

Dokumentacja końcowa projektu z przedmiotu
Analiza Algorytmów[AAL]
Tron Yertle

Paweł Kamiński

15 stycznia 2016

1 Opis problemu

W jakiej kolejności należy ustawić żółwie jeden na drugim, aby utworzyć tron króla Yertle. Każdy z 5607 żółwi ma inną wagę i inną wytrzymałość. Zadaniem jest zbudować najwyższy możliwy stos żółwi.

2 Dane

2.1 Dane wejściowe

Kolekcja danych o parametrach co najwyżej 5607 żółwi (wytrzymałość, waga). Wytrzymałość oznacza ile żółw może udźwignąć włączając w to jego własną wagę.

2.2 Wynik

Liczba całkowita oznaczająca maksymalną liczbę żółwi, jakie można ułożyć na stosie, nie przekraczając przy tym wytrzymałości żadnego z żółwi.

3 Metody rozwiązania problemu

We wszystkich rozwiązaniach Tron Yertle jest układany od góry - najpierw wrzucamy żółwie, które będą na samej górze, a następnie szukamy kolejnych żółwi, które będą w stanie utrzymać już do tej pory ułożony stos.

3.1 Programowanie dynamiczne

3.1.1 Złożoność

Złożoność asymptotyczna wynosi

$$O(n^2) \tag{1}$$

Złożoność dokładna wynosi w przybliżeniu:

$$T(n) = n \log_2(n) + 2n^2 + n \quad (2)$$

3.1.2 Opis

Na samym początku algorytmu żółwie są sortowane. Operator porównania jest zdefiniowany następująco:

```
bool Turtle::operator<(const Turtle& t) const
{
    if (getWeight() == t.getWeight())
        return getStrength() < t.getStrength();
    return getWeight() < t.getWeight();
}
```

Następnie tworzona jest tablica dwuwymiarowa o rozmiarze $N \times N$ (N - liczba dostępnych żółwi), w której są umieszczane aktualnie znalezione optymalne rozwiązania. Celem optymalizacji jest minimalizacja wagi tronu przy jak największej liczbie użytych do jego zbudowania żółwi.

1				
1	3			
1	3	6		
1	3	6	10	
1	3	6	10	15

Rysunek 1: Przykładowa tabela stworzona przez algorytm dla żółwi o wartościach (waga, wytrzymałość): (1,1) (2,3) (3,6) (4,10) (5,15). Widać, że z podanych żółwi możemy zbudować tron o wysokości 5 i wadze 15.

Każda kolejna iteracja algorytmu to kolejny wiersz tabeli. Do każdej kolejnej kolumny tabeli wpisywana jest wartość minimalna z porównania:

$\min(\text{table}[i][j], \text{table}[i-1][j])$

Jeśli żółw może udźwignąć wartość z poprzedniej iteracji, to w aktualne pole wybieramy:

$\min(\text{table}[i][j], \text{table}[i-1][j-1] + \text{turtles}[i].\text{getWeight}())$

gdzie i, j to odpowiednio aktualny wiersz i kolumna, a $\text{turtles}[i]$ to aktualnie rozpatrywany żółw.

3.1.3 Kod

```
table[0][1] = turtles[0].getWeight();
for (unsigned int i = 1; i < N; ++i)
{
    for (unsigned int j = 1; j <= i + 1; ++j)
    {
        table[i][j] = min(table[i][j], table[i - 1][j]);
        if (turtles[i].getCapacity() >= table[i - 1][j - 1])
        {
            table[i][j] = min(table[i][j],
                               table[i - 1][j - 1] + turtles[i].getWeight());
        }
    }
}

// looking for index of last element in last row(height of Throne)
for (unsigned int i = N; i >= 0; --i)
{
    if (table[N - 1][i] != numeric_limits<unsigned int>::max())
    {
        return i;
    }
}
```

3.2 Algorytm naiwny

3.2.1 Złożoność

Złożoność asymptotyczna wynosi

$$O(n^2) \quad (3)$$

Złożoność dokładna wynosi w przybliżeniu:

$$T(n) = n^2 + n \quad (4)$$

3.2.2 Opis

Złożoność Rozwiązanie polega na przejściu w pętli tyle razy ile jest żółwi, za każdym razem wybierając najbliższego żółwia (jeśli żółwie mają taką samą wagę, to wybieramy żółwia z mniejszą wytrzymałością). Znalezionego żółwia następnie umieszczamy na stos pod warunkiem, że jest w stanie go unieść. Jeśli żółw, który jest w danej chwili na szczycie stosu może unieść więcej niż aktualnie znaleziony żółw i da radę unieść także nowego, to zostaje on na szczycie stosu, a pod niego wkłada się nowego żółwia.

3.2.3 Kod

```
// putting first turtle on stack
auto firstTurtle = min_element(turtles.begin(), turtles.end());

// if there are no turtles in vector, return 0
if (firstTurtle == turtles.end())
    return 0;

unsigned int stackWeight = firstTurtle->getWeight();
unsigned int stackHeight = 1;

Turtle lastTurtle = *firstTurtle;

// erase first turtle from vector
turtles.erase(firstTurtle);

for (int i = 0; i < turtles.size(); ++i)
{
    // looking for the lightest turtle
    auto it = min_element(turtles.begin(), turtles.end());

    // if lastTurtle in stack has got better capacity than turtle
    // that we found and new turtle can hold stack -
    // we can maybe swap them on the top of stack
    if (lastTurtle.getCapacity() > it->getCapacity()
        && it->getCapacity() >= stackWeight)
    {
        // if lastTurtle can hold the stack with new turtle
        if(lastTurtle.getStrength() > stackWeight + it->getWeight())
        {
            stackWeight += it->getWeight();
            ++stackHeight;

            // erasing current turtle from vector
            turtles.erase(it);
            continue;
        }
    }
}

// if new turtle can hold current stack - push it on it
if (it->getCapacity() >= stackWeight)
{
    ++stackHeight;
}
```

```

        stackWeight += it->getWeight();
        lastTurtle = *it;
    }

    // erasing current turtle from vector
    turtles.erase(it);
}

```

```
return stackHeight;
```

3.3 Algorytm z presortowaniem

3.3.1 Złożoność

Złożoność asymptotyczna wynosi

$$O(n \log_2(n)) \quad (5)$$

Złożoność dokładna wynosi w przybliżeniu:

$$T(n) = n \log_2(n) + n \quad (6)$$

3.3.2 Opis

Algorytm bardzo przypomina przedstawione powyżej podejście naiwne. Na samym początku kolekcja żółwi jest sortowana według wagi. Następnie algorytm przechodzi po wszystkich żółwiach w pętli i próbuje je wrzucać na stos(sprawdzanie czy można wrzucić na stos jest analogiczne do algorytmu naiwnego).

3.3.3 Kod

```

sort(turtles.begin(), turtles.end());

unsigned int stackWeight = 0, stackHeight = 0;
auto lastTurtle = turtles.begin();
for (auto it = turtles.begin(); it != turtles.end(); ++it)
{
    if (lastTurtle->getCapacity() > it->getCapacity()
        && it->getCapacity() >= stackWeight)
    {
        if (lastTurtle->getStrength() > stackWeight + it->getWeight())
        {
            stackWeight += it->getWeight();
            ++stackHeight;
            continue;
        }
    }
}

```

```

    }

    }
    if (it->getCapacity() >= stackWeight)
    {
        ++stackHeight;
        stackWeight += it->getWeight();
        lastTurtle = it;
    }
}
return stackHeight;

```

4 Moduły źródłowe

Program został podzielony na następujące moduły:

ResultsTable

Klasa tworząca tabelkę z wynikami. Posiada metodę, która po podaniu do niej jako argument jednej z klas dziedziczących po Throne, generuje żółwie, wywołuje algorytm rozwiązujący problem oraz zapamiętuje czas jego wykonania. Funkcja iteruje po kilku rozmiarach problemu i następnie rysuje w konsoli tabelkę.

Throne

Klasa abstrakcyjna przechowująca zbiór żółwi. Umożliwia ich generowanie oraz posiadają metodę do rozwiązania problemu. Funkcja, która rozwiązuje problem jest czysto wirtualna i powinna być dostarczona przez klasy potomne. Klasa implementuje generator żółwi, który generuje je z rozkładu jednostajnego (rozkład został wybrany metodą prób i błędów. Zwracane przez niego dane umożliwiały zbudowanie najwyższej wieży. Rozkład normalny dawał podobne wyniki, ale odrobinę gorsze).

ThroneDynamicProgramming

Klasa dziedzicząca po Throne, rozwiązuje problem za pomocą programowania dynamicznego.

ThroneFast

Klasa dziedzicząca po Throne, rozwiązuje problem w najszybszy sposób.

ThroneNaive

Klasa dziedzicząca po Throne, rozwiązuje problem za pomocą algorytmu naiwnego.

Timer

Singleton, klasa timera odmierzającego czas, wykorzystywanego do

sprawdzenia czasu wykonania algorytmów. Czas jest odliczany w nanosekundach.

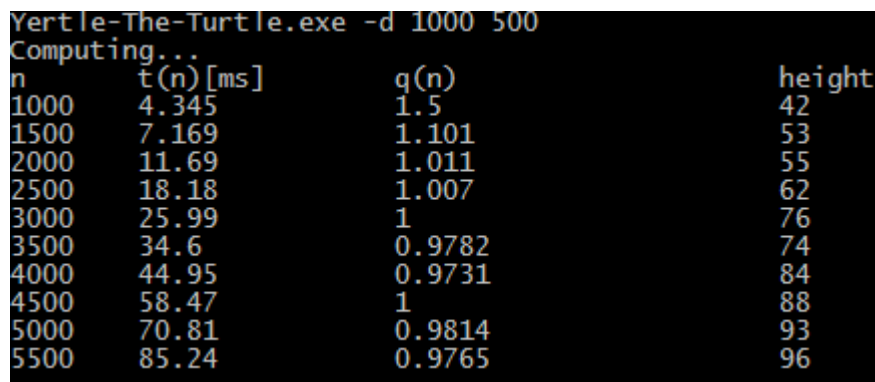
Turtle

Klasa reprezentująca pojedynczego żółwia posiadającego atrybuty takie jak waga i wytrzymałość.

5 Przykładowe wyniki pomiarów czasu

5.1 Programowanie dynamiczne

Algorytm programowania dynamicznego działa z kwadratową złożonością. Ma również dużą złożoność pamięciową, ponieważ alokuje tablicę o rozmiarze $N \times N$, gdzie N to liczba żółwi. W algorytmie zastosowano małą optymalizację, dzięki czemu zmniejszono rozmiar tablicy o około połowę (zaalokowano dolną połowę tablicy, ponieważ wartości powyżej jej przekątnej są nieistotne, można to zauważyć na rysunku 1). Algorytm działa w podobnym czasie jak algorytm naiwny.

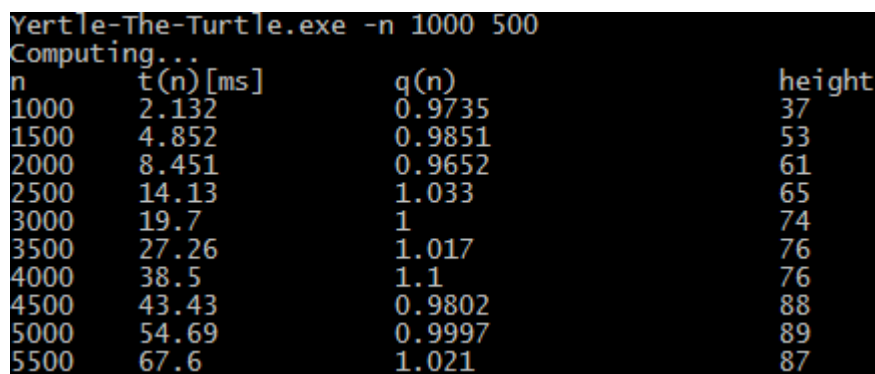


Yertle-The-Turtle.exe -d 1000 500			
Computing...			
n	t(n) [ms]	q(n)	height
1000	4.345	1.5	42
1500	7.169	1.101	53
2000	11.69	1.011	55
2500	18.18	1.007	62
3000	25.99	1	76
3500	34.6	0.9782	74
4000	44.95	0.9731	84
4500	58.47	1	88
5000	70.81	0.9814	93
5500	85.24	0.9765	96

Rysunek 2: Przykładowe wykonanie programu dla algorytmu programowania dynamicznego, przy początkowym rozmiarze problemu 1000 i kolejnych iteracjach co 500

5.2 Algorytm naiwny

Algorytm naiwny działa z kwadratową złożonością. Ma jednak o wiele mniejszą złożoność pamięciową. Czas jego wykonania jest bardzo podobny do czasu wykonania algorytmu korzystającego z programowania dynamicznego.

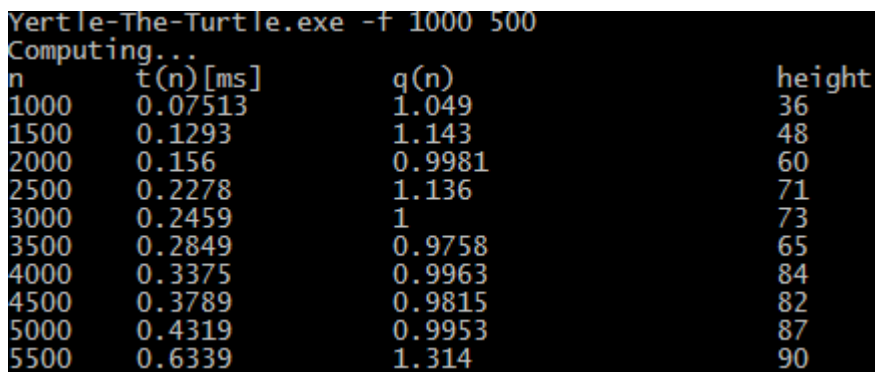


Yertle-The-Turtle.exe -n 1000 500			
Computing...			
n	t(n) [ms]	q(n)	height
1000	2.132	0.9735	37
1500	4.852	0.9851	53
2000	8.451	0.9652	61
2500	14.13	1.033	65
3000	19.7	1	74
3500	27.26	1.017	76
4000	38.5	1.1	76
4500	43.43	0.9802	88
5000	54.69	0.9997	89
5500	67.6	1.021	87

Rysunek 3: Przykładowe wykonanie programu dla algorytmu naiwnego, przy początkowym rozmiarze problemu 1000 i kolejnych iteracjach co 500

6 Algorytm z presortowaniem

Algorytm wykonuje się zdecydowanie najszybciej, różnica wynosi około 2 rzędy wielkości w stosunku do pozostałych dwóch algorytmów.



Yertle-The-Turtle.exe -t 1000 500			
Computing...			
n	t(n) [ms]	q(n)	height
1000	0.07513	1.049	36
1500	0.1293	1.143	48
2000	0.156	0.9981	60
2500	0.2278	1.136	71
3000	0.2459	1	73
3500	0.2849	0.9758	65
4000	0.3375	0.9963	84
4500	0.3789	0.9815	82
5000	0.4319	0.9953	87
5500	0.6339	1.314	90

Rysunek 4: Przykładowe wykonanie programu dla algorytmu z presortowaniem, przy początkowym rozmiarze problemu 1000 i kolejnych iteracjach co 500