

Sprawozdanie z listy 3 (AiSD)

Szymon Wojtasik, nr albumu: 287306

1 Problem rozcinania pręta

1.1 Idea i "naiwność" rozwiązańia

Teoretycznie problem pręta wydaje się trywialny – tniemy tam, gdzie zysk jest największy. Jednak algorytm naiwny (rekurencyjny) ma pewną ideę, by zrozumieć dlaczego samo wywoływanie funkcji bez zapamiętywania jest "ciężkie".

Kody będę prezentował w uproszczeniu, żeby było czytelniej.

```
1 int naiwny_algorytm(const vector<int>& p, int n) {
2     if (n == 0) return 0;
3     int q = INT_MIN;
4     for (int i = 1; i <= n; i++) {
5         q = max(q, p[i] + naiwny_algorytm(p, n - i));
6     }
7     return q;
8 }
```

W całym tym algorytmie najważniejsze dla mnie było:

⇒ rekurencyjne wywoływanie samego siebie w pętli (linijka 5)

⇒ brak jakiegokolwiek zapisu wyników pośrednich

Wskutek tego, dla $n = 30$ czas wykonania staje się nieznośny, co jest bardzo nieoptymalne względem wersji dynamicznych.

1.2 Odkrycie spamiętywania

Przejdzmy do modyfikacji, która jest zdecydowanie ciekawsza. Wersja iteracyjna buduje rozwiązanie od dołu, eliminując powtórzenia.

```
1 for (int j = 1; j <= n; j++) {
2     int q = INT_MIN;
3     for (int i = 1; i <= j; i++) {
4         if (q < p[i] + r[j - i]) {
5             q = p[i] + r[j - i];
6             s[j] = i;
7         }
8     }
9     r[j] = q;
10 }
```

Tutaj kluczowe było dla mnie:

⇒ wykorzystanie tablicy r do odczytu gotowych wyników (linijka 4)

⇒ tablica s (linijka 6), która pozwala potem odtworzyć gdzie wykonaliśmy cięcie.

Dzięki temu algorytm działa błyskawicznie nawet dla dużych danych.

1.3 Odzyskiwanie wyniku

Samo wyliczenie maksymalnego zysku to nie wszystko. Trzeba jeszcze wiedzieć, jakie to były kawałki.

```
1 void print_solution(const vector<int>& s, int n) {
2     while (n > 0) {
3         cout << s[n] << " ";
4         n = n - s[n];
5     }
6 }
```

Zauważmy, że pętla ‘while’ po prostu odejmuje długość odciętego kawałka od n .

⇒ Dzięki tablicy s nie musimy ponownie przeszukiwać drzewa rozwiązań. ⇒ Złożoność odzyskiwania to $O(n)$, co jest pomijalne przy samym obliczaniu ($O(n^2)$).

2 Najdłuższy Wspólny Podciąg

2.1 Macierz kierunków

W przypadku LCS (Longest Common Subsequence) mamy do czynienia z macierzą 2D. Mała zagwojtka pojawia się przy rekonstrukcji wyniku – musimy wiedzieć skąd przyszliśmy.

```
1 if (X[i] == Y[j]) {
2     c[i][j] = c[i - 1][j - 1] + 1;
3     b[i][j] = SKOS;
4 } else {
5     if (c[i - 1][j] >= c[i][j - 1]) {
6         c[i][j] = c[i - 1][j];
7         b[i][j] = GORA;
8     }
9 }
```

W tym algorytmie najważniejsze było:

⇒ "skos" oznacza znalezienie wspólnego znaku (zwiększamy wynik)

⇒ tablica b służy jako mapa drogowa do cofania się (backtracking)

2.2 Pułapka rekurencji

W wersji rekurencyjnej łatwo zapomnieć o najważniejszym warunku, który zmienia wszystko.

```
1 if (c[i][j] != -1) return c[i][j];
```

Gdybyśmy pominęli ten warunek (linijka wyżej), algorytm działałby wykładniczo, tak samo jak naiwne cięcie pręta.

⇒ Wniosek: Zainicjowanie macierzy wartościami -1 jest kluczowe, aby odróżnić stan "nieobliczony" od stanu "zysk równy 0".

3 Wybór zajęć

3.1 Zachłanność a Dynamika

Tutaj następuje ciekawe zderzenie podejść. Algorytm zachłanny wymaga jedynie posortowania zajęć po czasie zakończenia f .

```
1 for (int m = n - 1; m >= 1; m--) {
2     if (f[m] <= s[k]) {
3         A.push_back(m);
4         k = m;
5     }
6 }
```

Natomiast podejście dynamiczne (DP) dla tego problemu jest zaskakująco nieefektywne ($O(n^2)$).

⇒ Odkrycie: nie zawsze "cięższy" algorytm (DP) jest lepszy. Sortowanie + jedna pętla wygrywa przy $N > 5000$.

3.2 Warunek sortowania

Warto tu wspomnieć, że sortowanie po czasie startu s nie zadziałałoby tak łatwo w podejściu zachłannym "od przodu".

⇒ Algorytm musi wybierać to, co kończy się najszybciej (lub zaczyna najpóźniej, jeśli idziemy od tyłu), żeby zostawić miejsce dla innych.

⇒ Złożoność determinuje tutaj sortowanie: $O(n \log n)$.

4 Kodowanie Huffmana (Ternarne)

4.1 Odwrócona logika kolejki

Standardowa biblioteka C++ ('priorityqueue') zdejmuje elementy największe. My w Huffmaniu potrzebujemy najmniejszych (najrzadszych znaków).

```
1 struct Compare {
2     bool operator()(Node* l, Node* r) {
3         return l->freq > r->freq;
4     }
5 };
```

Tutaj musiałem zastosować "trik" z odwróceniem znaku nierówności.

⇒ Dzięki temu na wierzchu kolejki (top) lądują elementy o najmniejszej wadze.

4.2 Modyfikacja i parzystość

Standardowy Huffman łączy 2 węzły. Wersja ternarna (3 dzieci) ma pewną zagwostkę matematyczną. Żeby na końcu został dokładnie jeden korzeń, liczba liści musi pasować do redukcji o 2 w każdym kroku.

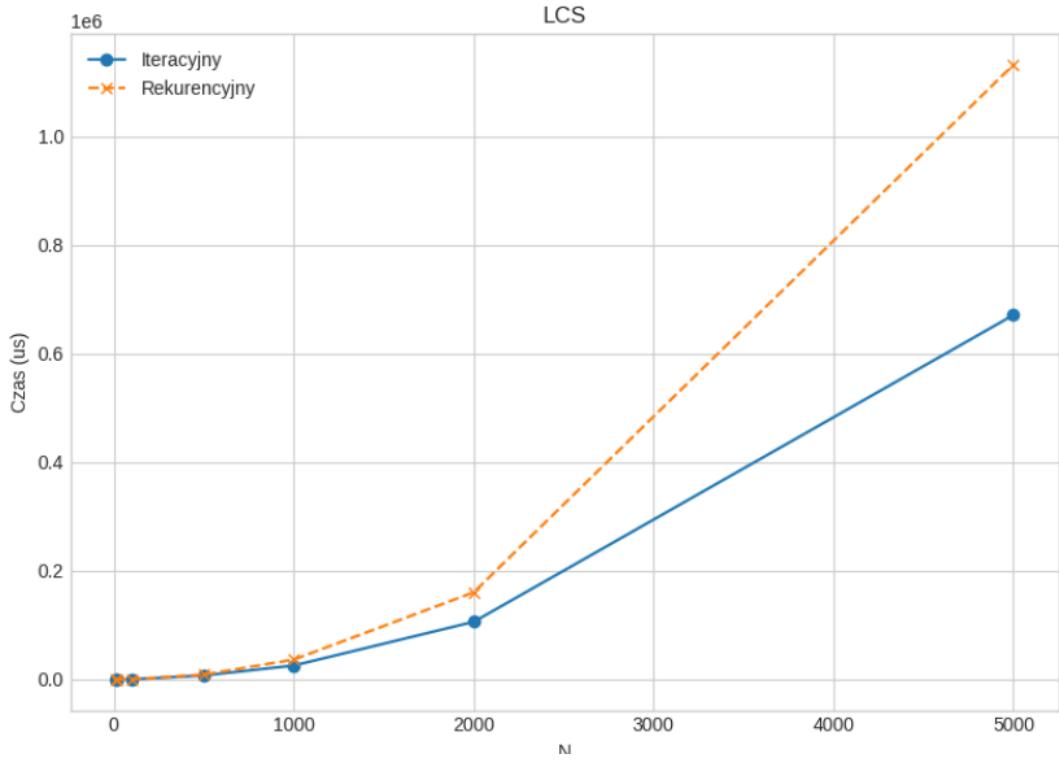
```
1 if (Q.size() % 2 == 0) {  
2     Q.push(new Node('\0', 0));  
3 }
```

W tym kodzie najważniejsze dla mnie było:

⇒ kodowanie alfabetem $\{0, 1, 2\}$ zamiast binarnym, co skraca wysokość drzewa.

⇒ mniejsza wysokość drzewa to średnio krótsze kody

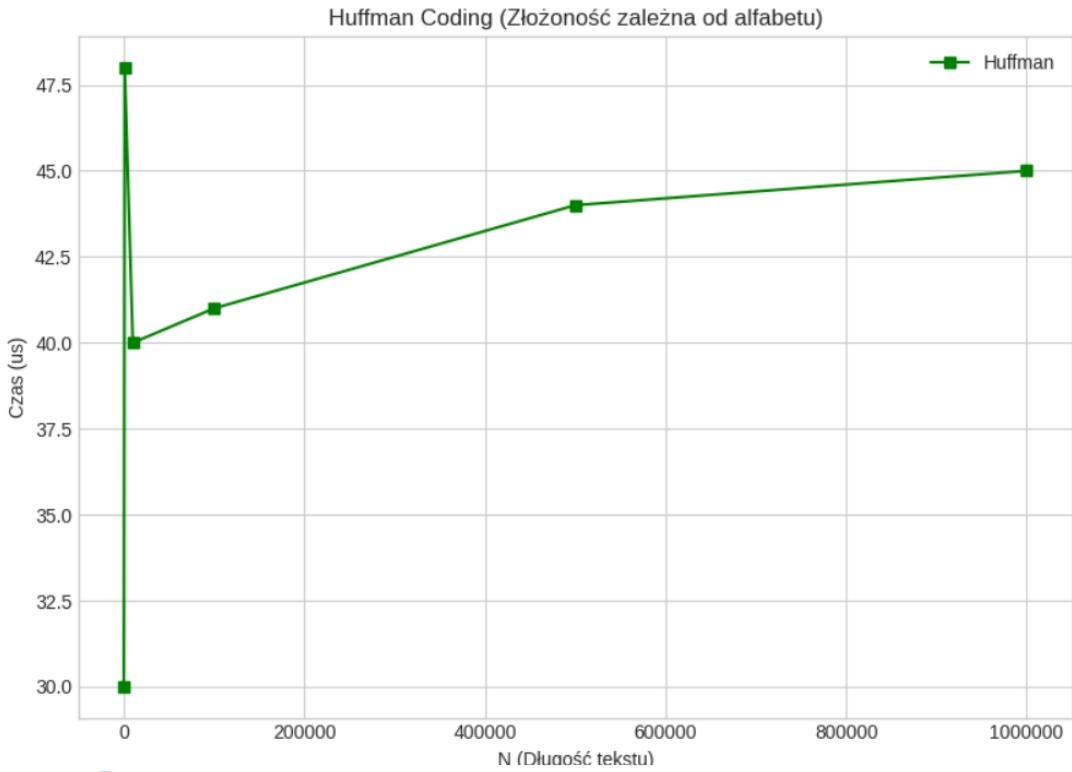
5 Wykres porównawczy - LCS



Patrząc na wykres dla LCS, rzuca się w oczy liniowa zależność obu podejść, ale z wyraźnym rozwarstwieniem.

- ⇒ Obie metody mają tę samą złożoność $O(n \cdot m)$, co widać po kształcie krzywych.
- ⇒ Wersja rekurencyjna (linia przerywana) jest wolniejsza. Iteracja na pustej tablicy zawsze wygrywa wydajnością, mimo że asymptotycznie to to samo.

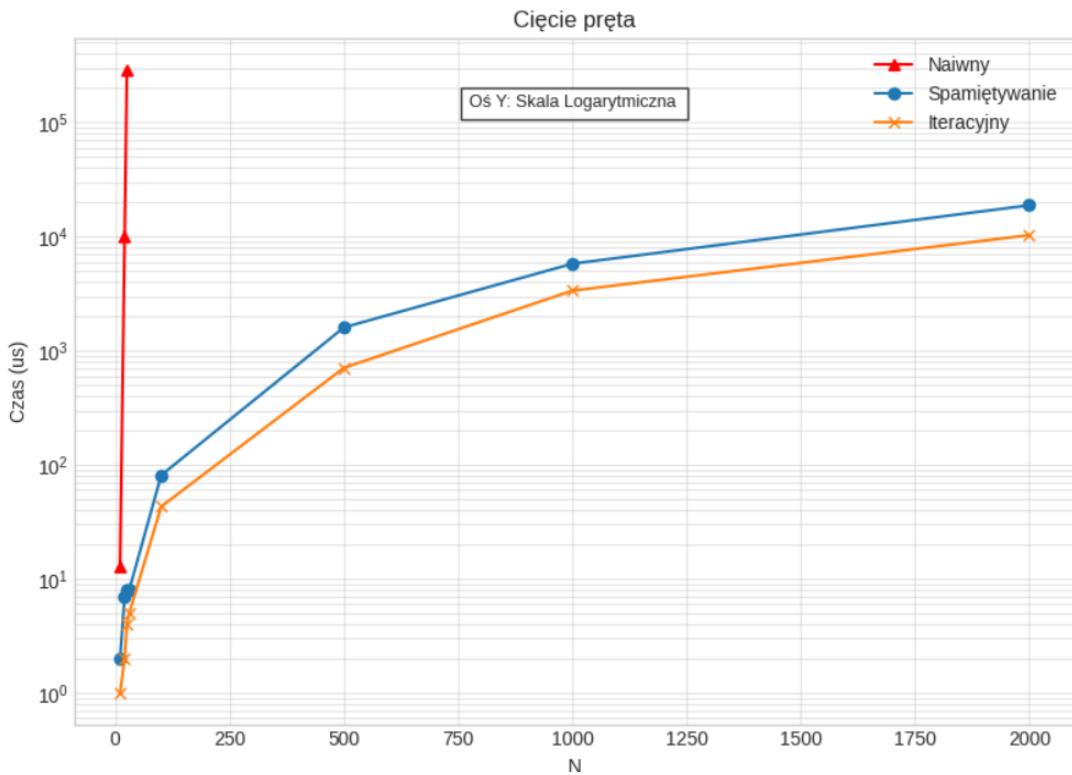
6 Wykres porównawczy - Huffman



Tutaj mamy do czynienia z anomalią w porównaniu do poprzednich algorytmów. Wykres jest niemal płaski, a czasy są w granicach 30-50 mikrosekund nawet dla $N = 1000000$.

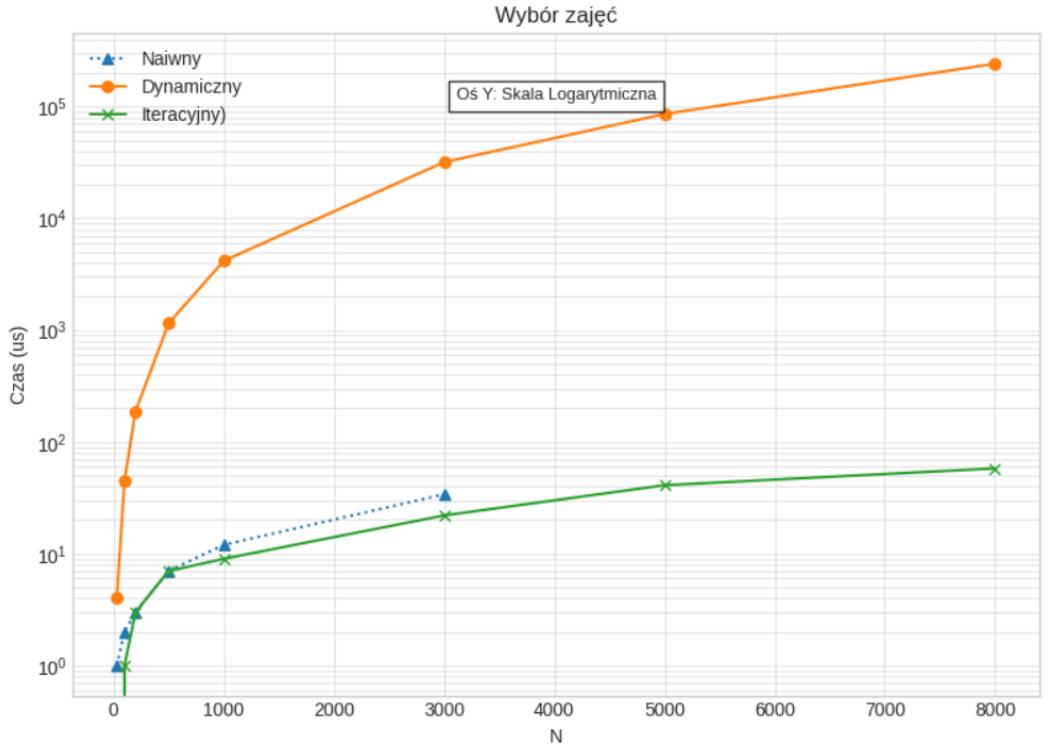
⇒ Potwierdza to to, że złożoność Huffmana zależy od wielkości alfabetu (liczby unikalnych znaków), a nie bezpośrednio od długości tekstu.

7 Wykres porównawczy - Rod Cutting



- Czerwona linia (algorytm naiwny) pnie się pionowo w górę już przy $N = 30$. Sugeruje to złożoność wykładniczą.
- Wersje iteracyjna i ze spamiętywaniem (niebieska i pomarańczowa) wyglądają przy tym niemal jak linie płaskie, mimo że $N = 2000$.
- **Wniosek:** W tym przypadku programowanie dynamiczne nie jest optymalizacją tylko jedynym sposobem, by algorytm w ogóle działał dla $N > 30$.

8 Wykres porównawczy - Activity Selection



Tutaj sytuacja jest odwrotna i bardzo ciekawa.

- Algorytm dynamiczny rośnie znacznie szybciej niż zachłanny. Dla $N = 8000$ różnica jest gigantyczna (skala logarytmiczna na osi Y).
- Podejście iteracyjne jest niemal przyklejone do osi X.
- Wniosek:** Używanie programowania dynamicznego do problemu, który ma strukturę matroidu i rozwiązuje się zachłannie jest błędem.

9 Tabela zbiorcza

zbiorcza tabela czasów									
Problem	Activity Selection			Huffman	LCS		Rod Cutting		
Algorytm	Activity_DP	Activity_Iter	Activity_Naiwny	Huffman_Time	LCS_Iter	LCS_Rekur	CutRod_Iter	CutRod_Naiwny	CutRod_Spm
N									
10	-	-	-	-	12.0	4.0	1.0	13.0	2.0
20	-	-	-	-	25.0	18.0	2.0	10182.0	7.0
25	-	-	-	-	-	-	4.0	289866.0	8.0
30	4.0	0.0	1.0	-	-	-	5.0	-	8.0
100	45.0	1.0	2.0	30.0	351.0	336.0	43.0	-	81.0
200	188.0	3.0	3.0	-	-	-	-	-	-
500	1151.0	7.0	7.0	-	7315.0	8902.0	708.0	-	1602.0
1000	4176.0	9.0	12.0	48.0	25661.0	36635.0	3363.0	-	5790.0
2000	-	-	-	-	106140.0	160513.0	10312.0	-	18819.0
3000	31900.0	22.0	34.0	-	-	-	-	-	-
5000	85908.0	41.0	-	-	671220.0	1132166.0	-	-	-
8000	242606.0	58.0	-	-	-	-	-	-	-
10000	-	-	-	40.0	-	-	-	-	-
100000	-	-	-	41.0	-	-	-	-	-
500000	-	-	-	44.0	-	-	-	-	-
1000000	-	-	-	45.0	-	-	-	-	-

10 Wnioski końcowe

Realizacja tej listy zadań doprowadziła mnie do kilku przemyśleń na temat doboru algorytmów:

1. **Pułapka rekurencji naiwnej** – Najważniejsza lekcja płynie z problemu cięcia pręta. Różnica między $O(2^n)$ a $O(n^2)$ to nie jest kwestią przypadku. Bez spamiętywania niektóre algorytmy są po prostu bezużyteczne.
2. **Nie zawsze dynamiczny jest szybszy** – Jeśli problem ma strukturę pozwalającą na zachłanność, to sortowanie i prosta pętla ($O(n \log n)$) dominują wydajnością podejścia dynamiczne. Widać to wyraźnie w tabeli, gdzie dla $N = 8000$ różnica jest kolosalna (242ms vs 0.05ms).
3. **Koszty** – Przy LCS zauważylem, że nawet przy tej samej złożoności obliczeniowej, implementacja ma znaczenie. Rekurencja ze spamiętywaniem jest wygodniejsza w zapisie, ale iteracja jest lżejsza dla procesora.
4. **Dane** – Przypadek Huffmana pokazał, że musimy patrzeć na to, co determinuje złożoność. Czasem N jest mniej istotne niż K .

Podsumowując, lista ta nauczyła mnie, że nie ma idealnego rozwiązania. Programowanie dynamiczne jest potężne tam, gdzie problemy się zazębają (Rod Cutting, LCS), ale jest balastem tam, gdzie wystarczy zwykła optymalizacja (Activity Selection).