyż wymagałoby to wiedzy z zakresu teorii języków formalnych i złożoności obliczeniowej. Zamiast tego będziemy chcieli nabrać trochę intuicji co do tego czy w ogóle warto starać się rozwiązywać dany problem czy może jest to strata naszego czasu.

Definicja 5. *Problemem decyzyjnym nazywamy problem którego rozwiązanie przyjmuje jedną z dwóch wartości - TAK, NIE*

Zauważmy że problemy decyzyjne możemy utożsamiać z podzbiorami pewnego uniwersumi, w ten sposób że problem jest zbiorem wartości, dla których odpowiedź to TAK.
PRZYKŁAD. Problem polegający na rozstrzygnięciu czy dana liczba naturalna p jest liczbą pierwszą utożsamilibyśmy ze zbiorem liczb pierwszych.

Definicja 6. *Dla danej funkcji kosztu f, problemem optymalizacyjnym nazywamy problem którego rozwiązaniem jest wartość z danego uniwersum, minimalizująca wartość funkcji kosztu.*

PRZYKŁAD. Dla danego grafu G oraz wierzchołków u i v, wyznaczyć najkrótszą drogę między u i v. Naszą funkcją kosztu jest długość drogi,a uniwersum to wszystkie drogi łączące u i v.

Definicja 7 (Klasa NP)**.** *Klasą NP nazywamy zbiór problemów decyzyjnych L t. że istnieje algorytm wielomianowy A dla którego prawdziwe jest następujące zdanie:*

$$x \in L \iff \textit{istnieje } y \textit{ t. że } |y| < |x|^c \textit{ oraz } A \textit{ akceptuje } (x, y)$$

W powyższej definicji możemy myśleć o y jako o podpowiedzi dla algorytmu, lub nawet gotowym rozwiązaniu, natomiast A jest weryfikatorem, który używając podpowiedzi próbuje udzielić odpowiedzi, gdzie A akceptuje (x, y) oznacza że odpowiedź dla danych wejściowych x to TAK.

PRZYKŁAD. Pokażmy że problem decyzyjny, polegający na sprawdzeniu czy w grafie G istnieje cykl Hamiltona jest w NP.

Dowód. Najpierw musimy wymyślić weryfikator, tzn. wielomianowy algorytm sprawdzający istnienie cyklu Hamiltona w grafie. Nasz będzie dość prosty - dla danego grafu x, oraz y - pewnej drogi w x, zwyczajnie sprawdzimy czy y jest

cyklem hamiltona, a jeśli tak to zwrócimy TAK(to znaczy nasz algorytm będzie akceptować parę (x, y)). Łatwo zauważyć że czas działania algorymtu jest ograniczony przez $O(|V| + |E|)$.

Dowód $\Rightarrow$
Niech x będzie grafem zawierającym cykl Hamiltona. Wówczas naszym y bedzie właśnie ten cykl. Wtedy $|y|$ jest ograniczona przez $O(|V| + |E|)$ oraz zgodnie z opisem działania naszego algorytmu, A zaakceptuje (x, y).
Dowód w drugą stronę jest równie prosty, więc go pominiemy. □

Zauważmy że podpowiedź nie zawsze jest potrzebna. Np. w przypadku problemu polegającym na sprawdzeniu czy liczba jest podzielna przez 2, możemy wziąć jakąkolwiek podowiedź, zignorować ją a następnie udzielić odpowiedzi w czasie wielomianowym. O takich problemach mówimy że są klasy P.

Definicja 8 (Redukcje wielomianowe)**.** *Mówimy że problem L jest wielomianowo redukowalny do problemu Q jeśli:*

1. $\exists f \forall x \quad x \in L \Rightarrow f(x) \in Q$
2. *Istnieje wielomianowy algorytm obliczający funkcję f*

Mimo że na pierwszy rzut oka może się to wydać niezrozumiałe, sens intuicyjny jest bardzo prosty. Chcielibyśmy "przetłumaczyć" problem L na problem Q, to znaczy użyć rozwiązania problemu Q tak abyśmy mogli za jego pomocą rozwiązać problem L. Jeśli uda nam się znaleźć taką funkcję(o której myślimy jak o algorytmie) to znaczy że znaleźliśmy redukcję problemu L do problemu Q. Aby redukcja była wielomianowa, musi być spełniony drugi warunek, tzn. musimy umieć obliczyć tę funkcję w czasie wielomianowym.

Definicja 9 (Problem NP-zupełny)**.** *Problem L jest problemem NP-zupełnym jeśli:*

1. $L \in NP$
2. *Każdy problem z klasy NP jest wielomianowo redukowalny do L*

O ile udowodnienie pierwszego warunku nie wydaje się specjalnie trudne(już raz to zrobiliśmy), tak drugi warunek może już sprawiać kłopoty. Zazwyczaj jednak nie dowodzi się tego bezpośrednio. Zamiast tego korzysta się z następującego faktu:

Lemat 10. *Jeśli L, jest problemem NP, Q jest problemem NP-zupełnym oraz L jest redukowalny wielomianowo do Q to L jest problemem NP-zupełnym*

Dowód. Weźmy dowolny problem $A \in NP$, zredukujmy go do Q a następnie do L. Czas obliczeń jest wówczas ograniczony przez sumę wielomianów, a więc przez wielomian. □

Jednak aby skorzystać z tego lematu trzeba najpierw znaleźć problem który jest NP-zupełny. Jednym z pierwszych takich problemów którego NP-zupełność udowodniono był problem SAT(spełnialność formuł logicznych), a jak się później okazało, wiele innych problemów możana do niego zredukować. My zostawimy to bez dowodu.

To czy $P = NP$ jest wciąż otwartym, problemem milenijnym, którego rozwiązanie jest warte 1 000 000$. Mimo że nikomu jeszcze nie udało się tego udowodnić, cały(duża większość) świat informatyki/matematyki wierzy że $P \neq NP$. Wiara jest na tyle mocna że wydawane są prace naukowe oraz dowodzone są twierdzenia przy założeniu że $P \neq NP$.

O problemach NP-zupełnych można myśleć jak o takich problemach które są co najmniej tak samo trudne jak wszystkie inne problemy z klasy NP. Ogólnie rzecz biorąc, są to problemy bardzo trudne obliczeniowo, dla których nie istnieje żaden algorytm wielomianowy rozwiązujący je, co więcej prawdopodobnie nigdy nie będzie istniał, no chyba że $P = NP$. Morał z tego jest taki, że jeśli wiemy że dany problem jest NP-zupełny, to raczej nie warto tracić czasu na rozwiązywanie go.

Jako dodatek, poniżej prezentujemy liste najbardziej znanych problemów NP-zupełnych(lub NP-trudnych):

1. Problem SAT
2. Problem cyklu Hamiltona
3. Problem trójkolorowalności grafu
4. Problem komiwojażera(NP-trudny)
5. Problem kliki
6. Problem plecakowy(NP-trudny)
7. Problem sumy podzbioru
8. Problem minimalnego pokrycia zbioru(NP-trudny)

2.12 Sieci przełączników Benesa-Waksmana

W tym rozdziale zajmiemy się sieciami przełączników Benesa-Waksmana. Moją one zastosowanie w sieciach komputerowych.

2.12.1 Budowa

Sieć składa się z przełączeników. Każdy z przełączeników ma dwa możliwe stany.

- W stanie 1 przełącznik przesyła dane z wejścia i na wyjście i ($i \in \{0, 1\}$)
- W stanie 2 przełącznik przesyła dane z wejścia i na wyjście $i + 1 \ mod \ 2$ ($i \in \{0, 1\}$)

<Tutaj wstawić obrazki ze stanami i jakąś przekładową sieć(wydaje mi się, że to lepiej objaśni, niż mój najlepszy opis)>

2.12.2 Kontrukcja sieci tworzącej wszystkie możliwe permutacje zbioru

Dla ułatwienia będziemy się zajmować zbiorami w postaci 2^n ($n \in \mathbb{N}$).

Konstrukcja będzie oparta na zasadzie dziel i zwyciężaj i będzie sprowadzała problem do rekurencyjnego zbudowania sieci wielkości 2^{n-1}, a następnie odpowiedniego połączenia portów.

$n = 1$

Dla zbioru wielkości 2 jeden przełącznik generuje każdą możliwą permutacje; stan 1 generuje identycznośc; stan 2 drugą permutację.

$n > 1$

<Tutaj znowu wstawiłbym rysunek idei algorytmu>

2.12.3 Własności wygenerowanej sieci

Głębokość sieci wyraża się równaniem

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

z tego wynika, że $G(n) = 2 \log n - 1$

Ilość przełączeników w sieci wyraża się równaniem

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

Wykorzystując punkt 2. (a) z twierdzenia o rekurencji uniwersalnej możemy stwierdzić, że $P(n) = \Theta(n \log n)$.

2.12.4 Dowód poprawności konstrukcji

TODO

2.12.5 Sortowanie

TODO

2.13 Pokrycie zbioru

Problem pokrycia zbioru jest problemem optymalizacyjnym związany z problemem alokacji zasobów. Przedstawimy zachłanny algorytm o logarytmicznym współczynniku aproksymacji rozwiązujący ten problem.

Dane dla problemu pokrycia zbioru to para $(U, \mathcal{S})$ oraz funkcja kosztu c. U, zwane uniwersum, jest skończonym zbiorem elementów, a $\mathcal{S}$ jest rodziną podzbiorów U, taką że:

$$\bigcup_{i=1}^{n} S_i = U$$

Mówimy, że podzbiór $S_i \in \mathcal{S}$ pokrywa elementy należące do S_i. Z kolei $c : S_i \to \mathbb{R}$, każdemu podzbiorowi S_i określa cenę pokrycia swoich elementów.

Problem polega na znalezieniu podrodziny $\mathcal{T} \subseteq \mathcal{S}$, której elementy pokrywają cały zbiór U:

$$U = \bigcup_{T \in \mathcal{T}} T$$

Spośród wszystkich takich rozwiązań interesuje nas to, którego koszt $c(\mathcal{T})$ jest minimalny:

$$min(c(\mathcal{T})) = \sum\nolimits_{T \in \mathcal{T}} c(T)$$

Mając zdefiniowaną cenę podzbior możemy zdefiniować cenę rynkową elementu, która będzie naszym kryterium wyboru podzbiorów. Niech $e_1, e_2, ..., e_n$ będą elementami U w porządku pokrycia przez algorytm. Przez cenę rynkową f_i elementu będziemy rozumieć średni koszt nowo pokrywanego elementu e_i przez rozpartywany podzbiór S_i:

$$f_i = \frac{c(S_{j_i})}{\left| S_{j_i} \setminus \bigcup_{j_i < k} S_k \right|},$$

gdzie S_i jest to $i-ty$ zbiór wybierany przez algorytm, S_{j_i} jest pierwszym zbiore m pokrywającym e_i, czyli
$j_i = min\left\{ 1 \leqslant k < n : e_i \in S_k \setminus \bigcup_{l=1}^{k-1} S_l \right\}$.

Mówiąc cena rynkowa zbioru mamy na myśli cenę rynkową elementów tego zbioru.

Algorytm 11: Zachłanny algorytm dla problemu pokrycia zbioru

Input: U - uniwersum, $\mathcal{S}$ - rodzina podzbiorów U
Output: $\mathcal{T}$, takie że $c(\mathcal{T})$ jest minimalne
$\mathcal{T} \leftarrow \emptyset$
while $\mathcal{T} \neq U$ **do**
  Oblicz cenę rynkową dla wszystkich zbiorów
  Wybierz zbiór A o najniższej cenie rynkowej
  $\mathcal{T} \leftarrow \mathcal{T} \cup A$
end

Lemat 11. *$f_i \leqslant \frac{c(OPT)}{n-i+1}$, gdzie OPT jest rozwiązaniem optymalnym.*

Dowód. Gdyby do pokrycia elementu e_i oraz wszystkich pozostałych elementów, czyli $e_{i+1}, e_{i+2}, ..., e_n$ użyłby rodziny zbiorów OPT, to cena rynkowa dla każdego z tych elementów wyniosłaby $\frac{c(OPT)}{libcza\ nowo\ pokrytych\ elementw}$, czyli $\frac{c(OPT)}{n-i+1}$.
W szczególności istnieje zbiór $Y \in OPT$ taki, że cena rynkowa pokrywanego elementu jest nie większa niż dla całego OPT. Zatem algorytm zachłanny wybiera do pokrycia e_i zbiór o cenie rynkowej pokrywanych elementów $\leqslant \frac{c(OPT)}{n-i+1}$. □

Koszt algorytmu zachłannego ALG

$$\begin{aligned} c(ALG) &= \sum_{i=1}^{n} f_i \\ &\leqslant \sum_{i=1}^{n} \frac{c(OPT)}{n-i+1} \\ &= c(OPT) \cdot \sum_{i=1}^{n} \frac{1}{n-i+1} \\ &= c(OPT) \cdot \sum_{i=1}^{n} \frac{1}{i} \\ &= c(OPT) \cdot H_n \\ &\leqslant c(OPT) \cdot \log(n+1) \end{aligned}$$

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	✓				