Sprawozdanie AiSD Lista 3

Michał Kasjaniuk, 287392

Styczeń 2026

1 Wprowadzenie

Celem niniejszego sprawozdania jest implementacja, analiza oraz porównanie wydajności algorytmów: `CUT ROD`, `LCS`, `ACTIVITY SELECTOR`, `HUFFMAN`, jak rownież ich modyfikacji opartych na różnych paradygmatach programowania. Zwrócimy szczególną uwagę na programowanie dynamiczne, algorytmy zachłanne oraz podejście rekurencyjne.

2 Opis algorytmów

2.1 CUT ROD

Problem rozcinania pręta polega na znalezieniu takiego sposobu podziału pręta o długości n na mniejsze kawałki, aby zmaksymalizować łączny zysk ze sprzedaży. Zakładamy, że znane są ceny p_i dla każdego kawałka o długości i. Problem ten posiada własność optymalnej podstruktury, co pozwala na zastosowanie wzoru rekurencyjnego:

$$r_n = \max_{1 \le i \le n} (p_i + r_{n-i}) \tag{1}$$

W ramach zadania zaimplementowano trzy podejścia do rozwiązania tego problemu:

1. **Algorytm naiwny (rekurencyjny):** Bezpośrednia implementacja wzoru rekurencyjnego. Metoda ta wielokrotnie rozwiązuje te same podproblemy, co prowadzi do wykładniczej złożoności obliczeniowej $O(2^n)$. Jak wykazały testy, jest to podejście nieefektywne.

```
int CUT_ROD_NAIVE(const vector<int>& p, int n) {
    if (n == 0) return 0;
    int q = INT_MIN;
    for (int i = 1; i <= n; i++) {
        q = max(q, p[i - 1] + CUT_ROD_NAIVE(p, n - i));
    }
    return q;
}
```

2. **Rekurencja ze spamiętywaniem (Memoization):** Podejście, w którym wyniki rozwiązanych podproblemów są zapisywane w tablicy pomocniczej. Przed wykonaniem obliczeń algorytm sprawdza, czy wynik dla danej długości jest już znany, co redukuje złożoność do $O(n^2)$.

3. **Wersja iteracyjna (Bottom-up):** Podejście oparte na programowaniu dynamicznym, polegające na wypełnianiu tablicy wyników od najmniejszych podproblemów do n. Złożoność wynosi również $O(n^2)$, jednak w praktyce jest to metoda najszybsza ze względu na brak narzutu wywołań rekurencyjnych.

```
pair<int, vector<int>> CUT_ROD_BOTTOM_UP(const vector<int>& p, int n) {
vector<int> r(n + 1);
vector<int> s(n + 1);
r[0] = 0;

for (int j = 1; j <= n; j++) {
    int q = INT_MIN;

    for (int i = 1; i <= j; i++) {
```

```
            if (q < p[i - 1] + r[j - i]) {
                q = p[i - 1] + r[j - i];
                s[j] = i;
            }
        }
        r[j] = q;
    }


    vector<int> cuts;
    int temp_n = n;
    while (temp_n > 0) {
        cuts.push_back(s[temp_n]);
        temp_n = temp_n - s[temp_n];
    }

    return {r[n], cuts};
}
```

2.2 Analiza wydajności CUT ROD i modyfikacji

Poniższa tabel i wykres obrazują złożoność obliczeniową omawianych funkcji. Poniższa tabela przedstawia wyniki pomiarów czasu wykonywania algorytmów dla różnych rozmiarów pręta n. Dla algorytmu naiwnego (Naive), przy $n \geq 30$, czas obliczeń stał się zbyt długi, co oznaczono symbolem „-". Potwierdza to jego wykładniczą złożoność $O(2^n)$. Wersje dynamiczne (Memoization i Iterative) działają błyskawicznie nawet dla $n = 3000$.

Rozmiar n	**Naive [s]**	**Memo [s]**	**Iter [s]**
10	0.000005	0.000001	0.000001
20	0.005996	0.000003	0.000002
25	0.166894	0.000006	0.000002
30	-	0.000004	0.000003
100	-	0.000023	0.000013
500	-	0.000505	0.000284
1000	-	0.001887	0.001225
2000	-	0.007539	0.004513
3000	-	0.017479	0.010301

Table 1: Porównanie czasów działania algorytmów Cut Rod

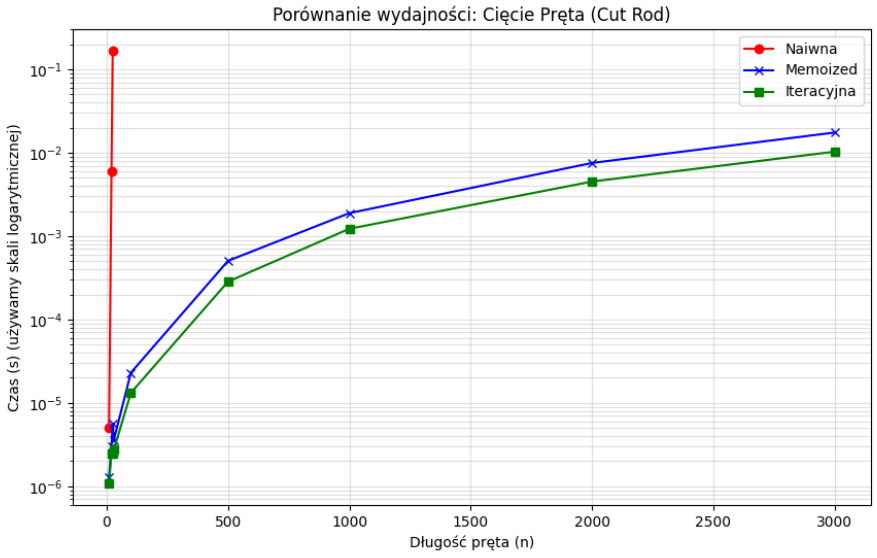

Figure 1: Wykres zależności czasu od rozmiaru danych dla algorytmu Cut Rod w skali logarytmicznej

2.3 Najdłuższy Wspólny Podciąg (LCS)

Problem polega na znalezieniu najdłuższego ciągu znaków, który jest podciągiem dwóch danych ciągów X i Y. W przeciwieństwie do podłańcucha, elementy podciągu nie muszą ze sobą sąsiadować, ale muszą zachować kolejność. Niech $c[i, j]$ oznacza długość LCS dla prefiksów X o długości i oraz Y o długości j. Problem ten posiada własność optymalnej podstruktury, a wzór rekurencyjny wygląda następująco:

$$c[i,j] = \begin{cases} 0 & \text{gdy } i = 0 \text{ lub } j = 0 \\ c[i-1, j-1] + 1 & \text{gdy } x_i = y_j \\ \max(c[i-1, j], c[i, j-1]) & \text{gdy } x_i \neq y_j \end{cases} \tag{2}$$

W ramach laboratorium zaimplementowano i przeanalizowano dwa warianty algorytmu wyznaczania Najdłuższego Wspólnego Podciągu, różniące się sposobem sterowania obliczeniami oraz zarządzaniem pamięcią:
W ramach laboratorium zaimplementowano dwa warianty algorytmu wyznaczania LCS:

1. **Wersja rekurencyjna ze spamiętywaniem (Memoization):**
   Jest to podejście typu „top-down". Aby uniknąć wykładniczej złożoności $O(2^n)$ wynikającej z wielokrotnego rozwiązywania tych samych podproblemów, algorytm zapamiętuje obliczone wyniki w tablicy pomocniczej. Przed każdym wywołaniem rekurencyjnym sprawdzane jest, czy wynik dla danej pary indeksów jest już znany. Redukuje to czas działania do $O(m \cdot n)$.

2. **Wersja iteracyjna (Bottom-Up):**
   Podejście klasyczne dla programowania dynamicznego, polegające na wypełnianiu tablicy wyników od przypadków bazowych do rozwiązania końcowego za pomocą pętli. Metoda ta charakteryzuje się tą samą złożonością $O(m \cdot n)$, jednak w praktyce jest szybsza i bardziej stabilna, ponieważ eliminuje narzut pamięciowy związany ze stosem wywołań rekurencyjnych.

```
vector<vector<int>> LCS_ITERATIVE(const string& X, const string& Y) {
int m = X.length();
int n = Y.length();

vector<vector<int>> c(m + 1, vector<int>(n + 1, 0));

for (int i = 1; i <= m; i++) {
    for (int j = 1; j <= n; j++) {
        if (X[i - 1] == Y[j - 1]) {
            c[i][j] = c[i - 1][j - 1] + 1;
        } else {
            c[i][j] = max(c[i - 1][j], c[i][j - 1]);
        }
    }
}
return c;
}
```

2.4 Analiza wydajności LCS i modyfikacji

Przeprowadzono testy porównawcze dla obu zaimplementowanych wersji algorytmu LCS (Iteracyjnej oraz Rekurencyjnej ze spamiętywaniem). Wyniki przedstawiono w Tabeli 2.

Oba algorytmy wykazują kwadratową złożoność obliczeniową $O(n^2)$, co potwierdza fakt, że przy dwukrotnym zwiększeniu rozmiaru danych (np. z 2000 na 4000), czas wykonania rośnie około czterokrotnie (z $\approx 0.08s$ na $\approx 0.34s$). Wersja iteracyjna okazuje się nieznacznie wydajniejsza dla bardzo dużych danych, eliminując narzut związany z wywołaniami rekurencyjnymi.

Rozmiar n	Iteracja [s]	Rekurencja [s]
10	0.000008	0.000005
50	0.000076	0.000066
100	0.000353	0.000256
500	0.005015	0.006110
1000	0.020921	0.022784
2000	0.084429	0.090328
3000	0.181102	0.206461
4000	0.341274	0.347385
5000	0.507833	0.533345

Table 2: Porównanie czasów działania algorytmów LCS

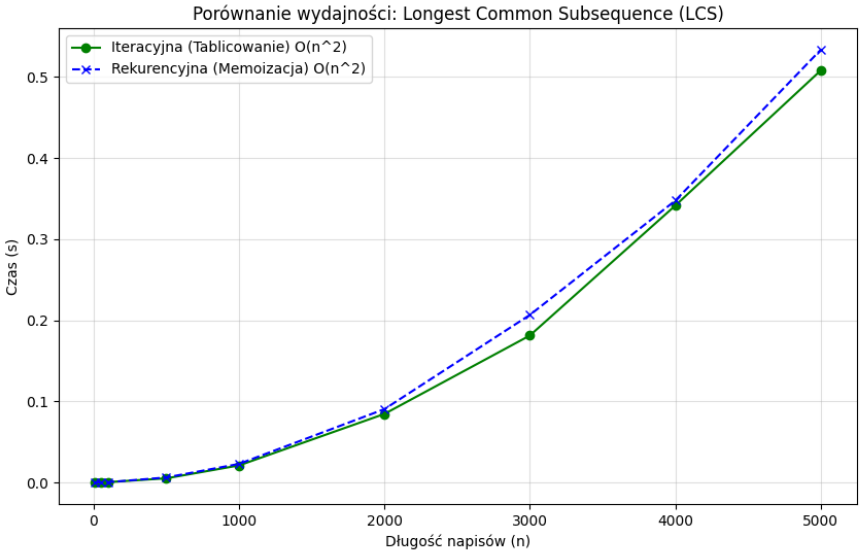

Figure 2: Zależność czasu wykonywania od długości ciągów dla LCS

2.5 Problem Wyboru Zajęć (Activity Selector)

Zadanie polega na wyborze maksymalnego liczebnie zbioru niekolidujących zajęć. Zgodnie z treścią zadania, algorytmy zostały przystosowane do pracy na danych wejściowych posortowanych niemalejąco względem czasu rozpoczęcia ($s_1 \leq s_2 \leq \cdots \leq s_n$).

Wymusiło to zmianę standardowej strategii zachłannej. Zamiast wybierać zajęcia kończące się najwcześniej (idąc od lewej), zastosowano podejście symetryczne: wybierane jest zajęcie, które zaczyna się najpóźniej spośród dostępnych (idąc od końca posortowanej tablicy). Gwarantuje to pozostawienie jak największej ilości czasu dla wcześniejszych zadań.

Zaimplementowano trzy warianty rozwiązania:

1. **Algorytm Zachłanny (Iteracyjny):** Przetwarza tablicę od ostatniego elementu $(n-1)$ do pierwszego (0). Dodaje do wyniku zajęcie, jeśli kończy się ono przed czasem rozpoczęcia ostatnio wybranego zadania. Złożoność: $O(n \log n)$ (koszt sortowania).

2. **Algorytm Zachłanny (Rekurencyjny):** Realizuje tę samą logikę co wersja iteracyjna, wykorzystując funkcję pomocniczą do przeszukiwania tablicy w tył w poszukiwaniu pierwszego zgodnego zajęcia.

3. **Programowanie Dynamiczne (DP):** Jest to rozwiązanie "konkurencyjne", niewykorzystujące własności zachłannej. Algorytm działa analogicznie do problemu najdłuższego rosnącego podciągu (LIS). Dla każdego zajęcia i sprawdzamy wszystkie wcześniejsze zajęcia $j < i$ i jeśli są zgodne, przedłużamy optymalny łańcuch. Metoda ta ma złożoność $O(n^2)$, co w testach okazało się znacznie wolniejsze od podejścia zachłannego.

```
Algorytm Zachlanny (Iteracyjny)
vector<int> ACTIVITY_SELECTOR_ITERATIVE(vector<Activity> activities) {
```

```
    if (activities.empty()) return {};
    sort(activities.begin(), activities.end(), compareByStart);
    vector<int> result;
    int n = activities.size();

    Activity lastSelected = activities[n - 1];
    result.push_back(lastSelected.id);

    for (int i = n - 2; i >= 0; i--) {
        // Jesli zajecie konczy sie przed startem wybranego -> pasuje
        if (activities[i].finish <= lastSelected.start) {
            result.push_back(activities[i].id);
            lastSelected = activities[i];
        }
    }
    return result;
}
```

2.6 Analiza wydajności problemu wyboru zajęć (Activity Selector)

Porównano wydajność algorytmu zachłannego (w wersji iteracyjnej i rekurencyjnej) z podejściem programowania dynamicznego. Wyniki zestawiono w Tabeli 3.

Zgodnie z przewidywaniami teoretycznymi, algorytm zachłanny okazuje się drastycznie szybszy od dynamicznego. Dla $n = 3000$ wersja zachłanna wykonuje się w ułamku milisekundy ($\approx 0.0002s$), podczas gdy wersja dynamiczna potrzebuje na to ponad 100 razy więcej czasu ($\approx 0.024s$).

Analiza danych dla podejścia dynamicznego (DP) wskazuje na złożoność kwadratową $O(n^2)$. Trzykrotny wzrost danych (z 1000 na 3000) powoduje w przybliżeniu dziewięciokrotny wzrost czasu obliczeń ($0.0025s \rightarrow 0.024s$), co jest zgodne z zależnością $3^2 = 9$. Algorytmy zachłanne wykazują zależność liniową $O(n)$.

Table 3: Porównanie czasów dla Activity Selector Problem

Rozmiar n	**Iteracja [s]**	**Rekurencja [s]**	**PD [s]**
10	0.000002	0.000001	0.000002
100	0.000008	0.000006	0.000032
500	0.000041	0.000030	0.000637
1000	0.000083	0.000064	0.002576
2000	0.000177	0.000143	0.010528
3000	0.000280	0.000242	0.024250

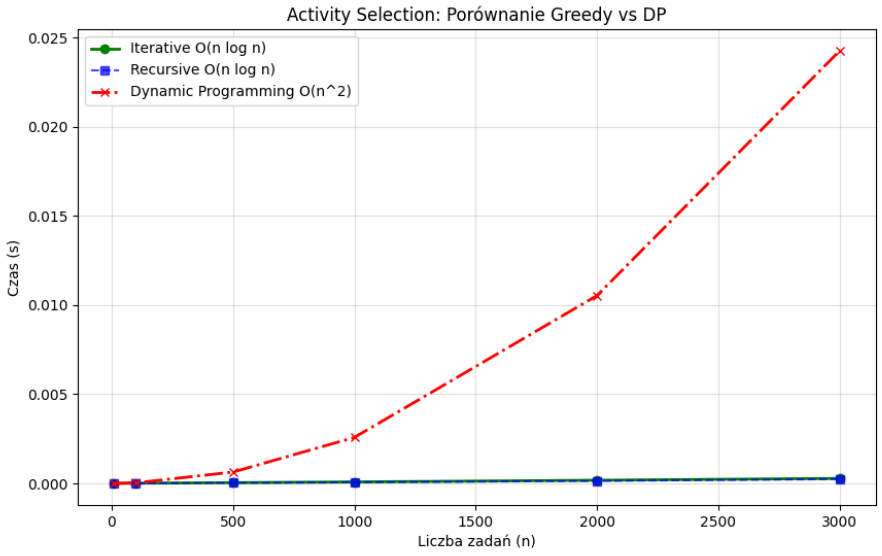

Figure 3: Porównanie wydajności algorytmów dla problemu wyboru zajęć

2.7 Algorytm Huffmana (Wariant Ternarny)

Ostatnim zaimplementowanym algorytmem jest algorytm Huffmana w zmodyfikowanej wersji ternarnej (trójkowej). Standardowy algorytm Huffmana buduje drzewo binarne, przypisując znakom kody złożone z bitów 0 i 1. Wersja ternarna buduje drzewo, w którym każdy węzeł może mieć do trzech potomków, a kody wynikowe składają się z symboli $\{0, 1, 2\}$.

Problem polega na skonstruowaniu optymalnego kodu prefiksowego dla zadanego zbioru znaków i ich częstości występowania. Algorytm zachłanny buduje drzewo od dołu do góry:

1. Umieść wszystkie węzły (znaki) w kolejce priorytetowej posortowanej rosnąco według częstości.
2. Sprawdź parzystość liczby węzłów. Ponieważ w każdym kroku łączymy 3 węzły w 1 (redukcja o 2), algorytm wymaga nieparzystej liczby węzłów startowych, aby zakończyć się na jednym korzeniu. Jeśli liczba węzłów jest parzysta, dodawany jest sztuczny węzeł (dummy node) o częstości 0.
3. Dopóki w kolejce jest więcej niż 1 węzeł:
   - Pobierz trzy węzły o najmniejszych częstościach (x, y, w).
   - Utwórz nowy węzeł nadrzędny z o częstości równej sumie $x + y + w$.
   - Ustaw x, y, w jako dzieci węzła z (odpowiednio lewe, środkowe, prawe).
   - Wstaw z z powrotem do kolejki.

Złożoność obliczeniowa tego algorytmu wynosi $O(n \log n)$ ze względu na operacje na kolejce priorytetowej (kopcu), gdzie n to liczba unikalnych znaków.

```
struct Node {
    char ch;
    int freq;
    Node *left, *middle, *right;
    Node(char c, int f) : ch(c), freq(f),
         left(nullptr), middle(nullptr), right(nullptr) {}
};


Node* HUFFMAN(const map<char, int>& C) {
    int n = C.size();

    priority_queue<Node*, vector<Node*>, Compare> Q;

    for (auto const& [key, val] : C) {
        Q.push(new Node(key, val));
    }

    if (Q.size() % 2 == 0) {
        Q.push(new Node('\0', 0));
        n++;
    }

    while (Q.size() > 1) {
        Node* z = new Node('\0', 0);
        Node* x = Q.top(); Q.pop();
        z->left = x;
        Node* y = Q.top(); Q.pop();
        z->middle = y;
        Node* w = Q.top(); Q.pop();
        z->right = w;
        z->freq = x->freq + y->freq + w->freq;
        Q.push(z);

    }
    return Q.top();
}
```

3 Wnioski

Analiza zaimplementowanych algorytmów pozwala na następujące spostrzeżenia:

- **Programowanie dynamiczne to konieczność przy trudnych problemach.** W zadaniu rozcinania pręta (Cut Rod) zwykła rekurencja działała wykładniczo ($O(2^n)$) i zawieszała się już przy 30 elementach. Zastosowanie tablicy (podejście dynamiczne) przyspieszyło algorytm do ułamków sekundy.
- **Algorytm zachłanny jest najszybszy.** W problemie wyboru zajęć (Activity Selector) podejście zachłanne ($O(n \log n)$) okazało się bezkonkurencyjne – było ponad 100 razy szybsze niż programowanie dynamiczne ($O(n^2)$). Warto go używać, gdy tylko struktura problemu na to pozwala.
- **Iteracja wygrywa z rekurencją.** Przy algorytmie LCS wersja na pętlach (iteracyjna) była nieco szybsza od rekurencji ze spamiętywaniem. Pozwala ona uniknąć narzutu pamięciowego związanego z wywoływaniem funkcji.
- **Huffman Ternarny wymaga korekty danych.** Implementacja kodowania trójkowego (znaki 0, 1, 2) wymaga nieparzystej liczby węzłów na starcie. Aby algorytm działał poprawnie dla dowolnych danych, konieczne było dodanie „sztucznego węzła” w przypadku parzystej liczby elementów.