sprawozdanie lista 1

Jakub Knaspki

November 2025

1 Wprowadzenie

Sprawozdanie przedstawia analizę trzech algorytmów: CUT ROD (problem cięcia pręta), LCS (najdłuższy wspólny podciąg) oraz ACTIVITY SELECTOR (wybór zajęć). Dla każdego algorytmu zaimplementowano różne wersje w celu porównania ich efektywności.

2 Algorytm CUT ROD

Problem cięcia pręta polega na znalezieniu maksymalnego dochodu z sprzedaży pręta o długości n, znając ceny kawałków o różnych długościach.

2.1 Algorytm Naiwny

Algorytm naiwny wykorzystuje podejście czysto rekurencyjne bez żadnej optymalizacji. Dla każdego możliwego cięcia pręta dokonuje pełnego przeszukiwania wszystkich możliwych podziałów, co prowadzi do wielokrotnego rozwiązywania tych samych podproblemów. W efekcie charakteryzuje się wykładniczą złożonością czasową $O(2^n)$, co czyni go praktycznie nieużytecznym dla większych wartości n. Podspodem widac właśnie ten algorytm

```
int Naive_Cut_Rod(int p[], int n) {
 if (n==0){
        return 0;
    }
 int q =INT_MIN;
    for (int i=1;i<=n;i++) {
  int Nwart=p[i-1]+Naive_Cut_Rod(p,n-i);
  if (Nwart>q)
  {
            q=Nwart;
        }
 }
 return q;
```

2.2 Algorytm ze spamiętywaniem

Wersja ze spamiętywaniem jest strukturalnie bardzo podobna do podejścia naiwnego, ponieważ również wykorzystuje rekurencję. Kluczową różnicą jest dodanie tablicy pomocniczej, która przechowuje już obliczone wyniki podproblemów. Dzięki temu każde podproblem jest rozwiązywane tylko raz, co bardzo zmniejsza złożoność czasową do $O(n^2)$.

2.3 Algorytm Iterujacy

Algorytm iteracyjny (od dołu do góry). W przeciwieństwie do rekurencyjnego ze spamiętywaniem, podejście iteracyjne rozwiązuje problem w sposób całkowicie iteracyjny, zaczynając od najmniejszych podproblemów i stopniowo budując rozwiązanie aż do docelowej długości pręta. W algorytmie tym kluczową rolę odgrywa wektor `s[]`, który przechowuje optymalne pierwsze cięcie dla każdej długości, umożliwiając późniejsze odtworzenie całego rozwiązania. Najistotniejszą częścią algorytmu jest wewnętrzna pętla, która dla każdej długości `j` przegląda wszystkie możliwe pierwsze cięcia `i` ,i wybiera to, które maksymalizuje sumę ceny kawałka o długości `i` oraz optymalnego rozwiązania dla pozostałej części `j-i`:

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

To właśnie ten fragment bezpośrednio realizuje zasadę optymalności, gwarantując znalezienie optymalnej wartości oraz zapamiętanie wyboru prowadzącego do niej.

2.4 Porównanie

Dla małej listy [2,6,11,12]

```
Naiwny
13
w tym algorytmie masz 49 przypisan
w tym algorytmie masz 54 porownan

 Spamietywanie
13
1 3
w tym algorytmie masz 40 przypisan
w tym algorytmie masz 43 porownan

 Iteracjia
13
1 3
w tym algorytmie masz 37 przypisan
w tym algorytmie masz 32 porownan
```

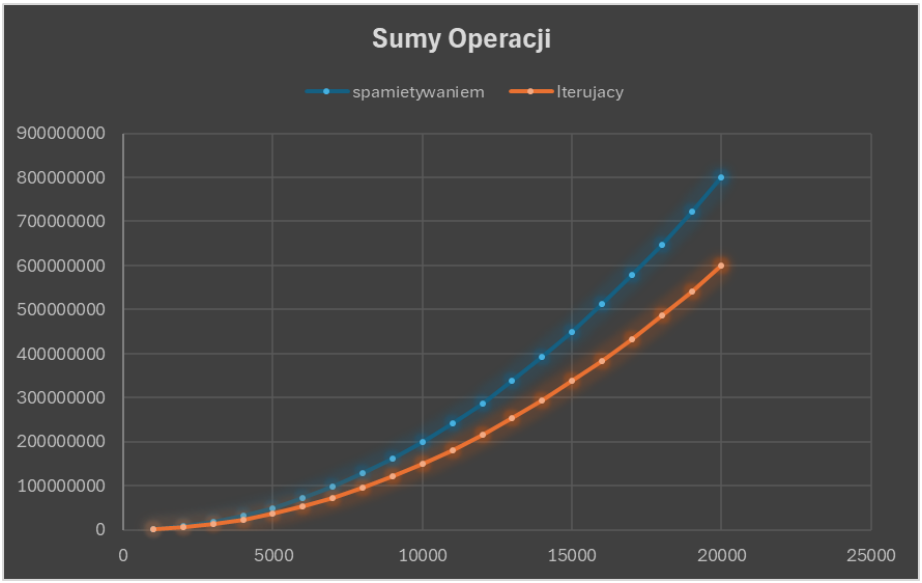
Dla tego rozmiaru problemu algorytm naiwny nie wykazuje jeszcze ekstremalnej nieefektywności. Natomiast widać już dużą różnicę dla [2, 6, 11, 12, 20, 24]:

```
    Naiwny
24
w tym algorytmie masz 205 przypisan
w tym algorytmie masz 222 porownan

 Spamietywanie
24
6
w tym algorytmie masz 73 przypisan
w tym algorytmie masz 79 porownan

 Iteracjia
24
6
w tym algorytmie masz 63 przypisan
w tym algorytmie masz 57 porownan
```

Widoczny jest znaczący wzrost liczby operacji w algorytmie naiwnym przy większych danych. Wersje zoptymalizowane wykonują zdecydowanie mniej operacji, z wersją iteracyjną osiągającą najniższe wartości.



Rysunek 1: Różnica pomiędzy algorytmem iteracyjnym a wersją ze spamiętywaniem

Obie zoptymalizowane wersje algorytmu CUT ROD działają w złożoności $O(n^2)$ i przechowują wyniki podproblemów w tablicy pomocniczej. Różnica polega na organizacji obliczeń: wersja ze spamiętywaniem używa rekurencji i oblicza wartości „na żądanie", podczas gdy wersja iteracyjna wypełnia tablicę systematycznie, bez narzutu rekurencyjnego. Eliminacja tego narzutu przekłada się na mniejszą liczbę operacji i lepszą wydajność, co widać na załączonym wykresie (rys. X) – dla każdego badanego rozmiaru danych (0–25000) suma operacji w wersji iteracyjnej jest niższa, a różnica rośnie wraz ze skalowaniem problemu.

3 Algorytm LCS (Najdłuższy Wspólny Podciąg)

Algorytm LCS (ang. *Longest Common Subsequence*) służy do wyznaczenia najdłuższego podciągu, który występuje w obu zadanych ciągach znaków. Podciąg, w przeciwieństwie do podsłowa, nie musi składać się z kolejnych znaków — dopuszczalne są przerwy. Algorytm ma szerokie zastosowanie, m.in. w systemach kontroli wersji, porównywaniu sekwencji DNA czy analizie tekstu.

W ramach sprawozdania zaimplementowano dwie wersje algorytmu: **rekurencyjną ze spamiętywaniem** oraz **iteracyjną** opartą na programowaniu dynamicznym. Obie wersje nie tylko obliczają długość najdłuższego wspólnego podciągu, ale również — dzięki odpowiedniemu śledzeniu decyzji — odtwarzają jego konkretną postać. W kolejnych podrozdziałach szczegółowo opisano działanie każdej z implementacji oraz zaprezentowano wyniki eksperymentów.

3.1 Wersja rekurencyjna ze spamiętywaniem (RLCS)

Algorytm rekurencyjny ze spamiętywaniem oblicza długość najdłuższego wspólnego podciągu dla dwóch ciągów X i Y, zaczynając od ich pełnych długości. W tablicy `c2` przechowywane są już obliczone wyniki dla par indeksów (i, j), aby uniknąć wielokrotnego rozwiązywania tych samych podproblemów. Dodatkowo w tablicy `b2` zapisywane są kierunki, które później umożliwiają odtworzenie samego podciągu.

Najistotniejszy fragment algorytmu stanowi część rekurencyjna, w której decydujemy, co zrobić, gdy znaki na pozycjach $i-1$ i $j-1$ są równe lub różne:

```
if(X[i-1] == Y[j-1]) {
    int temp = RLCS(X, Y, i-1, j-1, c2, b2) + 1;
    c2[i][j] = temp;
    b2[i][j] = '\\';
    return temp;
} else {
    int wart1 = RLCS(X, Y, i-1, j, c2, b2);
    int wart2 = RLCS(X, Y, i, j-1, c2, b2);
    if(wart1 >= wart2) {
        c2[i][j] = wart1;
        b2[i][j] = '|';
        return wart1;
    } else {
        c2[i][j] = wart2;
        b2[i][j] = '-';
        return wart2;
    }
}
```

Gdy znaki są identyczne, długość LCS zwiększa się o 1 i przechodzimy do podproblemów z pominięciem tych znaków (indeksy $i-1$, $j-1$). W przeciwnym

razie wybieramy lepszy wynik spośród dwóch możliwości: pominięcie znaku z ciągu X (indeksy $i-1$, j) lub pominięcie znaku z ciągu Y (indeksy i, $j-1$). Znak zapisany w tablicy `b2` wskazuje, która ścieżka została wybrana, co jest kluczowe dla późniejszego odtworzenia rozwiązania.

3.2 Wersja iteracyjna (ILCS)

Wersja iteracyjna algorytmu LCS opiera się na programowaniu dynamicznym. Tworzy dwie tablice: `c` do przechowywania długości LCS dla prefiksów ciągów oraz `b` do zapamiętywania kierunków (podobnie jak w wersji rekurencyjnej). Tablica `c` jest wypełniana w sposób systematyczny, zaczynając od najmniejszych prefiksów.

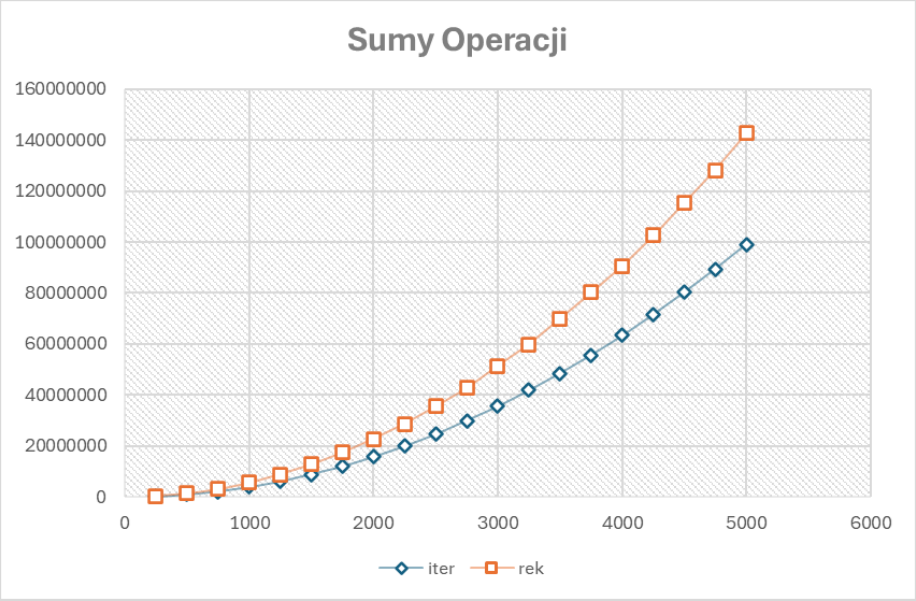
Kluczowy fragment kodu odpowiada za wypełnianie tablic `c` i `b`:

```
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] = '\\';
        } else {
            if(c[i-1][j] >= c[i][j-1]) {
                c[i][j] = c[i-1][j];
                b[i][j] = '|';
            } else {
                c[i][j] = c[i][j-1];
                b[i][j] = '-';
            }
        }
    }
}
```

Dla każdej pary indeksów (i, j) sprawdzamy równość znaków.

Jeśli są one takie same, to wartość `c[i][j]` jest ustalana na podstawie `c[i-1][j-1]` zwiększonej o 1. W przeciwnym przypadku wybieramy większą wartość spośród `c[i-1][j]` i `c[i][j-1]`. Kierunek zapisany w `b[i][j]` pozwala na późniejsze odtworzenie podciągu za pomocą funkcji `Print_Sol`.

Należy zwrócić uwagę, że wersja iteracyjna nie korzysta z rekurencji, przez co ma mniejszy narzut czasowy i jest bardziej przyjazna dla dużych danych wejściowych. Złożoność czasowa obu wersji wynosi $O(m \cdot n)$.



Rysunek 2: Enter Caption

3.3 Przyrównanie dwóch Algorytmów

Wykres porównuje sumy operacji wersji iteracyjnej (niebieskie romby) i rekurencyjnej ze spamiętywaniem (pomarańczowe kwadraty) algorytmu LCS dla ciągów do 5000 znaków. Obie krzywe rosną zgodnie z złożonością $O(m \cdot n)$, jednak wersja iteracyjna wykonuje mniej operacji, a różnica ta zwiększa się dla większych danych. Potwierdza to wyższą efektywność podejścia iteracyjnego.

4 Algorytm Activity Selector

Algorytm Activity Selector służy do wyboru maksymalnego podzbioru wzajemnie kompatybilnych aktywności (nie nakładających się czasowo) ze zbioru dostępnych zajęć. W klasycznym wariancie zakłada się, że aktywności są posortowane według czasu zakończenia, co umożliwia zastosowanie strategii zachłannej. W ramach niniejszej pracy zaimplementowano modyfikację tego algorytmu, która operuje na danych wejściowych posortowanych względem czasu rozpoczęcia. Zaprezentowano trzy podejścia: rekurencyjne, iteracyjne oraz uproszczone programowanie dynamiczne. Każda z wersji została szczegółowo opisana w poniższych podrozdziałach.

4.1 Wersja rekurencyjna (ActivitySelectorR)

Wersja rekurencyjna realizuje strategię "dziel i zwyciężaj". Po posortowaniu aktywności rosnąco według czasu rozpoczęcia, algorytm rozpoczyna od ostatniego elementu (o najpóźniejszym początku) i rekurencyjnie poszukuje poprzedniej aktywności, która kończy się przed rozpoczęciem obecnej. Kluczowy fragment odpowiedzialny za to wyszukiwanie i rekurencyjne wywołanie przedstawiono poniżej:

```
int m = k-1;
while (m >= 0 && tab[m].k > tab[k].s) {
    m--;
}
if (m >= 0) {
    wyniki[licznik] = tab[m].id;
    licznik++;
    ActivitySelectorR(tab, m, wyniki, licznik);
}
```

Pętla `while` przesuwa indeks `m` w lewo, tak długo, jak aktywność `tab[m]` kończy się po rozpoczęciu aktywności `tab[k]`. Gdy znajdzie kompatybilną aktywność (lub dojdzie do początku tablicy), dodaje jej identyfikator do tablicy wyników i rekurencyjnie kontynuuje proces od znalezionego indeksu. Jest to implementacja zachłannego wyboru działająca "wstecz".

4.2 Wersja iteracyjna (ActivitySelectorI)

Wersja iteracyjna realizuje dokładnie tę samą logikę co rekurencyjna, jednak w sposób od końca do początku, używając prostej pętli. Eliminuje to narzut związany z wywołaniami rekurencyjnymi. Kod rozpoczyna od wybrania ostatniej aktywności i iteracyjnie przeszukuje tablicę w poszukiwaniu poprzedniej kompatybilnej.

```
int licznik = 0;
int ostatni_idx = n-1;
wyniki[licznik] = tab[ostatni_idx].id;
licznik++;
for (int m = n-2; m >= 0; m--) {
    if (tab[m].k <= tab[ostatni_idx].s) {
        wyniki[licznik] = tab[m].id;
        licznik++;
        ostatni_idx = m;
    }
}
```

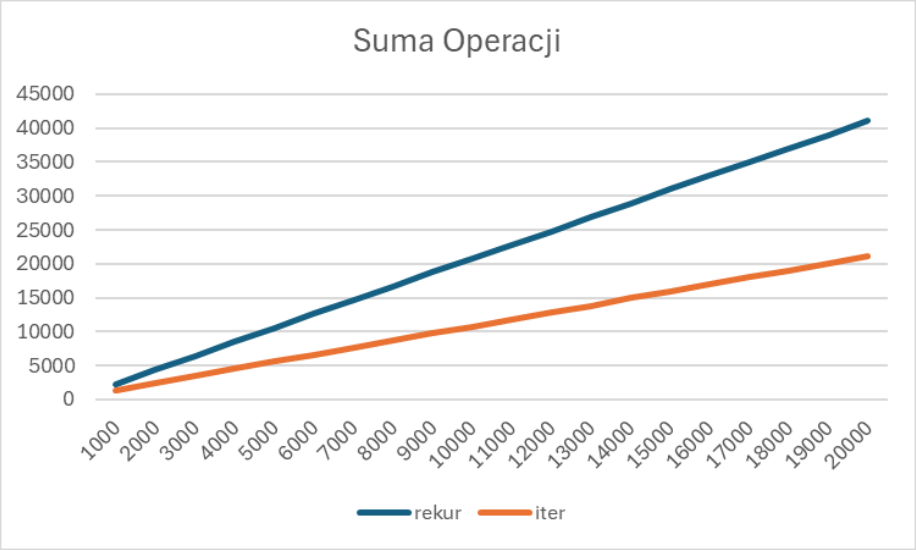
Warunek `if` sprawdza, czy aktywność `tab[m]` kończy się przed rozpoczęciem ostatnio wybranej (`tab[ostatni_idx]`). Jeśli tak, dodaje ją do wyników i uaktualnia indeks ostatnio wybranej. W ten sposób algorytm iteracyjnie buduje rozwiązanie od końca.

4.3 Wersja z programowaniem dynamicznym

Dla celów porównawczych zaimplementowano także uproszczony wariant oparty na programowaniu dynamicznym. W tej wersji nie odtwarza się dokładnego zestawu aktywności, a jedynie oblicza maksymalną możliwą ich liczbę.

```
for (int i = 1; i < n; i++) {
    for (int j = 0; j < i; j++) {
        if (tab[j].k <= tab[i].s) {
            if (dt[j] + 1 > dt[i]) {
                dt[i] = dt[j] + 1;
            }
        }
    }
}
```

Tablica `dt` przechowuje maksymalną liczbę aktywności, które można wybrać kończąc na pozycji `i`. Dla każdej pary `(j, i)`, gdzie `j < i`, jeśli aktywność `j` jest kompatybilna z `i`, algorytm sprawdza, czy wybór ścieżki przez `j` zwiększa maksymalną liczbę aktywności kończącą się na `i`. Złożoność tego podejścia wynosi $O(n^2)$, co czyni je mniej wydajnym od wersji zachłannych działających w czasie $O(n)$.



Rysunek 3: Enter Caption

4.4 Porównanie

Podobnie jak w problemie CUT ROD, uproszczona wersja dynamiczna działa w czasie $O(n^2)$ i jest niepraktyczna dla dużych danych. Wykres porównuje sumy operacji dla wersji iteracyjnej (pomarańczowa) i rekurencyjnej (niebieska) w zakresie 1000–20000 aktywności. Obie krzywe rosną liniowo ($O(n)$), potwierdzając efektywność strategii zachłannej. Wersja iteracyjna wykonuje nieco mniej operacji, co wynika z braku narzutu rekurencyjnego. W praktyce obie wersje są szybkie, przy czym iteracyjna jest odrobinę wydajniejsza.