Algorytmy i Struktury Danych
Sprawozdanie 2

Emilia Łagoda
Numer albumu: 287361

Spis treści

0.1 Wstęp 2
0.2 Metodyka Badań 2
0.3 Cut Rod 2
0.3.1 Naiwny Cut Rod 2
0.3.2 Memorized Cut Rod 4
0.3.3 Bottom-Up Cut Rod 6
0.3.4 Wnioski 8
0.4 LCS 8
0.4.1 Iterative LCS 8
0.4.2 Recursive LCS 10
0.4.3 Wnioski 12
0.5 Activity Selector 12
0.5.1 Recursive Activity Selector 12
0.5.2 Iterative Activity Selector 15
0.5.3 Dynamic Activity Selector 17
0.5.4 Wnioski 19
0.6 Kod Huffmanna 19
0.6.1 Ternarny Kod Hufmanna 22
0.6.2 Wnioski 24

0.1 Wstęp

W tym ćwiczeniu, porównywane będą różne wersje czterech algorytmów grafowych ze szczególną uwagą na czas wykonywania oraz zachowanie przy dużym zakresie rozmiaru danych. D osprawdzanych algorytmów należy rozcinanie pręta (Cut Rod), najdłuższy wspólny podciąg (Longest Common Sequence, tzw. LCS), algorytm polegający na wyborze aktywności względem planu godzinowego, oraz kod Huffmana.

0.2 Metodyka Badań

Dla każdego kodu, sprawdzaną zależnością będzie czas wykonywania algorytmu dla danych o liniowo rosnącym ciągu długości. Dla każdego algorytmu będzie wykonane dwadzieścia testów i pięć powtórzeń pomiarów.

0.3 Cut Rod

Celem problemu przecinania pręta jest zmaksymalizowanie zysku ze sprzedaży jego fragmentów, czyli optymalizacja podziału odcinka o długości n, tak aby suma cen pododcinków była jak najwyższa.

0.3.1 Naiwny Cut Rod

Naiwne rozwiązanie tego problemu polega na rekurencyjnym sprzedawaniu odcinka z lewego końca pręta, a następnie sprawdzeniu optymalnego podziału dla pozostałości. W taki sposób algorytm najpierw odcina kawałek $n = 1$, i wywołuje rekursję dla pręta o długości $n - 1$, po sprawdzeniu tego drzewa, odcina kawałek $n = 2$, i tworzy kolejne poddrzewo. Po rozważeniu każdego możliwego podziału, wynik ma postać maksymalnej sumy cen. Istnieje $2^(n - 1)$ takich podziałów, więc tyle też jest węzłów tego drzewa, przez co złożoność tego podejścia to $O(2^n)$.

```
int naive_cut_rod(int p[], int n){

    if (n == 0){
        return 0;
    }

    int q = INT_MIN;

    for (int i = 0; i < n; i++){
        q = max(q, p[i] + naive_cut_rod(p, n-i-1));
    }

    return q;
}
```

Średnia wyników

Tabela 1: Średnia wyników dla naiwnego Cut Rod

Długość tablicy n	**Czas [ms]**
10	0,0096
11	0,0162
12	0,0244
13	0,062
14	0,1278
15	0,2288
16	0,4558
17	1,1022
18	1,8898
19	3,7414
20	7,293
21	14,7586
22	27,1116
23	48,998
24	96,6236
25	200,1548
26	384,3816
27	761,4528
28	1472,924
29	3114,758

Analiza wykresów

Rysunek 1: Zależność czasu wykonywania od długości n dla algorytmu Quick Sort

Zgodnie z założeniami teoretycznymi, algorytm wykazuje złożoność $O(2^n)$.

0.3.2 Memorized Cut Rod

Memorized Cut Rod wykorzystuje zapamiętywanie wyników rekursji w celu przyśpieszenia późniejszych wywołań rekursji, w których dany wynik byłby potrzebny. Dzięki temu, każde poddrzewo jest przechodzone dokładnie raz, więc aby rozwiązać podproblem o rozmiarze i, wykonujemy i iteracji. Dla pręta o długości n, mamy najwyżej n podproblemów, to prowadzi do złożoności $O(n^2)$ w najgorszym przypadku, a w najlepszym zbliżoną do $O(n)$.

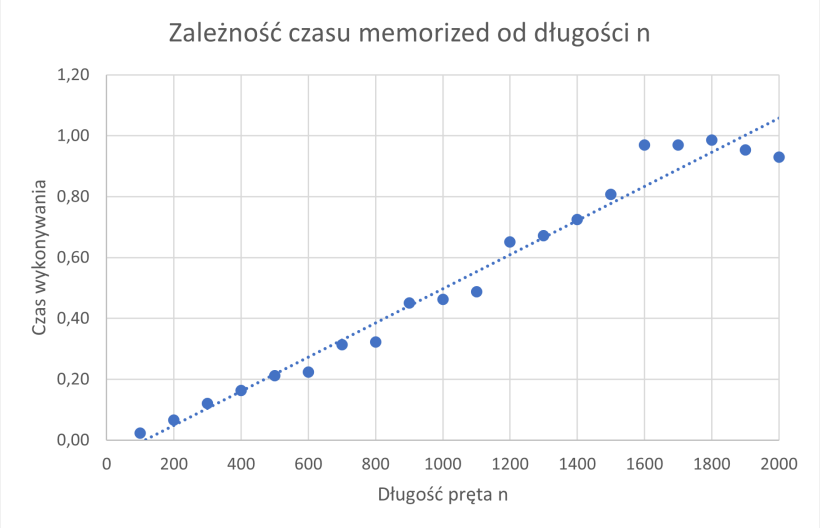
```
int memorized_cut_rod(int p[], int n, int r[]){
    int q;

    if (r[n] >= 0){
        return r[n];
    }

    if (n == 0){
        q = 0;
    }
    else{
        q = INT_MIN;

        for (int i = 0; i < n; i++){
            q = max(q, p[i] + memorized_cut_rod(p, n-i-1, r));
        }
    }

    r[n] = q;

    return q;
}
}
```

Średnia wyników

Tabela 2: Średnia wyników dla Memorized Cut Rod

Długość tablicy n	**Czas [ms]**
100	0,02
200	0,07
300	0,12
400	0,16
500	0,21
600	0,22
700	0,31
800	0,32
900	0,45
1000	0,46
1100	0,49
1200	0,65
1300	0,67
1400	0,73
1500	0,81
1600	0,97
1700	0,97
1800	0,99
1900	0,95
2000	0,93

Analiza wykresów



Rysunek 2: Zależność czasu wykonywania od długości n dla Memorized Cut Rod

Dla tego zestawu danych, zależność przyjmuje formę schodkową, podobną do zachowania algorytmów o złożoności $O(nlogn)$. Jest to częściowo zgodne z teoretyczną złożonością,

ponieważ $n < nlogn < n^2$, więc leży względnie pomiędzy best-case i worst-case zachowaniem algorytmu.

0.3.3 Bottom-Up Cut Rod

Tutaj również wyniki są zapamiętywane. Podejście bottom-up polega na wcześniejszym obliczeniu optymalnych podziałów dla krótszych prętów, które później będą potrzebne przy rozwiązaniach dla dłuższych. Złożoność algorytmu to podobnie do algorytmu z zapamiętywaniem, ponieważ przez zagnieżdżone pętle jest to $O(n^2)$.

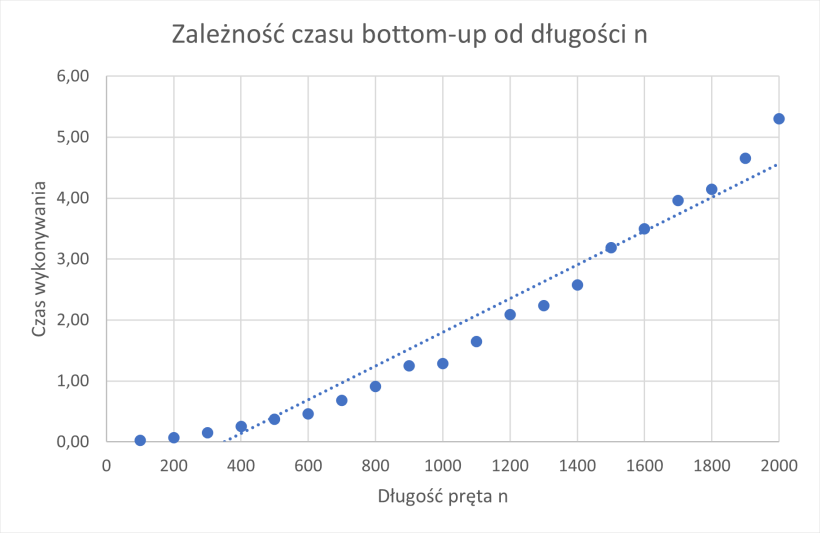
```
int bottom_up_cut_rod(int p[], int n){
    int r[n+1];
    int q;

    r[0] = 0;

    for (int j = 1; j <= n; j++){
        q = INT_MIN;
        for (int i = 0; i < j; i++){
            q = max(q, p[i] + r[j-i-1]);
        }
        r[j] = q;
    }

    return r[n];
}
```

Średnia wyników

Tabela 3: Średnia wyników dla Bottom-Up Cut Rod

Długość tablicy n	**Czas [ms]**
100	0.02
200	0.07
300	0.15
400	0.26
500	0.37
600	0.46
700	0.68
800	0.91
900	1.25
1000	1.28
1100	1.65
1200	2.09
1300	2.23
1400	2.58
1500	3.19
1600	3.49
1700	3.96
1800	4.15
1900	4.66
2000	5.30

Analiza wykresów



Rysunek 3: Zależność czasu wykonywania od długości n dla algorytmu Bottom-Up Cut Rod

Algoryt przybiera formę funkcji kwadratowej, co jest zgodne z założoną złożonością $O(n^2)$. Mimo małego gradientu, co można przypisać zbyt małej próbie danych wejściowych, występuje zauważalna parabola.

0.3.4 Wnioski

Najszybszym algorytmem okazuje się być wersja memorized, która w najgorszym przypadku ma złożoność podobną do wersji bottom-up, czyli $O(n^2)$.

0.4 LCS

Algorytm Longest Common Sequence, czy też LCS, jak nazwa wskazuje ma na celu znalezienie najdłuższego wspólnego podciągu wszystkich podanych ciągów znaków. Podciąg tu jest definiowany jako łańcuch wspólnych znaków w określonej kolejności, co oznacza, że mogą pomiędzy nimi występować znaki nienależące do pociągu.

0.4.1 Iterative LCS

Iteracyjny LCS polega na wypełnianiu tablicy rozwiązań dla coraz dłuższych początkowych podciągów. W taki sposób algorytm najpierw bierze podciąg długości 1 pierwszego ciągu i porównuje go z każdym podciągiem drugiego ciągu, zapisując w tablicy długość LCS dla tej pary. Po wypełnieniu całej tablicy, odtwarza rozwiązanie cofając się od prawego dolnego rogu do lewego górnego, śledząc kierunki zapisane w pomocniczej tablicy. Istnieje $m\ddot{O}n$ podproblemów, każdy rozwiązywany w czasie stałym, więc złożoność tego podejścia to $O(mn)$.

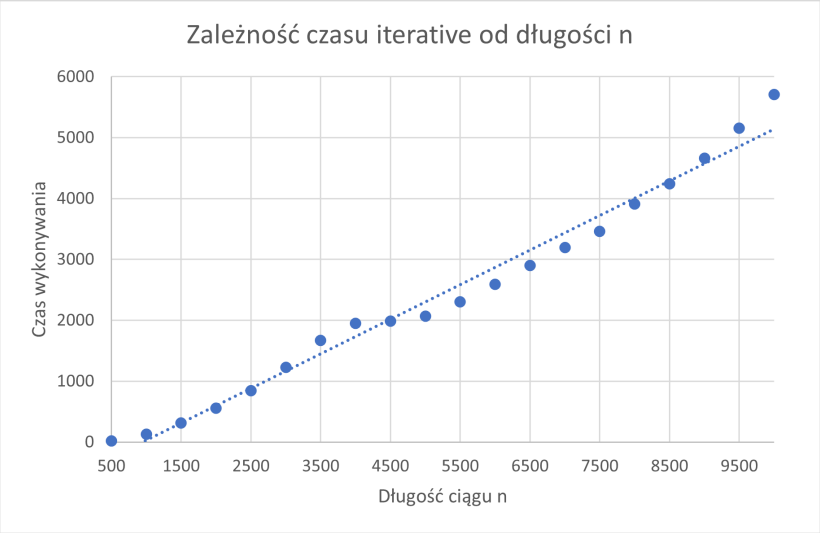
```
string LCS_iterative(string X, string Y) {
    int m = X.length();
    int n = Y.length();

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

    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;
                b[i][j] = 'd';  // diagonal
            }
            else if (c[i-1][j] >= c[i][j-1]) {
                c[i][j] = c[i-1][j];
                b[i][j] = 'u';  // up
            }
            else {
                c[i][j] = c[i][j-1];
                b[i][j] = 'l';  // left
            }
        }
    }
```

Średnia wyników

Tabela 4: Średnia wyników dla Iterative LCS

Długość tablicy n	Czas [ms]
500	20.9322
1000	131.794
1500	311.8902
2000	554.115
2500	845.0902
3000	1226.6162
3500	1667.184
4000	1946.904
4500	1984.332
5000	2066.046
5500	2305.618
6000	2593.048
6500	2897.956
7000	3196.498
7500	3460.006
8000	3911.796
8500	4239.448
9000	4664.2
9500	5159.274
10000	5705.98

Analiza wykresów



Rysunek 4: Zależność czasu wykonywania od długości ciągu dla algorytmu Iterative LCS

Wykazuje zależność podobną do liniowej, ale rosnący gradient dla drugiej połowy zestawu danych sugeruje, że może mieć złożoność nieliniową, co byłoby zgodne z założeniem teoretycznym, jednak zakres danych wejściowych jest zbyt mały aby to zaobserwować.

0.4.2 Recursive LCS

To podejście polega na rekurencyjnym porównywaniu końcowych podciągów, ale z zapamiętywaniem już obliczonych wyników. W taki sposób algorytm najpierw sprawdza czy rozwiązanie dla pary (i, j) już istnieje, jeśli nie, porównuje ostatnie znaki ciągów i wywołuje rekursję dla odpowiednio skróconych ciągów. Po obliczeniu wartości dla wszystkich możliwych par indeksów, odtwarza rozwiązanie rekurencyjnie. Istnieje $m\ddot{O}n$ podproblemów, więc złożoność tego podejścia to $O(mn)$, jednak ze względu na rekurencyjne wywołania zużywa więcej pamięci niż wersja iteracyjna.

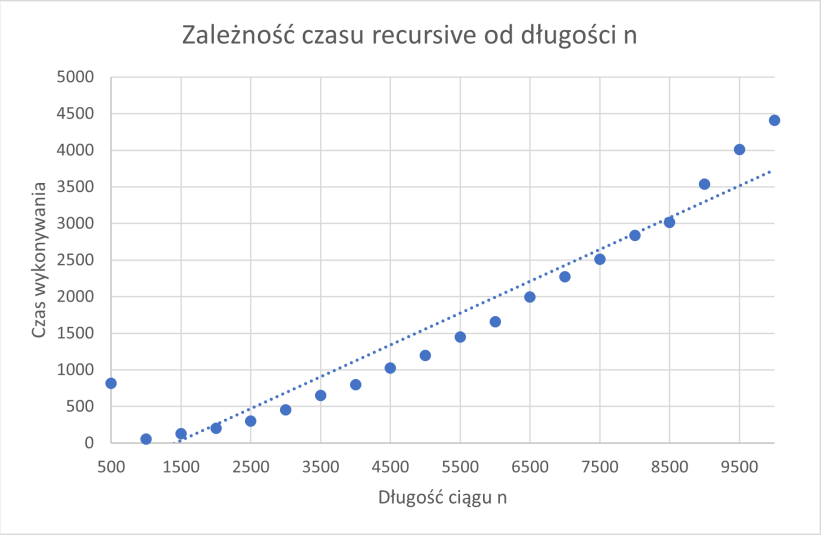
```
vector<vector<int>> memo;
string X_global, Y_global;

int LCS_recursive_helper(int i, int j) {
    if (i == 0 || j == 0) return 0;

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

    if (X_global[i-1] == Y_global[j-1]) {
        memo[i][j] = 1 + LCS_recursive_helper(i-1, j-1);
    } else {
        memo[i][j] = max(LCS_recursive_helper(i-1, j),
                         LCS_recursive_helper(i, j-1));
    }

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

Średnia wyników

Tabela 5: Średnia wyników dla Recursive LCS

Długość tablicy n	Czas [ms]
500	815.9904
1000	53.4254
1500	131.2608
2000	202.8398
2500	299.1258
3000	456.1596
3500	648.2342
4000	796.0108
4500	1024.2448
5000	1198.6554
5500	1449.468
6000	1656.638
6500	1992.714
7000	2273.652
7500	2513.258
8000	2838.718
8500	3014.935
9000	3534.976
9500	4008.214
10000	4409.962

Analiza wykresów



Rysunek 5: Zależność czasu wykonywania od długości ciągu dla Recursive LCS

Tutaj nieliniowa złożoność jest jasno zauważalna, co jest zgodne z założeniami teoretycznymi, dla dwóch ciągów o podobnej długości, będzie zbliżona do funkcji kwadratowej.

0.4.3 Wnioski

Oba algorytmy mają taką samą teroetyczną złożoność, jednak przy implementacji, wersja iteracyjna jest podobna do funkcji o złożoności $O(n)$. Mimo bardziej liniowej zależności, dla tych samych danych wejściowych, podejście iteracyjne jest wolniejsze, co można przypisać zapamiętywaniu par (i, j) w implementacji rekurencyjnej.

0.5 Activity Selector

Activity Selector należy do tak zwanych algorytmów zachłannych, czyli w każdym kolejnym kroku wybiera obecne najlepsze rozwiązanie częściowe. Celem zadania jest optymalne dobranie podzbioru dostępnych aktywności, tak aby zmaksymalizować ilość wybranych aktywności bez kolizji ich godzin rozpoczęcia i zakończenia.

0.5.1 Recursive Activity Selector

Podejście rekurencyjne polega na wyborze zajęcia kończącego się najwcześniej, a następnie iteracyjnym szukaniu kolejnych zajęć rozpoczynających się po jego zakończeniu. W taki sposób algorytm najpierw znajduje zajęcie o minimalnym czasie zakończenia, dodaje je do wyniku, a potem liniowo przegląda pozostałe zajęcia w posortowanej kolejności. Po sprawdzeniu każdego zajęcia od pozycji $earliest_finish_index$ do końca, wynik ma postać maksymalnego zbioru zajęć o niekonfliktujących czasach. Istnieje n takich porównań dla każdego wybranego zajęcia, więc złożoność tego podejścia to $O(nlogn)$.

```
vector<int> recursive_selector_sorted
(const vector<Activity>& activities, int k) {
    int n = activities.size();
    int m = k + 1;

    while (m < n && activities[m].start < activities[k].finish) {
        m++;
    }

    if (m < n) {
        vector<int> result = {activities[m].index};
        vector<int> rest = recursive_selector_sorted(activities, m);
        result.insert(result.end(), rest.begin(), rest.end());
        return result;
    }

    return {};
}

//pomocnicza do wywoŁania rekurencyjnego
vector<int> recursive_selector_modded
(const vector<Activity>& activities) {
    int n = activities.size();
    if (n == 0) return {};
```

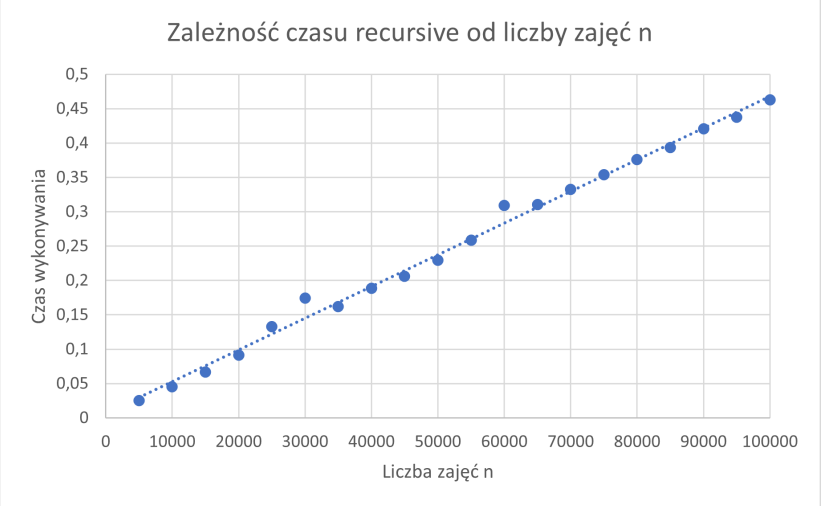
```
    //najczybciej si Ź ko cz
    int earliest_finish_index = 0;
    for (int i = 1; i < n; i++) {
        if (activities[i].finish <
        activities[earliest_finish_index].finish) {
            earliest_finish_index = i;
        }
    }

    //pierwszy wyb r
    vector<int> result = {activities[earliest_finish_index].index};

    // szukamy kolejnych zaj Ź
    int current_finish = activities[earliest_finish_index].finish;
    for (int i = earliest_finish_index + 1; i < n; i++) {
        if (activities[i].start >= current_finish) {
            result.push_back(activities[i].index);
            current_finish = activities[i].finish;
        }
    }

    return result;
}
```

Średnia wyników

Tabela 6: Średnia wyników dla Recursive Activity Selector

Długość tablicy n	**Czas [ms]**
5000	0.0252
10000	0.0454
15000	0.0668
20000	0.0916
25000	0.1326
30000	0.1746
35000	0.1618
40000	0.1884
45000	0.2062
50000	0.2294
55000	0.2586
60000	0.309
65000	0.3106
70000	0.3326
75000	0.354
80000	0.3758
85000	0.3936
90000	0.4206
95000	0.4376
100000	0.463

Analiza wykresów



Rysunek 6: Zależność czasu wykonywania od ilości zajęć dla Recursive Activity Selector

Zależność czasu od ilości zajęć przypomina funkcję liniową, co nie jest zgodne z założeniami teoretycznymi, jednak można zauważyć skoki charakterystyczne dla $O(nlogn)$ na przestrzeni całego zakresu danych, co sugeruje, że dla danych o większym rozmiarze, zależność może być

nieliniowa.

0.5.2 Iterative Activity Selector

Algorytm iteracyjny działa podobnie do podejścia z rekursją. Algorytm najpierw znajduje zajęcie kończące się najwcześniej w czasie $O(n)$, a następnie dodaje kolejne niekonfliktujące zajęcia. Zaczynając od najwcześniej kończącego się zajęcia, przegląda pozostałe zajęcia w ich oryginalnej kolejności i tworzy zbiór wynikowy. Po rozważeniu każdego zajęcia, wynik ma postać maksymalnego zbioru możliwych zajęć. Istnieje $n-1$ takich porównań po znalezieniu początkowego zajęcia, więc złożoność tego podejścia to $O(n)$ po znalezieniu minimalnego czasu zakończenia (lub $O(n\check{s})$ jeśli szukamy minimalnego w każdej iteracji).

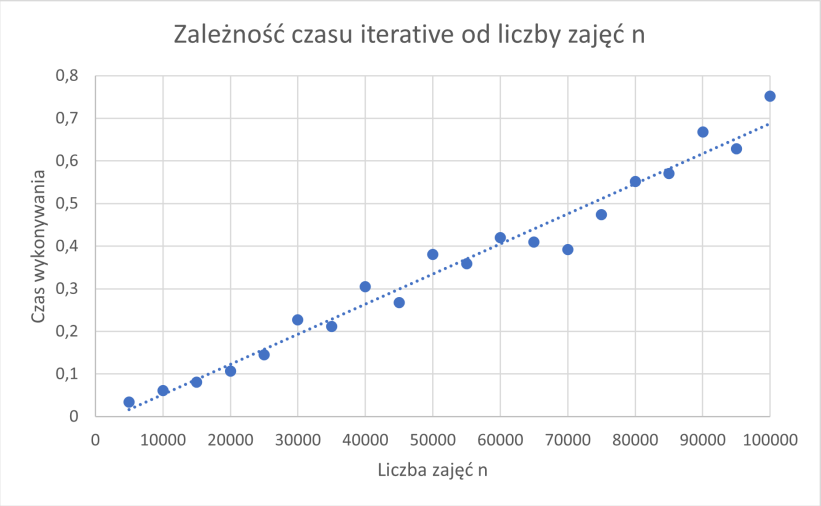
```
vector<int> iterative_selector_modded
(const vector<Activity>& activities) {
    int n = activities.size();
    if (n == 0) return {};

    //musimy znale    zaj Ście kt re ko czy si Ś najwcze Żniej
    int earliest_finish_index = 0;
    for (int i = 1; i < n; i++) {
        if (activities[i].finish <
        activities[earliest_finish_index].finish) {
            earliest_finish_index = i;
        }
    }

    vector<int> A = {activities[earliest_finish_index].index};
    int last_finish = activities[earliest_finish_index].finish;

    //pozosta Će zaj Ścia (ju  posortowane wg start)
    for (int i = earliest_finish_index + 1; i < n; i++) {
        if (activities[i].start >= last_finish) {
            A.push_back(activities[i].index);
            last_finish = activities[i].finish;
        }
    }

    return A;
}
```

Średnia wyników

Tabela 7: Średnia wyników dla Iterative Activity Selector

Długość tablicy n	Czas [ms]
5000	0.0252
10000	0.0454
15000	0.0668
20000	0.0916
25000	0.1326
30000	0.1746
35000	0.1618
40000	0.1884
45000	0.2062
50000	0.2294
55000	0.2586
60000	0.309
65000	0.3106
70000	0.3326
75000	0.354
80000	0.3758
85000	0.3936
90000	0.4206
95000	0.4376
100000	0.463

Analiza wykresów



Rysunek 7: Zależność czasu wykonywania od ilości zajęć dla zmodyfikowanego Bucket Sort

Tutaj również złożoność przypomina liniową, ale skoki są jeszcze bardziej zauważalne niż w przypadku podejścia rekursywnego. To zachowanie ponownie sugeruje, że zakres danych był zbyt mały aby odpowiednio zaobserwować złożoność implementacji algorytmu.

0.5.3 Dynamic Activity Selector

Algorytm programowania dynamicznego polega na obliczeniu maksymalnego zbioru zajęć kończących się na i-tym zajęciu. Algorytm najpierw sortuje zajęcia według czasu rozpoczęcia w $O(nlogn)$, a następnie dla każdego i-tego zajęcia sprawdza wszystkie wcześniejsze zajęcia i oblicza maksymalny rozmiar zbioru zawierającego i-te zajęcie. W taki sposób, zaczynając od pierwszego zajęcia, wypełnia tablicę dp wartościami $1..n$, a na koniec odtwarza rozwiązanie. Po rozważeniu każdej pary zajęć (i, j), gdzie $j < i$, wynik ma postać maksymalnej liczby zajęć które można wybrać. Istnieje $n(n-1)/2$ takich porównań, więc złożoność tego podejścia to $O(n$š$)$, plus $O(nlogn)$ na sortowanie.

```
vector<int> dynamic_selector(const vector<Activity>& activities) {
    int n = activities.size();
    if (n == 0) return {};

    //kopia z sortowaniem wg czasu rozpocz Ścia
    vector<Activity> sorted_activities = activities;
    sort(sorted_activities.begin(), sorted_activities.end(),
         [](const Activity& a, const Activity& b) {
             return a.start < b.start;
         });

    // dp[i] - maksymalna liczba zaj Ź  kt re mo na wybra ,
    // ko cz cych si Ź na i-tym zaj Źciu (lub wcze Żniej)
    vector<int> dp(n, 1);
    vector<int> prev(n, -1);

    //algorytm dynamiczny
    for (int i = 1; i < n; i++) {
        dp[i] = 1;
        prev[i] = -1;

        for (int j = 0; j < i; j++) {
            if (sorted_activities[j].finish
            <= sorted_activities[i].start) {
                if (dp[j] + 1 > dp[i]) {
                    dp[i] = dp[j] + 1;
                    prev[i] = j;
                }
            }
        }
    }

    int max_index = 0;
    for (int i = 1; i < n; i++) {
        if (dp[i] > dp[max_index]) {
            max_index = i;
        }
```

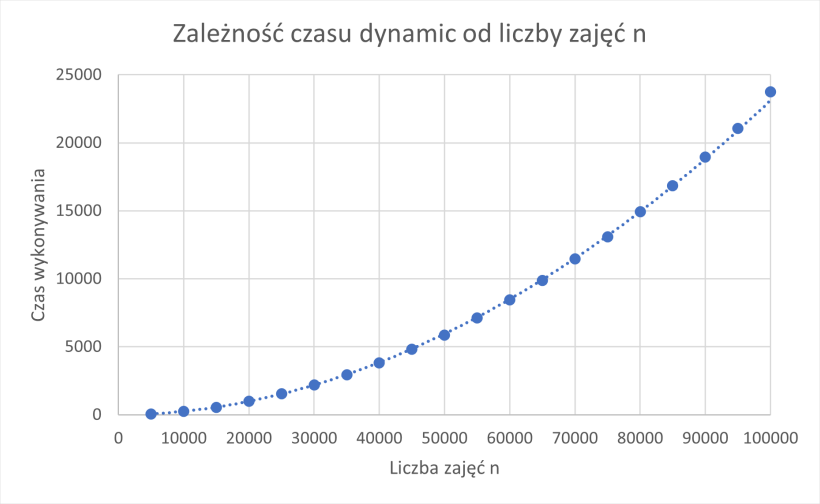
```
    }

    vector<int> result;
    int current = max_index;
    while (current != -1) {
        result.push_back(sorted_activities[current].index);
        current = prev[current];
    }

    reverse(result.begin(), result.end());
    return result;
}
```

Średnia wyników

Tabela 8: Średnia wyników dla Dynamic Activity Selector

Długość tablicy n	**Czas [ms]**
5000	0.0252
10000	0.0454
15000	0.0668
20000	0.0916
25000	0.1326
30000	0.1746
35000	0.1618
40000	0.1884
45000	0.2062
50000	0.2294
55000	0.2586
60000	0.309
65000	0.3106
70000	0.3326
75000	0.354
80000	0.3758
85000	0.3936
90000	0.4206
95000	0.4376
100000	0.463

Analiza wykresów



Rysunek 8: Zależność czasu wykonywania od ilości zajęć dla Dynamic Activity Selector

W przypadku podejścia z programowaniem dynaicznym, zależność czasu od liczby zajęć jest zgodna z założeniami teoretycznymi złożoności $O(n^2)$.

0.5.4 Wnioski

Mimo niedokładnych wykresów ze względu na za mały zakres danych przy podejściu iteracyjnym i rekurencyjnym, obie są szybsze od podejścia dynamicznego.

0.6 Kod Huffmanna

Kody Huffmanna to metoda często spotykana przy bezstratnej kompresji danych. Niech X będzie zbiorem symboli i niech $f(x)$ będzie częstotliwością danego symbolu w tekście, który kodujemy. Podejście z wykorzystaniem priority queue można podzielić na trzy fazy:

1. Tworzymy węzeł dla każdego symbolu i jego prawdopodobieństwa

2. Umieszczamy wszystkie węzły w kolejce priorytetowej uporządkowanej według rosnących częstotliwości

3. Dopóki w kolejce znajduje się więcej niż jeden węzeł, łączymy ze sobą dwa węzły o najmniejszej częstotliwości, tworząc nowy węzeł wewnętrzny, którego częstotliwość jest sumą częstotliwości węzłów składowych. Nowy węzeł dodajemy z powrotem do kolejki priorytetowej.

4. Po utworzeniu drzewa, każdemu symbolowi jest przypisywany kod. Przechodząc od korzenia do liścia, dodajemy 0 przy przejściu do lewego dziecka i 1 przy przejściu do prawego dziecka

Złożoność tego algorytmu jest zależna jedynie od liczby wejściowych symboli. Każda kolejka priorytetowa ma złożoność $O(logn)$, a trawersowanie drzewa ma złożoność $O(n)$, co daje końcową złożoność $O(nlogn)$.

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

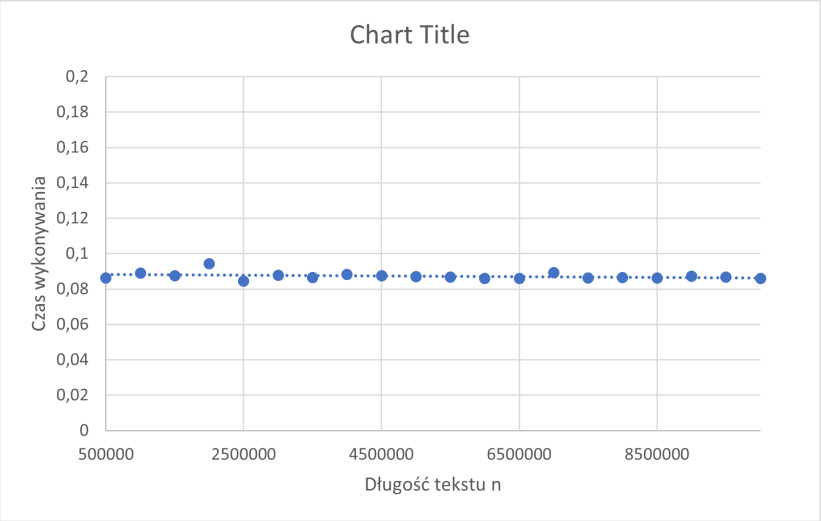
```
    Node(char s, int f) : symbol(s), freq(f),
    left(nullptr), right(nullptr) {}
    Node(Node* l, Node* r) : symbol('\0'),
    freq(l->freq + r->freq), left(l), right(r) {}
};

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

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

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

    while (pq.size() > 1) {
        Node* left = pq.top(); pq.pop();
        Node* right = pq.top(); pq.pop();
        pq.push(new Node(left, right));
    }

    return pq.top();
}

void get_codes(Node* root, string code,
unordered_map<char, string>& codes) {
    if (!root) return;

    if (!root->left && !root->right) {
        codes[root->symbol] = code;
    }

    get_codes(root->left, code + "0", codes);
    get_codes(root->right, code + "1", codes);
}
```

Średnia wyników

Tabela 9: Średnia wyników dla kodu Huffmanna

Długość tablicy n	**Czas [ms]**
500000	0.0862
1000000	0.089
1500000	0.0876
2000000	0.0942
2500000	0.0844
3000000	0.0878
3500000	0.0864
4000000	0.0882
4500000	0.0874
5000000	0.087
5500000	0.0866
6000000	0.086
6500000	0.086
7000000	0.0892
7500000	0.0862
8000000	0.0864
8500000	0.0862
9000000	0.0872
9500000	0.0866
10000000	0.086

Analiza wykresów



Rysunek 9: Zależność czasu wykonywania od długości tekstu dla algorytmu kodu Huffmanna

Zgodnie z założeniem, algorytm i jego czas wykonywania jest zależny od ilości symboli wejściowych, a nie długości samego tekstku, więc na wykresie jest podobny do wykresu funkcji stałej.

0.6.1 Ternarny Kod Hufmanna

Ternarna wersja kodowania Huffmanna zmienia jedynie zakres znaków, którymi kodowany jest tekst. Zamiast kodowania na system binarny, gdzie wykorzystywane są ciągi zero-jedynkowe, w ternarnym kodzie używane są zera, jedynki i dwójki. To wpływa na budowę drzewa. Węzły mogą mieć dziecko lewe, prawe i środkowe, wtedy przechodząc do dziecka po lewej dodajemy 0, przechodząc do środkowego dodajemy 1, a dla prawego 2.

```
struct Node {
    char symbol;
    int freq;
    vector<Node*> children;

    Node(char s, int f) : symbol(s), freq(f) {}
    Node(vector<Node*> kids) : symbol('\0'), freq(0), children(kids) {
        for (Node* child : kids) {
            freq += child->freq;
        }
    }
};

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

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

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


    while (pq.size() > 1 && (pq.size() - 1) % 2 != 0) {
        pq.push(new Node('\0', 0)); // pusty w Źze Ć
    }

    while (pq.size() > 1) {
        vector<Node*> children;
        for (int i = 0; i < 3 && !pq.empty(); i++) {
            children.push_back(pq.top());
            pq.pop();
        }
        pq.push(new Node(children));
    }
```

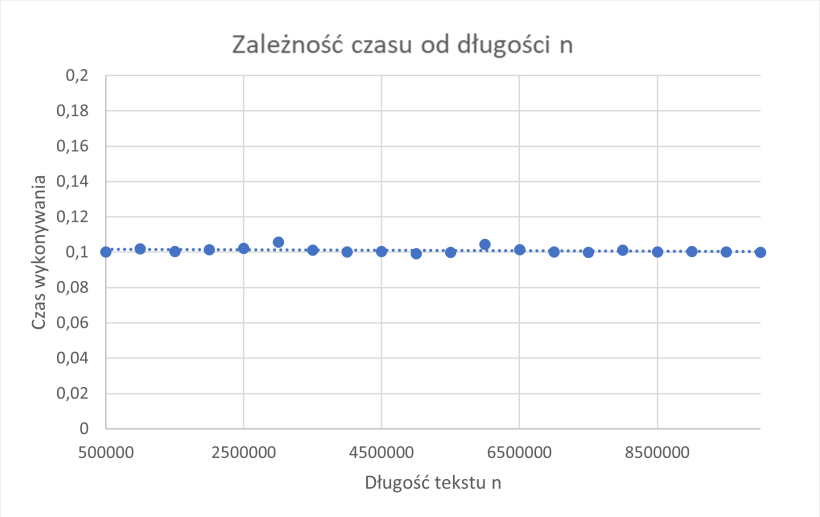
```
    return pq.top();
}

void get_ternary_code(Node* root, string code,
unordered_map<char, string>& codes) {
    if (!root) return;

    if (root->symbol != '\0' && root->children.empty()) {
        codes[root->symbol] = code;
        return;
    }

    for (size_t i = 0; i < root->children.size(); i++) {
        get_ternary_code(root->children[i], code + to_string(i), codes);
    }
}
```

Średnia wyników

Tabela 10: Średnia wyników dla ternarnego kodu Huffmanna

Długość tablicy n	**Czas [ms]**
500000	0.1002
1000000	0.1018
1500000	0.1004
2000000	0.1014
2500000	0.1022
3000000	0.1056
3500000	0.1012
4000000	0.1002
4500000	0.1004
5000000	0.0992
5500000	0.0998
6000000	0.1044
6500000	0.1014
7000000	0.1002
7500000	0.1
8000000	0.1012
8500000	0.1002
9000000	0.1004
9500000	0.1002
10000000	0.1

Analiza wykresów



Rysunek 10: Zależność czasu wykonywania od długości tekstu dla ternarnego kodu Huffmanna

Podobnie do standardowej wersji kodu, złożoność nie jest zależna od długości tekstu, więc i tutaj wykazuje zachowanie stałe.

0.6.2 Wnioski

Oba algortymy wykonują zadanie w podobnym czasie, jednak kod binarny jest niewiele szybszy od ternarnego, co można przypisać temu, że komputery szybciej wykonują operacje w systemie binarnym, mimo tego, że ternarny powinien skracać głębokość drzewa.