

Skrypt z Algorytmów i struktur danych

*Zbiór mniej lub bardziej ciekawych algorytmów
i struktur danych, jakie bywały omawiane na wy-
kładzie (albo i nie).*

PRACA ZBIOROWA POD REDAKCJĄ
KRZYSZTOFA PIECUCHA

Korzystać na własną odpowiedzialność.

Spis treści

1	Zrobione	7
1.1	Twierdzenie o rekurencji uniwersalnej	8
1.1.1	Przykłady wykorzystania twierdzenia	9
1.2	Sortowanie bitoniczne	10
1.3	Algorytm macierzowy wyznaczania liczb Fibonacciego	16
1.4	Algorytm Strassena	18
1.5	Model afinicznych drzew decyzyjnych	21
1.6	Problem plecakowy	25
1.7	Lazy Select	28
2	Under construction	33
2.1	Drzewce	34
2.2	Kopce binarne	39
2.3	Algorytm rosyjskich wieśniaków	41
2.4	Sortowanie topologiczne	44
2.5	Algorytmy sortowania	45
2.5.1	Quick sort	47
2.6	Minimalne drzewa rozpinające	48
2.6.1	Cut Property i Circle Property	48
2.6.2	Cycle property	48
2.6.3	Cut property	48
2.6.4	Algorytm Prima	49
2.6.5	Algorytm Kruskala	49
2.6.6	Algorytm Borůvky	49
2.7	Algorytm Dijkstry	50
2.7.1	Działanie	50
2.8	Dowód poprawności algorytmu	50
2.9	Analiza	51
2.10	Problemy	51

2.11	Problemy NP	52
2.12	Sieci przełączników Benesa-Waksmana	55
2.12.1	Budowa	55
2.12.2	Konstrukcja sieci tworzącej wszystkie możliwe permutacje zbioru	55
2.12.3	Własności wygenerowanej sieci	55
2.12.4	Dowód poprawności konstrukcji	56
2.12.5	Sortowanie	56
2.13	Pokrycie zbioru	57
2.14	Przynależność słowa do języka	59
2.15	Pokrycie wierzchołkowe	63
2.16	Algorytm znajdowania dwóch najbliższych punktów	64
2.16.1	Podejście siłowe	64
2.16.2	Podejście Dziel i Zwyciężaj	64
2.17	Kopce dwumianowe w wersji leniwej	66
2.17.1	Różnice w implementacji	66
2.17.2	Analiza złożoności	66
2.18	Counting sort	68
3	W ogóle nie zaczęte	71
3.1	Złożoność obliczeniowa	72
3.2	Model obliczeń	73
3.3	Programowanie dynamiczne na drzewach	74
3.4	Algorytm szeregowania	75
3.5	Drzewa Splay	76
3.6	Zbiory rozłączne	77
3.7	Drzewa przedziałowe	78
3.8	Tablice hashujące	79
3.9	Algorytm Shift-And	80
3.10	Wyszukiwanie wzorca z wykorzystaniem automatów skończonych	81
3.11	Algorytm Knutha-Morrisa-Pratta	82
3.12	Algorytm Karpa-Rabina	83
3.13	Drzewa czerwono-czarne	84
3.14	B-drzewa	85
3.15	Słownik statyczny	86
3.16	Geometria obliczeniowa	87
3.17	Kopce Fibonacciego	88
3.18	Szybka Transformata Fouriera	89
3.19	Drzewo van Emde Boasa	90
3.20	Sortowanie kubełkowe	91
3.21	Model drzew decyzyjnych	92

3.22	Dolna granica znajdowania min-maksa	93
3.23	Dolne granice	94
3.24	Optymalna kolejność mnożenia macierzy	95
3.25	Drzewa rozpinające drabiny	96
3.26	Cykl Hamiltona	97
3.27	Problem spełnialności formuł logicznych	98
3.28	Trójwymiarowe skojarzenia	99
3.29	Stokrotki	100
3.30	Otoczka wypukła	101
3.31	Najdłuższy wspólny podciąg	102
3.32	Algorytm Karatsuby	103
3.33	Algorytm Borůvky	104
3.33.1	Działanie	104
3.33.2	Pseudokod	104
3.33.3	Dowód poprawności	104
3.33.4	Złożoność czasowa	105
3.34	Algorytm Prima	106
3.35	Algorytm Kruskala	107
3.36	Statystyki pozycyjne	108
3.37	Algorytm magicznych piątek	109
3.38	Drzewa AVL	110
3.39	Izomorfizm drzew	111
3.40	Sortowanie ciągów różnej długości	112

Dodatek A Porównanie programów przedmiotu AiSD na różnych uczelniach	113
---	------------

Dodatek B Zadania na programowanie dynamiczne i algorytmy zachłanne	115
--	------------

Rozdział 1

Zrobione

1.1 Twierdzenie o rekurencji uniwersalnej

MARCIN BARTKOWIAK, KRZYSZTOF PIECUCH

Popularną metodą rozwiązywania zadań jest strategia 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  $n \leq 1$  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 chcielibyśmy uzyskać wzór zwarty. Do tego celu służy poniższe twierdzenie, znane jako Twierdzenie o rekurencji uniwersalnej.

Twierdzenie 1. *Niech $T(n) = aT(n/b) + \Theta(n^k \log^p n)$ oraz $a \geq 1$, $b > 1$, $k \geq 0$ 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) *$p \geq 0$ 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 2.5 zapoznaliśmy 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 powinien być osobny akapit, tabelka czy może coś jeszcze innego.

W rozdziale 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 przypadku $a = 2$, $b = 2$, $k = 1$ i $p = 1$. Ponieważ $a = b^k$ i $p > -1$ korzystamy z punktu 2a. Otrzymujemy złożoność $\Theta(n \log^2 n)$.

W rozdziale 3.32 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 rozdziale 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

KRZYSZTOF PIECUCH

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 równoległe 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 elementu 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  $i \leftarrow 0$  to  $n - 1$  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 tę 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 tę 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 tablic $A[0..n/2-1]$ oraz $A[n/2..n-1]$ będą tworzyły ciągi bitoniczne. Ponadto jeśli wartość zmiennej `up` jest ustawiona na `true` to każdy element tablicy $A[0..n/2-1]$ będzie niewiekszy 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 nie większy 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 $f: \mathbb{R} \rightarrow \mathbb{R}$ taką, że $\forall_{a,b \in \mathbb{R}} a \leq b \Rightarrow f(a) \leq f(b)$. 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 karząc 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.

Założmy, 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 istnieje w tablicy element $S[i]$ większy od elementu $S[i+1]$. Rozważmy następującą funkcję:

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

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</i>	<i>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 $i \geq n/2$. Rozważmy funkcję:

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

Wtedy twierdzenie nie byłoby prawdziwe dla tablicy $f(T)$ (zero-jedynkowej). Po-
nownie - 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)
		<i>j</i>			<i>i</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 rodzaj 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 nie większy od każdego elementu z ciągu drugiego.
- 0^k1^l oraz $k \leq n/2$. 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 nie większy od każdego elementu z ciągu drugiego.
- 0^k1^l oraz $k \geq n/2$. Otrzymamy ciągi $0^{n/2}$ oraz $0^{k-n/2}1^l$. Znowu - oba są bitoniczne i każdy element z pierwszego jest nie większy od każdego z drugiego.
- $0^k1^l0^m$ oraz $k \geq n/2$. Wtedy otrzymujemy ciągi $0^{n/2}$ oraz $0^{k-n/2}1^l0^m$. Spełniają one tezę twierdzenia.

- $0^k 1^l 0^m$ oraz $m \geq n/2$. Ciągi, które otrzymamy wyglądają tak: $0^{n/2}$ oraz $0^k 1^l 0^{m-n/2}$. Są to ciągi, które nas cieszą.
- $0^k 1^l 0^m$ oraz $l \geq n/2$. Dostaniemy wtedy ciągi $0^k 1^{l-n/2} 0^m$ oraz $1^{n/2}$. Są to ciągi, które spełniają naszą tezę.
- $0^k 1^l 0^m$ oraz $k, l, m \leq n/2$. 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

RAFAŁ FLORCZAK

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 $\Theta(n)$. Zaczniemy od zdefiniowania ciągu Fibonacciego:

$$F_n = \begin{cases} n, & \text{jeśli } n \leq 1 \\ F_{n-1} + F_{n-2}, & \text{wpp.} \end{cases}$$

Teraz, znajdziemy 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:

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

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

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

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 .

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n+1} \times \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n \times \begin{bmatrix} 1 \\ 0 \end{bmatrix} \stackrel{z.I.}{=} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} F_{n+1} \\ F_n \end{bmatrix} \stackrel{1.1}{=} \begin{bmatrix} F_{n+2} \\ F_{n+1} \end{bmatrix}$$

□

Algorytm 5: Procedura `get_fibonacci`

Input: n

Output: n -ta liczba Fibonacciego

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

$$M' \leftarrow \text{exp_by_squaring}(M, n - 1)$$

$$M'' \leftarrow M' \times \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

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:

$$G_{n+1} = a_0 G_n + a_1 G_{n-1} + \dots + a_k G_{n-k}$$

wygląda ona następująco:

$$M = \begin{bmatrix} a_0 & a_1 & a_2 & \dots & a_{k-1} & a_k \\ 1 & 0 & 0 & \dots & 0 & 0 \\ 0 & 1 & 0 & \dots & 0 & 0 \\ 0 & 0 & 1 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 1 & 0 \end{bmatrix}$$

Dowód tej konstrukcji pozostawiamy Czytelnikowi jako ćwiczenie.

Symbol mnożenia macierzy jest inny niż w rozdziale o liczbach fibonacciego.

1.4 Algorytm Strassena

KRZYSZTOF PIECUCH

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

Output: $C = A \cdot B$

```

for  $i \leftarrow 1$  to  $n$  do
    for  $j \leftarrow 1$  to  $p$  do
         $C[i][j] \leftarrow 0$ 
        for  $r \leftarrow 1$  to  $m$  do
             $C[i][j] \leftarrow C[i][j] + A[i][r] \cdot B[r][j]$ 
        end
    end
end

```

Powyższy algorytm działa w czasie $\Theta(n \cdot p \cdot m)$ ($\Theta(n^3)$ dla macierzy kwadratowych o boku n). Korzystając ze sprytniej 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 podzielimy 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 osiąga lepszą złożoność asymptotyczną przez pozbycie się jednego z mnożeń. Algorytm ten liczy następujące macierze:

$$M_1 = (A_{1,1} + A_{2,2}) \cdot (B_{1,1} + B_{2,2})$$

$$M_2 = (A_{2,1} + A_{2,2}) \cdot B_{1,1}$$

$$M_3 = A_{1,1} \cdot (B_{1,2} - B_{2,2})$$

$$M_4 = A_{2,2} \cdot (B_{2,1} - B_{1,1})$$

$$M_5 = (A_{1,1} + A_{1,2}) \cdot B_{2,2}$$

$$M_6 = (A_{2,1} - A_{1,1}) \cdot (B_{1,1} + B_{1,2})$$

$$M_7 = (A_{1,2} - A_{2,2}) \cdot (B_{2,1} + B_{2,2})$$

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

$$C_{1,1} = M_1 + M_4 - M_5 + M_7$$

$$C_{1,2} = M_3 + M_5$$

$$C_{2,1} = M_2 + M_4$$

$$C_{2,2} = M_1 - M_2 + M_3 + M_6$$

Wykonując proste przekształcenia arytmetyczne, możemy dowieść poprawności powyższych równań.

Używając powyższych wzorów, możemy skonstruować algorytm rekurencyjny. Będzie on dzielił 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

KRZYSZTOF PIECUCH

Zdefiniujmy następujący problem (ang. element uniqueness). 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: $c, a_0, a_1, \dots, a_{n-1}$, a następnie zapytać czy

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

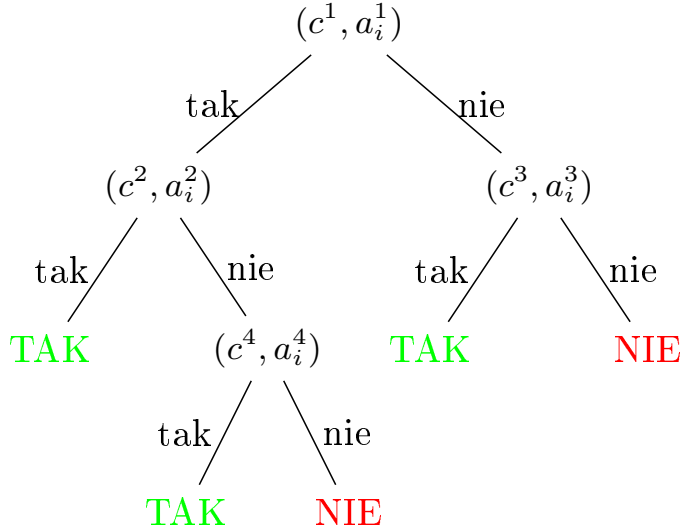
gdzie t_i to elementy tablicy T . Gdybyśmy użyli terminologii algebraicznej, to powiedzielibyśmy, że t jest punktem w przestrzeni \mathbb{R}^n , lewa strona powyższej nierówności to przekształcenie afiniczne, a zbiór wszystkich punktów z \mathbb{R}^n , które spełniają tę nierówność to półprzestrzeń afiniczna. Jeśli na Algebrze nie wyrobiłście 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 $0 \leq \alpha \leq 1$ punkt $\alpha \cdot u + (1 - \alpha)v$ także należy do C .



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

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:

Lemat 4. *Zbiór punktów osiagających liść l w afijnym drzewie decyzyjnym, jest zbiorem wypukłym.*

Dowód. Weźmy dowolne aficzne 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 osiagają liść l . W końcu weźmy dowolną liczbę rzeczywistą $0 \leq \alpha \leq 1$. 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

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

oraz, że

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

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

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

a ponieważ $1 - \alpha \geq 0$ możemy drugie równanie przemnożyć przez $1 - \alpha$:

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

Teraz sumując oba równania otrzymujemy:

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

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. \square

Wyobraźmy sobie, że na płaszczyznę \mathbb{R}^2 kładziemy półpłaszczyzny. Wtedy przecięcie dowolnej liczby półpłaszczyzn jest zbiorem wypukłym. Podobnie jeśli w przestrzeni \mathbb{R}^3 wyznaczmy 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 uniqueness to musi posiadać conajmniej $n!$ liści. Oznacza to, że wysokość takiego drzewa musi wynosić conajmniej $O(n \log n)$.

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

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 uniqueness. Oznacza to, że liczba liści w takim drzewie musi wynosić przynajmniej $n!$. Zatem wysokość takiego drzewa musi wynosić co najmniej $\Omega(n \log n)$.

1.6 Problem plecakowy

MARCIN BARTKOWIAK, KRZYSZTOF PIECUCH

Wyobraźmy sobie, że jesteśmy złodziejem. Pod przykryciem nocy, udało nam się dotrzeć w ustalone miejsce. Wszystko idzie wyjątkowo gładko. Wpisujemy kod, który otrzymaliśmy od sprzątaczk i jesteśmy już w środku. Wtem okazuje się, że zapomnieliśmy listy przedmiotów, które mieliśmy ukraść. Próbuje 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 $\sum_{i=1}^M x_i \cdot w_i \leq W$. 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: $x_i \in \mathbb{N}$, $x_i \leq c_i$. Problem ten nazywamy czasem ograniczonym, dyskretnym problemem plecakowym.
- Cyberprzestrzeń. W tym przypadku możemy ukraść dowolną ilość przedmiotu danego typu (np. klucze 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 ($0 \leq x_i \leq c_i$). 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 otrzymujemy 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:

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

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 policzania 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.

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

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 $O(W \cdot \sum_{m \leq M} c_m)$.

Tu i we wzorze
wyżej wyjechałem
na margines :C

1.7 Lazy Select

KRZYSZTOF PIECUCH

Naszym celem w tym rozdziale będzie znalezienie mediany w nieuporządkowanej tablicy w czasie liniowym. Przedstawimy algorytm zrandomizowany, który albo (z dużym prawdopodobieństwem) zwróci poprawną medianę, albo stwierdzi, że poszukiwania zakończyły się fiaskiem. Proszę mieć na uwadze fakt, że algorytm może posłużyć do znalezienia k -tego elementu, dla dowolnej wartości k .

Mediany będziemy szukać w zbiorze A i tradycyjnie założymy, że ma ona n elementów. Ponadto oznaczymy sobie szukaną medianę jako m . Na początku założymy, że mamy pewne dodatkowe informacje zesłane nam przez Boga, wyrocznię, wróżkę albo super-doktoranta (kto co woli). Super-doktorant pokazuje nam dwa elementy ze zbioru A : l oraz u . Ponadto wyrocznia gwarantuje nam, że te dwa elementy mają następujące własności:

- $l \leq m \leq u$
- $C = |\{a \in A : l \leq a \leq u\}| \leq 4 \cdot n^{3/4}$

Czyli mediana zbioru A znajduje się między podanymi elementami, oraz ilość elementów pomiędzy l a u jest nie większa od $4 \cdot n^{3/4}$ (elementy te oznaczamy jako zbiór C). Czy z taką boską pomocą student poradzi sobie ze znalezieniem mediany w zbiorze? Niewykluczone. Wystarczy bowiem tylko posortować teraz elementy leżące pomiędzy l a u . Jeśli wybierzemy swoją ulubioną efektywną metodę sortowania, to zajmie nam to $\Theta(n^{3/4} \log n^{3/4})$ czasu. Czyli $o(n)$ (małe $o(n)$ oznacza czas ostro mniejszy od liniowego - algorytm działa szybciej niż liniowo). Następnie z posortowanego zbioru wybierzemy odpowiedni element. W tym celu policzmy d_l - liczbę elementów mniejszych od l w zbiorze A oraz d_u - liczbę elementów większych od u w zbiorze A . Mediana zbioru A to $n/2 - d_l + 1$ -szy element posortowanego zbioru C .

W swoim (słynnym) skrypcie do programowania funkcyjnego Marcin Kubica napisał, że informatyka to dziedzina magii. W "Nowej encyklopedii powszechnej PWN" możemy znaleźć następujące określenie magii: "zespół działań zasadniczo pozaempirycznych, symbolicznych, których celem jest bezpośrednie osiągnięcie (...) pożądanych skutków (...)". Wyróżniamy przy tym następujące składniki działań magicznych:

- zrytualizowane działania (manipulacje)
- materialne obiekty magiczne (amulety, eliksiry itp.)
- reguły obowiązujące przy praktykach magicznych (zasady czystości, reguły określające czas i miejsce rytuałów)

- formuły tekstowe mające moc sprawczą (zaklęcia)

Programowanie mieści się w ramach ostatniego z powyższych punktów. Programy komputerowe są zapisanymi w specjalnych językach zaklęciami. Zaklęcia te są przeznaczone dla specjalnego rodzaju duchów żyjących w komputerach, zwanymi procesami obliczeniowymi. Ze względu na to, że komputery są obecnie produkowane seryjnie, stwierdzenie to może budzić kontrowersyjne. Zastanówmy się jednak chwilę, czym charakteryzują się duchy. Są to obiekty niematerialne, zdolne do działania. Procesy obliczeniowe ewidentnie spełniają te warunki: nie można ich dotknąć, ani zobaczyć, nic nie ważą, a można obserwować skutki ich działania, czy wręcz uzyskać od nich interesujące nas informacje. Nota bene, w trakcie zajęć można się spotkać również z przejawami pozostałych wymienionych składników działań magicznych. “Logowanie” się do sieci oraz wyłączanie komputera to działania o wyraźnie rytualnym charakterze. Przedmioty takie jak indeks, czy karta zaliczeniowa wydają się mieć isticie magiczną moc.

Czemu wam o tym pisze? Bo teraz zacznie się magia :) Jeśli bowiem nikt nam nie poda wartości l oraz u to będziemy musieli zabawić się w wróżkę i sami te wartości wyczarować. Zaczniemy od utworzenia zbioru R do którego zaczniemy wrzucać losowo z powtórzeniami elementy zbioru A z jednakowym prawdopodobieństwem. Będziemy robić to tak długo, aż zbiór R będzie miał $n^{3/4}$ elementów. Następnie zbiór ten posortujemy w czasie $o(n)$. Jako wartość l weźmiemy $n^{3/4}/2 - \sqrt{n}$ -szy element zbioru R , a jako wartość r weźmiemy $n^{3/4}/2 + \sqrt{n}$ -szy element zbioru R .

Co mogło pójść źle? Nasze wartości l oraz u miały spełniać dwie własności. Mediana miała znajdować się pomiędzy tymi wartościami oraz liczba elementów pomiędzy tymi wartościami miała być mniejsza od $4 \cdot n^{3/4}$. Pierwszy warunek nie będzie spełniony jeśli którakolwiek z liczb d_l albo d_u jest większa od $n/2$. Policzmy prawdopodobieństwo tego, że $d_l > n/2$. Drugie prawdopodobieństwo liczy się symetrycznie.

Przez Y oznaczmy sobie zbiór elementów R mniejszy bądź równy medianie. Formalnie: $Y = \{x \in R : x \leq m\}$. Skoro $l_d > n/2$ to $|Y| < n^{3/4}/2 - \sqrt{n}$. Niech X_i będzie zmienną losową oznaczającą, że i -ty element wybrany do zbioru R był mniejszy bądź równy medianie. Wtedy $P(X_i = 1) = ((n+1)/2)/n = 1/2 + 1/2 \cdot n^{-1/2}$. Oczywiście $|Y| = \sum X_i$. Stąd łatwo możemy policzyć wartość oczekiwaną oraz wariancję $|Y|$.

$$E[|Y|] = \frac{1}{2} \cdot n^{\frac{3}{4}} + \frac{1}{2} n^{-\frac{1}{4}}$$

$$Var[|Y|] = n^{\frac{3}{4}} \cdot Var[X_i] < \frac{1}{4} \cdot n^{\frac{3}{4}}$$

Teraz korzystając z nierówności Czybyszewa otrzymujemy interesującą nas war-

Algorytm 7: Procedura lazy_select

Input: A, n

Output: m - mediana tablicy A

for $i \leftarrow 0$ *to* $n^{3/4}$ **do**

$R[i] \leftarrow A[\text{random}(n)]$

end

sort(R, $n^{3/4}$)

$l \leftarrow R[n^{3/4}/2 - \sqrt{n}]$

$u \leftarrow R[n^{3/4}/2 + \sqrt{n}]$

$s \leftarrow d_l \leftarrow d_r \leftarrow 0$

for $i \leftarrow 0$ *to* n **do**

if $A[i] < l$ **then** $d_l \leftarrow d_l + 1$

else if $A[i] > u$ **then** $d_u \leftarrow d_u + 1$

else

$C[s] = A[i]$

$s \leftarrow s + 1$

end

end

if $d_l > n/2$ *or* $d_u > n/2$ *or* $s > 4 * n^{3/4}$ **then return fail**

sort(C, s)

return $C[n/2 - d_l + 1]$

tość:

$$\begin{aligned}
P(fail_1) &= P\left(|Y| < \left(\frac{1}{2} \cdot n^{\frac{3}{4}} - \sqrt{n}\right)\right) \\
&= P\left(\left(\frac{1}{2} \cdot n^{\frac{3}{4}} - |Y|\right) > \sqrt{n}\right) \\
&\leq P\left(\left(\frac{1}{2} \cdot n^{\frac{3}{4}} + \frac{1}{2} \cdot n^{-\frac{1}{4}} - |Y|\right) > \sqrt{n}\right) \\
&= P\left((E[|Y|] - |Y|) > \sqrt{n}\right) \\
&\leq P\left(|E[|Y|] - |Y|| > \sqrt{n}\right) \\
&\leq Var[|Y|]/(\sqrt{n})^2 \\
&\leq \frac{1}{4} \cdot n^{\frac{3}{4}}/n \\
&= \frac{1}{4} \cdot n^{-\frac{1}{4}}
\end{aligned}$$

Teraz rozważymy drugą sytuację, która mogła pójść źle - zbiór C okazał się za duży. Oznacza to, że albo conajmniej $2 \cdot n^{3/4}$ elementów C jest większych od mediany albo, że conajmniej $2 \cdot n^{3/4}$ elementów C jest mniejszych od mediany. Mamy zatem tak samo jak w poprzednim akapicie dwie symetryczne sytuacje. Rozważymy pierwszą z nich.

Dowód będzie analogiczny do dowodu poprzedniego. Przez Y oznaczmy sobie zbiór elementów R większych od $n/2 + 2 \cdot n^{3/4}$ -szego elementu zbioru A (w zbiorze posortowanym). Skoro conajmniej $2 \cdot n^{3/4}$ elementów C jest większych od mediany to znaczy, że $|Y| \geq n^{3/4} - (n^{3/4}/2 + \sqrt{n}) = n^{3/4}/2 - \sqrt{n}$. Niech X_i będzie zmienną losową oznaczającą, że i -ty element wybrany do zbioru R jest większy od $n/2 + 2 \cdot n^{3/4}$ -szego elementu zbioru A . Wtedy $P(X_i = 1) = (n/2 - 2 \cdot n^{3/4})/n = 1/2 - 2 \cdot n^{-1/4}$. I tak jak ostatnio $|Y| = \sum X_i$. Teraz wartość oczekiwana oraz wariancja:

$$\begin{aligned}
E[|Y|] &= \frac{1}{2} \cdot n^{\frac{3}{4}} - 2 \cdot \sqrt{n} \\
Var[|Y|] &= n^{\frac{3}{4}} Var[X_i] < \frac{1}{4} \cdot n^{\frac{3}{4}}
\end{aligned}$$

Wykonując podobne obliczenia jak ostatnio otrzymujemy wartość:

$$P(fail_2) \leq \frac{1}{4} \cdot n^{-\frac{1}{4}}$$

Ostatecznie

$$P(fail) \leq 2 \cdot P(fail_1) + 2 \cdot P(fail_2) \leq n^{-\frac{1}{4}}$$

Otrzymaliśmy algorytm, który działa w czasie $\Theta(n)$ i zwraca poprawną medianę lub z prawdopodobieństwem mniejszym od $n^{-1/4}$ zwraca, że się pomylił.

Rozdział 2

Under construction

2.1 Drzewce

BARTŁOMIEJ BETKA

Podobnie jak drzewa AVL oraz drzewa czerwono-czarne drzewce rozwiązują problem potencjalnego niezrównoważenia "zwykłego" BST. Osiągają one jednak ten cel w zupełnie inny sposób. Zamiast kontrolować wysokość drzewa i jego poddrzew drzewce opierają się na randomizacji, która, jak zaraz pokażemy, zapewnia asymptotycznie równie dobre oczekiwane własności.

Definicja 5. *Drzewiec to drzewo binarne, którego każdy węzeł posiada zarówno klucz jak i priorytet. Względem kluczy drzewiec jest drzewem przeszukiwań binarnych a względem priorytetów jest on kopcem.*

Podstawowe operacje, które implementują drzewce:

- **Find** Wygląda dokładnie tak, jak w drzewie BST (względem kluczy).
- **Insert** Najpierw wstawiamy wartość (wraz z losowym kluczem) jak do drzewa BST, następnie rotacjami/zamianą z ojcem przywracamy porządek kopcowy.
- **Delete** Znajdujemy klucz, rotacjami spychamy go do liścia, następnie usuwamy ten liść (co nie zaburza ani porządku BST ani kopca).
- Pomocniczo potrzebne są również operacje rotacji służące do przywracania porządku kopcowego.

Na końcu rozdziału znajduje się pseudo-kod, który pokazuje jak można zaimplementować powyższe metody i pozwala przekonać się, że jest to znacznie łatwiejsze niż w przypadku analogicznych operacji na drzewach AVL czy drzewach czerwono-czarnych.

Teraz udowodnimy kilka własności drzewców.¹

Twierdzenie 3. *Dla każdego zbioru par (klucz, priorytet) istnieje dokładnie jeden drzewiec.*

Dowód. Oczywiście para, w której znajduje się najwyższy priorytet musi znajdować się w korzeniu drzewa. W jego lewym poddrzewie znajdować się muszą wszystkie pary o mniejszych kluczach, a w prawym wszystkie pary o większych kluczach, które, indukcyjnie, tworzą drzewce według tej samej zasady. \square

¹Przy analizie drzewców zakładamy, że wszystkie priorytety są różne.

Z twierdzenia tego i jego dowodu wprost wynika, że drzewiec tworzy BST o takim kształcie, jakby kolejne klucze były dodawane w kolejności rosnących priorytetów.

Twierdzenie 4. *Oczekiwana wysokość drzewca jest $O(\log n)$.*

Dowód.

□

Twierdzenie 5. *Oczekiwana ilość rotacji przy insert/delete < 2 .*

Dowód. Nasz dowód rozpoczniemy od pomocniczej definicji:

Definicja 6. *Lewym/Prawym Kręgosłupem drzewa nazywamy ścieżkę od korzenia do węzła z najmniejszym/największym kluczem (złożoną wyłącznie z lewych/prawych krawędzi).*

Rozważmy teraz długość kręgosłupów dzieci dodawanego węzła w czasie operacji *insert*. Nie trudno zauważyć, że początkowo oba są długości 0 oraz, że każda rotacja w lewo wydłuża prawy kręgosłup lewego dziecka o 1. Analogicznie każda rotacja w prawo wydłuża lewy kręgosłup prawego drzewa o 1. Wynika stąd, że ilość rotacji potrzebnych przy wstawianiu węzła jest równa długości lewego kręgosłupa prawego dziecka i prawego kręgosłupa lewego dziecka nowo wstawionego węzła (po zakończeniu procedury).

Spróbujmy teraz określić wartości oczekiwane tych parametrów. Weźmy dwa różne węzły x i y należące do drzewca, oznaczmy $i = y.key$ i $k = x.key$, i wyznaczmy zmienną losową

$$X_{ik} = \begin{cases} 1 & y \in \text{prawego kręgosłupa lewego poddrzewa } x \\ 0 & \text{wpp.} \end{cases}$$

Pokażemy teraz, że $X_{ik} = 1 \iff y.priority > x.priority \wedge y.key < x.key \wedge (\forall z \ y.key < z.key < x.key \implies y.priority < z.priority)$

Implikacja w prawo wynika wprost z definicji drzewca. Rozważmy zatem implikację w drugą stronę. Ponieważ $y.priority > x.priority \wedge y.key < x.key$ to y należy do lewego poddrzewa x . Teraz, gdyby y nie należał do prawego kręgosłupa lewego poddrzewa x to musiałby istnieć z t.ż. $y.key < z.key < x.key$ i $y.priority > z.priority$, co daje sprzeczność.

Bez straty ogólności załóżmy teraz, że klucze są kolejnymi dodatnimi liczbami naturalnymi $([1, n])$ i spróbujmy obliczyć $P(X_{ik})$.

Rozważmy teraz węzły o kluczach z $[i, k]$, jeśli chcemy, żeby $\forall z \in [i+1, k-1] \ x.priority < y.priority < z.priority$ to spośród wszystkich $(k-i+1)!$ permutacji interesują nas te, w których $y.priority$ i $x.priority$ są kolejno na dwóch pierwszych pozycjach, takich permutacji jest $(k-i-1)!$. Stąd $P(X_{ik}) = \frac{(k-i-1)!}{(k-i+1)!} = \frac{1}{(k-i+1)(k-i)}$.

Teraz policzmy:

$$\mathbf{E}[\sum_{i=1}^k X_{ik}] = \sum_{i=1}^k \mathbf{E}[X_{ik}] = \sum_{i=1}^k \frac{1}{(k-i+1)(k-i)} = \sum_{i=1}^k \frac{1}{i(i+1)} = \sum_{i=1}^k \left(\frac{1}{i} - \frac{1}{i+1}\right) = 1 - \frac{1}{k}$$

Po przeprowadzeniu analogicznego rozumowania dla lewego kręgosłupa prawego poddrzewa (klucze z $[k, n]$) otrzymujemy wartość oczekiwaną $1 - \frac{1}{n-k+1}$.

Co razem daje nam oczekiwaną ilość rotacji równą $1 - \frac{1}{k} + 1 - \frac{1}{n-k+1} = 2 - \frac{1}{k} - \frac{1}{n-k+1} < 2$.

Identyczna liczba rotacji dla delete wynika wprost z powyższego oraz Twierdzenia 3. \square

Powyższe twierdzenia pokazują, że wysokość drzewców, a tym samym operacje na nich nie tylko mają dobrą złożoność asymptotyczną, ale też, że (oczekiwany) narzut związany z rotacjami jest ograniczony przez niewielką stałą.

Algorytm 8: rotateLeft (rotacja w prawo jest analogiczna)

Input: root
new_root \leftarrow *root.right*
root.right \leftarrow *new_root.left*
new_root.left \leftarrow *root*
return *new_root*

Algoritmo 9: insert

```
Input: root, value if root = null then
|   return Node(value, random())
end
else if root.key = value then
|   return root
end
else if root.key > value then
|   root.left ← insert(root.left, value)
|   if root.left.priority < root.priority then
|   |   root ← rotateLeft(root)
|   end
end
else if root.key < value then
|   root.right ← insert(root.right, value)
|   if root.right.priority < root.priority then
|   |   root ← rotateRight(root)
|   end
end
return root
```

Algorytm 10: delete

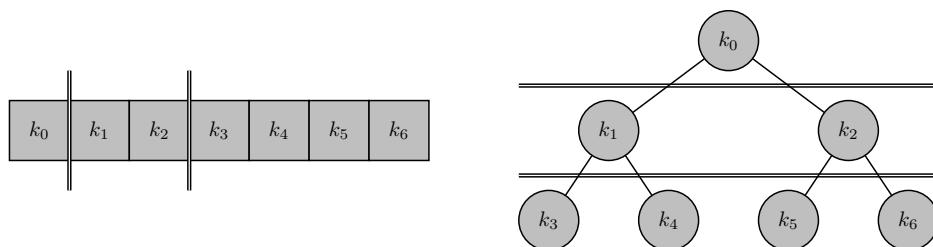
```
Input: root, value
if root = null then
  | return null
end
else if root.key > value then
  | root.left ← delete(root->left, value)
end
else if root.key < value then
  | root.right ← delete(root->right, value)
end
else if root.key = value then
  | if root.left = null then
  | | return root.right
  | end
  | else if root.right = null then
  | | return root.left
  | end
  | else if root.left.priority < root.right.priority then
  | | root ← rotateLeft(root)
  | | root.left ← delete(root.left, value)
  | end
  | else
  | | root ← rotateRight(root)
  | | root.right ← delete(root.right, value)
  | end
end
return root
```

2.2 Kopce binarne

MATEUSZ CIESIÓŁKA

Kopiec binarny to struktura danych, która reprezentowana jest jako prawie pełne drzewo binarne² 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.

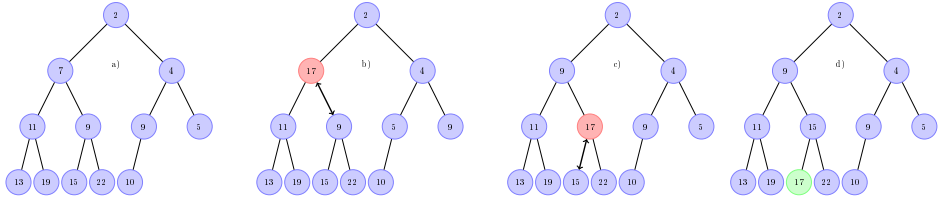
Warto zauważyć, że tak reprezentowane drzewo pozwala na łatwy dostęp do powiązanych węzłów. Synami węzła o indeksie i są węzły $2i + 1$ oraz $2i + 2$, natomiast jego ojcem jest $\lfloor \frac{i-1}{2} \rfloor$.

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.

Chciałbym, aby skrypt był 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.

²To 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.

Algorithm 11: 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

```

2.3 Algorytm rosyjskich wieśniaków

PRZEMYSŁAW JONIAK

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 kolumnie b z tych wierszy dla których wartości w kolumnie 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.

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

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

Faktycznie, otrzymaliśmy wynik poprawny. Spójrzmy raz jeszcze 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 .

W kolejnych paragrafach podamy algorytm i w celu jego udowodnienia sformułujemy *niezmiennik* oraz wykażemy jego prawdziwość.

Algorytm 12: 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

Twierdzenie 6. 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 wejściem do pętli mamy: $a'_0 = a$, $b'_0 = b$ oraz $wynik_0 = 0$, to oczywiście: $a'_0 \cdot b'_0 + wynik_0 = a \cdot b + 0 = a \cdot b$. \square

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.

- a'_i parzyste. Instrukcja *if* się nie wykona, w $i + 1$ iteracji $wynik_i$ pozostanie niezmienny, 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:

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

Nie możesz czegoś takiego założyć :)

- a'_i nieparzyste:

$$wynik_{i+1} = wynik_i + b'_i$$

$$a'_{i+1} = a'_i \operatorname{div} 2 = \frac{a'_i - 1}{2}$$

$$b'_{i+1} = b'_i \cdot 2$$

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 niezmiennik 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 jednostne 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 $O(\log a \cdot \log ab)$.

2.4 Sortowanie topologiczne

MATEUSZ CIESIÓŁKA, KRZYSZTOF PIECUCH



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.

2.5 Algorytmy sortowania

ANNA KARAŚ

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 $\text{merge}(A, p, q, r)$, gdzie A jest tablicą, a p, q, r są indeksami takimi, że $p \leq q < r$. 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 skalanych 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 skalanego 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

Algorytm 13: 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 $i \leq q$ oraz $j \leq r$ **do**
 if $A[i] \leq A[j]$ **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 $i \leq q$ **do**
 $C[k] \leftarrow A[i], i \leftarrow i + 1, k \leftarrow k + 1$
end
while $j \leq r$ **do**
 $C[k] \leftarrow A[j], j \leftarrow j + 1, k \leftarrow k + 1$
end

zbioru scalamy je za pomocą opisanej wcześniej procedury scalania podzbiorów uporządkowanych i kończymy algorytm. Zbiór jest posortowany.

Złożoność. Chociaż algorytm sortowania przez scalanie działa poprawnie nawet wówczas, gdy n jest nieparzyste, dla uproszczenia analizy założymy, 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 te-

Algorytm 14: 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

go $2T(\frac{n}{2})$ z etapu "zwyciężaj", otrzymujemy następującą rekurencję dla $T(n)$:

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

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.

2.5.1 Quick sort

In progress

2.6 Minimalne drzewa rozpinające

MATEUSZ CIESIÓŁKA

Todo, todo, todo...

2.6.1 Cut Property i Circle Property

Udowodnijmy dwie własności, które okazują się być niezwykle przydatne w dowodach dotyczących minimalnych drzew rozpinających – MST.

2.6.2 Cycle property

Niech C będzie dowolnym cyklem w ważonym grafie G . Załóżmy, że wszystkie wagi są różne.

Twierdzenie 7. *Jeżeli krawędź $e_k \in C$ jest najcięższą spośród krawędzi z C , to $e \notin MST(G)$.*

Dowód. Załóżmy nie wprost, że utworzyliśmy drzewo MST (G) w którym znajduje się krawędź e . Usuńmy ją. W ten sposób otrzymaliśmy dwa rozłączne drzewa, nazwijmy je T_1 i T_2 .

Krawędź e należała do cyklu C . Stąd wynika, że istniała druga krawędź f , która tworzyła „most” między T_1 a T_2 . Ponadto, z założenia, ma ona mniejszą wagę od e .

Dodajmy krawędź f do naszego lasu $\{T_1, T_2\}$. Otrzymaliśmy spójne drzewo MST o koszcie mniejszym od pierwotnego drzewa MST. Sprzeczność. \square

2.6.3 Cut property

Założenia takie same, jak w przypadku Cycle property: C – dowolny cykl w ważonym grafie G o różnych wagach.

Twierdzenie 8. *Podzielmy wszystkie wierzchołki cyklu na dwa rozłączne zbiory C_1 i C_2 (czyli dokonajmy cięcia). Jeżeli e jest najlżejszą krawędzią spośród łączących te dwa zbiory, to znajdzie się ona w $MST(G)$.*

Dowód. Załóżmy nie wprost, że mamy drzewo $T = MST(G)$, które nie zawiera e . Dodanie tej krawędzi utworzy cykl. Zatem istnieje druga krawędź f , która znajduje się między podzbiarami C_1 oraz C_2 .

Rozważmy drzewo $T \setminus \{f\} \cup \{e\}$. Tym sposobem otrzymaliśmy drzewo MST o mniejszej wadze. Sprzeczność. \square

- 2.6.4 Algorytm Prima
- 2.6.5 Algorytm Kruskala
- 2.6.6 Algorytm Borůvky

2.7 Algorytm Dijkstry

MIKOŁAJ SŁUPIŃSKI

Powstało wiele algorytmów pozwalających wyznaczyć najkrótszą ścieżkę w grafie z krawędziami ważonymi. Wśród nich na szczególną uwagę zasługuje algorytm Dijkstry.

2.7.1 Działanie

Niech $G = (V, E)$ będzie grafem ważonym. Dodatkowo musimy założyć, że waga $w(u, v) \geq 0$ dla wszystkich krawędzi $(u, v) \in E$.

Niech S będzie takim zbiorem wierzchołków, których najkrótsza odległość od źródła s została już określona. Algorytm Dijkstry wybiera kolejne wierzchołki $u \in V - S$ z minimalnym oszacowaniem najkrótszej ścieżki, dodaje u do S , i rozluźnia wszystkie ścieżki pozostawiając u .

Pseudokod: TODO

Algorytm zachowuje niezmiennik, że $Q = V - S$ na początku każdej iteracji pętli while.

Algorytm Dijkstry stosuje zachłanne podejście zawsze wybierając najbliższy wierzchołek w $V - S$, który dodaje do zbioru S .

2.8 Dowód poprawności algorytmu

Aby dowieść poprawności algorytmu Dijkstry skorzystamy z następującego niezmiennika pętli: Na początku każdej iteracji pętli while $d[v] = \delta(s, v)$ dla każdego wierzchołka $v \in S$.

Wystarczy udowodnić, że dla każdego wierzchołka $u \in V$ mamy $d[u] = \delta(s, u)$ w momencie kiedy u jest dodane do zbioru S . Kiedy już udowodnimy, że $d[u] = \delta(s, u)$, polegamy na ograniczeniu górnym własności, aby pokazać, że równość jest potem zachowana.

Inicjalizacja: Na samym początku, S jest zbiorem pustym, więc niezmiennik jest oczywiście zachowany.

Utrzymanie: Chcemy pokazać, że z każdą iteracją zachowana jest równość $d[u] = \delta(s, u)$ dla wszystkich wierzchołków dodanych do zbioru S . Załóżmy nie wprost, że u jest pierwszym takim wierzchołkiem, że $d[u] \neq \delta(s, u)$ w momencie go do zbioru S . Weźmy dowolny $u \neq s$, ponieważ s jest pierwszym wierzchołkiem dodanym do S i $Ed[s] = \delta(s, s) = 0$. Skoro $u \neq s$, to S nie jest zbiorem pustym

zaraz przed dodaniem u do S . Musi istnieć ścieżka od s do u gdyż inaczej $d[u] = \delta(s, u) = \inf$ i doszlibyśmy do sprzeczności z naszym założeniem, że $d[u] \neq \delta(s, u)$. Skoro istnieje conajmniej jedna ścieżka, istnieje też ścieżka najkrótsza. Przed dodaniem u do S , ścieżka p łączy wierzchołek w S , powiedzmy s , z wierzchołkiem w $V - S$, powiedzmy y . Rozważmy pierwszy wierzchołek y należący do p t. że $y \in V - S$ i niech $x \in S$ będzie poprzednikiem y . Możemy podzielić ścieżkę p na dwie podścieżki, p_1 łączącą s z x oraz p_2 łączącą y z u (ścieżki te mogą być pozbawione krawędzi).

Chcemy udowodnić, że $d[y] = \delta(s, y)$ gdy u jest dodane do S . Aby to zrobić zauważmy, że $x \in S$. Skoro u jest pierwszym wierzchołkiem, dla którego $d[u] \neq \delta(s, u)$, to $d[x] = \delta(s, x)$ w momencie kiedy x został dodany do S . Krawędź (x, y) została wtedy zrelaksowana, z czego wynika powyższa równość.

Możemy teraz uzyskać sprzeczność pozwalającą nam udowodnić, że $d[u] = \delta(s, u)$. Skoro y występuje przed u na najkrótszej ścieżce od s do u , a wszystkie wagi krawędzi są nieujemne (w szczególności krawędzi należących do p_2). Otrzymujemy $\delta(s, y) \leq \delta(s, u)$, więc TODO

Ale skoro oba wierzchołki u i y były w $V - S$ gdy ustaliliśmy u mamy $d[u] \leq d[y]$. Zatem, obie nierówności są tak na prawdę równościami, dzięki którym $d[y] = \delta(s, y) = \delta(s, u) = d[u]$. W rezultacie $d[u] = \delta(s, u)$, co przeczy naszemu wyborowi u . Wnioskujemy, że $d[u] = \delta(s, u)$ gdy dodamy u do S , a własność ta jest zachowana od tego momentu.

Zakończenie: Na końcu, Q jest puste, co w połączeniu z naszym niezmiennikiem, że $Q = V - S$ implikuje, że $S = V$, zatem $d[u] = \delta(s, u)$ dla każdego wierzchołka $u \in V$.

2.9 Analiza

Skoro wiemy już jak działa algorytm Dijkstry oraz wiemy, że działa poprawie należy zastanowić się z jaką prędkością on działa. TODO

2.10 Problemy

TODO

2.11 Problemy NP

WOJCIECH BALIK

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 7. *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 8. *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 9 (Klasa NP). *Klasą NP nazywamy zbiór problemów decyzyjnych L t. że istnieje algorytm wielomianowy A dla którego prawdziwe jest następujące zdanie:*

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

W powyższej definicji możemy myśleć o y jako o odpowiedzi dla algorytmu, lub nawet gotowym rozwiązaniu, natomiast A jest weryfikatorem, który używając odpowiedzi 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 algorytmu jest ograniczony przez $O(|V| + |E|)$.

Dowód \Rightarrow

Niech x będzie grafem zawierającym cykl Hamiltona. Wówczas naszym y będzie 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. \square

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

Definicja 10 (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 11 (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. \square

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 można 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 klik
6. Problem plecakowy (NP -trudny)
7. Problem sumy podzbioru
8. Problem minimalnego pokrycia zbioru (NP -trudny)

2.12 Sieci przełączników Benesa-Waksmana

MARCIN BARTKOWIAK

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

2.12.1 Budowa

Sieć składa się z przełączników. Każdy z przełączników 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 $i + 1 \bmod 2$ ($i \in \{0, 1\}$)

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

2.12.2 Konstrukcja 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 permutację; stan 1 generuje identyczność; 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 przełączników 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

MICHAŁ WIERZBICKI

Problem pokrycia zbioru jest problemem optymalizacyjnym związany z problemem alokacji zasobów. Przedstawimy zachłanny algorytm o logarytmicznym współczynnikiem 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 $c : S_i \rightarrow \mathbb{R}$, każdemu podzbiorkowi 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:

$$c(\mathcal{T}) = \sum_{T \in \mathcal{T}} c(T)$$

Mając zdefiniowaną cenę podzbior możemy zdefiniować cenę rynkową elementu, która będzie naszym kryterium wyboru podzbiorów. Niech e_1, e_2, \dots, 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 :

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

gdzie S_i jest to i -ty zbiór wybierany przez algorytm, S_{j_i} jest pierwszym zbiorem pokrywającym e_i , czyli

$$j_i = \min \left\{ 1 \leq k < n : e_i \in S_k \setminus \bigcup_{l=1}^{k-1} S_l \right\}.$$

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

Algorytm 15: 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. $f_i \leq \frac{c(OPT)}{n-i+1}$, gdzie OPT jest rozwiązaniem optymalnym.

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

Koszt algorytmu zachłannego ALG

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

2.14 Przynależność słowa do języka

PRZEMYSŁAW JONIAK

TODO: co oznacza gwiazdka nad zbiorem?

W tym rozdziale przedstawimy algorytm sprawdzające czy dane słowo należy do języka generowanego przez gramatykę bezkontekstową. Algorytm ten działa w czasie $\Theta(n^3)$ względem długości słowa i jego idea jest oparta na technice programowania dynamicznego. Na początek wprowadźmy parę definicji i oznaczeń.

Definicja 12. *Gramatyka bezkontekstowa to taka czwórka $\langle N, T, P, S \rangle$, że:*

- *N i T to skończone zbiory rozłączne. N nazywamy zbiorem **nieterminali**, a T zbiorem **terminali**.*
- *P - zbiór **produkcji** - to podzbiór $(N \times (N \cup T))^*$*
- *S to **symbol startowy** - wyróżniony element ze zbioru produkcji.*

Będziemy przyjmować, że elementy zbioru terminali to duże litery alfabetu angielskiego, np. $N = \{A, B, S\}$, a elementy zbioru terminali to małe litery, np. $T = \{a, b, c, \epsilon\}$ (z wyjątkiem epsilon - jest to *znak pusty*). Zbiór produkcji to zbiór par (L, R) , gdzie L jest terminalem, a R jest ciągiem złożonym z terminali i nieterminali. Parę (L, R) będziemy zapisywali w postaci $L \rightarrow R$, np.: $A \rightarrow aAb$, $A \rightarrow c$, $B \rightarrow b$, $A \rightarrow aSbB$, $S \rightarrow A$. Jeżeli w zbiorze produkcji jeden terminal pojawia się "po prawej strzałki" więcej niż raz, np. $B \rightarrow ab$, $B \rightarrow b$, to zapisujemy te dwie produkcje w skrócie: $B \rightarrow ab|b$. Poprawną produkcją nie jest $AB \rightarrow b$, ponieważ z lewej strony strzałki znajdują się dwa nieterminale.

Aby wyprowadzić konkretne słowo, np. $aacbb$, to musimy zacząć od symbolu startowego S i kolejno "podmieniać" terminale zgodnie ze zbiorem produkcji:

$$S \Rightarrow A \Rightarrow aAb \Rightarrow aaAbb \Rightarrow aacbb$$

W powyższym przykładzie kolejno skorzystaliśmy z produkcji: $S \rightarrow A$, $A \rightarrow aAb$ (dwukrotnie), $A \rightarrow c$. Słowo $aabb$ zostało *wyprowadzone* w gramatyce G .

Definicja 13. *Niech $G = \langle N, T, P, S \rangle$, $a, b, c \in (N \cup T)^*$ oraz $A \in N$. Ze słowa aAb można w G **wyprowadzić** słowo acb jeżeli $A \rightarrow c$ jest produkcją z P . Zapisujemy to: $aAb \Rightarrow acb$.*

Jeżeli rozważymy zbiór wszystkich słów, które da się wyprowadzić gramatyce G , to zbiór ten nazywamy **językiem** $L(G)$ nad gramatyką G :

Definicja 14. *Niech $G = \langle N, T, P, S \rangle$. Język $L(G)$ generowany przez gramatykę G to:*

$$L(G) = \{w : w \in T^* \wedge S \Rightarrow^* w\}$$

$O \Rightarrow^*$, czyli o tranzytywnym domknięciu relacji \Rightarrow , możemy myśleć jako wielokrotnym zaaplikowaniu (iterowaniu) relacji \Rightarrow .

W naszym algorytmie będziemy używać gramatyk bezkontekstowych, w których po prawej stronie każdej produkcji znajdują się albo dwa terminale albo jeden nieterminal:

Definicja 15. *Gramatyka jest w postaci Chomsky’ego jeżeli każda produkcja jest w jednej z poniższych postaci:*

- $A \rightarrow BC$ (typ I)
- a (typ II)

gdzie A, B, C są nieterminalami i a jest terminalem.

Zauważmy, że każdą gramatykę bezkontekstową można przedstawić w postaci Chomsky’ego. Wystarczy każdą produkcję niebędącą w pożądanej postaci rozpiąć na kilka innych, poprawnych produkcji.

Mając daną gramatykę G w postaci Chomsky’ego oraz słowo w możemy zadać pytanie: czy słowo $w = a_1a_2\dots a_n$ należy do języka generowanego przez tę gramatykę? Jeżeli w jest długości jeden, to znaczy, że w jest terminalem i wystarczy sprawdzić, czy w G istnieje produkcja $S \rightarrow w$. Jeżeli długość w jest większa od 1 i jeżeli w należałoby do języka, to ostatnia produkcja w wyprowadzeniu w musiała mieć postać $X \rightarrow YZ$ (bo gramatyka jest w postaci Chomsky’ego). W takim razie słowo w da się podzielić na dwie części $a_1a_2\dots a_i$ oraz $a_{i+1}\dots a_n$, takie, że pierwszą da się wyprowadzić z Y , a drugą z Z . Nie znamy indeksu i , więc musimy sprawdzić wszystkie możliwe jego wartości. Następnie tę procedurę powtarzamy zarówno dla Y jak i Z .

Niestety, może się okazać, że wiele takich samych fragmentów słowa w możemy obliczać wielokrotnie. Takie podejście rekurencyjne skutkuje wykładniczym czasem działania. Aby temu zapobiec zastosujemy programowanie dynamiczne. Zaczniemy analogicznie jak w algorytmie obliczania n -tej liczby Fibonacciego: rozpoczniemy od małych fragmentów, a pośrednie wyniki obliczane iteracyjne będziemy spamiętywać.

Szkic algorytmu Mamy daną gramatykę $G = \langle N, T, P, S \rangle$ oraz słowo $w = a_1a_2\dots a_n$. Chcemy się dowiedzieć czy $w \in L(G)$.

- Na początku przeglądamy fragmenty w długości jeden, czyli a_i dla $i = 1..n$. Dla każdego a_i musimy sprawdzić czy istnieje taka produkcja, w której po prawej stronie występuje a_i . Jeżeli istnieje, to zapamiętujemy nieterminal po lewej stronie produkcji.
- Teraz będziemy rozważać kolejno wszystkie fragmenty w długości 2, 3, ... n :

- Fragmentów długości 2 jest $n - 1$: $a_1a_2, a_2a_3, \dots, a_{n-1}a_n$.
- Fragmentów długości 3 jest $n - 2$: $a_1a_2a_3, a_2a_3a_4, \dots, a_{n-2}a_{n-1}a_n$; itd.
- Będą dwa fragmenty długości $n - 1$ (w bez kolejno: ostatniego i pierwszego znaku) oraz jeden długości n - całe słowo w .

Weźmy fragment $a_i a_{i+1} \dots a_j$ ($1 \leq i < j \leq n$). Gdyby ten fragment należał do języka, to dało by się go wyprowadzić pewną produkcją $X \rightarrow YZ$. Aby to sprawdzić, musimy ciachnąć $a_i a_{i+1} \dots a_j$ na dwie niepuste połowy. Możemy to zrobić na $j - i$ sposobów:

$$\begin{array}{cc} a_i | a_{i+1} a_{i+2} \dots a_j & a_i a_{i+1} | a_{i+2} \dots a_j \\ a_i \dots a_k | a_{k+1} \dots a_j & a_i \dots a_{n-1} | a_n \end{array}$$

Dla każdego podziału sprawdzamy czy zarówno prawa strona jak i lewa strona dała się wcześniej wyprowadzić - takie sprawdzenie jest darmowe, wcześniej już to policzyliśmy (albo i nie). Jeżeli te części istnieją, to wystarczy sprawdzić czy istnieje taka produkcja $X \rightarrow YZ$, że z X możemy wyprowadzić pierwszą część fragmentu: $a_i \dots a_k$, a z Y można wyprowadzić drugą: $a_{k+1} \dots a_j$. Jak istnieje, to zapamiętujemy nieterminale po prawej stronie produkcji.

- w należy do języka, jeżeli okaże się, że symbol startowy wyprowadza w .

Wszystkich fragmentów słowa w jest rzędu n^2 i dla każdego fragmentu wykonujemy operacji proporcjonalnie do jego długości. Stąd wykonamy $\Theta(n^3)$ operacji.

Zauważmy, że obliczenia możemy zorganizować w tabeli $n \times n$. W komórkach przekątnej wpisujemy wyniki kroku pierwszego: nieterminale, które wyprowadzają pojedynczy znak. W kolejnych przyprzekątnych obliczamy fragmenty długości $2, 3, \dots, n$. Jeżeli w komórce $[1, n]$ znajdzie się nieterminał S , to dane słowo jest wyprowadzalne.

Przykład Niech dana będzie gramatyka G , w której: $T = \{a, b\}$, $N = \{S, A, B\}$, a zbiór produkcji wygląda następująco:

$$P = \{S \rightarrow SS | AB, A \rightarrow AS | AA | a, B \rightarrow SB | BB | b\}$$

oraz dane słowo $w = aabbab$.

Najpierw rozważamy wszystkie fragmenty długości 1: a, a, b, b, a, b . Każdy z nich da się wyprowadzić albo z produkcji $A \rightarrow a$ albo z $B \rightarrow b$. Skoro $w_{1,1} = a$ oraz $A \rightarrow a$, to do komórki $(1, 1)$ tabeli wpisujemy A . Analogicznie wypełniamy resztę przekątnej:

Teraz fragmenty długości 2: aa, ab, bb, ba, ab .

	1	2	3	4	5	6
1	{A}					
2		{A}				
3			{B}			
4				{B}		
5					{A}	
6						{B}

- Fragment $w_{1,2} = aa$ można tylko na jeden sposób podzielić na dwie części: a oraz a . Z poprzedniego kroku wiemy, że a dało się już wyprowadzić z nie-terminala A . Istnieje produkcja $A \rightarrow AA$, więc do komórki (1,2) wpisujemy A .
- $w_{2,3} = ab$. Dzielimy na pół. Z poprzedniej iteracji wiemy, że da się wyprowadzić słowo a oraz b z kolejno terminala A oraz B . Istnieje produkcja $S \rightarrow AB$, więc do komórki (2,3) wpisujemy S .
- Analogicznie wypełniamy resztę przyprzekątnej. Zauważmy jednak, że np. przy $w_{4,5}$ istnieją nietermianle, z których da się wyprowadzić b oraz a , ale nie istnieje produkcja, która wyprowadza te nieterminale (nie ma produkcji postaci: $X \rightarrow BA$). Zatem do komórki (4,5) nic nie wpisujemy:

	1	2	3	4	5	6
1	{A}	{A}				
2		{A}	{S}			
3			{B}	{B}		
4				{B}	-	
5					{A}	{S}
6						{B}

Teraz będziemy rozważać fragmenty długości 3: aab, abb, bba, bab .

•

2.15 Pokrycie wierzchołkowe

KRZYSZTOF STARZYK

MPK Wrocław chciałoby dokonać inspekcji wszystkich torów tramwajowych w mieście (wystarczy odwiedzić przystanek by dokonać inspekcji incydentnych) torów. Podwykonawca (TOREX) żąda opłaty za każdy odwiedzony przystanek z osobna więc MPK chciałoby zminimalizować liczbę przystanków które będzie musiała odwiedzić ekipa TOREXu (zachowując przy tym podstawowe zadanie jakim jest inspekcja WSZYSTKICH torów). MPK Wrocław cannot into informatyka więc nie rozwiąże tego problemu (rozwiąże go podwyżką cen biletów). My co prawda nie potrafimy w czasie wielomianowym dostarczyć żadanego rozwiązania, ale wiemy jak się do tego zabrać.

Bardziej formalnie:

Definicja 16. *Pokryciem wierzchołkowym (dal. PW) grafu $G = (V, E)$ nazywamy zbiór V' tż.: $V' \subseteq V \wedge (\forall e \in E, \exists v \in V' : v \in e)$.*

Problem pokrycia wierzchołkowego (dal. PPW) będziemy rozpatrywać na dwa sposoby: optymalizacyjnym (jakie jest najmniejsze PW) i decyzyjnym (czy istnieje PW rozmiaru k); posiadając wszelakie zastosowania praktyczne jest oczywiście problemem NP-zupełnym.

Jeżeli przystanki tramwajowe potraktujemy jako wierzchołki a tory między nimi jako krawędzie, to PW nazwiemy taki podzbiór przystanków że wszystkie tory mają przynajmniej jeden koniec kończący się na przystanku z naszego podzbioru.

Przybliżone rozwiązanie: W dość prosty sposób możemy uzyskać przybliżone rozwiązanie (co najmniej?) dwukrotnie gorsze od optymalnego w sensie mocy otrzymanego zbioru). Idea jest następująca: dla każdej krawędzi e w E' ($= E$) weźmy dwa wierzchołki które e łączy, dodajmy je do zbioru rozwiązań i usuńmy z E' wszystkie krawędzie incydentne do nich.

16: Przybliżone rozwiązanie PPW

Input: $G = (V, E)$ **Output:** S - pewne PW dla grafu G $E' = E$ **while**
 $E' \neq \{\}$ **do**
 e' - dowolna krawędź łącząca wierzchołki (u, v) $S = S \cup u$ $S = S \cup v$
 $E' = E' \setminus$ (wszystkie krawędzie incydentne do u i v)
end

2.16 Algorytm znajdowania dwóch najbliższych punktów

TOMASZ NANOWSKI

Teraz zajmiemy się problemem znalezienia pary najmniej odległych punktów w zadanym zbiorze $Q = \{(x_1, y_1), \dots, (x_n, y_n)\}$. Interesować nas będzie odległość euklidesowa, czyli szukamy takich indeksów i, j , że $d(p_i, p_j) = \min\{d(p_k, p_l) \mid 1 \leq p < l \leq n\}$, gdzie $d(p_k, p_l) = \sqrt{(x_k - x_l)^2 + (y_k - y_l)^2}$

2.16.1 Podejście siłowe

W podejściu siłowym potrzebujemy wyznaczyć i porównać wszystkie odległości pomiędzy punktami. Jest ich $\binom{|Q|}{2}$, czyli w taki sposób problem można rozwiązać w czasie $O(n^2)$. W następnym rozdziale pokażemy jak zrobić to szybciej.

2.16.2 Podejście Dziel i Zwyciężaj

Wiemy już, że problem ten można naiwnie rozwiązać w czasie $\Theta(n^2)$. Zastanówmy się jak wykorzystując strategię Dziel i Zwyciężaj zrobić to szybciej. Już po chwili zastanowienia widać, że nie jest to takie trywialne, ponieważ po podziale zbioru na 2 części i znalezieniu dla nich rozwiązań, i tak musielibyśmy sprawdzić wszystkie odległości pomiędzy tymi zbiorami. Ale czy na pewno?

Nasz algorytm będzie wywoływał się rekurencyjnie, więc w celu uniknięcia wielokrotnego sortowania, wykorzystamy dwie tablice X i Y , które zawierać będą wszystkie punkty z Q posortowane odpowiednio po x -owej i y -owej współrzędnej. W tym miejscu istotnym jest, aby punkty ze wszystkich tablic były odpowiednio *połączone* między sobą. Dzięki temu przy podziale zbioru na dwie części: Q_L i Q_R odtworzenie tablic X_L, Y_L i X_R, Y_R będzie możliwe w czasie $O(|Q|)$. Wystarczy przeglądać po kolei elementy z tablic X oraz Y i przerzucać je do mniejszych odpowiedników, w zależności czy punkt jest w części L czy R .

Algorytm 17: Implementacja procedury NAJMNIEJ-ODLEGŁA-PARA

Input: Q - zbiór punktów, X, Y

Output: najmniejsza odległość między punktami

if $|Q| < 2$ then

 | return ∞

end

if $|Q| = 2$ then

 | return $Q[1] - Q[2]$

end

wykorzystując tablicę X znajdź prostą l dzielącą zbiór Q na
dwa prawie równoliczne zbiory

podziel Q na zbiory Q_L i Q_R względem prostej l odpowiednio
po jej lewej i prawej stronie

wyznacz tablice X_L, Y_L i X_R, Y_R

$d_L \leftarrow \text{NAJMNIEJ-ODLEGŁA-PARA}(Q_L, X_L, Y_L)$

$d_R \leftarrow \text{NAJMNIEJ-ODLEGŁA-PARA}(Q_R, X_R, Y_R)$

$d \leftarrow \min(d_L, d_R)$

$Y' \leftarrow$ punkty z Y odległe o co najwyżej d od prostej l

for $i = 1$ to $|Y'|$ do

 for $j = 1$ to $\min(7, |Y'| - i)$ do

 if $P[i] - P[i + j] < d$ then

 | $d \leftarrow |P[i] - P[i + j]|$

 end

 end

end

return d

2.17 Kopce dwumianowe w wersji leniwej

MARCIN BARTKOWIAK

Kopce dwumianowe w wersji leniwej różnią się strukturalnie od wersji gorliwej, tym, że w danym momencie możemy mieć więcej niż jeden kopiec B_k na liście.

2.17.1 Różnice w implementacji

insert

Stwórz drzewo składające się wyłącznie z danego elementu a następnie wywołaj *meld* z kopcem właściwym.

meld

Operacja meld polega na połączeniu list drzew dwóch kopców.

extract-min

W wersji leniwej, podobnie jak w Kopcach Fibonacciego to tutaj będziemy wykonywać całą pracę związaną z utrzymaniem struktury kopców.

Idea algorytmu:

1. Usuń min z listy wierzchołków, a następnie dodaj do listy wierzchołków jego dzieci.
2. Stwórz pustą tablicę B wielkości największemu stopniowi drzewa niezbędnego, w kopcu o poprawnej strukturze trzymającym wszystkie elementy($\lceil \log n \rceil$)
3. Iterując po każdym drzewie w kopcu sprawdź, czy to nie minimum(i ustaw wskaźnik minimum jeśli jest), a następnie sprawdź w tablicy B jest element o indeksie jego wielkości. Jeśli nie ma wstaw go do tablicy. Jeśli istnieje połącz dane drzewa i rekurencyjnie wstaw nowe drzewo do tablicy.

Pozostałe operacje

Pozostałe operacje implementujemy identycznie jak w gorliwych kopcach dwumianowych.

2.17.2 Analiza złożoności

Zdefiniujmy funkcję potencjału $\Phi = \#drzew\ w\ kopcu$

meld

Meld w oczywisty sposób nie zmienia sumy potencjałów kopców, jedynym kosztem będzie przepięcie wskaźników, więc złożoność tej funkcji to $\Theta(1)$

insert

Dodając drzewo zwiększamy potencjał o jeden. Następnie będziemy musieli wykonać meld, które kosztuje jedną operację.

$$\Delta(\Phi) + 1 = 1 + 1 = 2 \in \Theta(1)$$

extract-min

Na początku będziemy musieli wstawić wszystkie dzieci od minimalnego elementu; zajmie to $O(\log n)$.

Oznaczmy T jako wszystkich drzew po tej operacji.

Dominującym kosztem rzeczywistym łączenia będzie iteracja po wszystkich drzewach (w czasie $\Theta(T)$).

Niech $\Delta(\Phi)$ oznacza różnicę potencjałów między kopcem po dodaniu dzieci minimalnego elementu, a kopcem po złączeniu drzew tego samego stopnia. Koszt amortyzowany wyrażać się więc będzie wzorem.

$$\Delta(\Phi) + O(\log n) + \Theta(T) = O(\log(n)) - T + O(\log(n)) + \Theta(T) = O(\log(n))$$

2.18 Counting sort

DOMINIKA WÓJCIK

W tym rozdziale będzie można zapoznać się z algorytmem sortowania przez zliczanie (ang. *counting sort*). W metodzie zakładamy, że każdy z n elementów ciągu jest liczbą całkowitą z przedziału od 0 do k . Sortowanie przez zliczanie działa w czasie $\Theta(n + k)$. Jeśli jednak założymy, że $k \in O(n)$, możemy powiedzieć, że jest to czas liniowy.

Idea. W sortowaniu przez zliczanie głównym pomysłem jest wyznaczenie dla każdego x z ciągu wejściowego liczby elementów mniejszych od x . Dlaczego? Załóżmy, że policzyliśmy, że jest 10 elementów mniejszych od x . Z tego możemy już wywnioskować, że element x będzie na 11 miejscu w tablicy. Jeśli chcemy dopuścić powtórzenia w ciągu wejściowym musimy dokonać drobnej modyfikacji i zamiast pamiętać liczby mniejsze od x , pamiętamy te mniejsze lub równe. Spójrzmy na pseudokod algorytmu.

Algorytm 18: Procedura counting sort

```
Input: tablica  $A$ , liczba  $k$ 
Output: posortowana tablica  $B$ 
for  $i \leftarrow 0..k$  do
  |  $C[i] \leftarrow 0$ ;
end
for  $j \leftarrow 1..A.length$  do
  |  $C[A[j]] \leftarrow C[A[j]] + 1$ ;
end
for  $i \leftarrow 1..k$  do
  |  $C[i] \leftarrow C[i] + C[i - 1]$ ;
end
for  $j \leftarrow A.length..1$  do
  |  $B[C[A[j]]] \leftarrow A[j]$ 
  |  $C[A[j]] \leftarrow C[A[j]] - 1$ ;
end
```

Opis. A teraz pokażemy, czemu dokładnie służą kolejne pętle for w pseudokodzie.

1. Na początku inicjujemy tablicę C .
2. W następnej pętli for przechodzimy po kolei po wszystkich elementach i dla wartości x danego elementu, wartość tablicy $C[x]$ zwiększamy o jeden. A

zatem po wykonaniu drugiej pętli tablica C zawiera liczbę wystąpień elementów np. x występuje $C[x]$ razy.

3. Obliczamy ile jest elementów mniejszych lub równych x . Odbywa się to przez sumowanie komórek $C[x]$ z wartościami mniejszych indeksów tablicy.
4. W ostatniej pętli `for` umieszczamy elementy na właściwych pozycjach w tablicy wyjściowej B . Jeśli w tablicy wejściowej A nie ma powtórzeń, to dla każdego $A[j]$ wartość $C[A[j]]$ jest poprawnym ostatecznym numerem pozycji w tablicy B . Jeśli jednak dopuszczamy powtórki musimy tę wartość rekrementować za każdym razem gdy wartość $A[j]$ jest wstawiana do tablicy B . Dzięki temu następny element o tej samej wartości (jeśli istnieje) zostanie wstawiony do tablicy B na pozycję o numerze o jeden mniejszym.

Złożoność. Bedziemy określać złożoność czasową po przez oszacowanie każdej kolejnej pętli. Pierwsza działa w czasie $\Theta(k)$, druga w czasie $\Theta(n)$, kolejna w czasie $\Theta(k)$ a ostatnia w $\Theta(n)$. Całkowity czas działania procedury to $\Theta(k + n)$ ale w praktyce najczęściej $k \in O(n)$ a wtedy złożoność czasowa to $\Theta(n)$.

Stabilność. Stabilność to własność sortowania, dzięki której elementy o tych samych wartościach występują w tablicy wynikowej w takiej samej kolejności jak w ciągu wejściowym. Sortowanie przez zliczanie jest stabilne, co jest ważną zaletą pozwalającą wykorzystać tą procedurę w sortowaniu pozycyjnym.

Uwagi.

1. Gdy przyjrzymy się dokładnie algorytmowi sortowania przez zliczanie możemy zastanawiać się, dlaczego postępujemy w tak dziwny sposób? Przecież mając zliczone wystąpienia każdej wartości w licznikach, możemy je od razu przepisać do zbioru wyjściowego. Istotnie tak by było, gdyby chodziło jedynie o posortowanie liczb. Jest jednak inaczej. Celem nie jest posortowanie jedynie samych wartości elementów. Sortowane wartości są zwykle tzw. kluczami, czyli wartościami skojarzonymi z elementami, które wyliczono na podstawie pewnego kryterium sortując klucze chcemy posortować zawierające je elementy. Dlatego do zbioru wynikowego musimy przepisać całe elementy ze zbioru wejściowego, gdyż w praktyce klucze stanowią jedynie część (raczej małą) danych zawartych w elementach. Zatem algorytm sortowania przez zliczanie wyznacza docelowe pozycje elementów na podstawie reprezentujących je kluczy, które mogą się wielokrotnie powtarzać. Następnie elementy są umieszczane na właściwym miejscu w zbiorze wyjściowym.
2. Być może niektórych niepokoi fakt, że osiągnęliśmy lepszą złożoność czasową niż mówi o tym dolna granica sortowania, czyli $\Theta(n \lg n)$. Zauważmy jednak, że w tym wypadku mamy więcej danych o elementach. Wymagamy aby były to liczby całkowite z danego zakresu. Ponadto dolna granica

mówi nam o wykonywaniu algorytmu sortowania za pomocą porównań. W sortowaniu przez zliczanie nie korzystamy z porównań a zatem ta granica nie dotyczy tego algorytmu.

Podsumowanie. Sortowanie przez zliczanie działa w czasie liniowym, jest stabilne, wymaga aby elementy były liczbami całkowitymi z danego zakresu. Jest ono wykorzystywane w sortowaniu pozycyjnym.

Rozdział 3

W ogóle nie zaczęte

3.1 Złożoność obliczeniowa

JESZCZE NIKT

Todo, todo, todo...

3.2 Model obliczeń

JESZCZE NIKT

Todo, todo, todo...

3.3 Programowanie dynamiczne na drzewach

JESZCZE NIKT

Todo, todo, todo...

3.4 Algorytm szeregowania

JESZCZE NIKT

Todo, todo, todo...

3.5 Drzewa Splay

JESZCZE NIKT

Todo, todo, todo...

3.6 Zbiory rozłączne

JESZCZE NIKT

Todo, todo, todo...

3.7 Drzewa przedziałowe

JESZCZE NIKT

Todo, todo, todo...

3.8 Tablice hashujące

JESZCZE NIKT

Todo, todo, todo...

3.9 Algorytm Shift-And

JESZCZE NIKT

Todo, todo, todo...

3.10 Wyszukiwanie wzorca z wykorzystaniem automatów skończonych

JESZCZE NIKT

Todo, todo, todo...

3.11 Algorytm Knutha-Morrisa-Pratta

JESZCZE NIKT

Todo, todo, todo...

3.12 Algorytm Karpa-Rabina

JESZCZE NIKT

Todo, todo, todo...

3.13 Drzewa czerwono-czarne

JESZCZE NIKT

Todo, todo, todo...

3.14 B-drzewa

JESZCZE NIKT

Todo, todo, todo...

3.15 Słownik statyczny

JESZCZE NIKT

Todo, todo, todo...

3.16 Geometria obliczeniowa

JESZCZE NIKT

Todo, todo, todo...

3.17 Kopce Fibonacciego

JESZCZE NIKT

Todo, todo, todo...

3.18 Szybka Transformata Fouriera

JESZCZE NIKT

Todo, todo, todo...

3.19 Drzewo van Emde Boasa

JESZCZE NIKT

Todo, todo, todo...

3.20 Sortowanie kubełkowe

JESZCZE NIKT

Todo, todo, todo...

3.21 Model drzew decyzyjnych

JESZCZE NIKT

Todo, todo, todo...

3.22 Dolna granica znajdowania min-maksa

JESZCZE NIKT

Todo, todo, todo...

3.23 Dolne granice

JESZCZE NIKT

Todo, todo, todo...

3.24 Optymalna kolejność mnożenia macierzy

JESZCZE NIKT

Todo, todo, todo...

3.25 Drzewa rozpinające drabiny

JESZCZE NIKT

Todo, todo, todo...

3.26 Cykl Hamiltona

JESZCZE NIKT

Todo, todo, todo...

3.27 Problem spełnialności formuł logicznych

JESZCZE NIKT

Todo, todo, todo...

3.28 Trójwymiarowe skojarzenia

JESZCZE NIKT

Todo, todo, todo...

3.29 Stokrotki

JESZCZE NIKT

Todo, todo, todo...

3.30 Otoczka wypukła

JESZCZE NIKT

Todo, todo, todo...

3.31 Najdłuższy wspólny podciąg

JESZCZE NIKT

Todo, todo, todo...

3.32 Algorytm Karatsuby

JESZCZE NIKT

Todo, todo, todo...

3.33 Algorytm Borůvky

MICHAŁ BRONIKOWSKI

Algorytm Borůvky jest przykładem algorytmu zachłannego znajdującego minimalne drzewo rozpinające. Algorytm został opracowany przez Czeskiego matematyka Otakara Borůvke w 1926 roku. Miał pomóc zoptymalizować zasięg sieci elektrycznej w Czechach.

3.33.1 Działanie

Niech $G = (V, E)$ będzie grafem ważonym, w którym wszystkie krawędzie mają różne wagi. W pierwszym kroku algorytm Borůvky wybiera dla wszystkich wierzchołków $v \in V$ najlżejszą krawędź $e \in E$, która jest z nimi incydentna i dołącza wierzchołki wraz z wybranymi krawędziami do lasu F . W następnym kroku tworzymy graf G' będący kopią G , z tą różnicą, że wszystkie spójne składowe z F zostały ze sobą "połączone" w pojedyncze wierzchołki zwane superwierzchołkami. Powyższe czynności powtarzamy zamieniając G' na G , dopóki nie otrzymamy jednej spójnej składowej, będzie to nasze MST .

3.33.2 Pseudokod

Algorytm 19: Algorytm Borůvky

Input: \mathcal{G} - Graf, w którym szukamy MST

Output: \mathcal{T} , takie że \mathcal{T} jest MST \mathcal{G}

$\mathcal{T} \leftarrow \emptyset$

$\mathcal{F} \leftarrow \emptyset$

while $|\mathcal{G}(V)| > 1$ **do**

 Do \mathcal{F} dołącz wszystkie wierzchołki z \mathcal{G} wraz z incydentnymi krawędziami o najniższej wadze

 W \mathcal{F} wszystkie spójne składowe zamień na pojedyncze wierzchołki

$\mathcal{G}' \leftarrow \mathcal{G} \setminus \mathcal{F}$

$\mathcal{G} \leftarrow \mathcal{G}'$

end

3.33.3 Dowód poprawności

W każdym kroku algorytmu budujemy rozwiązanie będące MST , poprzez dodanie do niego spójnych składowych z F w postaci jednego wierzchołka, a z *Cut Property* wiemy, że te wierzchołki należą do MST

3.33.4 Złożoność czasowa

W pojedynczym kroku algorytmu wykonujemy maksymalnie m operacji, gdzie m jest ilością krawędzi w G . W każdym kroku algorytmu ilość wierzchołków zmniejsza się przynajmniej o połowę. W związku z tym złożoność algorytmu Borůvky wynosi $O(m \log n)$, gdzie n oznacza ilość wierzchołków w G .

3.34 Algorytm Prima

JESZCZE NIKT

Todo, todo, todo...

3.35 Algorytm Kruskala

JESZCZE NIKT

Todo, todo, todo...

3.36 Statystyki pozycyjne

JESZCZE NIKT

Todo, todo, todo...

3.37 Algorytm magicznych piątek

JESZCZE NIKT

Todo, todo, todo...

3.38 Drzewa AVL

JESZCZE NIKT

Todo, todo, todo...

3.39 Izomorfizm drzew

JESZCZE NIKT

Todo, todo, todo...

3.40 Sortowanie ciągów różnej długości

JESZCZE NIKT

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	✓				

Dodatek B

Zadania na programowanie dynamiczne i algorytmy zachłanne

Na egzaminie z Algorytmów i Struktur Danych, na drugiej części egzaminu lubią pojawiać się zadania na programowanie dynamiczne oraz algorytmy zachłanne. Z doświadczenia wiem, że najlepszą metodą na nauczenie się rozwiązywania tego typu zadań jest zrobienie dużej ilości zadań tego rodzaju. W niniejszym dodatku umieszczam mały zbiór zadań w nadziei, że pomoże on Czytelnikowi lepiej radzić sobie z zadaniami tego typu.

Optymalne mnożenie macierzy

O mnożeniu macierzy mowa była już w rozdziale 1.4. Przedstawiliśmy tam algorytm mnożenia macierzy, który dla macierzy o rozmiarze $n \times m$ oraz $m \times p$ działa w czasie $\Theta(n \cdot m \cdot p)$. Ponadto powinniśmy pamiętać z Algebry, że mnożenie macierzy jest przemienne, czyli, że $(A \cdot B) \cdot C = A \cdot (B \cdot C)$. Lub mówiąc inaczej - ustawienie nawiasów w ciągu iloczynów macierzy nie wpływa na wynik mnożenia. Jednakże ustawienie nawiasów wpływa na coś innego - na sumaryczną ilość operacji jaką wykona algorytm mnożący wszystkie macierze. Dla przykładu niech macierz A będzie rozmiaru 5×10 , macierz B rozmiaru 10×20 , a C rozmiaru 20×35 . Jeśli algorytm pierw przemnoży macierz A przez B a następnie wynik tego mnożenia przez C , wykona 4500 operacji. Jeśli natomiast najpierw przemnoży B przez C a następnie A przez wynik poprzedniego mnożenia, wtedy wykona 8750 operacji. Mając dany ciąg a_1, a_2, \dots, a_{n+1} należy wypisać takie nawiasowa-

nie macierzy A_1, A_2, \dots, A_n przy którym algorytm wykona jak najmniejszą ilość operacji. Zakładamy przy tym, że macierz A_i jest rozmiaru $a_i \times a_{i+1}$.

Wydawanie reszty

Przyjmijmy, że mamy następujące nominały: 50 groszy, 25 groszy, 10 groszy, 5 groszy oraz 1 grosz. Chcemy wydać resztę używając tych nominałów. Możemy to zrobić na różne sposoby. Dla przykładu, gdy chcemy wydać 11 groszy, możemy to zrobić używając 10-groszówki i 1-groszówki, dwóch 5-groszówek oraz 1-groszówki, jednej 5 groszówki oraz sześciu 1-groszówek lub jedenastu 1-groszówek. Ułóż algorytm, który dla zadanej reszty liczy ile jest sposobów na wydanie reszty za pomocą wspomnianych nominałów.

Łamanie patyka

Dany jest długi patyk, który należy połamać w określonych miejscach. Jak powszechnie wiadomo (?) im dłuższy patyk, tym trudniej go złamać. Firma profesjonalnie zajmująca się łamaniem patyków, każe sobie płacić proporcjonalnie dużo od długości patyka. W zależności od tego w jakiej kolejności każemy firmie łamać patyk, może zależeć to ile za takie łamanie zapłacimy. Dla przykładu powiedzmy, że mamy patyk o długości 10 m i musimy go złamać w 2, 4 i 7 metrach. Jeśli firma wpierw złamie go w 2-gim metrze, następnie w 4-tym a na końcu w 7-mym to zapłacimy za tą niewątpliwą przyjemność $10 + 8 + 6 = 24$. Jeśli natomiast najpierw poprosilibyśmy, aby firma złamała go w 4-tym metrze, a następnie w 2-gim i 7-mym to zapłacilibyśmy $10 + 4 + 6 = 20$. Mając daną długość patyka oraz miejsca przełamania policz najtańszą kolejność łamania patyka.

Wąskie liczby

Liczbę w systemie k -tkowym nazywamy wąską, jeśli każde dwie sąsiednie cyfry różnią się od siebie nie więcej niż o jeden. Dla danych liczb k oraz n policz ile jest n -cyfrowych liczb w systemie k -tkowym, które są wąskie.

Mądre słonie

Niektórym wydaje się, że im większy jest słoń, tym jest mądrzejszy. Aby zaprzeczyć temu twierdzeniu chcesz znaleźć w podanej bazie danych słoni jak najdłuższy ciąg w którym waga słoni rośnie, a IQ maleje.

Odwracanie naleśników

Dany jest talerz z naleśnikami. Naleśniki mają różne średnice i leżą na talerzu na wspólnym stosie. Chcemy posortować naleśniki, przy czym jedyną dopuszczalną operacją jest włożenie łypatki pod wybrany naleśnik a następnie odwrócenie go wraz z wszystkimi naleśnikami leżącymi wyżej. Należy znaleźć ciąg operacji odwracania naleśników, który doprowadzi do posortowania naleśników według średnic. Możesz założyć, że naleśniki są z serem.

Upośledzony Gustaw

Gustaw potrafi liczyć. Teraz uczy się jak liczby zapisuje się na papierze. Ponieważ Gustaw jest bardzo dobrym uczniem, zna już cyfry 1, 2, 3 oraz 4. Nie zorientował się on jeszcze, że 4 to jest inna cyfra niż 1, więc myśli, że “4” to inny sposób aby zapisać “1”. Pomimo tego wymyślił bardzo “ciekawą” grę. Mając daną liczbę n zastanawia się ile zna liczb, których cyfry sumują się do n . Jeśli na przykład n jest równe 2 to Gustaw zna pięć takich liczb: 11, 14, 41 oraz 2. Opracuj algorytm, który dla zadanej liczby n policzy ile liczb zna Gustaw, które sumują się do n .

Gra w klasy

W bogatej dzielnicy Sosnowca dzieci znalazły sobie nową zabawę. Na chodniku malują szachownicę o rozmiarach $n \times n$. Następnie na każde pole szachownicy kładą różną ilość groszy. Kolejnie jedno dziecko staje w dolnym lewym rogu szachownicy i zbiera wszystkie monety leżące na tym polu. Następnie wybiera pomiędzy dwoma kierunkami: prawo lub góra i skacze na wybrane przez siebie pole (o ile jest w stanie na nie doskoczyć; powiedzmy, że jest w każdym momencie skoczyć na pole nie dalsze niż o k). Pole to musi zawierać więcej groszy niż pole na którym stał uprzednio. Z tego pola zbiera wszystkie monety i zabawę kontynuuje dopóty ma możliwe ruchy. Opracuj algorytm, który mając daną szachownicę obliczy ile dany gracz jest w stanie zebrać groszy.

Pokrycie przedziałowe

Dane są przedziały liczbowe postaci (l_i, r_i) . Należy wybrać jak najmniejszą liczbę przedziałów tak, aby ich suma zawierała przedział $(0, M)$.

Problem chińskich pałeczków

Dlaczego łyżkę kojarzy się z jesienią? Bo je się nią. Ale nie w Chinach. Tam je się pałeczkami. Profesor L. wymyślił właśnie nowy system jedzenia w którym używa się 3 pałeczek. Jednej normalnej pary oraz jednego długiego patyka, którym zjada duże kawałki dziabiąc nim. Oczywiście dobrze by było, aby dwie małe pałeczki były o zbliżonym rozmiarze, natomiast rozmiar największej pałeczki nie ma znaczenia. Formalnie - jeśli $A \leq B \leq C$ to rozmiar pałeczek to niedobroć zestawu definiujemy jako $(B - A)^2$. Profesor L. organizuje właśnie imprezę na którą przyjdzie K gości. Chce on zaprezentować swoim gościom swój nowy system. Planuje teraz wybrać ze swojej kolekcji N pałeczek K zestawów, tak aby zminimalizować sumę niedobroci zestawów.

Konstruowanie BST

Mamy zadaną wysokość H oraz liczbę elementów N . Chcemy znaleźć taką permutację liczb $1, 2, \dots, N$, że wstawiając do początkowo pustego drzewa BST otrzymamy drzewo o wysokości nie większej niż H . Jeśli jest więcej takich permutacji to chcemy znaleźć leksykograficznie najmniejszą.

Optymalne BST

Mamy zadany zbiór par $\{(e_i, p_i)\}$. Chcemy z nich utworzyć drzewo w ten sposób, aby tworzyło ono drzewo BST po wartościach e_i . Ponadto dla każdego drzewa definiujemy jego wagę jako $\sum p_i \cdot d_i$ gdzie d_i to odległość i -tego wierzchołka od korzenia. Należy znaleźć drzewo BST złożone z zadanych elementów o minimalnej wadze.

Willy Wonka

Willy Wonka chce rozdać trójce dzieci cukierki. Cukierków jest N ($N \leq 30$) i każdy z nich ma swoją wagę w gramach ($w_i \leq W \leq 20$). Willy chce rozdać wszystkie cukierki i chce aby dzieciaki dostały mniej więcej po równo - tj. chce zminimalizować różnicę w wadze między dzieciakiem, który dostał najcięższy worek cukierków a dzieciakiem, który dostał worek najlżejszy.

Ustawianie domina

Chcemy ustawić szereg domina złożony z N kostek domina. Powiedzmy, że nasz szereg wygląda tak: "DD__DxDDD_D". Chcemy teraz wstawić kolejną kostkę w pole oznaczone jako "x". Ponieważ mylić się jest rzeczą ludzką - prawdopodobieństwo, że kostka przwróci się w lewo wynosi p_l a w prawo p_r . Zakładamy, że $0 < p_l + p_r \leq 0.5$. Gdy kostka się przewróci - przewraca blok kostek znajdujący się obok niego. W przykładzie - jeśli kostka którą stawiamy przewróci się w lewo - przewróci dodatkowo jedną kostkę. Jeśli w prawo - przewróci dodatkowo aż trzy kostki. Kostki te będziemy musieli postawić na nowo. Ile wynosi oczekiwana ilość kostek, które musimy postawić w optymalnej strategii ustawienia?

Hazard

Z sześciu symboli rozlosowywane są 3 (z powtórzeniami). Stawiamy $a \leq L$ złotych na jeden symbol (gdzie L to dostępny limit na zakład). Jeśli zostanie on wylosowany to odzyskujemy swoje a złotych plus dodatkowo zyskujemy a złotych za każde wystąpienie naszego symbolu wśród 3 wylosowanych. Jacek opracował Strategię Na Pewno Wygrywającą. Wygląda ona w ten sposób, że Jacek stawia w pierwszym zakładzie 1 złoty. Jeśli przegra to w następnym zakładzie stawia dwa razy więcej niż w zakładzie poprzednim. Łatwo można zauważyć, że gdy któryś zakład wygramy - wychodzimy na plus. Wtedy zaczynamy naszą strategię od nowa - stawiając ponownie złotówkę na zakład. Problemem jest limit na zakład. Gdy Jacek go osiągnie, zaczyna strategię od nowa - mając nadzieję, że jeszcze się odkuje. Jakie jest prawdopodobieństwo, że między K -tym zakładem, a M -tym chociaż raz będziemy na plusie?

Złote monety

W Bajtocji istnieje dziwny system monetarny. Każda moneta ma wybity nominal, który jest liczbą naturalną. Każdą monetę o nominale A można w Banku Bajtockim rozmiąć na monety $\lfloor A/2 \rfloor$, $\lfloor A/3 \rfloor$ oraz $\lfloor A/4 \rfloor$. Ponadto monety można zamieniać na polskie złotówki przy kursie jeden do jednego (nie można dokonywać zamian w przeciwną stronę). Mamy jedną monetę o nominale A . Ile złotówek możemy otrzymać za nią?

Deski i gwoździe

Dany jest zbiór przedziałów $\{(s_i, k_i)\}$. Pytamy się o najmniejszy zbiór punktów $\{p_j\}$ taki, że dla każdego i istnieje takie j , że $p_j \in (s_i, k_i)$.

Sumowanie

Dany jest zbiór liczb naturalnych $\{a_i\}$, który należy zsumować. Koszt zsumowania liczb a oraz b wynosi $a + b$. W jakiej kolejności należy sumować elementy, aby zminimalizować koszt sumowania?

Rzeźbiarz

Do pracowni rzeźbiarskiej właśnie dotarła marmurowa płyta o rozmiarach $H \times W$. Płytę tą można ciąć na dwa prostokąty wzdłuż jednego z jego boków. W pracowni potrzebne są określone rozmiary płyt $\{(h_i, w_i)\}$. Jeden rozmiar można wykorzystać wielokrotnie. Pracownia chce teraz tak pociąć dostarczoną płytę, aby zminimalizować ilość “odpadków”.

Dzikie kodowanie

Każdej literze przyporządkowujemy dodatnią liczbę całkowitą, określającą jej pozycję w alfabecie. Słowa kodujemy zapisując każdą literę za pomocą przypisanej jej liczby. Na przykład słowo “BEAN” zakodowalibyśmy jako “25114”. Problem jest z dekodowaniem słowa. Dla przykładu - “25114” można zdekodować na 6 sposobów. Dla danej liczby należy policzyć na ile sposobów można ją zdekodować.

Transport przez rzekę

Mamy dany statek, który może na raz przetransportować n samochodów z jednego brzegu rzeki na drugi. Transport trwa k jednostek czasu w jedną stronę, po którym statek musi powrócić na pierwszy brzeg. Mamy danych m samochodów oraz s_i - czas przyjazdu każdego samochodu nad rzekę. Pytamy się o to w jaki sposób należy transportować samochody, aby ostatni samochód przekroczył rzekę jak najwcześniej. Ile co najmniej kursów musi zrobić statek, aby osiągnąć taki czas?

Gra na grafie

Dany mamy graf pełny z pętelkami. Każda krawędź ma jeden z trzech kolorów. Na grafie znajdują się 3 pionki. Możemy przesunąć pionek z wierzchołka u na wierzchołek v jeśli krawędź między tymi wierzchołkami jest tego samego koloru co krawędź łącząca pozycje dwóch pozostałych pionków. Pytamy się o minimalną liczbę ruchów jakie trzeba wykonać, aby przesunąć wszystkie pionki na jedno pole.

Sklep z kwiatami

Mamy dany zbiór kwiatów oraz rząd donniczek. Musimy umieścić kwiaty w donniczkach zachowując następujące zasady:

- donniczek nie można przestawiać
- jeden rodzaj kwiatu może wystąpić w wielu donniczkach
- kwiaty muszą być ustawione w kolejności alfabetycznej (narcyzy muszą znaleźć się na lewo od róż)

Nie wszystkie kwiaty tak samo ładnie wyglądają w różnych donniczkach. Bardziej formalnie - mamy zadaną macierz $\{m_{i,j}\}$ określającą “ładność” wyglądu i -tego gatunku kwiatów w j -tej donniczce. Pytamy się o najładniejsze ustawienie kwiatów w donniczkach.

TODO: Dodać zadania z `średnie.pdf`