Dinamičko programiranje

Tkalčec, David

Undergraduate thesis / Završni rad

2017

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **Josip Juraj Strossmayer University of Osijek, Faculty of Electrical Engineering, Computer Science and Information Technology Osijek / Sveučilište Josipa Jurja Strossmayera u Osijeku, Fakultet elektrotehnike, računarstva i informacijskih tehnologija Osijek**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:200:140262

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-03-28**

Repository / Repozitorij:

Faculty of Electrical Engineering, Computer Science and Information Technology Osijek

zir.nsk.hr

Image not found or type unknown

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA

Sveučilišni studij računarstva

DINAMIČKO PROGRAMIRANJE

Završni rad

David Tkalčec

Osijek, 2017.

SADRŽAJ

1. UVOD 1

1.1. Zadatak završnog rada 1

2. DINAMIČKO PROGRAMIRANJE – TEORIJSKA PODLOGA 2

2.1. Osnovni pristupi dinamičkog programiranja 3

2.2. Dokazivanje optimalne podstrukture i zavisnosti podproblema 4

3. PROBLEM REZANJA DRVA – IMPLEMENTACIJA U PROGRAMSKOM JEZIKU 7

3.1. Rekurzivna implementacija 7

3.2. Implementacija pristupom „Odozgo prema dolje“ 9

3.3. Implementacija pristupom „Odozdo prema gore“ 11

4. ANALIZA I OBRADA REZULTATA 12

4.1. Korištena zaglavlja i funkcije za dobivanje rezultata 12

4.2. Analiza povratnih vrijednosti funkcija i vremena izvođenja 13

5. ZAKLJUČAK 16

LITERATURA 17

SAŽETAK 18

ABSTRACT 19

ŽIVOTOPIS 20

1. UVOD

U ovom završnom radu bit će objašnjeno što je dinamičko programiranje te će biti opisana osnovna ideja ove tehnike koja se temelji na rastavljanju različitih matematičkih i drugih problema na više problema manje složenosti čija se rješenja pohranjuju i koriste za rješavanje osnovnog problema. Bitna karakteristika koja razlikuje dinamičko programiranje od drugih sličnih metoda je ta da se svaki podproblem glavnog problema rješava najviše jednom čime se značajno brže može doći do konačnog rješenja. Nakon toga će biti definirano kako za određeni problem odrediti može li se on riješiti ovom tehnikom te će biti dani pristupi pomoću kojih se ova tehnika implementira u programskom jeziku. U glavnom dijelu rada bit će prikazano kako se dinamičko programiranje implementira u programskom jeziku C++ za konkretni primjer. Na kraju rada usporedit ćemo brzine izvođenja različitih implementacija te ćemo analizirati na koji način dinamičko programiranje doprinosi učinkovitijem rješavanju problema.

1.1. Zadatak završnog rada

Zadatak ovoga rada je detaljno opisati tehniku dinamičkog programiranja, opisati na koji način se ova tehnika koristi te na konkretnom primjeru detaljno pokazati i objasniti pristupe dinamičkog programiranja. Također ćemo na „Problemu rezanja drva“ implementirati dinamičko rješenje u programskom jeziku C++ te odrediti učinkovitost tako dobivenog rješenja.

2. DINAMIČKO PROGRAMIRANJE – TEORIJSKA PODLOGA

Dinamičko programiranje je naziv za tehniku rješavanja složenih algoritamskih problema kojom značajno možemo smanjiti složenost algoritama. Pojam „dinamičko programiranje“ uveo je matematičar Richard Belman, koji je sredinom pedesetih godina dvadesetog stoljeća proučavao problem tako što je proučavao hijerarhiju podproblema sadržanih u glavnom problemu te je rješavanje počinjao od najjednostavnijih.

Ovom tehnikom početni složeni problem rastavljamo na podprobleme (eng. Subproblems) manje složenosti koji su međusobno zavisni. Svaki podproblem se rješava najviše jednom, čime možemo izbjeći višestruko računanje numeričkih karakteristika istog stanja. Osnovna ideja na koju se oslanjaju algoritmi dinamičkog programiranja je da se svaki dobiveni međurezultat, odnosno rješenje podproblema, spremi te zatim kada se sljedeći puta naiđe na taj isti podproblem, izbjegne njegovo ponovno rješavanje. Postoji veliki broj problema koji možemo riješiti na ovaj način te je ova tehnika široko primjenjiva u praksi.

Dinamičko programiranje se često uspoređuje sa metodom „Podijeli pa vladaj“ (eng. Divide and conquer) koja također glavni problem rastavlja podprobleme manje složenosti te zatim rekurzivno dolazi do rješenja podproblema kako bi se njihovim spajanjem dobilo rješenje polaznog problema. Bitna razlika između ove dvije metode je ta da su podproblemi kod metode „Podijeli pa vladaj“ nezavisni, dok kod dinamičkog programiranja oni moraju imati optimalnu podstrukturu te su oni međusobno zavisni.

Prije samog rješavanja određenog problema vrlo je važno utvrditi možemo li za njegovo rješavanje primijeniti tehniku dinamičkog programiranja. Kao sta je ranije spomenuto dva osnovna uvjeta za primjenu ove tehnike su da problem ima optimalnu podstrukturu te da su podproblemi glavnog problema međusobno povezani(zavisni).

2.1. Osnovni pristupi dinamičkog programiranja

U prethodnom poglavlju je opisano kako je osnovna misao dinamičkog programiranja glavni problem(zadatak) podijeliti na više problema manje složenosti koji su međusobno povezani te nakon toga svaki od tih problema riješiti najviše jednom kako bi se došlo do konačnog rješenja. Rješavanje svakog podproblema najviše jednom postiže se pomoću dva osnovna pristupa rješavanju problema: pristup „Odozgo prema dolje“ (eng. Top-down approach) te pristup „Odozdo prema gore“ (eng. Bottom-up approach).

Pristup „Odozgo prema dolje“ temelji se na tome da se osnovni problem rastavi na podprobleme te da se to podproblemi riješe i pamte se njihova rješenja. U slučaju kada se naiđe na problem koji je već prethodno riješen, samo se očita spremljeno rješenje tog problema iz tablice rješenja. Ovakva tehnika pohrane rješenja podproblema naziva se memoizacija.

Kod pristupa „Odozdo prema gore“ najprije se rješavaju najjednostavniji podproblemi te se zatim njihova rješenja koriste za nalaženje složenijih i tako sve dok se ne dođe do konačnog rješenja. Ovaj pristup u nekim slučajevima može biti efikasniji u kontekstu potrošnje memorije, ali je ponekad teško odrediti koji su podproblemi potrebni za rješavanje danog problema.

U glavnom dijelu ovoga rada detaljno će biti objašnjena ova dva pristupa na konkretnom primjeru.

2.2. Dokazivanje optimalne podstrukture i zavisnosti podproblema

Za određeni problem kažemo da ima optimalnu podstrukturu ako do optimalnog rješenja toga problema možemo doći pomoću optimalnih rješenja njegovih podproblema.

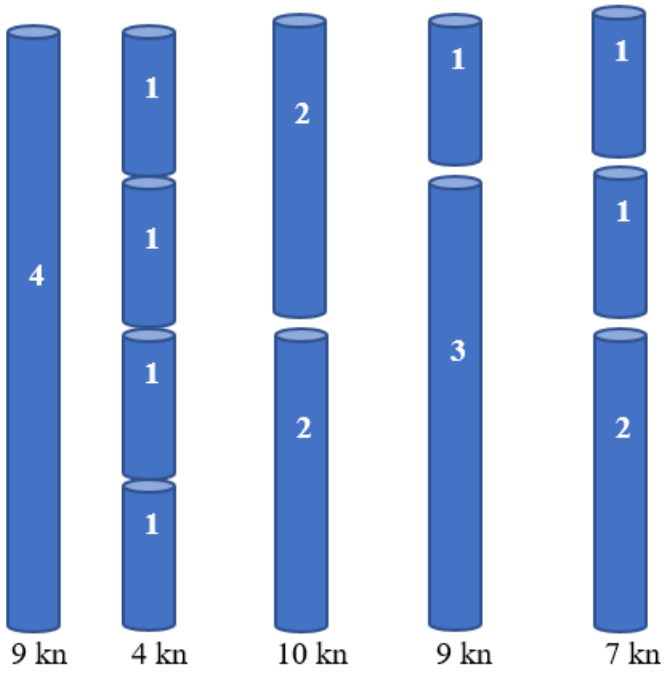
Problem za koji ćemo dokazati da ima optimalnu podstrukturu je „Problem rezanja drva“. Problem je sljedeći: Pretpostavimo da imamo drvo duljine n te da želimo razrezati drvo i prodati komade na način da zaradimo najviše novca. Komad drva duljine l ima vrijednost p_n kuna kako je prikazano u tablici 2.1.

Tab. 2.1. *Cijene drva za pojedine duljine.*

duljina (n)	1	2	3	4	5	6	7	8	9	10
cijena (p_n)	1	5	8	9	10	17	17	20	24	30

Jedan od načina kako doći do rješenja ovoga problema je da odredimo sve moguće kombinacije na koje možemo izrezati drvo određene duljine te zatim odredimo koja kombinacija donosi najveći profit.

Drvo duljine n možemo razrezati na 2^{n-1} načina s obzirom na to da svaki metar može biti rezan ili ne. Slika 2.1. prikazuje sve načine na koje možemo razrezati drvo duljine 4 metra.



Sl. 2.1. *Kombinacije za rezanje drva duljine 4 metra*
(preostala tri načina su samo permutacije već navedenih rezova).

Iz slike 2.1. možemo vidjeti da ćemo ostvariti najveći profit ako drvo duljine 4 metra prerežemo na 2 komada duljine 2 metra. To je ujedno i optimalno rješenje za drvo duljine 4 metra, uzimajući u obzir tablicu 2.1.

Prikažimo sada način određivanja maksimalnog prihoda za nekoliko drva različitih duljina radi lakšeg shvaćanja formiranja rezultata(Tab. 2.2.)

Tab. 2.2. *Računanje prihoda za duljine drva od 1 do 4.*

duljina drva (n)	sve kombinacije rezanja drva duljine n	[1]maksimalni prihod r_n za drvo duljine n
1	ne režemo	1
2	2, 1+1	5
3	3, 1+1, 2+1, 1+1+1	8
4	4, 1+3, 2+2, 3+1, 1+1+2, 1+2+1, 2+1+1, 1+1+1+1	10

Pomoću tablice 2.2. možemo formirati općeniti matematički izraz za računanje maksimalnog prihoda (r_n) za određenu duljinu drva (k):

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \dots, r_{n-1} + r_1) \quad (2\text{-}1)$$

gdje je: max – funkcija koja vrača najveću vrijednost od predanih joj vrijednosti

p_n – cijena drva duljine n

r_n – maksimalni prihod za cijenu drva duljine n

Izraz (2-1) možemo pojednostaviti na sljedeći način;

$$r_n = \max_{1<i<n}(p_i + r_{n-i}) \quad (2\text{-}2)$$

Tab. 2.3. *Ovisnost svakog prihoda r_n o prihodu r_4.*

duljina drva (n)	maksimalni prihod r_n za drvo duljine n	najmanji od
1	r_1	$p_1 + r_0$
2	r_2	$p_1 + r_1, p_2 + r_0$
3	r_3	$p_1 + r_2,\ p_2 + r_1,\ p_3 + r_0$
4	10	$p_1 + r_3,\ p_2 + r_2,\ p_3 + r_1, p_4 + r_0$

[1] Maksimalni prihod računamo pomoću Tablice 2.1.

U tablici 2.3. je prikazano kako svaki maksimalni prihod (r_n) ovisi o prihodu r_4. Svaki pojedinačni r_n računamo tako da za svaki mogući prvi rez (npr. $p_1 \ldots p_k$) izračunamo zbroj vrijednosti toga reza (npr p_1) i najvećega prihoda koji smo mogli dobiti od ostatka drveta. Na kraju samo izaberemo najveću sumu $(p_i + r_{n-i})$.

Primjeri računanja maksimalnog prihoda za drva duljine 4 i 5 metara prikazani su u tablici 2.4.

Tab. 2.4. *Računanje r_4 i r_5.*

$r_4 = \max(p_1 + r_3,\ p_2 + r_2,\ p_3 + r_1, p_4 + r_0)$ $= \max(1+8, 5+5, 8+1, 9+0)$ $= \max(9,10,9,9)$ $= 10$	$r_5 = \max(p_1 + r_4, p_2 + r_3, p_3 + r_2,$ $p_4 + r_1,\ p_5 + r_0)$ $= \max(1+10, 5+8, 8+5, 9+$ $1, 10+0)$ $= \max(11,13,13,10,10)$ $= 13$
Napomena: Pretpostavljamo da je $r_0 = 0$.	

Primijetimo da svaka vrijednost prihoda (r_n) ovisi samo o vrijednostima koje su iznad te vrijednosti u tablici (primjer tablica 2.3.). Sada jasno možemo vidjeti da „Problem rezanja drva“ ima optimalnu podstrukturu jer do optimalnog rješenja ovoga problema dolazimo pomoću optimalnih rješenja njegovih podproblema, odnosno rekurzivnim izvođenjem zadatka pomoću formule (2-2). Također vidimo da su podproblemi međusobno zavisni.

3. PROBLEM REZANJA DRVA – IMPLEMENTACIJA U PROGRAMSKOM JEZIKU

U ovome poglavlju implementirat ćemo tri različita načina rješavanja prethodno razrađenoga problema rezanja drva. Prvi način će biti pomoću rekurzije bez primjene tehnike dinamičkog programiranja, dok će druga dva načina biti primjenom pristupa „Odozgo prema dolje“ te pristupa „Odozdo prema gore“. Sva tri načina ćemo implementirati u programskom jeziku C++.

3.1. Rekurzivna implementacija

U programiranju rekurzija nastaje kada funkcija poziva samu sebe. Rekurzivna funkcija koja vraća maksimalni prihod koji možemo dobiti rezanjem drva određene duljine prikazana je na slici 3.1. Funkciji kao listu argumenata predajemo polje sa cijenama za pojedine duljine drva te duljinu drva koje želimo izrezati. Unutar for petlje koristimo matematičku formulu (2-2) koju smo ranije odredili te funkciju Max čija je implementacija prikazana na slici 3.2..

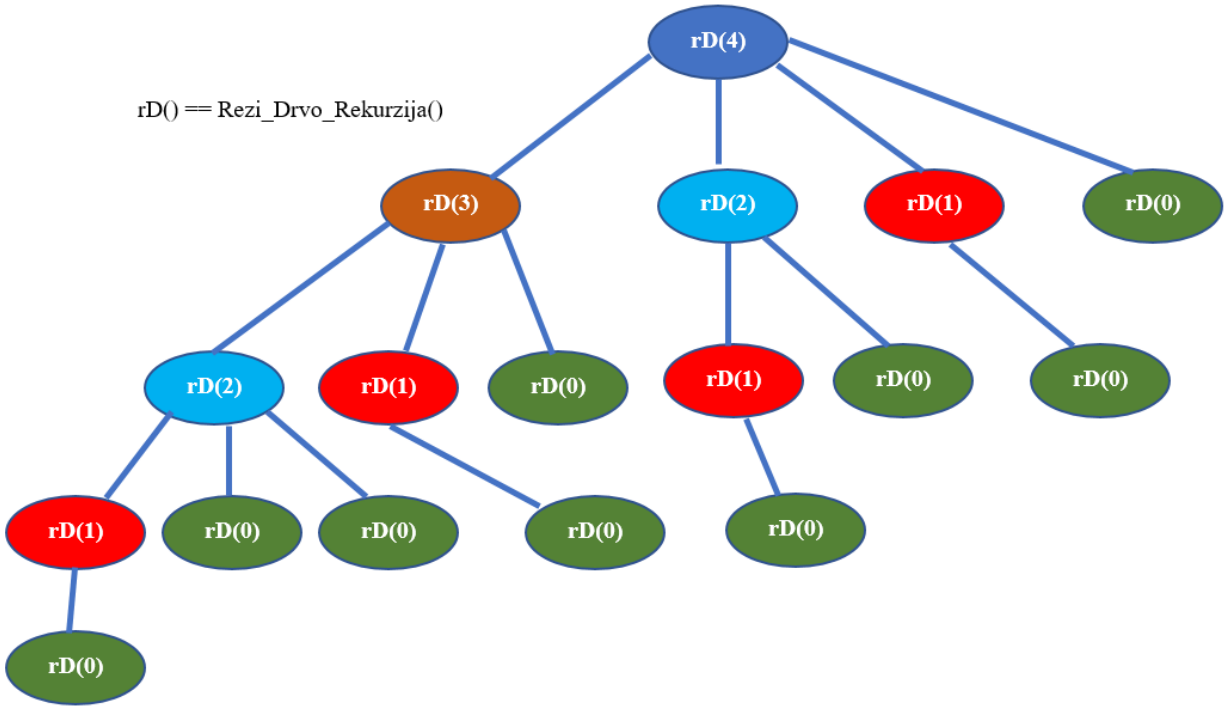
```
int Rezi_Drvo_Rekurzija(int cijene[], int duljina)
{
    if (duljina <= 0) return 0;
    int najveci_prihod = INT_MIN; //-2147483648
    int i;
    for (i = 0; i < duljina; i++)
    {
        najveci_prihod = Max(najveci_prihod, cijene[i] + Rezi_Drvo_Rekurzija(cijene, duljina - i-1));
    }
    return najveci_prihod;
}
```

Sl. 3.1. *Rekurzivna funkcija koja vraća maksimalni prihod.*

```
/*pomocna funkcija koja od 2 broja tipa integer
vraca onaj veci*/
int Max(int x, int y)
{
    if (x > y) return x;
    else return y;
}
```

Sl. 3.2. *Pomoćna funkcija za izračun najveće vrijednosti.*

Uzimajući u obzir implementaciju prikazanu na slici 3.1. možemo grafički pomoću rekurzivnog stabla prikazati način izvođenja dane funkcije (Sl 3.3.).



Sl. 3.3. *Prikaz rekurzivnog izvođenja funkcije.*

Na slici 3.3. možemo vidjeti kako se mnogi „isti“ podproblemi izvode više puta. Povećanjem duljine drva koje moramo razrezati, broj kombinacija koje moramo odrediti raste eksponencijalno. Isto tako raste i broj istih podproblema koji će se višestruko računati. U kontekstu vremenske složenosti, rješavanje ovoga problema na takav način je vrlo „skup“ proces. Vremenska složenost funkcije prikazane na slici 3.1. je eksponencijalna, odnosno $O(2^n)$.

3.2. Implementacija pristupom „Odozgo prema dolje“

Kao što je ranije spomenuto pristup „Odozgo prema dolje“ je jedan od dva osnovna pristupa dinamičkog programiranja. Na slici 3.5. možemo vidjeti kako se ovaj pristup implementira u programskom jeziku C++ za „Problem rezanja drva“.

```
int *Rezi_Drvo_Memoizacija_init(int cijene[], int duljina)
{
    int j;
    int *memorija = new int[duljina];
    for (j = 0; j < duljina; j++)
    {
        memorija[j] = INT_MIN;
    }
    return memorija;
}
```

Sl. 3.4. *Pomoćna funkcija za stvaranje 1D polja za memoizaciju.*

```
int Rezi_Drvo_Memoizacija(int cijene[], int duljina, int memorija[])
{
    if (memorija[duljina] > 0) return memorija[duljina];

    if (duljina <= 0) return 0;

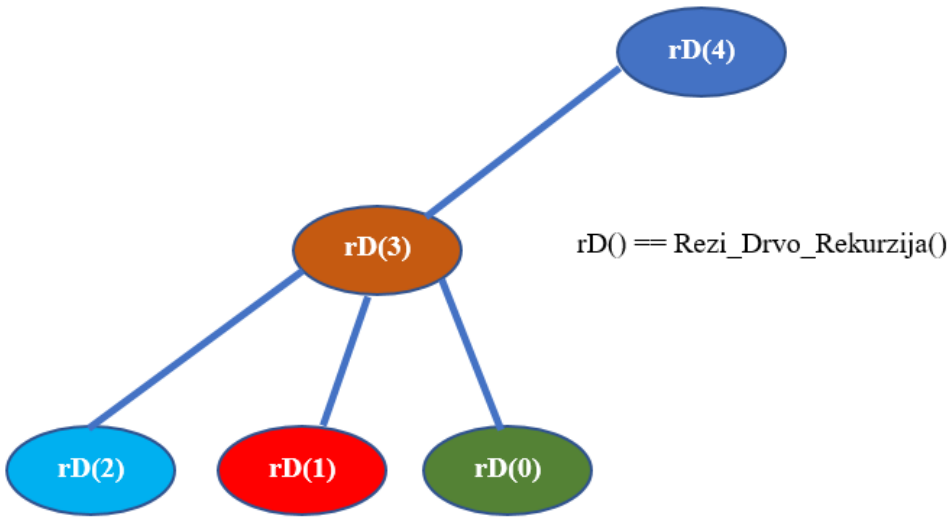
    int najveci_prihod = INT_MIN; //-2147483648
    int i;
    for (i = 0; i < duljina; i++)
    {
        najveci_prihod = Max(najveci_prihod, cijene[i] + Rezi_Drvo_Memoizacija(cijene, duljina - i-1 ,memorija));
    }
    memorija[duljina] = najveci_prihod;
    return najveci_prihod;
}
```

Sl. 3.5. *Prikaz funkcije za računanje maksimalnog prihoda pristupom „Odozgo prema dolje“.*

Vidimo da je programski kod vrlo sličan rekurzivnoj implementaciji. Glavna i najbitnija razlika je ta da je unutar ovoga koda dodano novo polje koje će služiti kao memorija. Na početku funkcije najprije provjeravamo postoji li rješenje zadanog problema u memoriji, ako postoji samo vračamo to rješenje. Ukoliko rješenje nije prethodno izračunato, rekurzivnim izvođenjem dolazimo do novog rješenja te to rješenje također pohranjujemo u memoriju.

Na slici 3.4. prikazana je pomoćna funkcija pomoću koje ćemo dinamički alocirati memoriju za novo 1D polje i zatim sve vrijednosti u polju koje predstavlja memorijsku tablicu za naš problem postaviti na minimalnu vrijednost cjelobrojnog tipa (koja iznosi -2147483648) te će takve vrijednosti predstavljati „praznu memoriju“.

Ovakav način izvođenja značajno je učinkovitiji u kontekstu vremenske složenosti iz razloga što se svaki podproblem rješava najviše jednom. Kao što smo ranije spomenuli rekurzivni algoritam ima eksponencijalnu vremensku složenost ($O(2^n)$). Dinamičkim programiranjem, odnosno pristupom „Odozgo prema dolje“ algoritam smo sveli na kvadratnu vremensku složenost ($O(n^2)$). Na slici 3.6. možemo vidjeti kako bi izgledalo stablo izvođenja ovako implementirane funkcije za „Problem rezanja drva“.



Sl. 3.6. *Stablo izvođenja funkcije primjenom dinamičke metode „Odozgo prema dolje“.*

3.3. Implementacija pristupom „Odozdo prema gore“

Pristup „Odozdo prema gore“ drugi je način na koji možemo implementirati rješenje našega „Problema rezanja drva“. Na slici 3.7. prikazana je implementacija funkcije za određivanje maksimalnog prihoda u programskom jeziku C++.

```
int Rezi_Drvo_BottomUp(int cijene[], int duljina)
{
    int *polje_za_max_vrijednosti= new int[duljina+1];
    polje_za_max_vrijednosti[0] = 0;
    int i,j, najveci_prihod;
    for (i = 1; i <= duljina; i++)
    {
        najveci_prihod = -1;
        for (j = 0; j < i; j++)
        {
            najveci_prihod = Max(najveci_prihod, cijene[j] + polje_za_max_vrijednosti[i-(j+1)]);
            polje_za_max_vrijednosti[i] = najveci_prihod;
        }

    }
    return polje_za_max_vrijednosti[duljina];
}
```

Sl. 3.7. *Prikaz funkcije za računanje maksimalnog prihoda pristupom „Odozdo prema gore“.*

Na početku funkcije dinamički alociramo memoriju za polje koje će sadržavati vrijednosti najvećih prihoda za pojedine duljine drva. Ovakav način pristupa problemu popunjava tablicu (polje) maksimalnih prihoda tako da najprije maksimalni prihod za drvo duljine nula postavimo na vrijednost 0 te se zatim redom računaju vrijednosti maksimalnog prihoda za drva duljine 1 metar (rD(1)), 2 metra (rD(2)) i tako redom do konačne vrijednosti maksimalnog prihoda. Svaka prethodno izračunata vrijednost se koristi za računanje sljedeće. Sada je jasnije zašto ovaj pristup zovemo „Odozdo prema gore“.

Po pitanju vremenske složenosti, ovaj algoritam također ima kvadratnu vremensku složenost ($O(n^2)$) kao i algoritam kod kojega je korišten pristup „Odozgo prema dolje“.

4. ANALIZA I OBRADA REZULTATA

U ovome poglavlju ćemo mjeriti vremena izvođenja prethodno implementiranih rješenja za „Problem rezanja drva“. Do sada je bilo opisano na koji način se izvode pojedine implementacije te koje su njihove vremenske složenosti. Sada će u programskom jeziku C++ te programskom okruženju Microsoft Visual Studio 2017 biti izmjerena vremena izvođenja pomoću već ugrađenih funkcija te će biti napravljena analiza i usporedba dobivenih rezultata.

4.1. Korištena zaglavlja i funkcije za dobivanje rezultata

Na slici 4.1. prikazana su sva potrebna zaglavlja za pokretanje ranije opisanih funkcija te mjerenje vremena njihova izvođenja. Zaglavlje *iostream* definira standardne klase i objekte za ulaz i izlaz podataka(cin, cout), dok zaglavlje *stdlib.h* definira neke osnovne funkcije i funkcije za upravljanje dinamičkom alokacijom memorije. Vremenska biblioteka *chrono* je ime zaglavlja, ali ujedno i ime pod-imenika. Elementi zaglavlja *chrono* nisu definirani u standardnom imeniku (*namespace* std) pa ih je potrebno posebno definirati. Ovo zaglavlje će nam omogućiti korištenje vremenskih funkcija potrebnih za računanje izvođenja naših funkcija.

```
#include<iostream>
#include<stdlib.h>
# include <chrono>

using namespace std;
using  ns = chrono::nanoseconds;
using get_time = chrono::steady_clock;
```

Sl. 4.1. *Korištene biblioteke i imenici.*

Vrijeme će biti mjereno pomoću funkcija prikazanih na slici 4.2.

```
auto start = get_time::now();
Rezi_Drvo_Rekurzija(cijene, size);
auto end = get_time::now();
auto diff = end - start;
cout << "Vrijeme izvodenja funkcije:  " <<
    chrono::duration_cast<ns>(diff).count() << " ns " << endl;
```

Sl. 4.2. *Mjerenje vremena trajanja funkcije.*

4.2. Analiza povratnih vrijednosti funkcija i vremena izvođenja

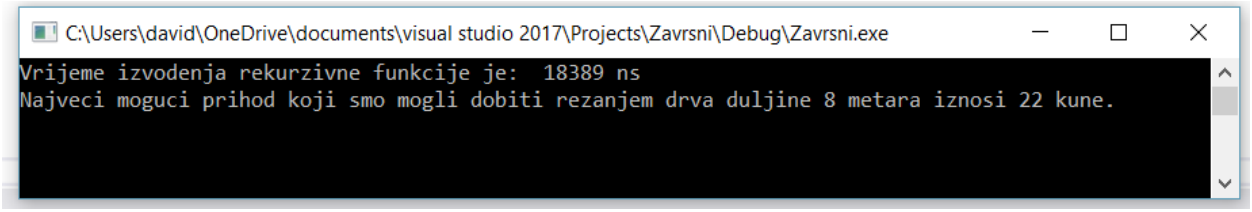
```
int main()
{
    int cijene[] = { 1, 5, 8, 9, 10, 17, 17, 20 };
    int duljina_drva = sizeof(cijene) / sizeof(cijene[0]);

    auto start = get_time::now();
    Rezi_Drvo_Rekurzija(cijene, duljina_drva);
    auto end = get_time::now();
    auto diff = end - start;
    cout << "Vrijeme izvodenja rekurzivne funkcije je:  " <<
        chrono::duration_cast<ns>(diff).count() << " ns " << endl;

    cout << "Najveci moguci prihod koji smo mogli dobiti rezanjem drva duljine "
        << duljina_drva << " metara iznosi "<< Rezi_Drvo_Rekurzija(cijene, duljina_drva) <<" kune." <<endl;

    int a;
    cin >> a;
    return 0;
}
```

Sl. 4.3. *Mjerenje vremena rekurzivne funkcije za rezanje drva.*



Sl. 4.4. *Izlaz programa za testiranje povratne vrijednosti rekurzivne funkcije i vremena izvođenja.*

Na slici 4.3. prikazana je glavna funkcija pomoću koje smo mjerili vrijeme izvođenja rekurzivne funkcije. Pokretanjem navedenog programa dobili smo rezultat prikazan na slici 4.4. Možemo vidjeti da vrijeme izvođenja rekurzivne funkcije iznosi 18389 ns. Također možemo vidjeti da je povratna vrijednost rekurzivne funkcije ispravna.

```
int main()
{
    int cijene[] = { 1, 5, 8, 9, 10, 17, 17, 20 };
    int duljina_drva = sizeof(cijene) / sizeof(cijene[0]);

    int *memorija = Rezi_Drvo_Memoizacija_init(cijene, duljina_drva);

    auto start = get_time::now();
    Rezi_Drvo_Memoizacija(cijene, duljina_drva, memorija);
    auto end = get_time::now();
    auto diff = end - start;
    cout << "Vrijeme izvodenja funkcije koja koristi memoizaciju je:  " <<
        chrono::duration_cast<ns>(diff).count() << " ns " << endl;

    cout << "Najveci moguci prihod koji smo mogli dobiti rezanjem drva duljine "
        << duljina_drva << " metara iznosi "
        << Rezi_Drvo_Memoizacija(cijene, duljina_drva, memorija) << " kune." << endl;

    cout << "U polju koje pretstavlja memoriju nalaze se sljedeći podaci: ";
    for (int j = 1; j < duljina_drva + 1; j++)
    {
        cout << " " << memorija[j];
    }
    int a;
    cin >> a;
    return 0;
}
```

Sl. 4.5. *Mjerenje vremena funkcije koja koristi pristup „Odozgo prema dolje“ za problem rezanja drva.*

```
C:\Users\david\OneDrive\documents\visual studio 2017\Projects\Zavrsni\Debug\Zavrsni.exe
Vrijeme izvodenja funkcije koja koristi memoizaciju je:  3849 ns
Najveci moguci prihod koji smo mogli dobiti rezanjem drva duljine 8 metara iznosi 22 kune.
U polju koje pretstavlja memoriju nalaze se sljedeŠi podaci:  1 5 8 10 13 17 18 22
```

Sl. 4.6. *Izlaz programa za testiranje povratne vrijednosti i vremena izvođenja funkcije koja koristi pristup „Odozgo prema dolje“.*

Na slici 4.5. prikazana je glavna funkcija za mjerenje vremena funkcije koja koristi pristup „Odozgo prema dolje“. U ovoj funkciji nakon što smo definirali i deklarirali polje sa cijenama, bilo je potrebno pomoću pomoćne funkcije Rezi_Drvo_Memoizacija_init() dinamički alocirati prostor za polje koje ćemo koristiti za memoizaciju. Nakon toga smo kao i u prethodnom primjeru mjerili vrijeme izvođenja funkcije. Kao rezultat smo dobili vrijeme izvođenja koje iznosi 3849 ns

(Sl. 4.6.). Možemo vidjeti da je ovo vrijeme značajno manje od onoga kod obične rekurzivne implementacije gdje je ono iznosilo 18389 ns. Zaključujemo da smo ovakvom implementacijom dobili mnogo efikasnije rješenje.

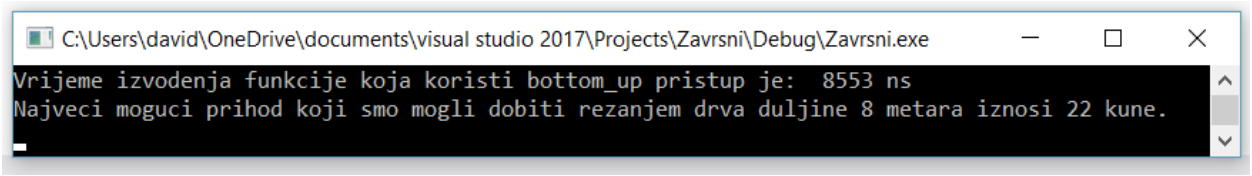
```
int main()
{
    int cijene[] = { 1, 5, 8, 9, 10, 17, 17, 20 };
    int duljina_drva = sizeof(cijene) / sizeof(cijene[0]);

    auto start = get_time::now();
    Rezi_Drvo_BottomUp(cijene, duljina_drva);;
    auto end = get_time::now();
    auto diff = end - start;
    cout << "Vrijeme izvodenja funkcije koja koristi bottom_up pristup je:  " <<
        chrono::duration_cast<ns>(diff).count() << " ns " << endl;

    cout << "Najveci moguci prihod koji smo mogli dobiti rezanjem drva duljine "
        << duljina_drva << " metara iznosi "
        << Rezi_Drvo_BottomUp(cijene, duljina_drva) << " kune." << endl;

    int a;
    cin >> a;
    return 0;
}
```

Sl. 4.7. *Mjerenje vremena funkcije koja koristi pristup „Odozdo prema gore“ za problem rezanja drva.*



Sl. 4.8. *Izlaz programa za testiranje povratne vrijednosti i vremena izvođenja funkcije koja koristi pristup „Odozdo prema gore“.*

Na slici 4.7. prikazana je glavna funkcija za mjerenje vremena izvođenja funkcije koja koristi pristup „Odozdo prema gore“. Kao rezultat smo dobili vrijeme izvođenja koje iznosi 8553 ns. Možemo vidjeti da je vrijeme izvođenja funkcije pomoću ovoga pristupa nešto sporije u odnosu na pristup „Odozgo prema dolje“.

5. ZAKLJUČAK

U ovome radu opisana je tehnika dinamičkog programiranja. Vidjeli smo da je ovo vrlo učinkovita tehnika pogodna za rješavanje određenog tipa složenih problema. Osnovna ideja ovakvog načina rješavanja problema je izbjegavanje višestrukog izračunavanja iste vrijednosti pomoću dodatnog prostora u koji spremamo međurezultate. U glavnom dijelu rada razradili smo „Problem rezanja drva“ te smo ga implementirali pomoću rekurzivne metode te pomoću tehnike dinamičkog programiranja. Vidjeli smo da rekurzivnom metodom dobivamo vrlo sporu implementaciju koja nije učinkovita za probleme sa velikim brojem ulaznih podataka. Nakon toga smo implementirali dva dinamička pristupa: „Pristup „Odozdo prema gore“ te pristup „Odozgo prema dolje“. Rezultati dobiveni ovakvim pristupima su značajno učinkovitiji po pitanju vremenske složenosti. Oba pristupa dinamičkog programiranja pohranjuju rezultate rješenja podproblema. Kod pristupa „Odozgo prema dolje“ gdje koristimo memoizaciju, polje rješenja popunjavamo „na zahtjev“ odnosno onda dok dođemo do onog rješenja koje nije u memoizacijskom polju dok kod pristupa „Odozdo prema gore“, sva mjesta u polju popunjavamo jedno po jedno, počevši od prvog mjesta u polju. U memoizacijskom pristupu sva polja memoizacijske tablice ne moraju nužno biti popunjena, dok se u pristupu „Odozdo prema gore“ popunjavaju sve vrijednosti elemenata polja. Navedena dva pristupa su vrlo slične učinkovitosti. Ovisno o problemu koji rješavamo ponekad može biti bolji prvi pristup, a ponekad drugi. Također postoje problemi koji se mogu efikasno riješiti samo jednim od ova dva pristupa.

LITERATURA

[1] Dinamičko programiranje, Wikipedija, https://en.wikipedia.org/wiki/Dynamic_programming , pristupljeno 17. lipnja 2017.

[2] Ivan Budiselić, Dinamičko programiranje i problemi raspoređivanja, Seminarski rad za poslijediplomski predmet, 19. rujna 2011.

[3] Dinamičko programiranje, Frane Kurtović i Martin Gluhak, FER Zagreb, https://www.fer.unizg.hr/_download/repository/Dinamika1_web.pdf, pristupljeno 10.lipnja 2017.

[4] Uvod u dinamičko programiranje, https://www.youtube.com/watch?v=W2ote4jCuYw&list=PLyEvk8ZeQDMVbsg7CEfT0NV3s3GkMx1vN , pristupljeno 20.lipnja 2017.

[5] Optimalna podstruktura, Wikipedija, https://en.wikipedia.org/wiki/Optimal_substructure , pristupljeno 17.lipnja 2017.

[6] Biblioteka „Chrono“ C++, http://www.cplusplus.com/reference/chrono/ , pristupljeno 21.lipnja 2017

SAŽETAK

Zadatak ovog rada bio je opisati i na konkretnom primjeru implementirati tehniku dinamičkog programiranja, Ovom tehnikom početni složeni problem rastavljamo na podprobleme manje složenosti koji su međusobno zavisni. Svaki podproblem se rješava najviše jednom, čime možemo izbjeći višestruko računanje numeričkih karakteristika istog stanja. Osnovna ideja na koju se oslanjaju algoritmi dinamičkog programiranja je da se svaki dobiveni međurezultat, odnosno rješenje podproblema, spremi te zatim kada se sljedeći puta naiđe na taj isti podproblem, izbjegne njegovo ponovno rješavanje. Ovakva tehnika pohrane rješenja podproblema naziva se memoizacija. Dva osnovna pristupa pomoću kojih možemo rješavati probleme dinamičkim programiranjem su: pristup „Odozgo prema dolje“ i pristup „Odozdo prema gore“. U glavnom dijelu rada navedeni pristupi su implementirani na primjeru „Rezanje drva“ te je izmjereno vrijeme izvođenja dobivenih funkcija. Na kraju smo zaključili da problem riješen ovom tehnikom ima znatno brže vrijeme izvođenja od onog riješenog nedinamičkom metodom.

Ključne riječi: C++ implementacija, dinamičko programiranje, memoizacija, optimalna struktura, podproblem.

ABSTRACT

Dynamic programming

The task of this paper was to describe a dynamic programming technique and then implement solution for specific problem. Using this technique, we break down the initial complex problem into a collection of simpler subproblems. Each subproblem is solved once and its solution had been stored. The next time the same subproblem occurs, instead of recomputing its solution, one simply looks up the previously computed solution. The technique of storing solutions to subproblems instead of recomputing them is called "memoization". There are two ways of doing this: with Top-down approach and with Bottom-up approach. In the main part of this paper these approaches were implemented in the example "Rod cutting" and the time taken to perform the obtained functions was measured. In the end, we have concluded that the problem solved by this technique has much faster execution time than one solved with other method.

Keywords: C++ implementation, dynamic programming, memorization, optimal substructure, subproblem.

ŽIVOTOPIS

David Tkalčec rođen je 17. studenog 1995. godine u Virovitici. Završio je Osnovnu školu Petra Preradovića u Pitomači te je nakon završene osnovne škole upisao opći smjer u Gimnaziji Petra Preradovića u Virovitici. Maturirao je 2014. godine, nakon čega je upisao Preddiplomski sveučilišni studij Računarstva na Fakultetu elektrotehnike, računarstva i informacijskih tehnologija u Osijeku.