WYDZIAŁ
INFORMATYKI
I TELEKOMUNIKACJI

Obiegi drzew

Veronika Tronchuk

I rok Informatyki

Nr albumu: 30019

Celem

ćwiczenia było zapoznanie się z wybranymi algorytmami obliczeniowymi oraz ich implementacją w języku C++. W ramach zadań wykonano problem plecakowy, obiegi drzew, algorytm Huffmana oraz optymalne mnożenie macierzy.

wykonano cztery zadania, które pozwoliły lepiej zrozumieć działanie algorytmów przeszukiwania, programowania dynamicznego oraz kodowania Huffmana. Każdy program został przetestowany i działa poprawnie. Wyniki potwierdzają, że zastosowane metody dają poprawne i optymalne rozwiązania.

Zadanie 1

```
#include <iostream>
#include <vector>
#include <cmath>
```

te biblioteki umożliwiają obsługę wejścia i wyjścia w konsoli, tworzenie wygodnych list (wektorów) oraz korzystanie z funkcji matematycznych.

```
int main() {
```

Początek głównego programu.

```
vector<int> wagi = { 4, 2, 3 };
vector<int> wartosci = { 10, 4, 7 };
int maxWaga = 5;
int n = wagi.size();
```

Tworzymy listę wag (wagi), listę wartości (wartosci), ustawiamy maksymalną wagę plecaka i obliczamy liczbę przedmiotów.

```
int najlepszaWartosc = 0;
vector<int> najlepszaKombinacja(n, 0);
```

Zmienna do przechowywania najlepszej wartości i lista do zapisu najlepszej kombinacji przedmiotów.

```
int konfiguracji = pow(2, n);
```

Obliczamy liczbę wszystkich możliwych kombinacji

```
for (int maska = 0; maska < konfiguracji; ++maska) {
```

Uruchamiamy pętlę po wszystkich możliwych kombinacjach przedmiotów.

```
int sumaWagi = 0, sumaWartosci = 0;
vector<int> kombinacja(n, 0);
```

Dla każdej kombinacji tworzymy zmienne do obliczenia sumy wagi i wartości oraz listę kombinacji.

```
for (int i = 0; i < n; ++i) {
    if (maska & (1 << i)) {
        sumaWagi += wagi[i];
        sumaWartosci += wartosci[i];
        kombinacja[i] = 1;
    }
}
```

Dla każdego przedmiotu sprawdzamy, czy jest wybrany w bieżącej kombinacji, i dodajemy jego wagę i wartość.

```
cout << "kombinacja: ";
for (int k : kombinacja) cout << k << " ";
cout << "waga: " << sumaWagi << " wartosc: " << sumaWartosci << endl;
```
=

Wyświetlamy kombinację, jej łączną wagę i wartość.

```
if (sumaWagi <= maxWaga && sumaWartosci > najlepszaWartosc) {
    najlepszaWartosc = sumaWartosci;
    najlepszaKombinacja = kombinacja;
}
```

Jeśli waga nie przekracza maksimum i wartość jest lepsza — aktualizujemy najlepszy wynik.

```
cout << "najlepsza kombinacja: ";
for (int k : najlepszaKombinacja) cout << k << " ";
cout << "wartosc: " << najlepszaWartosc << endl;
```

Wyświetlamy najlepszą znalezioną kombinację i jej wartość.

Wnioski

używa pełnego przeglądu, więc działa bardzo wolno przy dużej liczbie przedmiotów; dodatkowo brakuje wygodnego wczytywania danych od użytkownika lub z pliku oraz sprawdzania niepoprawnych wartości wagi lub wartości.

Zadanie 2

```
int n1 = 3, n2 = 5, n3 = 10;
```

Deklarujemy trzy zmienne: liczba przedmiotów (3, 5 i 10).

```
cout << "dla " << n1 << " przedmiotów: " << pow(2, n1) << " konfiguracji\n";
```

Wyświetlamy ile konfiguracji jest możliwych dla 3 przedmiotów

```
cout << "dla " << n2 << " przedmiotów: " << pow(2, n2) << " konfiguracji\n";
cout << "dla " << n3 << " przedmiotów: " << pow(2, n3) << " konfiguracji\n";
```

Analogicznie dla 5 i 10 przedmiotów

Wnioski

Ten kod po prostu liczy, ile jest możliwych wariantów dla kilku przedmiotów. Dobrze pokazuje, jak szybko rośnie liczba kombinacji, ale nie rozwiązuje problemu i nie sprawdza, czy te kombinacje pasują do plecaka.

```
#include <algorithm>
```

Biblioteka algorithm do sortowania.

```
struct Przedmiot {
    int waga;
    int wartosc;
    double ratio;
};
```

Tworzymy strukturę Przedmiot z wagą, wartością i współczynnikiem wartość/waga.

```
bool compare(Przedmiot a, Przedmiot b) {
    return a.ratio > b.ratio;
}
```

Funkcja do sortowania przedmiotów malejąco według stosunku wartość/waga.

```
vector<Przedmiot> przedmioty = {
    {4, 10}, {2, 4}, {3, 7}
};
int maxWaga = 5;
```

Tworzymy listę przedmiotów z wagą i wartością. Ustawiamy maksymalną wagę plecaka maxWaga = 5.

```
for (auto& p : przedmioty) p.ratio = (double)p.wartosc / p.waga;
sort(przedmioty.begin(), przedmioty.end(), compare);
```

Obliczamy stosunek wartość/waga dla każdego przedmiotu i sortujemy według tego współczynnika.

```
double lacznaWartosc = 0.0;
int wagaZajeta = 0;
```

Zmienne dla sumarycznej wartości i zajętej wagi.

```cpp
for (auto p : przedmioty) {
    if (wagaZajeta + p.waga <= maxWaga) {
        wagaZajeta += p.waga;
        lacznaWartosc += p.wartosc;
        cout << "pelny: waga " << p.waga << " wartosc " << p.wartosc << endl;
    }
    else {
        int pozostalaWaga = maxWaga - wagaZajeta;
        lacznaWartosc += p.ratio * pozostalaWaga;
        cout << "czesc: waga " << pozostalaWaga << " wartosc " << p.ratio * pozostalaWaga << endl;
        break;
    }
}
```

Dla każdego przedmiotu: jeśli mieści się cały — dodajemy wagę i wartość, jeśli nie — dodajemy część i kończymy.

```cpp
cout << "laczna wartosc: " << lacznaWartosc << endl;
```

Wyświetlamy sumaryczną wartość

Wnioski

W tym zadaniu zastosowano algorytm zachłanny dla plecaka łamliwego: przedmioty są sortowane według stosunku wartości do wagi i dodawane do plecaka w całości lub częściowo, aż do zapełnienia. Kod działa poprawnie dla tego wariantu, ale jego wadą jest to, że nie nadaje się do plecaka bez możliwości dzielenia przedmiotów — wtedy potrzebny jest pełny przegląd lub programowanie dynamiczne.

Zadanie 4

```
int ksDpTable(int weights[], int values[], int n, int capacity) {
    vector<vector<int>> dp(n + 1, vector<int>(capacity + 1, 0));
```

Funkcja ksDpTable tworzy tablicę DP (programowanie dynamiczne) o wymiarach n+1 wierszy i capacity+1 kolumn, wypełnioną zerami.

```
for (int i = 1; i <= n; ++i) {
    for (int w = 1; w <= capacity; ++w) {
        dp[i][w] = dp[i - 1][w];
```

Iterujemy po przedmiotach i i możliwym ciężarze w. Na początku wartość jest taka sama jak w poprzednim wierszu (bez nowego przedmiotu).

```
if (weights[i - 1] <= w) {
    dp[i][w] = max(dp[i][w], values[i - 1] + dp[i - 1][w - weights[i - 1]]);
}
```

Jeśli waga przedmiotu jest mniejsza lub równa w, wybieramy maksimum: brać przedmiot czy nie.

```
return dp[n][capacity];
```

Po wypełnieniu tablicy zwracamy maksymalną wartość dla n przedmiotów i pojemności capacity.

```
int weights[] = { 10, 20, 30 };
int values[] = { 60, 100, 120 };
int n = 3;
int capacity = 50;
```

ustalamy wagi, wartości, liczbę przedmiotów n oraz maksymalną pojemność plecaka capacity.

```
cout << "weights: ";
for (int i = 0; i < n; i++) cout << weights[i] << " ";
```

Wyświetlamy dane wejściowe: wagi, wartości i pojemność.

```
cout << "\nmax weight capacity: " << capacity << endl;
```

Wywołujemy funkcję DP, pokazujemy maksymalną wartość i zatrzymujemy konsolę.

Wnioski

Program zużywa dużo pamięci, jeśli jest dużo przedmiotów lub duża pojemność. Pokazuje tylko łączną wartość, ale nie wskazuje, które przedmioty wybrać. Może działać wolno przy dużych danych.