Politechnika Wrocławska

Sterowanie Procesami Dyskretnymi
E12-30a

Sprawozdanie 2
Algorytm Witi

Piotr Niedziółka
249023

26 marca 2021

Politechnika Wrocławska

Spis treści

1 Oddanie zadania

Celem ćwiczenia było opracowanie algorytmu witi.

Termin oddania - 26.03.2021 (brak okresu spóźnienia)

Całość można znaleźć również na portalu GitHub: Repozytorium
Pliki można pobrać z Google Drive: Pliki do pobrania

Aby program działał prawidłowo, konieczne jest posiadanie pliku z danymi *dane.txt* w miejscu, gdzie znajduje się program.

2 Opis algorytmu

Problem przedstawiony w tym ćwiczeniu został rozwiązany przy użyciu programowania dynamicznego. Algorytm ma za zadanie zminimalizować sumę kar w zależności od wagi problemu i planowanego jego uruchomienia. Jego złożoność obliczeniowa wynosi $O(2^n)$.

Program tworzy tablicę dynamiczną, a następnie w niej przechowywane są wyniki, które zwraca algorytm. Na koniec tablica jest usuwana, aby nie było wycieków pamięci. Program wypisuje optymalne rozwiązanie oraz ilość operacji, a także czas w jakim wykonał się algorytm.

Algorytm polega na sprawdzanie kosztów podproblemów, wybranie najmniejszej z nich i następnie sprawdzanie możliwości dla najmniejszego kosztu. Tak jest to ponawiane, aż do uzyskania ogólnego problemu - składającego się z mniejszych - gdzie wybierana jest najmniejsza uzyskana wartość, jest to minimalna kara problemu.

3 Kod programu

Poniżej przedstawiono napisany kod:

```
#include <iostream>
#include <fstream>
#include <time.h>
#include <ctime>
#include <algorithm>

using namespace std;

// liczba zadan
int n = 0;
int pomtime;

// do obslugi pliku
string s;
ifstream plik;

// struktura danych (wczytanych plikow)
typedef struct dane {
public:
    int t;  // czas wykonania zadania
    int w;  // wsp  czynnik kary
    int r;  //   dany   termin zako czenia zadania
    int x;  // kolejnosc
} dane;

dane witi[20];
```

```
27
28 // wczytywanie danych
29 void Wczytywanie()
30 {
31     // Wczytywanie z pliku
32     plik.open("data.txt");
33
34     while (s != "data.20")
35     {
36         plik >> s;
37     }
38     plik >> n;
39
40     for (int i = 0; i < n; i++)
41     {
42         witi[i].x = i;
43         plik >> witi[i].t >> witi[i].w >> witi[i].r;
44     }
45
46     plik.close();
47
48     // Wyswietlanie ilosci danych
49     cout << " >>> n = " << n << " <<<\n\n";
50
51     // Wyswietlenie procesow ( danych )
52     for (int i = 0; i < n; i++)
53     {
54         cout << witi[i].x << ") " << witi[i].t << " " << witi[i].w << " " << witi[i].r
    << " " << endl;
55     }
56 }
57
58 void Algorytm()
59 {
60     // pomocnicza liczaca ilosc operacji
61     int s = 0;
62
63     // zmienna przechowuj ca ilosc operacji w postaci
64     // poteg 2, np w przypadku 10 // 2^10
65     int bit = 1 << n;
66
67     // zmienne dynamiczne
68     int* F = new int[bit];
69
70     // Wyzerowanie pierwszego elementu tablicy dynamicznej
71     F[0] = 0;
72
73     // dla wszystkich bit w
74     for (int bitcount = 1; bitcount < bit; bitcount++)  // od 1 do 2^bit ( w przypadku
    10 // 1023)
75     {
76         // liczenie ilosci rozwi za
77         pomtime = 0;
78         for (int i = 0, b = 1; i < n; i++, b *= 2) // b = 1, 2, 4, 8, ..
79         {
80             if (bitcount & b)
81             {
82                 pomtime += witi[i].t;
83             }
84         }
85
86         // ustawienie ostatniej wartosci na maksymaln
87         F[bitcount] = 999999;
88
89         for (int j = 0, b = 1; j < n; j++, b *= 2)
```

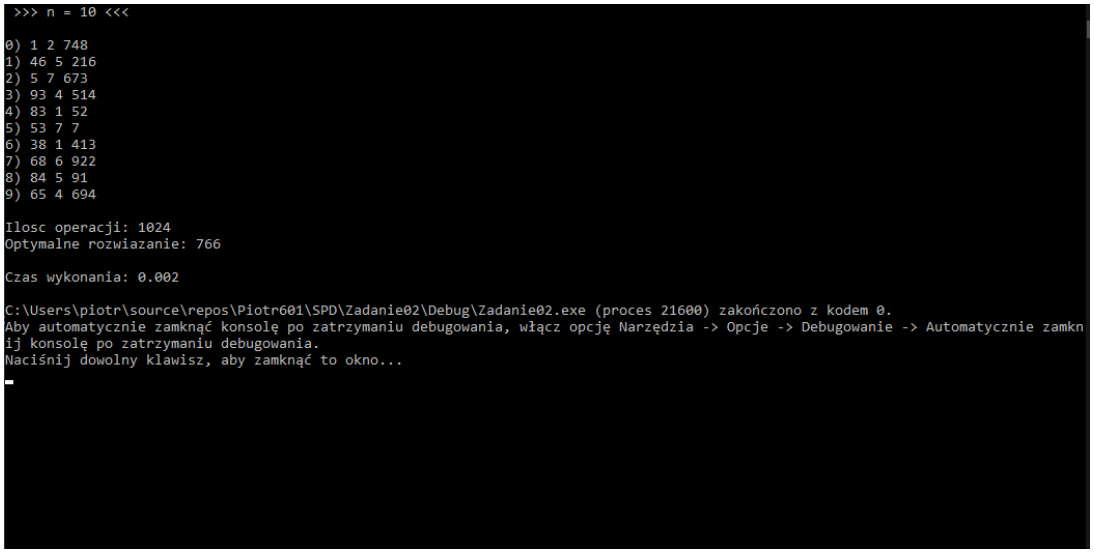
```
        {
            if (bitcount & b)
            {
                // wybranie mniejszej wartosci ( 99999,
                F[bitcount] = min( F[bitcount], F[bitcount - b] + witi[j].w * max(
    pomtime - witi[j].r, 0 ) ) ;

                //if ( F[bitcount] == F[bit - 1] )
                //{
                    //witi[j].x = F[bitcount - b];
                    //cout << witi[j].x << " ";
                //}
            }
        }
    }

    cout << "\nIlosc operacji: " << bit << endl;
    cout << "Optymalne rozwiazanie: " << F[bit - 1] << endl;

    delete[] F;
}


int main()
{
    // WCZYTYWANIE DANYCH
    Wczytywanie();

    // do mierzenia czasu
    int ilosc_powt = 1;
    time_t start, stop;
    double t = 0;

    // ALGORYTM
    for (int i = 0; i < ilosc_powt; i++)
    {
        start = clock();
        Algorytm();
        stop = clock();

        t += (double_t)(stop - start) / CLOCKS_PER_SEC;
    }

    // CZAS WYKONANIA SAMEGO ALGORYTMU
    cout << "\nCzas wykonania: " << t/ilosc_powt << endl;
}
```

Dane do wczytania przez program znajdują się również w repozytorium i mają nazwę *data.txt*.

4 Testy programu

Przeprowadzono testy programu. Wyniki wyszły takie same jak w pliku txt.

Dane 10:



Dane 13:

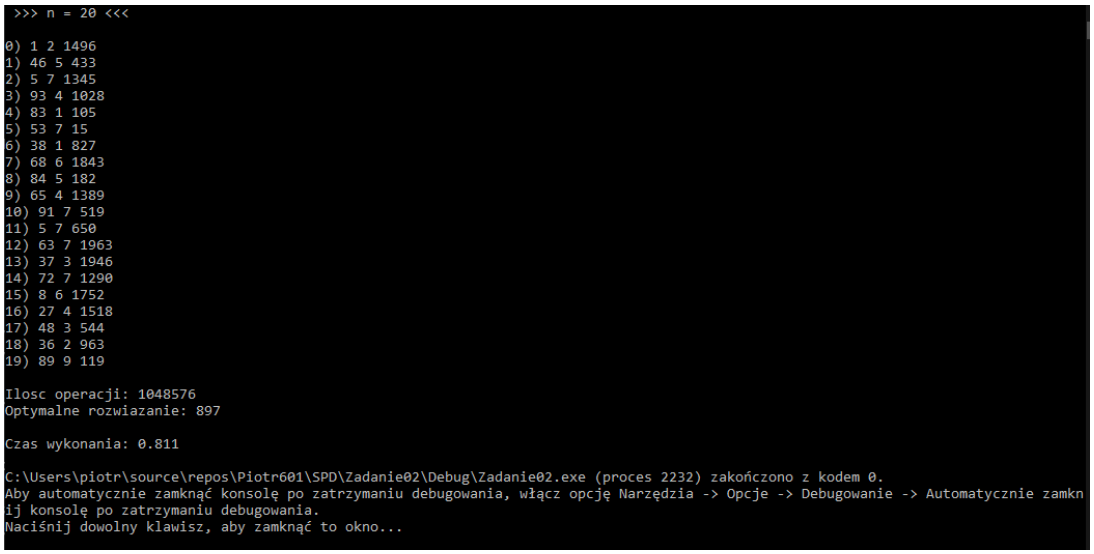
Dane 15:

```
>>> n = 15 <<<

0) 1 2 1122
1) 46 5 325
2) 5 7 1009
3) 93 4 771
4) 83 1 79
5) 53 7 11
6) 38 1 620
7) 68 6 1382
8) 84 5 136
9) 65 4 1041
10) 91 7 390
11) 5 7 487
12) 63 7 1472
13) 37 3 1460
14) 72 7 968

Ilosc operacji: 32768
Optymalne rozwiazanie: 440

Czas wykonania: 0.019

C:\Users\piotr\source\repos\Piotr601\SPD\Zadanie02\Debug\Zadanie02.exe (proces 15228) zakończono z kodem 0.
Aby automatycznie zamknąć konsolę po zatrzymaniu debugowania, włącz opcję Narzędzia -> Opcje -> Debugowanie -> Automatycznie zamkn
ij konsolę po zatrzymaniu debugowania.
Naciśnij dowolny klawisz, aby zamknąć to okno...
```

Dane 17:

```
>>> n = 17 <<<

0) 1 2 1272
1) 46 5 368
2) 5 7 1143
3) 93 4 874
4) 83 1 90
5) 53 7 12
6) 38 1 703
7) 68 6 1566
8) 84 5 154
9) 65 4 1180
10) 91 7 441
11) 5 7 552
12) 63 7 1668
13) 37 3 1654
14) 72 7 1097
15) 8 6 1489
16) 27 4 1290

Ilosc operacji: 131072
Optymalne rozwiazanie: 417

Czas wykonania: 0.093

C:\Users\piotr\source\repos\Piotr601\SPD\Zadanie02\Debug\Zadanie02.exe (proces 18576) zakończono z kodem 0.
Aby automatycznie zamknąć konsolę po zatrzymaniu debugowania, włącz opcję Narzędzia -> Opcje -> Debugowanie -> Automatycznie zamkn
ij konsolę po zatrzymaniu debugowania.
Naciśnij dowolny klawisz, aby zamknąć to okno...
```

Dane 20:



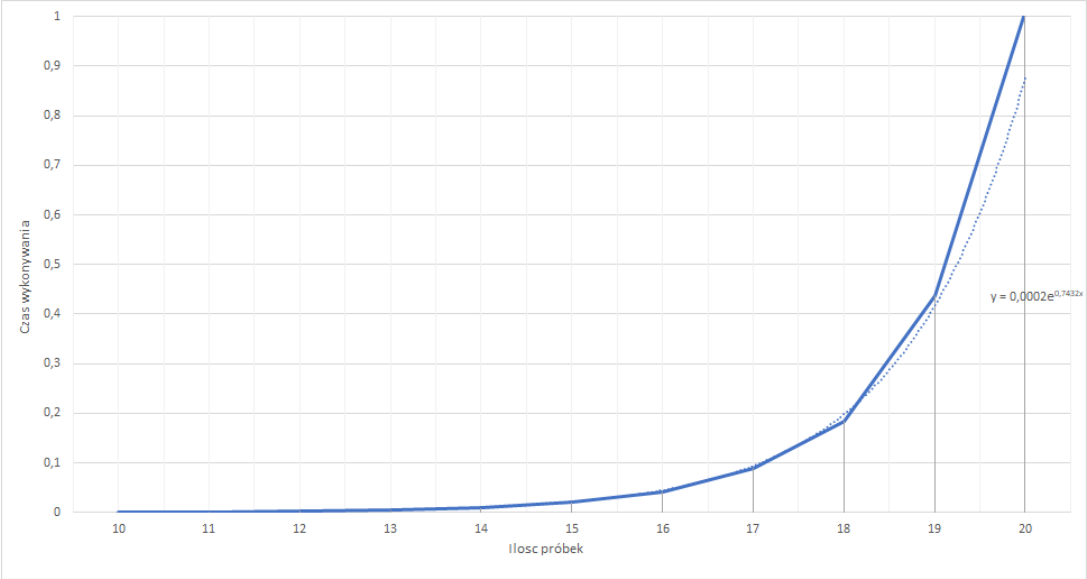
5 Wykresy

Poniżej przedstawiono wykresy utworzono na podstawie wyników otrzymanych z działania algorytmu.

Dla większych dokładności, wszystkie algorytmy włączono w pętli 1000 razy i podzielono przez 1000.

- **n**=10 Czas wykonania: 0.000549
- **n=11** Czas wykonania: 0.001105
- **n=12** Czas wykonania: 0.002448
- **n=13** Czas wykonania: 0.004753
- **n=14** Czas wykonania: 0.009697
- **n=15** Czas wykonania: 0.02008
- **n=16** Czas wykonania: 0.042156
- **n=17** Czas wykonania: 0.087786
- **n=18** Czas wykonania: 0.183759
- **n=19** Czas wykonania: 0.437706
- **n=20** Czas wykonania: 1.007

5.1 Algorytm

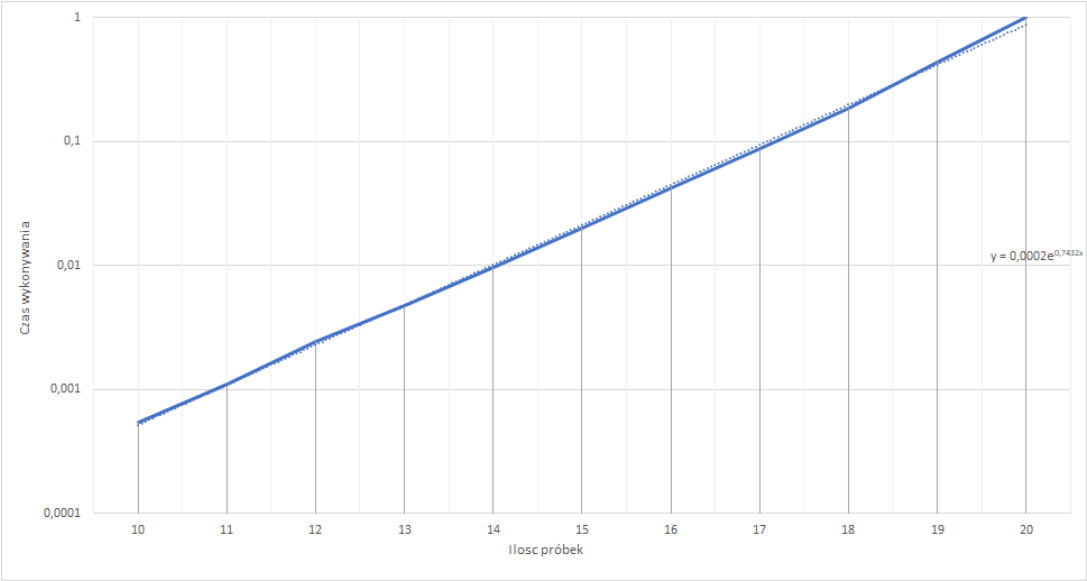


Rysunek 1: Wykres zależności ilości próbek od czasu działania algorytmu

Jak można zauważyć aproksymacja tego wykresu jest wykładniczą linią trendu.

$$y = 0,0002e^{0,7432x}$$

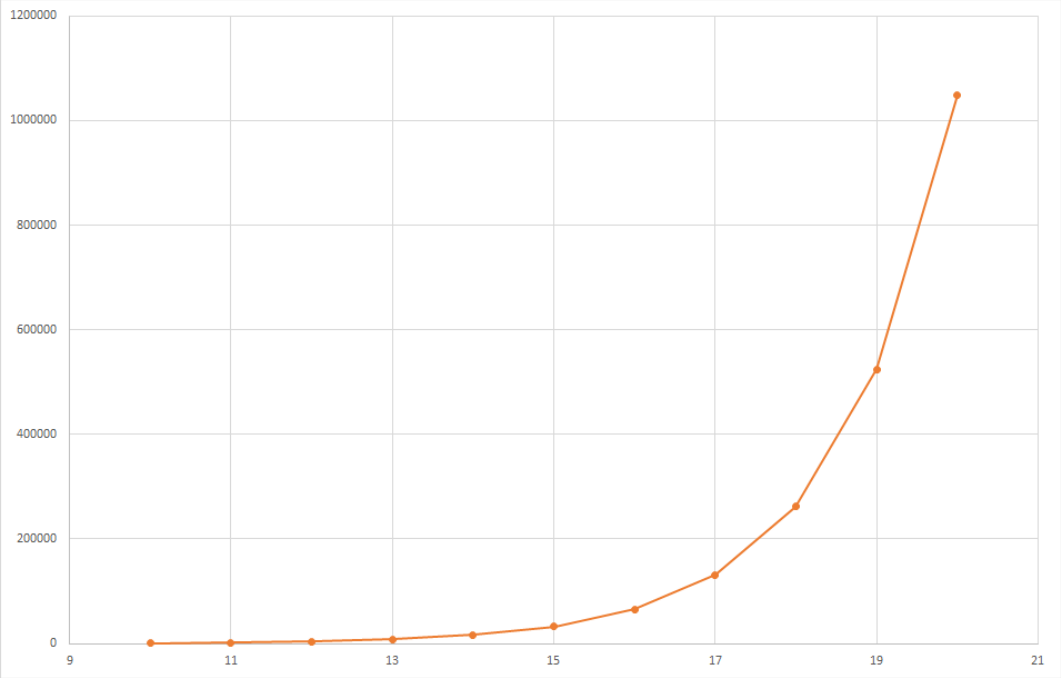
Wykres logarytmiczny również wskazuje na to, że jest to funkcja wykładnicza:



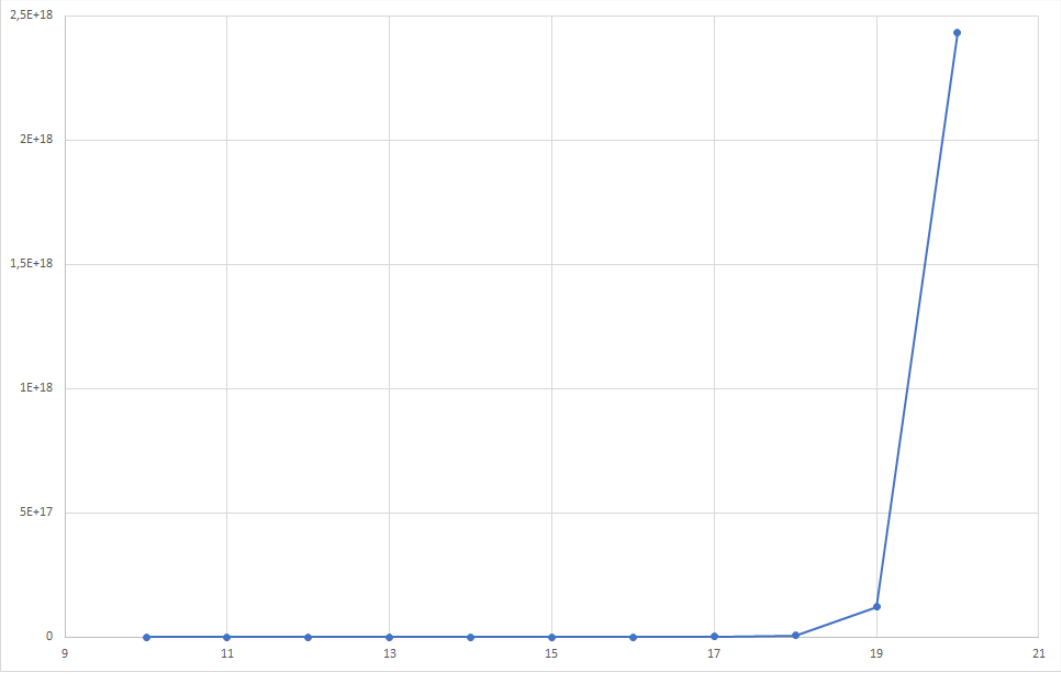
Rysunek 2: Logarytmiczny wykres zależności ilości próbek od czasu działania algorytmu

5.2 Ilość operacji

Poniżej znajdują się wykresy ilości wykonywanych operacji ze względu na ilość próbek:



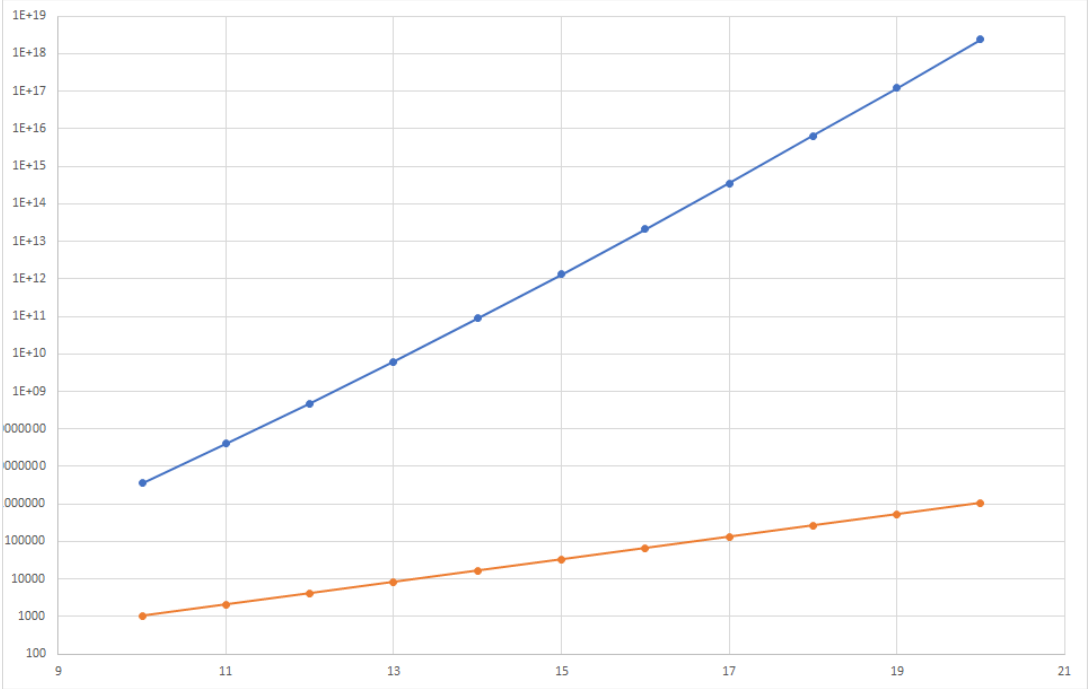
Rysunek 3: Wykres zależności ilości próbek od ilości operacji - 2^n



Rysunek 4: Wykres zależności ilości próbek od ilości operacji - $n!$

5.3 Porównanie

Dla lepszego zobrazowania porównano je na skali logarytmicznej:



Rysunek 5: Logarytmiczn porównanie obydwu zależności - 2^n i $n!$

6 Wyniki, ocena

Przedstawione powyżej wyniki potwierdzają złożoność algorytmu równą $O(2^n)$, a także potwierdzają poprawność algorytmu - wyniki optymalne wychodzą takie same jakie są zawarte w dokumencie z danymi.

Zgodnie z przedstawionymi zasadami proponowana ocena to 3.5 - bez spóźnienia.