Raport AiSD lista 3

Amelia Dorożko

8 stycznia 2025

1 Problem podziału pręta

1.1 CUT_ROD

Funkcja CUT_ROD jest prostą implementacją algorytmu rekurencyjnego, który rozwiązuje problem cięcia pręta na mniejsze kawałki w sposób "naivny", bez użycia żadnej optymalizacji pamięci.

1.2 MEMORIZED_CUT_ROD

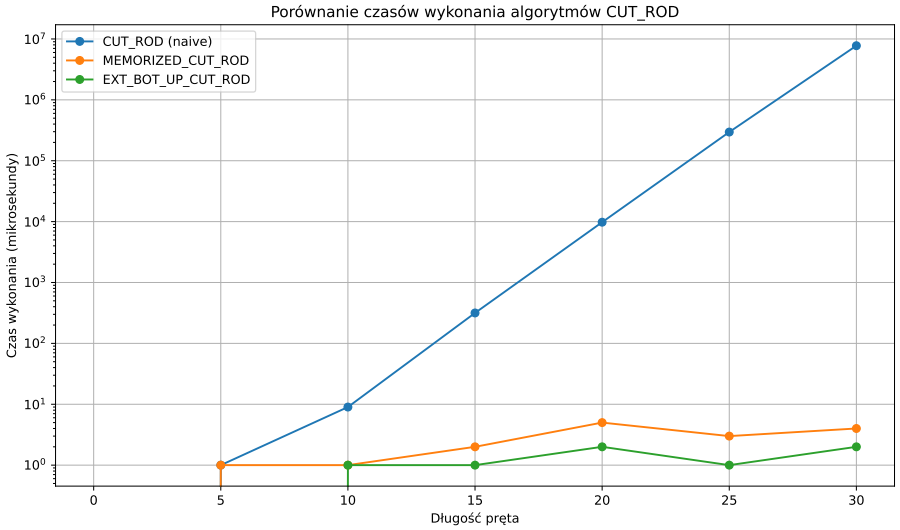
Funkcja MEMORIZED_CUT_ROD jest ulepszoną wersją funkcji rekurencyjnej, w której zastosowano memoizację. Celem jest unikanie powtarzających się obliczeń, które pojawiają się w naiwnej rekurencji. Zapamiętujemy już obliczone wyniki dla danego n i wykorzystujemy je, gdy są potrzebne ponownie. Chcąc odzyskać rozwiązanie przy użyciu tego algorytmu, korzystamy z poniższej funkcji PRINT_MEMORIZED_CUT_ROD.

```
1         void PRINT_MEMORIZED_SOLUTION(int p[], int n) {
2             int* r = new int[n + 1];
3             fill(r, r + n + 1, -1); // Inicjalizacja tablicy
4
5             int maxProfit = MEMORIZED_CUT_ROD(p, r, n);
6             cout << "Maksymalny zysk: " << maxProfit << endl;
7
8             cout << "Podzial: ";
9             while (n > 0) {
10                for (int i = 1; i <= n; i++) {
11                    if (r[n] == p[i-1] + r[n - i]) {
12                        cout << i << " ";
13                        n -= i;
14                        break;
15                    }
16                }
17            }
18            cout << endl;
19            delete[] r;
20        }
```

1.3 EXT_BOT_UP_CUT_ROD

Algorytm `EXT_BOT_UP_CUT_ROD` działa na zasadzie programowania dynamicznego. Wykorzystuje podejście bottom-up, co oznacza, że rozwiązanie problemu budowane jest od najprostszych przypadków, aż do przypadków bardziej złożonych. Algorytm przechodzi od najmniejszych problemów (pręt o długości 0) do głównego problemu (pręt o długości n), rozwiązując kolejne podproblemy i przechowując wyniki w tablicach, co pozwala uniknąć ich ponownego obliczania. Jego główną zaletą jest możliwość rekonstrukcji rozwiązania, dzięki czemu możemy uzyskać nie tylko optymalny zysk, ale także konkretne cięcia, które prowadzą do tego zysku. Chcąc odzyskać optymalne rozwiązanie, wykorzystujemy funkcję `PRINT_EXT_SOLUTION`, która korzysta z dwóch tablic zysku i optymalnego podziału.

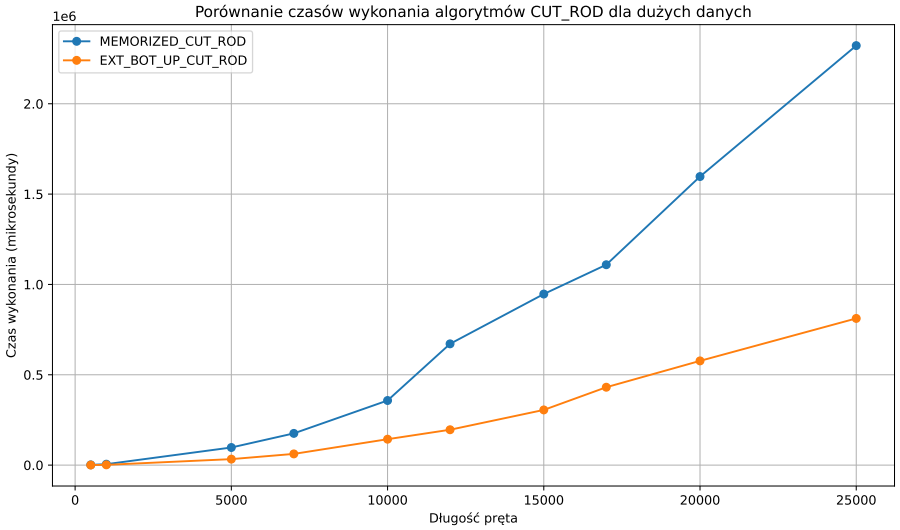
```
1         void PRINT_EXT_SOLUTION(int p[], int n) {
2             int* r = new int[n + 1];
3             int* s = new int[n + 1];
4
5             EXT_BOT_UP_CUT_ROD(p, r, s, n);
6
7             cout << "Maksymalny zysk: " << r[n] << endl;
8
9             cout << "Podzial: ";
10            while (n > 0) {
11                cout << s[n] << " ";
12                n -= s[n];
13            }
14            cout << endl;
15
16            delete[] r;
17            delete[] s;
18        }
```



Z analizy wynika, że algorytm `EXT_BOT_UP_CUT_ROD` działa efektywniej niż pozostałe przy większych danych.

Długość pręta	MEMORIZED_CUT_ROD	EXT_BOT_UP_CUT_ROD
500	1496	372
1000	5143	1546
5000	97467	33144
7000	175612	62039
10000	357622	143427
12000	671226	195770
15000	946803	305454
17000	1109070	431027
20000	1597139	576887
25000	2321849	811880

Tabela 1: Porównanie czasu działania algorytmów, dla danych o większym zakresie (mikrosekundy).



Pomimo podobnej złożoności obliczeniowej, algorytmy `MEMORIZED_CUT_ROD` i `EXT_BOT_UP_CUT_ROD` różnią się w sposób implementacji. W `MEMORIZED_CUT_ROD` rekurencja wiąże się z wyższym narzutem pamięciowym i czasowym. Z kolei `EXT_BOT_UP_CUT_ROD` eliminuje rekurencję, opierając się na podejściu iteracyjnym, co zmniejsza wymagania pamięciowe i poprawia zarządzanie przestrzenią.

Jeśli chodzi o wydajność, dla mniejszych danych różnica jest minimalna, jednak dla większych rozmiarów danych `EXT_BOT_UP_CUT_ROD` działa szybciej, ponieważ nie obciąża systemu rekurencją. Dodatkowo, algorytm bottom-up lepiej radzi sobie z większymi problemami i jest bardziej efektywny pod względem pamięciowym.

2 Najdłuższy wspólny podciąg

2.1 Rekurencyjny algorytm LCS (REC_LCS)

Algorytm rekurencyjny `REC_LCS` opiera się na zasadzie podziału problemu na mniejsze podproblemy. Dla dwóch ciągów wejściowych $s1$ i $s2$ o długościach odpowiednio m i n, najdłuższy wspólny podciąg można obliczyć rekurencyjnie w następujący sposób:

- Jeśli $m = 0$ lub $n = 0$, najdłuższy wspólny podciąg ma długość 0.
- Jeśli ostatnie znaki $s1[m-1]$ i $s2[n-1]$ są takie same, długość LCS jest równa $1 + LCS(m-1, n-1)$.
- W przeciwnym razie, długość LCS to $\max(LCS(m-1, n), LCS(m, n-1))$.

Aby poprawić efektywność, funkcja wykorzystuje technikę memoizacji, przechowując wyniki wcześniejszych wywołań w dynamicznie alokowanej tablicy $b[m][n]$. `REC_LCS` jest prostym i intuicyjnym do implementacji algorytmem, mimo tego, wymaga dużej liczby rekurencyjnych wywołań, co może czynić go nieefektywnym, ze względu przepełnienia stosu.

2.2 Iteracyjny algorytm LCS (IT_LCS)

Iteracyjny algorytm (IT_LCS) korzysta z dynamicznego programowania. Tablica $c[m+1][n+1]$ jest używana do przechowywania długości LCS dla wszystkich możliwych podciągów. Podejście to wypełnia tablicę wiersz po wierszu, stosując następujące zasady:

- Jeśli $X[i-1] = Y[j-1]$, to $c[i][j] = c[i-1][j-1] + 1$.
- W przeciwnym przypadku $c[i][j] = \max(c[i-1][j], c[i][j-1])$.

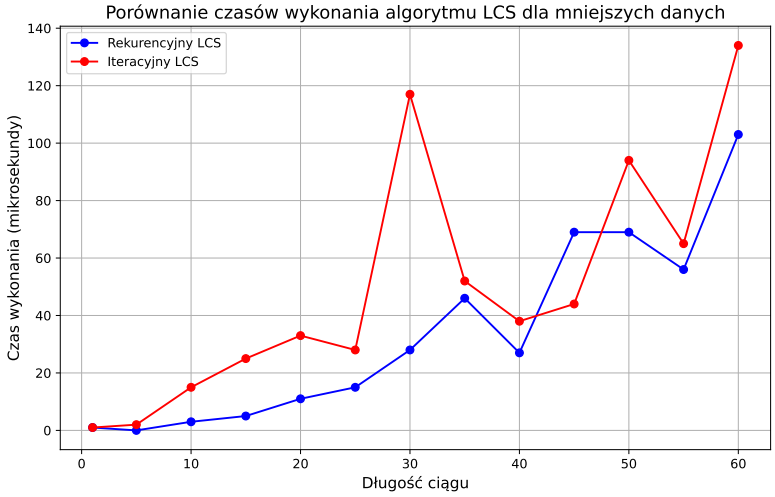
Dodatkowa tablica $b[m][n]$ przechowuje informacje o kierunku, skąd pochodzi optymalne rozwiązanie (`'
'`, `'|'`, `'-'`).

Dla dużych danych wejściowych algorytm iteracyjny działa znacznie szybciej niż rekurencyjny, co czyni go bardziej odpowiednim dla zastosowań praktycznych.

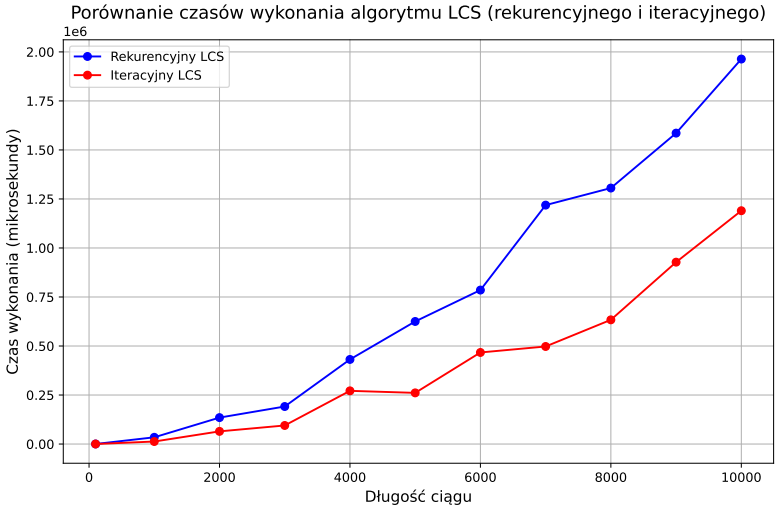
Długość ciagów	Czas rekurencyjny (mikrosekundy)	Czas iteracyjny (mikrosekundy)
100	281	246
1000	34384	12813
2000	134854	64630
3000	191414	94692
4000	431470	271382
5000	625166	260978
6000	785324	467062
7000	1218602	497589
8000	1305638	633539
9000	1585806	927642
10000	1963625	1190042

Tabela 2: Porównanie czasów wykonania algorytmu rekurencyjnego i iteracyjnego LCS dla różnych długości ciągów.

Dla małych danych algorytm `REC_LCS` może być szybszy, ponieważ wymaga mniejszej pamięci, nie musi tworzyć dużych tablic, a sama liczba obliczeń jest wystarczająco mała, by działać efektywnie. Co przedstawia wykres dla mniejszego zakresu napisów.



Z kolei dla większych danych jego wydajność spada, gdyż nie używa pamięci do zapisywania wyników pośrednich, co prowadzi do wielu powtórzeń obliczeń. W takich przypadkach algorytm iteracyjny jest bardziej efektywny.



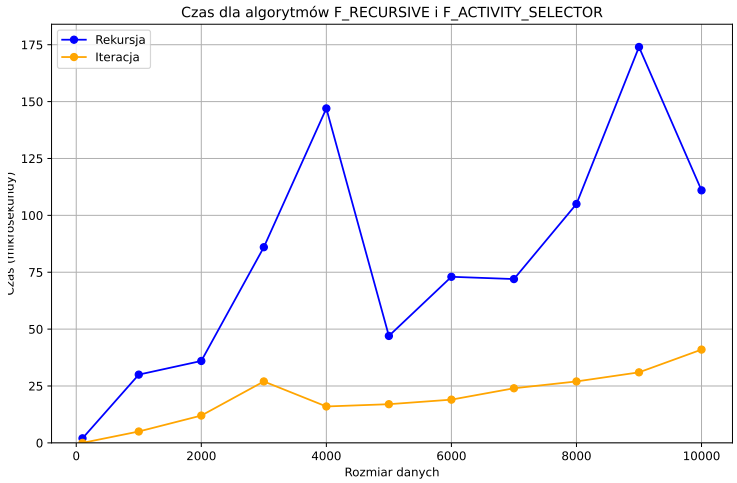
3 Problem wyboru zajęć

3.1 Algorytmy z posortowaną tablicą `f[]`

Funkcje `F_RECURSIVE_ACTIVITY_SELECTOR` oraz `F_ACTIVITY_SELECTOR` działają dla posortowanej tablicy zakończenia zajęć `f[]`. Te algorytmy selekcjonują możliwe zajęcia, które mogą się odbyć w określonym czasie, bazując na porównaniu czasów rozpoczęcia z czasami zakończenia.

Funkcja `F_RECURSIVE_ACTIVITY_SELECTOR` używa rekurencji. Początkowy indeks `k` jest równy 0. Algorytm zaczyna iterację od kolejnego możliwego zajęcia `s[m]`, sprawdzając, czy jego czas rozpoczęcia jest większy lub równy czasowi zakończenia zajęcia `f[k]`. Jeśli tak, to dane zajęcia są wybierane, a rekurencja kontynuuje dla `m`.

Funkcja `F_ACTIVITY_SELECTOR` działa iteracyjnie. Zaczynamy od pierwszego zajęcia (indeks 1), zapamiętujemy je jako wybrane. W kolejnych krokach algorytm sprawdza, czy czas rozpoczęcia zajęcia `s[m]` jest większy lub równy czasowi zakończenia zajęcia `f[k]`. Jeśli tak, to zajęcia są wybierane, a indeks `k` zostaje zaktualizowany.



3.2 Algorytmy z posortowaną tablicą `s[]`

Funkcje `S_RECURSIVE_ACTIVITY_SELECTOR` oraz `S_ACTIVITY_SELECTOR` działają dla posortowanej tablicy rozpoczęcia zajęć `s[]`. Te algorytmy selekcjonują zajęcia, które mogą się rozpocząć po zakończeniu wcześniejszych zajęć.

Funkcja `S_RECURSIVE_ACTIVITY_SELECTOR` działa rekurencyjnie. Rozpoczyna iterację od ostatniego zajęcia i sprawdza, czy czas zakończenia zajęcia `f[m]` jest większy od czasu rozpoczęcia zajęcia `s[k]`. Jeśli tak, to indeks `m` zostaje zaktualizowany, a rekurencja kontynuuje.

Funkcja `S_ACTIVITY_SELECTOR` działa iteracyjnie. Zaczynamy od ostatniego zajęcia (indeks n-1) i zapamiętujemy je jako wybrane. Algorytm sprawdza, czy czas zakończenia zajęcia `f[m]` jest mniejszy lub równy czasowi rozpoczęcia zajęcia `s[k]`. Jeśli tak, to dane zajęcia są wybierane, a indeks `k` zostaje zaktualizowany.

Przykładowe rozwiązanie z posortowaną tablicą `s[]`

Zajęcie	Czas rozpoczęcia	Czas zakończenia
1	6	15
2	10	18
3	18	28
4	20	23
5	30	34
6	33	42
7	37	45
8	45	55
9	46	53

Tabela 3: Dostępne zajęcia z ich czasami rozpoczęcia i zakończenia.

Wynik algorytmów

Metoda rekurencyjna

Wybrane zajęcia (w kolejności wyboru): $2, 4, 5, 7, 9$.

Metoda iteracyjna

Wybrane zajęcia (w kolejności wyboru): $2, 4, 5, 7, 9$.

Dynamiczne rozwiązanie problemu

Funkcja `DYNAMIC_A_S` realizuje algorytm selekcji aktywności dynamicznej, który wybiera maksymalną liczbę aktywności, nie nakładających się na siebie.

Funkcja `DYNAMIC_A_S` korzysta z dwóch tablic:

- `dp`: Tablica przechowująca maksymalną liczbę aktywności do danego momentu.
- `prev`: Tablica przechowująca indeks poprzedniej aktywności w optymalnym rozwiązaniu.

Algorytm iteracyjnie przechodzi przez wszystkie aktywności, korzystając z funkcji `last`, która znajduje ostatnią aktywność nie nakładającą się na aktualną. Wynik jest wyświetlany w postaci wybranych aktywności.

Poniżej przedstawiono kod funkcji `DYNAMIC_A_S`:

```
1
2           int last(int s[], int f[], int i) {
3               for (int j = i - 1; j >= 0; j--) {
4                   if (f[j] <= s[i]) {
5                       return j;
6                   }
7               }
8               return -1;
```

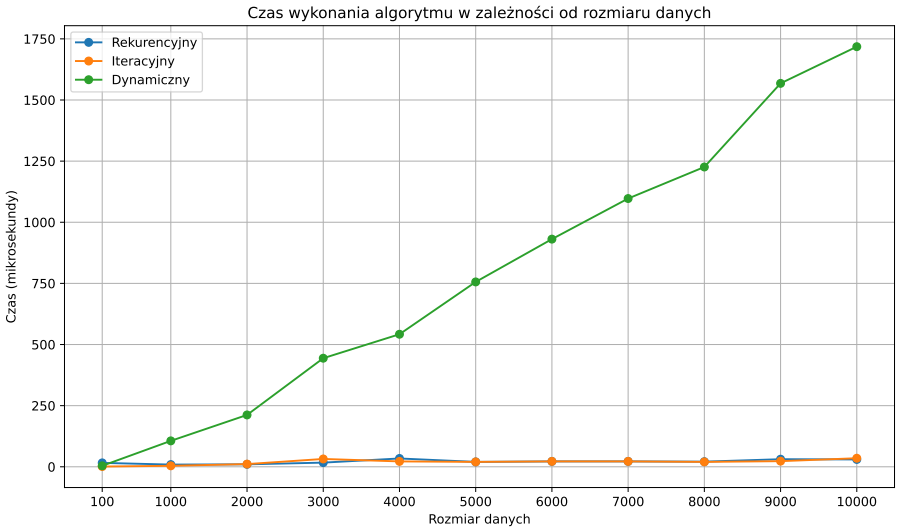
```
}

void DYNAMIC_A_S(int s[], int f[], int n) {
    int dp[n];
    int prev[n];
    dp[0] = 1;
    prev[0] = 0;

    for (int i = 1; i < n; i++) {
        int k = last(s, f, i);
        if (k != -1 && dp[k] + 1 > dp[i - 1]) {
            dp[i] = dp[k] + 1;
            prev[i] = i;
        } else {
            dp[i] = dp[i - 1];
            prev[i] = prev[i - 1];
        }
    }

    cout << "Wybrane zajecia:\n";
    int i = n - 1;
    while (i > 0) {
        if (prev[i] == i) {
            cout << "(" << s[i] << ", " << f[i] << ") ";
            i = last(s, f, i);
        } else {
            i--;
        }
    }
```

Porównując z poprzednimi algorytmami, dynamiczne podejście do problemu nie jest optymalne.



W każdym przypadku algorytmy działają dla posortowanej listy zajęć i do-

konują selekcji zgodnie z określonym kryterium, czy to czasu zakończenia $f[]$ czy rozpoczęcia $s[]$.