Algorytmy i struktury danych sprawozdanie z laboratorium nr 3

Aleksandra Cichecka
numer albumu: 287362

Styczen 2026

Spis treści

1 Wstep

W dokumencie zawarta jest analiza kilku wybranych algorytmów zachłannych oraz algorytmów opartych na programowaniu dynamicznym. Celem pracy było zaimplementowanie algorytmów w różnych wersjach, a także odzyskania rozwiązań gdzie było to możliwe. Algorytmy na których się skupimy to:

- CUT ROD, w wersji naiwnej, ze spamiętywaniem oraz iteracyjnej
- LCS, w wersji rekurencyjnej ze spamiętywaniem oraz iteracyjnej
- ACTIVITY SELECTOR, w wersji rekurencyjnej, iteracyjnej, zmodyfikowanej, a także tej opartej na programowaniu dynamicznym
- HUFFMAN, w wersji klasycznej oraz zmodyfikowanej

Przeprowadzone zostały także testy, które dzięki którymi można porównać efektywność algorytmów.

2 Cut-Rod

2.1 Opis działania algorytmu

Celem problemu przecinania pręta jest zmaksymalizowanie zysku ze sprzedaży jego fragmentów, czyli na wyznaczeniu optymalnego sposobu podziału pręta o zadanej długości n na fragmenty, które maksymalizują łączną wartość sprzedaży.

2.1.1 Wersja naiwna

Wersja naiwna polega na sprawdzeniu wszystkich możliwych sposobów pierwszego cięcia pręta, a następnie wykonania rekurencyjnie tego samego problemu dla pozostałej części. Jest to wersja naiwna ponieważ nie zapamiętuje wyników i liczy wszystko od nowa. Liczba możliwych podziałów wynosi $2^{(n-1)}$, przez co złożoność tego podejścia to $O(2^n)$

2.1.2 Wersja ze spamietywaniem

Wersja ze spamiętywaniem, jak sama nazwa wskazuje, polega na zapamiętaniu wyników rekursji w celu przyśpieszenia późniejszych jej wywołań, gdy dany wynik jest potrzebny. Każda wartość obliczana jest tylko raz. Prowadzi to do złożoności czasowej $O(n^2)$

W tej wersji stosujemy również odzyskiwanie rozwiązania. Poza maksymalnym zyskiem, dowiadujemy się też na jakie kawałki należy podzielić ten pręt aby taki zysk osiągnąć.

2.1.3 Wersja iteracyjna

Wersja iteracyjna stosuje podejście od dołu do góry. Zamiast rekurencyjnego rozwiązywania problemów od największego rozmiaru, zaczyna od najmniejszego podproblemu. Tutaj również wyniki są zapamiętywane oraz rozwiązania są odzyskiwane. Złożoność algorytmu podobnie jak wersji z spamiętywaniem wynosi $O(n^2)$.

2.2 Fragmenty kodu

```
int NAIVE_CUT_ROD(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]+NAIVE_CUT_ROD(p, n-i));
    }
    return q;
}
```

Implementacja wersji naiwnej

```
MEMORIZED_CUT_ROD(int p[], int r[], int s[], int n)
{
    if (r[n]>=0) return r[n];

    int q;

    if(n==0)
    {
        q=0;
        s[0]=0;
    } else
    {
        q=INT_MIN;
        for(int i=1; i<=n; i++)
        {
            int current =p[i]+MEMORIZED_CUT_ROD(p, r, s, n-i);
            if(current >q )
            {
                q=current;
                s[n]=i; //zapamietuje optymalne ciecie
            }
        }
    }
    r[n]=q;
    return q;
}
```

Implementacja wersji ze spamiętywaniem

```
void PRINT_SOLUTION(int s[], int n)
{
    while(n>0)
    {
        cout << s[n];
        n=n-s[n];
        if(n>0) cout << " + ";
    }
    cout << endl;
}
```

Funckja do odzyskiwania rozwiązań

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

    //dla kazdej dlugosci szuka maksymalny zysk
    for(int i=1; i<=n; i++)
    {
        int q=INT_MIN;

        //sprawdzanie mozliwych pierwszch ciec
        for(int j=1; j<=i; j++)
        {
            if(q<p[j] + r[i-j])
            {
                q=p[j]+r[i-j];
                s[i]=j;
            }
        }
        r[i]=q;
    }
    return r[n];
}
```

Implementacja wersji iteracyjnej

2.3 Testy i ich analiza

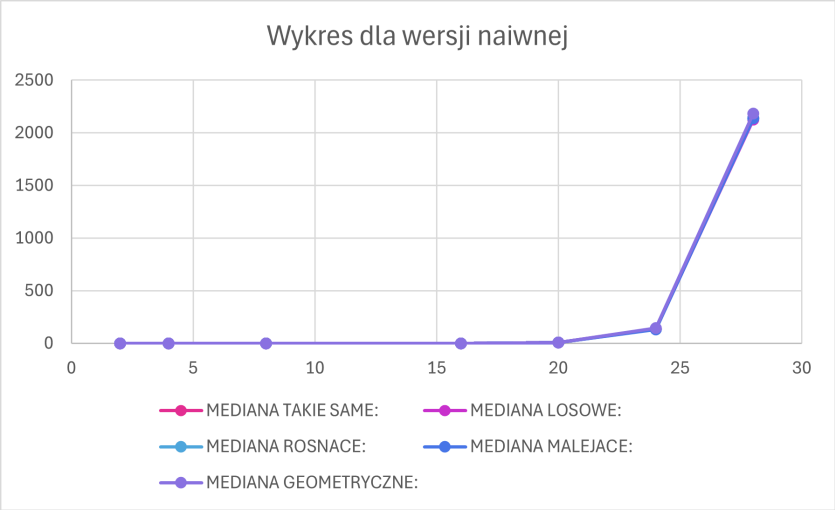
2.3.1 Opis testów

Będziemy mierzyć czas wykonania w milisekundach dla różnych długości pręta. Dla naiwnej będą to [2, 4, 8, 16, 20, 24, 28], ograniczona z powodu złożoności. Dla wersji ze spamiętywaniem [2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384], a dla iteracyjnej dodatkowo [32768, 65536]. Test każdej długości zostanie wykonany dla 5 rodzajów cen. Gdy każdy fragment ma taką samą cenę (10), losową (od 1 do 50), rosnącą liniowo, malejącą liniowo oraz rosnącą wykładniczo. Każdy pomiar zostanie wykonany po 10 razy, a następnie do analizy pobieramy medianę z wyników. W testach nie uwzględniamy odzyskiwania rozwiązań.

2.3.2 Analiza testów wersji naiwnej

	2	**4**	**8**	**16**	**20**	**24**	**28**
MEDIANA TAKIE SAME:	0	0	0	0	9	138	2137,5
MEDIANA LOSOWE:	0	0	0	0	8	134	2126,5
MEDIANA ROSNACE:	0	0	0	0	7,5	134	2137,5
MEDIANA MALEJACE:	0	0	0	0	8	132	2136,5
MEDIANA GEOMETRYCZNE:	0	0	0	0	8	143	2185

Tabela 1: Mediana czasów wykonania (w ms) naiwnej wersji algorytmu Cut Rod dla różnych długości pręta

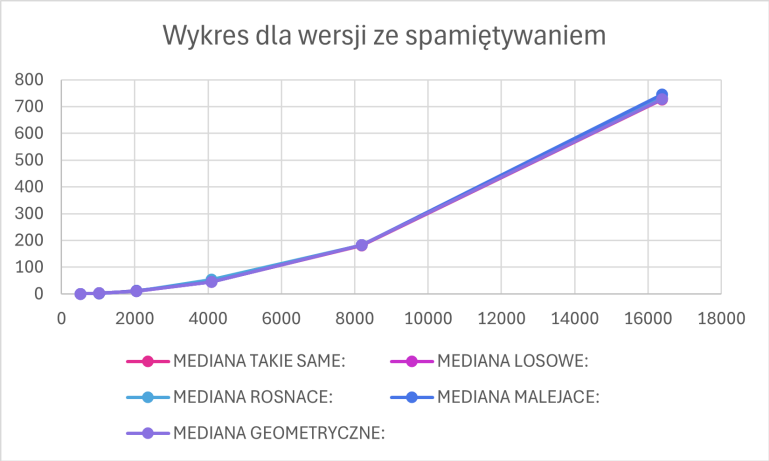


Dla małych długości (do 16) czas jest zerowy niezależnie od typu danych, co świadczy o szybkości algorytmu dla małych danych. Widoczny jest gwałtowny wzrost od długości 20. Dalszy brak istotnych różnic między typami danych wejściowych. Największa różnica pojawia się przy długości 28 dla wzrostu wykładniczego (geometrycznego), jest on o około 50ms dłuższy niż pozostałe. Stąd wynika że czas wykonania zależy głównie od długości pręta. Z uwagi na złożoność $O(2^n)$ ciężko jest przeprowadzić testy dla większych długości.

2.3.3 Analiza testów wersji ze spamiętywaniem

	512	1024	2048	4096	8192	16384
MEDIANA TAKIE SAME:	0	2	12	45	181,5	725,5
MEDIANA LOSOWE:	0	3	11	45	182	730,5
MEDIANA ROSNACE:	0	2,5	11	53	182	732
MEDIANA MALEJACE:	0	2	12	45,5	182	744,5
MEDIANA GEOMETRYCZNE:	0	3	10,5	45	183	728,5

Tabela 2: Mediana czasów wykonania (w ms) w wersji ze spamiętywaniem Cut Rod dla różnych długości pręta

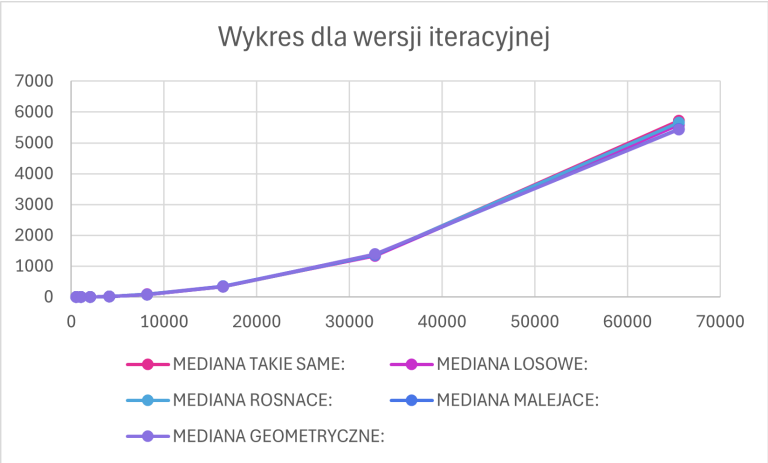


Dla długości poniżej 512 czas jest zerowy niezależnie od typu danych. Widoczny wzrost jest dopiero od długości 1024. Dalszy brak istotnych różnic między typami danych. Wyraźnie widać złożoność $O(n^2)$.

2.3.4 Analiza testów wersji iteracyjnej

	512	1024	2048	4096	8192	16384	32768	65536
MEDIANA TAKIE SAME:	0	1	5,5	20,5	83,5	334	1354	5721
MEDIANA LOSOWE:	0	0	5	21,5	95	352	1336	5578,5
MEDIANA ROSNACE:	0	0,5	5	21	84,5	339,5	1369	5661
MEDIANA MALEJACE:	0	2	5	21	83	344	1382	5452,5
MEDIANA GEOMETRYCZNE:	0	0,5	5	21	83	336	1387,5	5434

Tabela 3: Mediana czasów wykonania (w ms) w wersji iteracyjnej Cut Rod dla różnych długości pręta



Podobnie jak poprzednio, poniżej długości 512 czas jest zerowy niezależnie od typów danych. Powolny wzrost zaczyna się przy długości 1024. Dalszy brak istotnych różnic między typami danych. Wyraźnie widać złożoność $O(n^2)$.

2.3.5 Wnioski testów algorytmu Cut Rod

Niezależnie od rodzaju cen nie widać znaczących różnic między typami danych wejściowych. Ze względu na wzrost wykładniczy najwolniejsza jest wersja naiwna. Najszybsza za to jest wersja iteracyjna, dla której można było przeprowadzić testy nawet dla długości o 65 tysiącach. Dla małych danych wejściowych zarówno wersja iteracyjna jak i ze spamiętywaniem jest dobrym wyborem.

3 LCS

3.1 Opis działania algorytmu

Algorytm Longest Common Subsequence, polega na znalezieniu najdłuższego wspólnego podciągu spośród podanych ciągów znaków, które występują w tej samej kolejności. Elementy podciągów nie muszą przy tym leżeć obok siebie. W obydwu wersjach niżej używanych odzyskujemy rozwiązania. Poza długością takiego podciągu dostajemy jeszcze jak on wygląda.

3.1.1 Wersja rekurencyjna ze spamietywaniem

Wersja rekurencyjna ze spamiętywaniem oblicza LCS porównując ostatnie znaki prefiksów. Jeśli są równe dodaje 1 do wyniku dla krótszych ciągów, w przeciwnym razie wybiera maksimum z odrzucenia znaku z X lub Y. Dodatkowo używa spamiętywania żeby nie liczyć tych samych podproblemów wielokrotnie. Złożoność obliczeniowa to $O(n * m)$.

3.1.2 Wersja iteracyjna

Wersja iteracyjna wypełnia dwuwymiarową tablicę w porządku od lewej do prawej i od góry do dołu, dla każdej pary pozycji sprawdza czy odpowiadające znaki ciągów są równe. Jeśli tak to bierze wartość z przekątnej plus 1. W przeciwnym razie wybiera większa wartość z góry lub z lewej. Złożoność obliczeniowa to także $O(n * m)$.

3.2 Fragmenty kodu

```
1
2       if (X[i-1] == Y[j-1])
3       {
4           result = LCS_RECURSIVE(X, Y, c, b, i-1, j-1) + 1;
5           b[i][j] = '\\';
6       }
7       else
8       {
9           // oblciza dwie mozliwosci
10          int up = LCS_RECURSIVE(X, Y, c, b, i-1, j);
11          int left = LCS_RECURSIVE(X, Y, c, b, i, j-1);
12
13          // wybiera lepsza
14          if (up >= left)
15          {
16              result = up;
17              b[i][j] = '^';
18          } else
19          {
20              result = left;
21              b[i][j] = '<';
22          }
23      }
```

Główny fragment kodu wersji rekurencyjnej

```
1       for(int i=1; i<=m; i++)
2       {
3           for (int j=1; j<=n; j++)
4           {
5               //znaki sie zgadzaja
6               if(X[i-1]==Y[j-1])
7               {
8                   //dlugosc lcs rosnie o 1
9                   c[i][j] = c[i-1][j-1]+1;
10              } else
11              //znaki sie roznia
12              if(c[i-1][j]>= c[i][j-1])
13              {
14                  //wieksza wartosc jest z gory
15                  c[i][j]=c[i-1][j];
16              } else
17              {
18                  //wieksza wartosc jest z lewo
19                  c[i][j] = c[i][j-1];
```

```
20                }
21            }
22        }
```

Główna pętla wersji iteracyjnej

3.3 Testy i ich analiza

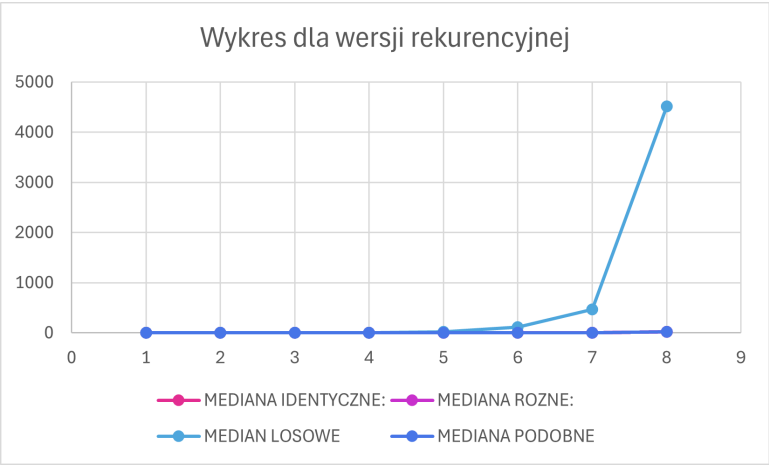
3.3.1 Opis testów

Będziemy mierzyć czas wykonania w milisekundach dla różnych długości ciągów. Zakładamy że oba ciągi mają te samą długość i mają w sobie litery małego alfabetu od a do z. sprawdzane długości to [16, 32, 64, 128, 256, 512, 1024, 2048,(tutaj koniec dla rekurencyjnej), 4096, 8192, 16384, 32768, 65536] Test dla każdej długości zostanie wykonany dla różnych rodzajów danych wejściowych. Sprawdzane rodzaje to: dwa losowe ciągi, dwa identyczne ciągi, dwa całkowicie różne oraz dwa częściowo podobne do siebie (w 50%). Każdy pomiar zostanie wykonany po 10 razy, a następnie do analizy pobierzemy medianę z wyników. W testach nie uwzględniamy odzyskiwania rozwiązań.

3.3.2 Analiza testów wersji rekurencyjnej

	16	**32**	**64**	**128**	**256**	**512**	**1024**	**2048**
MEDIANA IDENTYCZNE:	0	0	0	0	0,5	1	4	22,5
MEDIANA ROZNE:	0	0	0	0	0,5	1	4	22,5
MEDIAN LOSOWE	0	0	1	6	24,5	111	468,5	4513
MEDIANA PODOBNE	0	0	0	0	0	0	4	22

Tabela 4: Mediana czasów wykonania (w ms) w wersji rekurencyjnej LCS dla różnych długości ciągów

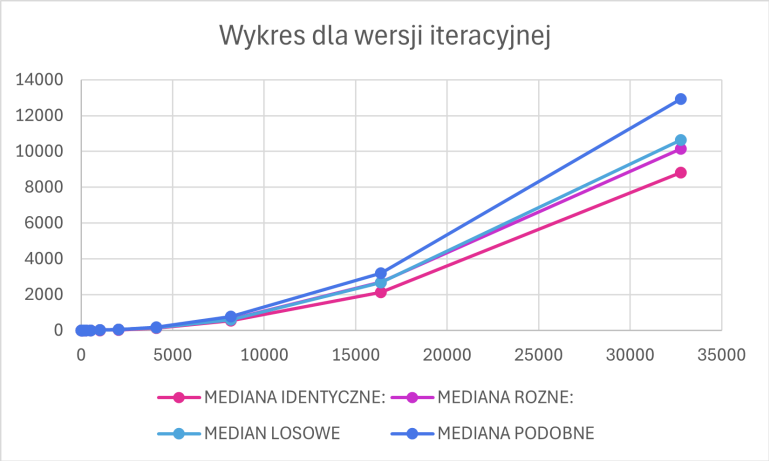


Dla długości poniżej 128 czas jest zerowy. Dopiero od długości 256 widzimy powolny wzrost. Największą różnice w wynikach testu widzimy dla przypadku gdzie obydwa ciągi były generowane losowo. Tutaj widzimy szybki wzrost wykładniczy, widzimy dokładną złożoność $O(2^n)$. Poza tym czasy w innych przypadkach są do siebie zbliżone.

3.3.3 Analiza testów wersji iteracyjnej

	64	**128**	**256**	**512**	**1024**	**2048**	**4096**	**8192**	**16384**	**32768**
MEDIANA IDENTYCZNE:	0	0	0	1	5,5	25	115	531	2121,5	8802,5
MEDIANA ROZNE:	0	0	0	1	8	34,5	138,5	617	2686	10151
MEDIAN LOSOWE	0	0	0	1	9	41,5	147,5	596	2684,5	10635,5
MEDIANA PODOBNE	0	0	0	2	11	46	178	764,5	3193	12913

Tabela 5: Mediana czasów wykonania (w ms) w wersji iteracyjnej LCS dla różnych długości ciągów



W przypadku wersji iteracyjnej nie ma znaczących odchyłów. Dla długości 256 i poniżej czas jest zerowy. Natomiast dla większych można zaobserwować wyraźną złożoność $O(n^2)$. Najszybciej algorytm wykonuje się dla ciągów identycznych, a najdłużej dla częściowo podobnych.

3.3.4 Wnioski testów algorytmu LCS

Na podstawie przeprowadzonych algorytmów możemy zauważyć, że jedynie dla wersji rekurencyjnej widzimy znaczenie wprowadzanych danych wejściowych. Ciągi losowe mają zdecydowanie dłuższy czas wykonywania niż inne rodzaje. W przypadku wersji iteracyjnej nie ma to dużego znaczenia. O ile nasze ciągi nie są losowe to każda wersja algorytmu będzie działać stosunkowo szybko.

4 Activity Selector

4.1 Opis działania algorytmu

Problem wyrobu zajęć polega a wybraniu jak największej liczby wzajemnie kompatybilnych zajęć. Każde zajęcie ma czas rozpoczęcia i czas zakończenia. Dwa zajęcia są kompatybilne jeżeli ich przedziały czasowe się nie nakładają. Jest to algorytm zachłanny, któryw każdym kroku wybiera najlepsze rozwiązania w danym momencie. Później zaimplementujemy również alternatywne rozwiązanie tego problemu z wykorzystaniem programowania dynamicznego.

4.1.1 Wersja rekurencyjna

Zakładamy że dane posortowane są rosnąco według czasu zakończenia. Wersja rekurencyjna szuka pierwszego zajęcia, które nie koliduje z poprzednimi. Następnie patrzy czy zajęcie zaczyna się po zakończeniu ostatniego wybranego. Jeżeli nie to je pomija, a jeżeli tak to wybiera je. Dodaje do rozwiązania i rekurencyjnie szuka kolejnych. Algorytm zwraca maksymalną liczbę zajęć które można wybrać bez kolizji czasowych.

4.1.2 Wersja iteracyjna

Zakładamy że dane posortowane są rosnąco według czasu zakończenia. W wersji iteracyjnej rozpoczynamy od wybrania pierwszego zajęcia z listy. Następnie przechodzimy przez wszystkie pozostałe i dla każdego sprawdzamy czy jego czas rozpoczęcia jest większy niż czas zakończenia ostatnio wybranego zajęcia. Jeżeli warunek jest spełniony to zwiększamy licznik i nowo wybrane zajęcie staje się naszym ostatnio wybranym. Algorytm zwraca maksymalną liczbe zajęć, która można wybrać bez kolizji czasowych.

4.1.3 Wersja zmodyfikowana

Poprzednie dwie wersje zakładały, że dane posortowane są rosnąca po czasie zakończenia. W wersji zmodyfikowanej sortujemy dane rosnąco ale po czasie rozpoczęcia. Do tej modyfikacji weźmiemy algorytm rekurencyjny. Podczas gdy standardowa rekurencja wybiera pierwsze zajęcia, które nie kolidują, tak zmodyfikowana przegląda wszystkie zajęcia i wybiera to które kończy się najwcześniej spośród niekolidujących.

4.1.4 Wersja z programowaniem dynamicznym

W programowaniu dynamicznym rozważane są wszystkie możliwości, rozbija problemy na mniejsze podproblemy a następnie zapamiętuje ich rozwiązania w tablicy. Zajęcia także są posortowane według czasu zakończenia rosnąco. Podczas gdy algorytm zachłanny wybiera lokalnie optymalne rozwiązanie, tak programowanie dynamiczne rozważa wszystkie możliwości i wybiera globalnie optymalne.

4.2 Fragmenty kodu

```
int RECURSIVE_ACTIVITY_SELECTOR(int s[], int f[], int k, int n)
{
    int m=k+1;
    //szuka pierwszego zajecia ktore nie koliduje
    while(m<=n && s[m]<f[k])
    {
        m++;
    }
    //jesli znalezlismy zajecie
    if(m<=n)
    {
        return 1+ RECURSIVE_ACTIVITY_SELECTOR(s,f,m,n);
    }
    return 0;
}
```

Funckja w wersji rekurencyjnej

```
int ACTIVITY_SELECTOR(int s[], int f[], int n)
{
    //zaczynamy od 1 wiec nie potrzebujemy f[0]=int_min i s[0]=0
    int wynik = 1; //liczba wybranych zajec
    int k=1; //ostatnie wybrane zajecie

    for(int m=2; m<=n; m++)
    {
        if(s[m]>=f[k])
        {
            wynik++;
            k=m;
        }
    }
    return wynik;
}
```

Funkcja w wersji iteracyjnej

```
int MOD_ACTIVITY_SELECTOR(int s[], int f[], int k, int n)
{
    //szukamy zajecia ktore nie koliduje z ostatnim wybranym
    int indeks=-1;
    int czas = INT_MAX;
    for(int i=k+1; i<=n; i++)
    {
        //czy zajecie zaczyna sie po zakonczeniu ostatniego
    wybranego
        if(s[i]>=f[k])
        {
            //czy zajecie konczy sie wczesniej niz dotychczasowe
    wybrane
            if(f[i] < czas)
            {
                czas=f[i];
                indeks=i;
            }
```

```
        }
    }
    if(indeks!=-1)
    {
        return 1+MOD_ACTIVITY_SELECTOR(s,f,indeks,n);
    }
    return 0;
}
```

Funkcja w wersji zmodyfikowanej

```
    //wlaczanie biezacego zajecia
    int wybieram=1;
    if(nie!=-1)
    {
        wybieram+=dp[nie];
    }
    //nie wlaczanie biezacego zajecia
    int niewybieram=dp[i-1];
    //wybiera i zpamaietuje lepsza opcje
    if(wybieram>niewybieram)
    {
        dp[i]=wybieram;
    } else
    {
        dp[i]=niewybieram;
    }
```

Fragment funkcji w programowaniu dynamicznym

4.3 Testy i ich analiza

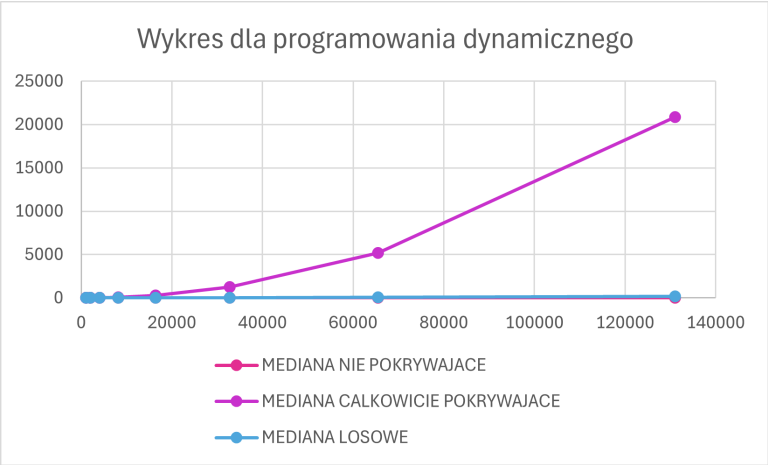
4.3.1 Opis testów

W testach nie bierzemy pod uwagę wersji zmodyfikowanej. Mierzymy czas wykonania algorytmu (w ms) dla różnej liczby aktywności: [16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536, 131072]. Testujemy programy dla 3 rodzajów danych wejściowych, na zajęciach które się nie pokrywają, na zajęciach które się pokrywają całkowicie, a także na losowych czasach rozpoczęcia i zakończenia. Wykonujemy po 10 pomiarów dla każdej wielkości, każdego rodzaju a następnie wyciągamy medianę,

4.3.2 Analiza testów w wersji z programowaniem dynamicznym

	512	**1024**	**2048**	**4096**	**8192**	**16384**	**32768**	**65536**	**131072**
M NIE POKRYWAJACE	0	0	0	0	0	0	0	0	1
M CALKOWICIE POKRYWAJACE	0	1	5	19	76	304	1260,5	5200,5	20866,5
M LOSOWE	0	0	0	0	0	3	13	51	199

Tabela 6: Mediana czasów wykonania (w ms) w wersji z programowaniem dynamicznym dla różnych rodzajów zajęć



Czas dla wielkości poniżej 1024 jest zerowy. Dopiero od 2048 widzimy rosnące wartości dla całkowicie pokrywających się zajęć. Jest to najgorszy przypadek dla tego algorytmu. W przypadku losowych zajęć też widzimy wzrost, ale bardzo powolny. Dla różnych ma on złożoność czasową $O(n^2)$, natomiast dla losowych coś pomiędzy $O(n)$, a $O(n^2$. Najlepszy przypadek to własnie $O(n)$ dla nie pokrywającej się.

4.3.3 Analiza testów dla wersji iteracyjnej i rekurencyjnej

W przypadku wersji iteracyjnej i rekurencyjnej wszystkie pomiary dały czas milisekund równy 0. Wynika to najprawdopodobniej z nieuwzględnienia czasu sortowania w pomiarach. Wtedy czas wykonania algorytmu zależałby od wybranego sortowania do ułożenia danych wejściowych. Jeżeli założymy, że dane wejściowe od początku byłyby podawane rosnąco to również by nam ten czas odpadł. Algorytm dla każdych wersji danych ma złożoność $O(n)$. Zatem niezależnie od ilości podanych zajęć będzie on się wykonywał bardzo szybko.

4.3.4 Wnioski testów algorytmu Activity Selector

Na podstawie przeprowadzonych testów widzimy ze wersja z programowaniem dynamicznym jest znacznie gorsza niż rekurencyjna czy iteracyjna. Ma o wiele dłuższy czas wykonywania, zwłaszcza dla danych losowych. Dla tych dwóch wersji jest ona zerowa, dowolną ilośc danych jest w stanie przetworzyć w bardzo szybkim tempie i podać oczekiwane rezultaty.

5 Huffman

5.1 Opis działania algorytmu

Algorytm Huffmana służy do kompresji bezstratnej danych. Tworzy optymalny kod prefiksowy, w którym częściej występujące znaki mają krótsze kody, a rzadziej występujące dłuższe kody.

5.1.1 Wersja klasyczna

Wersja klasyczna liczy ile razy każdy znak występuje w tekście. Następnie buduje drzewo i po kolei łączy dwa najrzadziej występowanie znaki w jeden tworząc z niego nowy węzeł. Generuje kody, gdzie lewa gałąź drzewa to '0', a prawa to '1'. Potem zamienia ciąg zer i jedynek według kodu z drzewa. Tutaj kolejkę priorytetowa zaimplementowaliśmy jako kopiec binarny korzystając z kodu używanego do Heap Sort z poprzednich list.

5.1.2 Wersja zmodyfikowana

Wersja zmodyfikowana działa tak samo jak klasyczna, jedynie tutaj zamiast kodu binarnego mamy kod ternarny, a zamiast drzewa binarnego buduje drzewo trójkowe.

5.2 Fragmenty kodu

```
1  Node* HUFFMAN(Node* nodes[], int n)
2  {
3      //inicjalizacja kolejki priorytetowej
4      heap = new Node*[n];
5
6      for(int i=0; i<n; i++)
7      {
8          heap[i] = nodes[i];
9      }
10
11     //buduje kopiec
12     BUILD_HEAP(heap, n);
13
14     //drzewo huffmana
15     for(int i=1; i<=n-1; i++)
16     {
17         Node* x = EXTRACT_MIN(heap); //najmniejsza czestotliwosc
18         Node* y = EXTRACT_MIN(heap); //kolejny z najmniejsza
19         Node* z = createNode('\0', x->freq + y->freq); //suma
       czestotliwosci dzieci
20         z->left = x;    // x lewe dziecko
21         z->right = y;   // y prawe dziecko
22
23         //wstawianie nowego wezla spowrotem do kopca
24         HEAP_INSERT(heap, z);
25     }
26     //zwracanie korzenia drzewa
```

```
27      Node* root = EXTRACT_MIN(heap);
28
29      return root;
30  }
```

Główna funkcja algorytmu Huffmana

5.3 Testy i ich analiza

Zostały przeprowadzone testy, które nie dały żadnych ciekawych rezultatów. Zakładamy że nasz teksty do kodowania tworzony był z małych liter alfabetu. Nawet dla rozmiarów 33mln+ czas wykonania wynosił 0. Z tego wynika że algorytm ten jest bardzo szybki, a różne rodzaje częstotliwości danych liter w tekście nie wpływa na czas działania.

6 Podsumowanie

Na podstawie przeprowadzonych powyżej testów można łatwo zauważyć, które wersje algorytmów opłaca się stosowac, a których raczej warto unikać. W przypadku Cut Rod najwolniejsza jest wersja naiwna, a najszybsza iteracyjna. Dla algorytmu LCS o ile nasze ciągi nie są losowe to każda wersja algorytmu będzie działac w miarę sprawnie. Jeżeli chodzi o Activity Selector to zdecydowanie warto unikać wersji z programowania dynamicznego. Natomiast Huffman niezaleznie od testowanej wersji zawsze koduje dane szybko.