Algorytmy i Struktury Danych
Sprawozdanie 3

Maja Kołakowska

287360

Spis treści

1 Wstęp

W tej pracy rozpatrzymy różne wersje następujących algorytmów:

- rozcinanie pręta (cut rod) – w wersji naiwnej, ze spamiętywaniem oraz iteracyjnej,
- najdłuższy wspólny podciąg (LCS) – w wersji rekurencyjnej ze spamiętywaniem oraz w wersji iteracyjnej,
- wybór zajęć (activity selector) – w wersji rekurencyjnej, iteracyjnej, zmodyfikowanej, aby działał na danych posortowanych względem czasu rozpoczęcia, oraz opartej na programowaniu dynamicznym,
- kody Huffmana – w wersji podstawowej oraz zmodyfikowanej, aby kodować ternarnie.

2 Opis metody badawczej

Celem pracy jest zbadanie i porównanie czasu wykonywania poszczególnych algorytmów dla ich różnych wersji. Ze względu na różnorodność algorytmów wielkość danych testowych oraz liczba wykonywanych testów będzie się różnić.

2.1 Cut Rod

Dla wersji naiwnej zostanie wykonane po 100 testów dla losowych cen i długości: 5, 8, 10, 12, 14, 16, 18, 20, 22, 24, a następnie obliczymy średni czas. Natomiast dla pozostałych wersji skupimy się na większych danych: 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 1100, 1200, 1300, 1400, 1500, 1600, 1700, 1800, 1900, 2000.

2.2 LCS

Będziemy wykonywać po 5 testów dla następujących długości ciągów: 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 1100, 1200, 1300, 1400, 1500, 1600, 1700, 1800, 1900, 2000, po czym obliczymy średni czas.

2.3 Activity selector

Będziemy wykonywać po 100 testów dla następujących długości ciągów: 5000, 10000, 15000, 20000, 25000, 30000, 35000, 40000, 45000, 50000, 55000,

60000, 65000, 70000, 75000, 80000, 85000, 90000, 95000, 100000, po czym obliczymy średni czas.

2.4 Kod Huffmana

Dla tych algorytmów nie zostały przewidziane testy.

3 Cut Rod

Problem rozcinania pręta polega na znalezieniu maksymalnego zysku, jaki można uzyskać poprzez podział pręta o długości n na mniejsze części, przy założeniu, że znane są ceny poszczególnych długości.

3.1 Naiwny Cut Rod

Naiwne rozwiązanie opiera się na rekursji i sprawdza wszystkie możliwe sposoby podziału pręta. Dla każdej długości i rozważany jest podział na część długości oraz pozostałą część; $n - i$. Algorytm ten cechuje się bardzo dużą złożonością czasową rzędu $O(2^n)$, ponieważ wielokrotnie oblicza te same podproblemy.

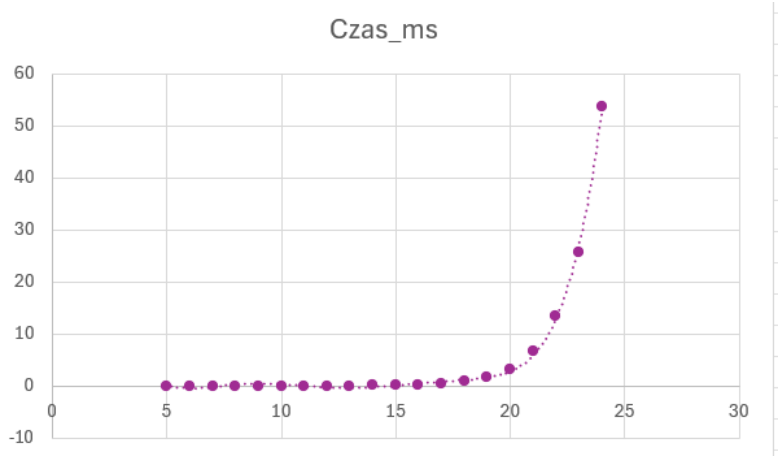
```
1 int naive_cut_rod (int p[], int n) {
2     if (n == 0) return 0;
3     int q = INT_MIN;
4     for (int i = 1; i <= n; i++) {
5       q = max(q, p[i-1]+naive_cut_rod(p,n-i));
6         }
7     return q;
8     }
```

3.2 Pomiary

Dla każdego z dwudziestu podanych uprzednio długości wykonano serię stu testów z pomiarem czasu. Wyniki tych pomiarów prezentuje Tabela 1.

Długość	Czas [ms]
5	0
6	0
7	0
8	0
9	0
10	0,01
11	0
12	0,02
13	0,02
14	0,06
15	0,1
16	0,2
17	0,39
18	0,81
19	1,55
20	3,1
21	6,56
22	13,32
23	25,61
24	53,62

Tabela 1: Czas wersji naiwnej



Rysunek 1: Wykres czasu wersji naiwnej

Funkcja została aproksymowana funkcją wykładniczą. Jak można zauważyć na podstawie wykresu, algorytm ma złożoność czasową $O(2^n)$, co jest

zgodne z założeniami.

3.3 Cut Rod ze spamientywaniem

Wersja ze spamiętywaniem eliminuje problem wielokrotnego liczenia tych samych podproblemów poprzez zapisywanie wcześniej obliczonych wyników w tablicy. Dzięki temu każdy podproblem rozwiązywany jest tylko raz, co redukuje złożoność czasową do $O(n^2)$.

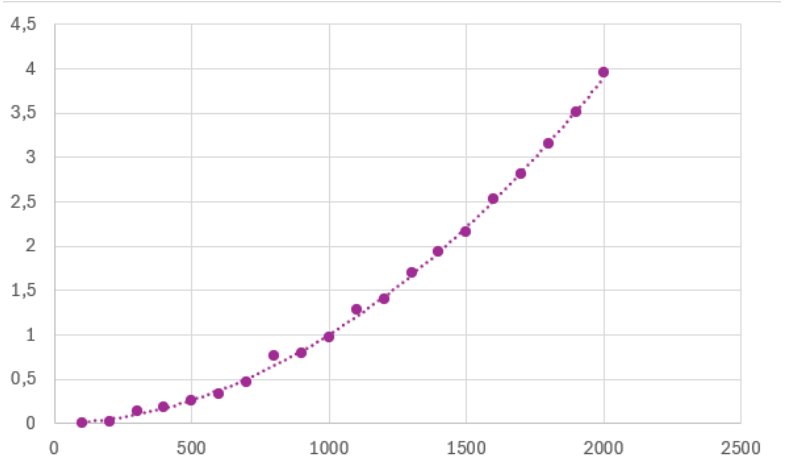
```
1  int memorized_cut_rod(int p[], int n, int r[], int s[]) {
2      if (r[n] >= 0) return r[n];
3
4      if (n == 0) {
5          r[0] = 0;
6          s[0] = 0;
7          return 0;
8      } else {
9          int q = INT_MIN;
10         for (int i = 1; i <= n; i++) {
11             int current = p[i-1] + memorized_cut_rod(p, n-i, r, s);
12             if (q < current) {
13                 q = current;
14                 s[n] = i;
15             }
16         }
17         r[n] = q;
18         return q;
19     }
20 }
```

3.4 Pomiary

Dla każdego z dwudziestu podanych uprzednio długości wykonano serię stu testów z pomiarem czasu. Wyniki tych pomiarów prezentuje Tabela 2.

Długość	Czas[ms]
100	0,01
200	0,02
300	0,14
400	0,19
500	0,26
600	0,34
700	0,47
800	0,77
900	0,8
1000	0,97
1100	1,29
1200	1,4
1300	1,7
1400	1,93
1500	2,16
1600	2,53
1700	2,81
1800	3,16
1900	3,51
2000	3,96

Tabela 2: Czas wersji ze spamietywaniem



Rysunek 2: Wykres czasu wersji ze spamientywaniem

Funkcja została aproksymowana funkcją kwadratową. Jak można zauważyć, algorytm ma złożoność czasową $O(n^2)$, co jest zgodne z założeniami.

3.5 Cut Rod Iteracyjny

Iteracyjne rozwiązanie problemu cut rod wykorzystuje podejście oddolne (bottom-up). Wyniki dla mniejszych długości pręta są obliczane najpierw i przechowywane w tablicy, a następnie wykorzystywane do obliczenia rozwiązania dla większych długości. Złożoność czasowa tej wersji również wynosi

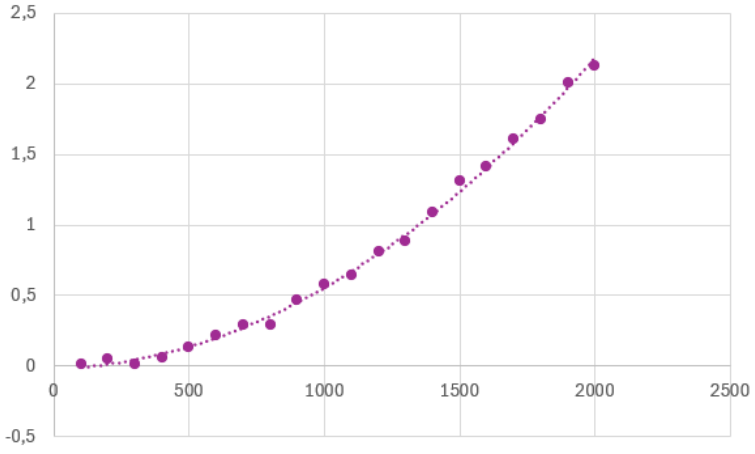
```
int ext_cut_rod(const int p[], int n, int r[], int s[]) {
    r[0] = 0;
    s[0] = 0;

    for (int j = 1; j <= n; j++) {
        int q = INT_MIN;
        s[j] = 0;

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

        r[j] = q;
    }
    return r[n];
}
```

3.6 Pomiary

Dla każdego z dwudziestu podanych uprzednio długości wykonano serię stu testów z pomiarem czasu. Wyniki tych pomiarów prezentuje Tabela 3.

Długość	Czas[ms]
100	0,01
200	0,05
300	0,01
400	0,06
500	0,13
600	0,22
700	0,29
800	0,29
900	0,47
1000	0,58
1100	0,64
1200	0,81
1300	0,88
1400	1,09
1500	1,31
1600	1,41
1700	1,61
1800	1,75
1900	2,01
2000	2,13

Tabela 3: Czas wersji iteracyjnej



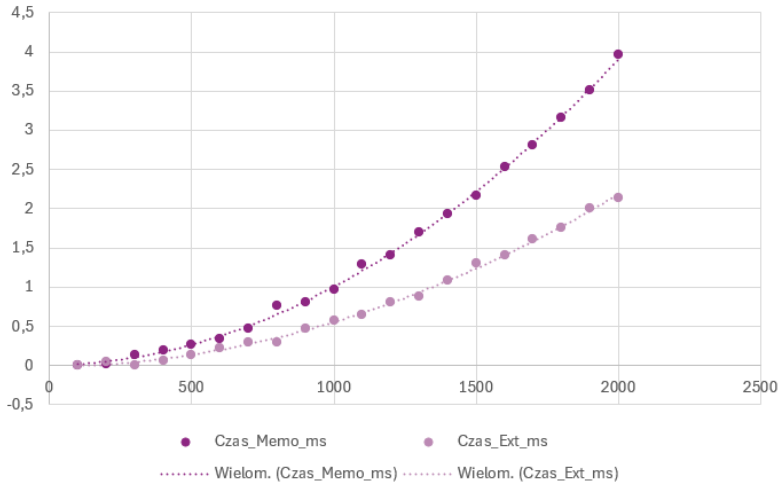
Rysunek 3: Wykres czasu wersji iteracyjny

Funkcja została aproksymowana funkcją kwadratową. Jak można zauważyć na podstawie wykresu, algorytm ma złożoność czasową $O(n^2)$, co jest

zgodne z założeniami.

3.7 Wnioski

Wersja naiwana nie jest porównywana, ponieważ już dla swojego zestawu danych osiąga podobne wyniki czasowe; można jednoznacznie stwierdzić, że jest najmniej efektywnym algorytmem. Na Rysunku 4 został przedstawiony zbiórczy wykres czasu dla wersji z spamiętywaniem oraz iteracyjnej. Na jego podstawie można stwierdzić, że algorytm iteracyjny jest szybszy.



Rysunek 4: Porównanie

4 LSC

Problem najdłuższego wspólnego podciągu polega na znalezieniu najdłuższej sekwencji znaków, która występuje w dwóch ciągach w tej samej kolejności, choć niekoniecznie w sposób ciągły.

4.1 LSC rekurencyjny

Ta wersja algorytmu LCS opiera się na rekurencyjnym porównywaniu końcowych podciągów, z zapamiętywaniem już obliczonych wyników. Algorytm najpierw sprawdza, czy rozwiązanie dla danej pary już istnieje, jeśli nie, porównuje ostatnie znaki ciągów i działa rekurencyjnie dla odpowiednio skróconych ciągów. Złożoność czasowa algorytmu wynosi $O(mn)$, gdzie gdzie m i n to długości obu ciągów.

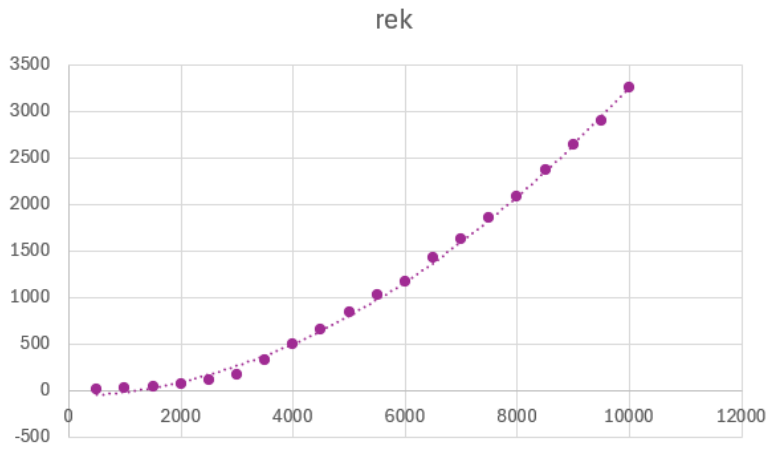
```
int LCS_rek(int i, int j) {
    if (i < 0 || j < 0) return 0;

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

    if (X[i] == Y[j]) {
        c[i][j] = LCS_rek(i-1, j-1) + 1;
        b[i][j] = 'd';
    }
    else {
        int gora = LCS_rek(i-1, j);
        int lewo = LCS_rek(i, j-1);

        if (gora >= lewo) {
            c[i][j] = gora;
            b[i][j] = 'u';
        }
        else {
            c[i][j] = lewo;
            b[i][j] = 'l';
        }
    }

    return c[i][j];
}
```

4.2 Pomiary

Dla każdego z dwudziestu podanych uprzednio długości wykonano serię stu testów z pomiarem czasu. Wyniki tych pomiarów prezentuje Tabela 4.

Długość	Czas rek [ms]
500	5,4
1000	18
1500	41,2
2000	72
2500	113,2
3000	162,6
3500	320,4
4000	491,6
4500	658
5000	833,6
5500	1029,6
6000	1166
6500	1418,6
7000	1618,4
7500	1848,2
8000	2084
8500	2372,6
9000	2642,4
9500	2892,6
10000	3252

Tabela 4: Czas wersji rekurencyjnej



Rysunek 5: Wykres czasu wersji rekurencyjnej

Funkcja została aproksymowana funkcją kwadratową. Jak można zauważyć na podstawie wykresu, algorytm ma złożoność czasową $O(n^2)$, co jest

zgodne z założeniami, dla dwóch ciągów podobnej długości.

4.3 LSC iteracyjny

Iteracyjna wersja algorytmu LCS wypełnia tablicę coraz dłuższymi podciągami, zaczynając od tych najprostrzych. Alogrytm najpierw bierze jednoelementowy podciąg i porównuje go z każdym podciągiem drugiego ciągu, zapisując w tablicy długości LCS. Złożoność czasowa również wynosi $O(mn)$, gdzie gdzie m i n to długości obu ciągów.

```
void LCS_iteracyjny(string X, string Y) {
    int m = X.size();
    int n = Y.size();

    vector<vector<int>> c(m, vector<int>(n, 0));
    vector<vector<char>> b(m, vector<char>(n, ' '));

    for (int j = 0; j < n; j++) {
        if (X[0] == Y[j]) {
            c[0][j] = 1;
            b[0][j] = 'd';
        } else if (j > 0) {
            c[0][j] = c[0][j-1];
            b[0][j] = 'l';
        }
    }

    for (int i = 0; i < m; i++) {
        if (X[i] == Y[0]) {
            c[i][0] = 1;
            b[i][0] = 'd';
        } else if (i > 0) {
            c[i][0] = c[i-1][0];
            b[i][0] = 'u';
        }
    }

    for (int i = 1; i < m; i++) {
        for (int j = 1; j < n; j++) {
            if (X[i] == Y[j]) {
                c[i][j] = c[i-1][j-1] + 1;
                b[i][j] = 'd';
            }
```

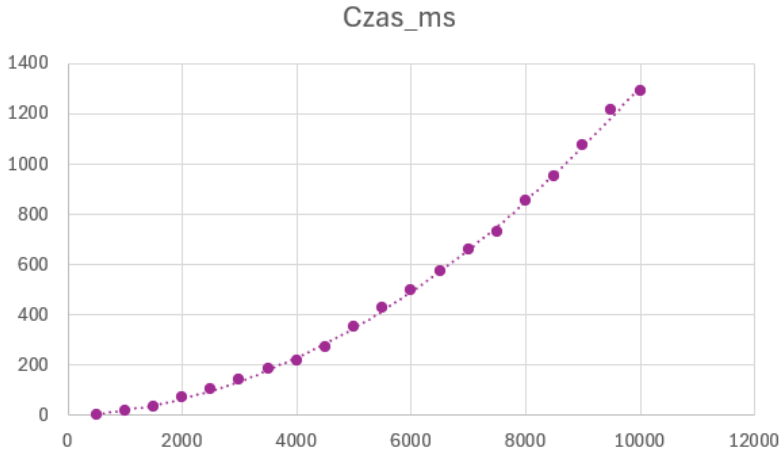
```
34              else if (c[i-1][j] >= c[i][j-1]) {
35                  c[i][j] = c[i-1][j];
36                  b[i][j] = 'u';
37              }
38              else {
39                  c[i][j] = c[i][j-1];
40                  b[i][j] = 'l';
41              }
42          }
43      }
44  }
```

4.4 Pomiary

Dla każdego z dwudziestu podanych uprzednio długości wykonano serię stu testów z pomiarem czasu. Wyniki tych pomiarów prezentuje Tabela 5.

Długość	**Czas [ms]**
500	5
1000	19
1500	36,2
2000	71
2500	106,2
3000	142,6
3500	184,6
4000	217,8
4500	273,8
5000	352,6
5500	426,4
6000	496,8
6500	575,2
7000	658
7500	728,8
8000	856,8
8500	951,4
9000	1075,8
9500	1216,2
10000	1293

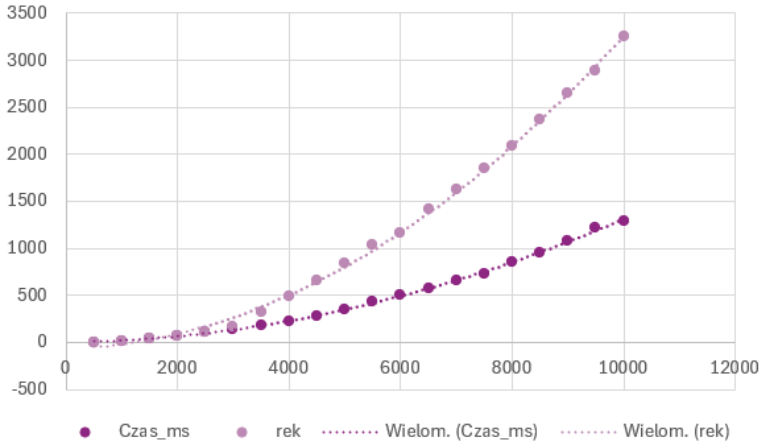
Tabela 5: Czas wersji iteracyjnej



Rysunek 6: Wykres czasu wersji iteracyjnej

Funkcja została aproksymowana funkcją kwadratową. Jak można zauważyć na podstawie wykresu, algorytm ma złożoność czasową $O(n^2)$, co jest zgodne z założeniami, dla dwóch ciągów podobnej długości.

4.5 Wnioski



Rysunek 7: Porównanie

Na Rysunku 7 został przedstawiony zbiórczy wykres czasu dla wersji rekurencyjnej oraz iteracyjnej. Na jego podstawie można stwierdzić, że algorytm iteracyjny jest szybszy.

5 Activity Selector

Problem wyboru zajęć polega na wybraniu maksymalnej liczby niekolidujących ze sobą aktywności, z których każda opisana jest czasem rozpoczęcia oraz zakończenia

5.1 Activity Selector rekurencyjny

Wersja rekurencyjna algorytmu wybiera kolejne aktywności poprzez wywołania funkcji rekurencyjnej. Dla każdej aktywności sprawdza, czy nie koliduje z ostatnio wybraną, a następnie wywołuje funkcję dla pozostałych aktywności. Po przejściu wszystkich elementów uzyskuje się maksymalny zbiór niekolidujących zajęć. Złożoność czasowa pozostaje liniowa $O(n)$ w przypadku danych posortowanych według czasu zakończenia.

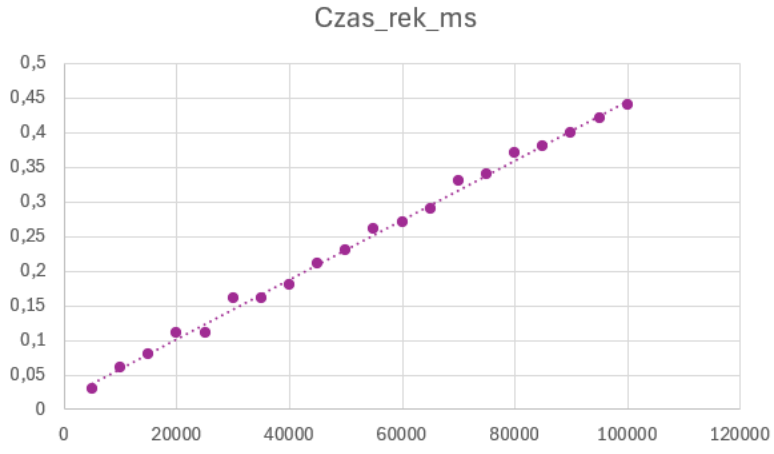
```
int RECURSIVE_ACTIVITY_SELECTOR(
    const int s[], const int f[],
    int n, int k, int result[]
) {
    int m = k + 1;

    while (m < n && k != -1 && s[m] < f[k])
        m++;

    if (m < n) {
        result[0] = m;
        return 1 + RECURSIVE_ACTIVITY_SELECTOR(s, f, n, m, result + 1);
    }
    return 0;
}
```

5.2 Pomiary

Dla każdego z dwudziestu podanych uprzednio długości wykonano serię stu testów z pomiarem czasu. Wyniki tych pomiarów prezentuje Tabela 6.

Długość	Czas rek [ms]
5000	0,03
10000	0,06
15000	0,08
20000	0,11
25000	0,11
30000	0,16
35000	0,16
40000	0,18
45000	0,21
50000	0,23
55000	0,26
60000	0,27
65000	0,29
70000	0,33
75000	0,34
80000	0,37
85000	0,38
90000	0,40
95000	0,42
100000	0,44

Tabela 6: Czas wersji rekurencyjnej



Rysunek 8: Wykres wersji rekurencyjnej

Funkcja została aproksymowana funkcją liniową. Jak można zauważyć na

podstawie wykresu, algorytm ma złożoność czasową $O(n)$, co jest zgodne z założeniami.

5.3 Activity Selector iteracyjny

Algorytm iteracyjny działa podobnie do wersji rekurencyjnej. Najpierw znajduje zajęcie kończące się najwcześniej, a następnie przegląda pozostałe aktywności w kolejności rosnącego czasu zakończenia, dodając każde, które nie koliduje z już wybranymi. Po rozważeniu wszystkich zajęć otrzymuje się maksymalny zbiór niekolidujących aktywności. Złożoność czasowa wynosi $O(n)$, jeśli mamy posortowone czasy zakończenia.

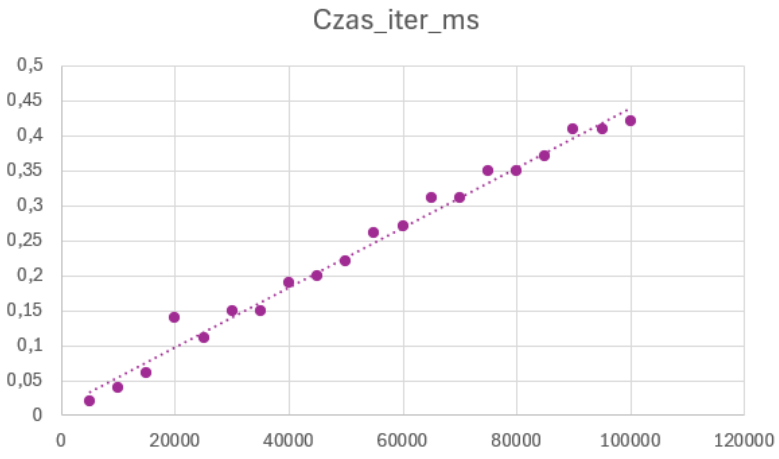
```
int ACTIVITY_SELECTOR(
    const int s[], const int f[],
    int n, int result[]
) {
    if (n == 0) return 0;

    result[0] = 0;
    int count = 1;
    int k = 0;

    for (int m = 1; m < n; m++) {
        if (s[m] >= f[k]) {
            result[count++] = m;
            k = m;
        }
    }
    return count;
}
```

5.4 Pomiary

Dla każdego z dwudziestu podanych uprzednio długości wykonano serię stu testów z pomiarem czasu. Wyniki tych pomiarów prezentuje Tabela 7.

Długość	Czas iter [ms]
5000	0,02
10000	0,04
15000	0,06
20000	0,14
25000	0,11
30000	0,15
35000	0,15
40000	0,19
45000	0,20
50000	0,22
55000	0,26
60000	0,27
65000	0,31
70000	0,31
75000	0,35
80000	0,35
85000	0,37
90000	0,41
95000	0,41
100000	0,42

Tabela 7: Czas wersji iteracyjnej



Rysunek 9: Wykres czasu wersji iteracyjnej

Funkcja została aproksymowana funkcją liniową. Jak można zauważyć na

podstawie wykresu, algorytm ma złożoność czasową $O(n)$, co jest zgodne z założeniami.

5.5 Activity Selector dynamiczny

Algorytm dynamiczny znajduje maksymalny zbiór niekolidujących aktywności, analizując kolejno każdą aktywność i sprawdzając, którą poprzednią aktywność można wybrać bez konfliktu. Dla każdej aktywności decyduje, czy lepiej ją włączyć do zbioru, czy pominąć, aby zmaksymalizować liczbę wybranych zajęć. Wybrane decyzje zapisywane są w dodatkowej tablicy, a wynikowy zbiór odtwarzany jest cofając się od końca. Wyszukiwanie poprzedniej niekolidującej aktywności realizowane jest za pomocą wyszukiwania binarnego, dzięki czemu czas działania algorytmu wynosi $O(nlogn)$.

```
int DYNAMIC_ACTIVITY_SELECTOR(const int s[], const int f[], int n, int result[]) {
    if (n < 1) return 0;

    vector<int> dp(n + 1, 0);
    vector<int> choice(n + 1, 0);

    dp[0] = 0;

    for (int i = 1; i <= n; i++) {

        int left = 0, right = i - 1, best = 0;
        while (left <= right) {
            int mid = (left + right) / 2;
            if (f[mid] <= s[i]) {
                best = mid;
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }

        int take = dp[best] + 1;
        int skip = dp[i-1];

        if (take > skip) {
            dp[i] = take;
            choice[i] = 1;
        } else {
            dp[i] = skip;
```

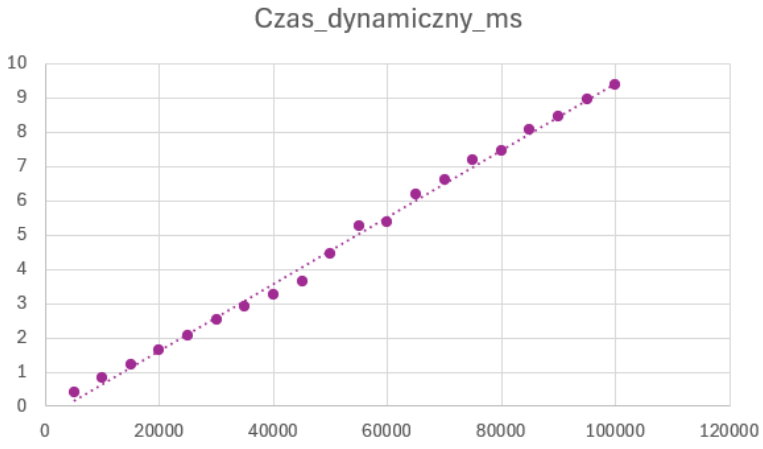
```
            choice[i] = 0;
        }
    }


    int count = dp[n];
    vector<int> temp;
    temp.reserve(count);

    int i = n;
    while (i > 0) {
        if (choice[i] == 1) {
            temp.push_back(i);

            int left = 0, right = i - 1, best = 0;
            while (left <= right) {
                int mid = (left + right) / 2;
                if (f[mid] <= s[i]) {
                    best = mid;
                    left = mid + 1;
                } else {
                    right = mid - 1;
                }
            }
            i = best;
        } else {
            i--;
        }
    }

    reverse(temp.begin(), temp.end());
    for (int j = 0; j < count; j++) {
        result[j] = temp[j];
    }

    return count;
}
```

5.6 Pomiary

Dla każdego z dwudziestu podanych uprzednio długości wykonano serię stu testów z pomiarem czasu. Wyniki tych pomiarów prezentuje Tabela 8.

Długość	Czas [ms]
5000	0,42
10000	0,81
15000	1,22
20000	1,63
25000	2,05
30000	2,51
35000	2,92
40000	3,26
45000	3,63
50000	4,43
55000	5,24
60000	5,39
65000	6,17
70000	6,62
75000	7,18
80000	7,45
85000	8,07
90000	8,44
95000	8,94
100000	9,36

Tabela 8: Czas wersji dynamicznej

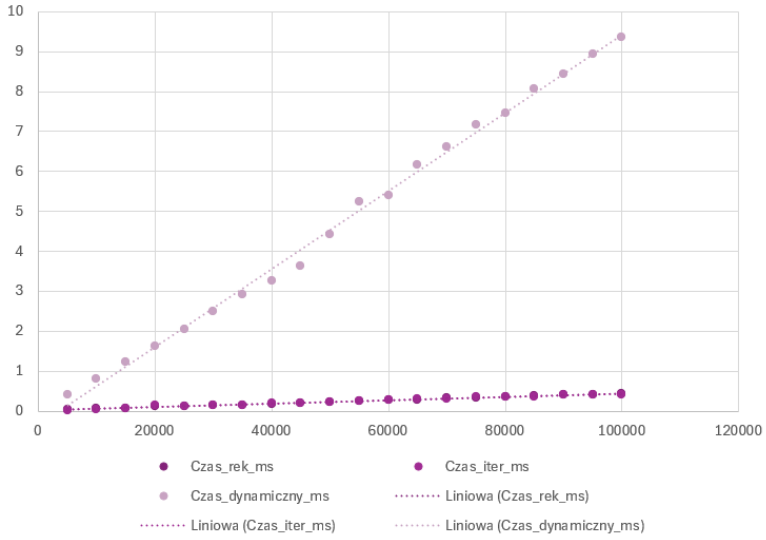


Rysunek 10: Wykres czasu wersji dynamicznej

Funkcja została aproksymowana funkcją liniową, ponieważ jest ona zbliżona do funkcji liniowo-logarytmicznej dla małych rozmiarów danych. Można

więc uznać, że wyniki są zgodne z założeniami.

5.7 Wnioski



Rysunek 11: Porównanie

Na Rysunku 11 został przedstawiony zbiórczy wykres czasu dla wersji rekurencyjnej, iteracyjnej oraz z zastosowaniem programowania dynamicznego. Na jego podstawie można stwierdzić, że algorytm iteracyjny i rekurencyjny działają w podobnym czasie i są szybsze od konkurencyjnego algoryt.

5.8 Activity Selector zmodyfikowany

Algorytm działa w odwrotnym kierunku do klasycznej wersji - zaczyna od zajęcia kończącego się najpóźniej, a następnie przegląda pozostałe aktywności od tyłu, dodając każde, które nie koliduje z już wybranymi. Wynik jest odwracany, aby uzyskać chronologiczną kolejność. Po rozważeniu wszystkich zajęć otrzymuje się maksymalny zbiór niekolidujących aktywności. Pomiary dla tego algorytmu zostały pominięte, ponieważ ma on taką samą złożoność jak klasyczna wersja.

```
1 int ACTIVITY_SELECTOR_GREEDY(const int s[], const int f[], int n, int result[]) {
2     if (n < 1) return 0;
3
4     result[0] = n;
5     int count = 1;
```

```
    int k = n;

    for (int m = n - 1; m >= 1; m--) {

        if (f[m] <= s[k]) {
            result[count] = m;
            count++;
            k = m;
        }
    }


    reverse(result, result + count);

    return count;
}
```

6 Kod Huffmana

6.1 Kod Huffmana binarny

```
struct Node {
    char symbol;
    int freq;
    Node *left, *right;

    Node(char s, int f) : symbol(s), freq(f), left(nullptr), right(nullptr) {}
    Node(Node* l, Node* r) : symbol('\0'), freq(l->freq + r->freq), left(l), right(r) {}
};

struct Compare {
    bool operator()(Node* a, Node* b) {
        return a->freq > b->freq;
    }
};

Node* buildHuffman(unordered_map<char, int>& freq) {
    priority_queue<Node*, vector<Node*>, Compare> pq;

    for (auto& p : freq) {
        pq.push(new Node(p.first, p.second));
    }
```

```
22
23      while (pq.size() > 1) {
24          Node* left = pq.top(); pq.pop();
25          Node* right = pq.top(); pq.pop();
26          pq.push(new Node(left, right));
27      }
28
29      return pq.top();
30  }
31
32  void getCodes(Node* root, string code, unordered_map<char, string>& codes) {
33      if (!root) return;
34
35      if (!root->left && !root->right) {
36          codes[root->symbol] = code;
37      }
38
39      getCodes(root->left, code + "0", codes);
40      getCodes(root->right, code + "1", codes);
41  }
```

Kody Huffmana to metoda spotykana przy kompresji danych. Podejście z wykorzystaniem kolejki priorytetowej można podzielić na następujące fazy:

1. Tworzymy węzeł dla każdego symbolu z jego częstotliwością występowania
2. Umieszczamy wszystkie węzły w kolejce priorytetowej uporządkowanej według rosnących częstotliwości za pomocą komparatora
3. Dopóki w kolejce znajduje się więcej niż jeden węzeł:
   - Wybieramy dwa węzły o najmniejszej częstotliwości i usuwamy je z kolejki
   - Tworzymy nowy węzeł wewnętrzny, którego częstotliwość jest sumą częstotliwości lewego i prawego dziecka
   - Nowy węzeł dodajemy z powrotem do kolejki priorytetowej
4. Gdy w kolejce pozostaje jeden węzeł, staje się on korzeniem drzewa Huffmana
5. Kody binarne generujemy rekurencyjnie funkcją `getCodes` – przechodząc do lewego dziecka dodajemy '0', do prawego '1'. Symbole znajdują się tylko w liściach drzewa

Złożoność tego algorytmu wynosi $O(n \log n)$.

6.2 Kod Huffmana trenarny

Trójkowy wariant kodów Huffmana wykorzystuje drzewo z trzema potomkami w każdym węźle wewnętrznym. Przed budową drzewa dodajemy fikcyjne węzły o częstotliwości 0, aby liczba węzłów spełniała warunek $(n-1) \bmod 2 = 0$. W każdej iteracji wybieramy i łączymy trzy węzły o najmniejszych częstotliwościach zamiast dwóch. Kody generujemy funkcją `getTernaryCodes`, przypisując '0', '1', '2' odpowiednio dla pierwszego, drugiego i trzeciego dziecka. Złożoność wynosi $O(n \log n)$.

```
struct TernaryNode {
    char symbol;
    int freq;
    TernaryNode *child0, *child1, *child2;

    TernaryNode(char s, int f) : symbol(s), freq(f), child0(nullptr), child1(nullptr), c
    TernaryNode(TernaryNode* c0, TernaryNode* c1, TernaryNode* c2)
        : symbol('\0'), freq(c0->freq + c1->freq + c2->freq), child0(c0), child1(c1), chi
};

struct CompareTernary {
    bool operator()(TernaryNode* a, TernaryNode* b) {
        return a->freq > b->freq;
    }
};

TernaryNode* buildHuffmanTernary(unordered_map<char, int>& freq) {
    priority_queue<TernaryNode*, vector<TernaryNode*>, CompareTernary> pq;

    for (auto& p : freq) {
        pq.push(new TernaryNode(p.first, p.second));
    }

    while ((pq.size() - 1) % 2 != 0) {
        pq.push(new TernaryNode('\0', 0));
    }

    while (pq.size() > 1) {
        TernaryNode* c0 = pq.top(); pq.pop();
        TernaryNode* c1 = pq.top(); pq.pop();
        TernaryNode* c2 = pq.top(); pq.pop();
        pq.push(new TernaryNode(c0, c1, c2));
    }

```

```
    return pq.top();
}

void getTernaryCodes(TernaryNode* root, string code, unordered_map<char, string>& codes)
    if (!root) return;

    if (!root->child0 && !root->child1 && !root->child2 && root->symbol != '\0') {
        codes[root->symbol] = code;
    }

    getTernaryCodes(root->child0, code + "0", codes);
    getTernaryCodes(root->child1, code + "1", codes);
    getTernaryCodes(root->child2, code + "2", codes);
}
```