Dinamičko programiranje i problemi raspoređivanja

Ivan Budiselić

Seminarski rad za poslijediplomski predmet
Postupci raspoređivanja

19. rujna 2011.

Sažetak

Seminarski rad daje praktični pogled na dinamičko programiranje kao tehniku dizajna algoritama i primjenu tehnike na problem raspoređivanja. Tehnika je uvedena kroz jednostavne primjere traženja n-tog Fibonaccijevog broja i rješavanje problema ruksaka. Osim uvoda u dizajn algoritama dinamičkim programiranjem, za svaki algoritam prikazan u radu navedena je i objašnjena vremenska i ponekad prostorna složenost u Landau notaciji.

Nakon detaljnih rješenja uvodnih primjera, općenito su opisani kriteriji točnosti i primjenjivosti dinamičkog programiranja i dva vida dinamičkog programiranja: memoizacija rekurzivnih funkcija i dinamičko programiranje od dna prema vrhu. Za oba vida su kroz rad navedeni brojni primjeri kao i ograničenja i prednosti u odnosu na alternativu.

Konačno, prikazana je primjena dinamičkog programiranja na pronalaženje optimalnog rješenja malih instanci dviju inačica problema raspoređivanja.

Sadržaj

Uvod

Dinamičko programiranje spominje se kao tehnika u operacijskim istraživanjima u okviru matematičkog optimiranja i kao algoritamska tehnika u računalnom programiranju. U ovom radu je prikazan praktični pogled na dinamičko programiranje, specifično u kontekstu primjene na pronalaženje optimalnih rješenja malih instanci problema raspoređivanja.

U primjeni dinamičkog programiranja, slično kao kod primjene tehnike *podijeli-pa-vladaj*, dobivena instanca problema sustavno se razlaže na jednostavnije instance istog problema. Rješenja jednostavnijih instanci koriste se onda za rješavanje početne instance problema. Međutim, dok je cilj tehnike podijeli-pa-vladaj instancu podijeliti na *nezavisne* instance, dinamičko programiranje primjenjivo je upravo u slučaju kada se jednostavnije instance preklapaju.

Dinamičkim programiranjem može se naivne algoritme za mnoge kombinatoričke probleme značajno ubrzati uz veću potrošnju spremničkog prostora. Iako formalno neispravno, za intuiciju se može reći da se dinamičkim programiranjem algoritam eksponencijalne vremenske složenosti svodi na polinomnu vremensku složenost uz povećanje prostorne složenosti za polinomni faktor. Čak i kada je visoka prostorna složenost prihvatljiva, dinamičko programiranje često nije prikladno za rješavanje nekog problema, tj. ne donosi ubrzanje u izvođenju. U radu su opisani uvjeti koje mora ispunjavati neki algoritam kako bi se njegova vremenska složenost mogla smanjiti dinamičkim programiranjem.

U najširem smislu, problemi raspoređivanja traže optimalnu organizaciju zadataka na nekom skupu strojeva, uz različite uvjete nad zadacima i različite funkcije cilja. Različite inačice problema raspoređivanja prirodno se pojavljuju u organizaciji proizvodnje, ali i u računalnim sustavima, na primjer u raspoređivaču zadataka u operacijskom sustavu. Značajan dio praktično

zanimljivih inačica problema raspoređivanja je NP-potpun ili NP-težak[1], pa se tipično rješava heurističkim algoritmima koji pronalaze "dovoljno dobra" rješenja problema. Ipak, pokazuje se da je za dio inačica moguće osmisliti algoritam zasnovan na dinamičkom programiranju koji pronalazi optimalan poredak u pseudopolinomnom vremenu[2] i može biti primjenjiv za manje instance problema čija optimalnost je od posebnog značaja.

Ostatak rada organiziran je na sljedeći način. U poglavlju 1 je prikazan uvod u tehniku dinamičkog programiranja kroz detaljnu analizu dva jednostavna problema: traženje n-tog člana Fibonaccijevog slijeda i rješavanje 0-1 problema ruksaka. Poglavlje 2 daje općeniti pregled dinamičkog programiranja kroz definiciju kriterija točnosti i primjenjivosti dinamičkog programiranja i analizu dva vida dinamičkog programiranja: memoizacije rekurzivnih funkcija i dinamičkog programiranja od dna prema vrhu. Konačno, u poglavlju 3 je prikazana primjena dinamičkog programiranja na pronalaženje optimalne organizacije zadataka u dvije inačice problema raspoređivanja.

[1]Definicija pojmova *NP-potpun* i *NP-težak* zasniva se na teoriji složenosti jezika. Klasa jezika P sadrži sve jezike koje je moguće u polinomnom broju koraka prihvatiti *determinističkim* Turingovim strojem. S druge strane, klasa jezika *NP* sadrži sve jezike koje je moguće prihvatiti u polinomnom broju koraka *nedeterminističkim* Turingovim strojem. Nadalje, neki jezik L je NP-potpun ako i samo ako je u klasi NP i svaki jezik L' iz NP je *polinomno-reducibilan* u L. Jezik L' je polinomno-reducibilan u L ako i samo ako se za svaki niz *w'* pripadnost jeziku *L'* može ispitati tako da se u polinomnom vremenu konstruira niz w i ispita njegova pripadnost jeziku L. Tada vrijedi $w' \in L' \Leftrightarrow w \in L$. Jezik L jest NP-težak ako i samo ako je svaki jezik L' polinomno-reducibilan u L. Pritom, sam jezik L ne mora biti u klasi NP. Definicije iz teorije jezika prenose se na područje složenosti algoritama definiranjem preslikavanja iz opisa instance problema u niz nekog jezika. U skladu s definicijama, NP-potpune i NP-teške probleme nije moguće riješiti u polinomnom vremenu na determinističkom stroju ako je $P \neq NP$.

[2]Pseudopolinomno vrijeme izvođenja polinomno ovisi o vrijednosti nekog *broja* u opisu instance problema. Kako se vrijednost broja u bilo kojoj brojevnoj bazi na traku zamišljenog Turingovog stroja može zapisati u logaritamski broj ćelija, polinoman broj koraja je eksponencijalna funkcija duljine ulaznog niza. Preciznije objašnjenje dano je u odjeljku 2.1.

Poglavlje 1

Uvod u dinamičko programiranje

Dinamičko programiranje blisko je vezano uz rekurziju i matematičku indukciju. Svaki problem koji se rješava dinamičkim programiranjem moguće je svesti na računanje određenog člana nekog slijeda brojeva, pri čemu definicija slijeda ovisi o problemu koji se rješava. U općenitom slučaju, slijed će biti višeparametarski, gdje svaki parametar odgovara nekoj dimenziji problema, ali višeparametarski slijed može se pretvoriti i u jednoparametarski slijed nešto složenije definicije. Definicijom slijeda opisuje se kako se neka instanca problema razlaže na manje instance istog problema i kako se rješenja tih manjih instanci povezuju u rješenje početne instance. Zbog toga kao koristan uvod u dinamičko programiranje može poslužiti jednostavan problem računanja n-tog člana jednoparametarskog slijeda brojeva.

1.1 Fibonaccijev slijed

Mnogi sljedovi brojeva mogu se opisati rekurzivno, kroz konačan broj početnih članova i definiciju n-tog člana kao funkcije prethodnih članova. Na primjer, poznati slijed Fibonaccijevih brojeva definiran je početnim članovima nula i jedan, te n-tim članom koji je jednak zbroju prethodna dva. Matematički bismo ovu definiciju slijeda Fibonaccijevih brojeva mogli zapisati ovako:

$$F_0 = 0,$$
$$F_1 = 1,$$
$$F_n = F_{n-1} + F_{n-2}, \forall n > 1.$$

Ovom definicijom, slijed je potpuno definiran jer za bilo koji n možemo izračunati F_n uzastopnim uvrštavanjem. Na primjer,

$$\begin{aligned} F_3 &= F_2 + F_1 \\ &= (F_1 + F_0) + F_1 \\ &= (1 + 0) + 1 \\ &= 2. \end{aligned}$$

Za neke rekurzivne definicije slijeda, uključujući i Fibonaccijev slijed, moguće je odrediti *zatvorenu formu* [1]. Zatvorena forma je jednadžba koja definira n-ti član slijeda isključivo kao funkciju broja n, neovisno o prethodnim članovima slijeda. Zatvorena forma često omogućuje brže računanje n-tog člana od uzastopnog uvrštavanja, ponekad čak i u konstantnom vremenu, tj. neovisno o n. Ipak, određivanje zatvorene forme često je matematički zahtjevno, a za mnoge sljedove i nemoguće. Alternativno, n-ti član slijeda može se odrediti uz pomoć računala, što je prikazano u nastavku na primjeru Fibonaccijevog slijeda.

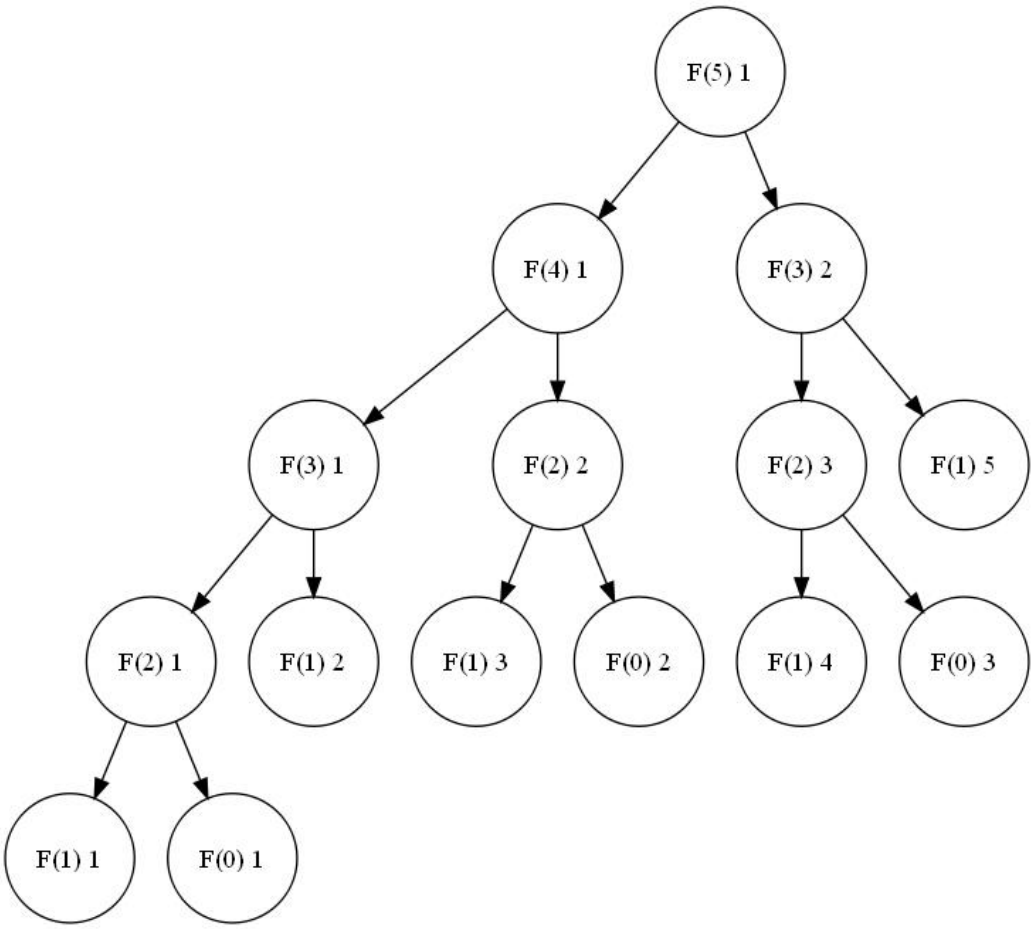
Naivni algoritam

```
1 Fibonacci(n):
2     ako je n = 0:
3         vrati 0
4     inace ako je n = 1:
5         vrati 1
6     inace:
7         vrati Fibonacci(n-1) + Fibonacci(n-2)
```

Ispis 1.1: Računanje n-tog Fibonaccijevog broja po definiciji.

Naivni algoritam za računanje n-tog člana Fibonaccijevog slijeda prikazan je pseudokodom u ispisu 1.1. Retci 2–5 osiguravaju ispravne vrijednosti za nulti i prvi član slijeda F_0 i F_1. Redak 7 računa n-ti član za $n > 1$ po definiciji, koristeći dva rekurzivna poziva funkcije. Sa aspekta čitljivosti, teško je zamisliti jasniji algoritam s obzirom da prikazani algoritam izravno modelira matematičku definiciju slijeda.

Slika 1.1 prikazuje stablo aktiviranja funkcije `Fibonacci` za $n = 5$. U zagradi u svakom čvoru nalazi se pripadni argument funkciji, a desno od



Slika 1.1: Stablo aktiviranja procedura za ispis 1.1 i $n = 5$.

zagrade se nalazi redni broj poziva funkcije s danim argumentom. Na primjer, funkcija se poziva dvaput s argumentom `3` i pet puta s argumentom `1`. Koristeći metodu supstitucije [2], lako je dokazati da je vrijeme izvođenja funkcije `Fibonacci` $\Theta(\phi^n)$[1] pri čemu je ϕ omjer zlatnog reza i iznosi nešto više od 1.618[2]. S obzirom da eksponencijalna funkcija raste vrlo brzo, ovaj algoritam nije primjeren za računanje n-tog Fibonaccijevog broja već za niske dvoznamenkaste n. Na primjer, ako bi se na nekom računalu funkcija `Fibonacci(30)` izvršavala jednu sekundu, poziv `Fibonacci(50)` izvršavao bi

[1]Definicija oznake Θ dana je u dodatku A.

[2]Iako je vrijeme izvođenja funkcije eksponencijalno, algoritam iz ispisa 1.1 ne računa n-ti Fibonaccijev broj u vremenu koje je eksponencijalna funkcija veličine ulaznih podataka. Naime, ulaz u algoritam je sam broj n, a njegov zapis u bilo kojoj brojevnoj bazi zauzima $\Theta(\log n)$ ćelija te je zbog toga algoritam mnogo lošiji od eksponencijalnog.

se više od četiri sata.

Osnovni razlog visoke vremenske složenosti algoritma iz ispisa 1.1 je višestruko izračunavanje rezultata za isti argument funkcije `Fibonacci`, što je prvi znak da se algoritam može ubrzati koristeći dinamičko programiranje.

Memoizirani naivni algoritam

Najjednostavniji način primjene dinamičkog programiranja na već gotovo rekurzivno rješenje je *memoizacija*. Memoizacija podrazumijeva pamćenje izračunatog rezultata za određene argumente funkcije kako bi se izbjeglo ponovno izračunavanje kada se funkcija pozove s istim argumentima.

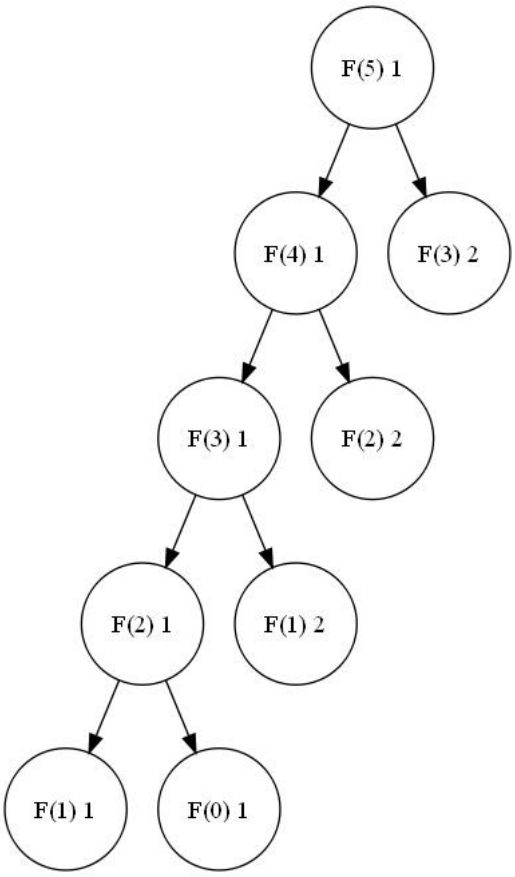
```
1 Fib je niz n+1 cijelih brojeva
2 inicijaliziraj Fib[i] := -1 za sve i <= n
3 Fib[0] := 0
4 Fib[1] := 1
5 Fibonacci(n):
6     ako je Fib[n] = -1:
7         Fib[n] := Fibonacci(n-1) + Fibonacci(n-2)
8     vrati Fib[n]
```

Ispis 1.2: Računanje n-tog Fibonaccijevog broja s memoizacijom.

Memoizirana funkcija `Fibonacci` prikazana je u ispisu 1.2. Retci 1–3 inicijaliziraju globalni niz brojeva u koji se pohranjuju izračunate vrijednosti. U retku 5, funkcija provjerava je li traženi rezultat već izračunat. Ako nije, rezultat se izračuna u retku 6 i pohrani na odgovarajuće mjesto u niz `Fib`. Konačno, u retku 7, kao rezultat funkcije vraća se vrijednost `Fib[n]`.

Na slici 1.2 prikazano je stablo aktiviranja funkcije `Fibonacci` za $n = 5$. Iz slike je očito da se poziv `Fibonacci(n-2)` u retku 6 uvijek izvršava u konstantnom vremenu, jer se tijekom izvršavanja poziva `Fibonacci(n-1)` postavi odgovarajuća vrijednost u `Fib[n-2]`. Indukcijom se lako može pokazati da je vrijeme izvođenja funkcije `Fibonacci(n)` $\Theta(n)$[3].

[3]S obzirom na to da je veličina ulaza u algoritam $\Theta(\log n)$, memoizirani algoritam računa n-ti Fibonaccijev broj u vremenu koje eksponencijalno ovisi o veličini ulaza. Kako je n-ti član Fibonaccijevog slijeda definiran linearnom rekurzivnom formulom, moguće ga je lako izračunati linearnim algoritmom u $\Theta(\log n)$ vremenu brzim potenciranjem karakteristične matrice, ali taj algoritam izlazi iz opsega ovog rada.



Slika 1.2: Stablo aktiviranja procedura za ispis 1.2 i $n = 5$.

Prostorna složenost memoiziranog algoritma je također $\Theta(n)$. U ovom slučaju, korištenjem dinamičkog programiranja nije došlo do asimptotskog povećanja prostorne složenosti s obzirom da i naivni algoritam ima istu prostornu složenost zbog programskog stoga koji se koristi za rekurzivne pozive.

Memoizacija se ponekad naziva dinamičkim programiranjem *od vrha prema dnu* jer se manja instanca problema rješava samo kada je njeno rješenje nužno za rješavanje veće instance. To svojstvo predstavlja osnovnu razliku u odnosu na "pravo" dinamičko programiranje, gdje se sustavno rješavaju instance od manjih prema većim, tj. *od dna prema vrhu*.

Rješenje dinamičkim programiranjem od dna prema vrhu

Za dinamičko programiranje od dna prema vrhu obično se koristi niz ili, u općenitom slučaju, višedimenzionalna matrica. Vrijednosti u nizu ili matrici sustavno se izračunavaju nekim redoslijedom koji ovisi o problemu koji se rješava.

```
1 Fib je niz n+1 cijelih brojeva
2 Fib[0] := 0
3 Fib[1] := 1
4 za i od 2 do n:
5     Fib[i] = Fib[i-1] + Fib[i-2]
```

Ispis 1.3: Računanje n-tog Fibonaccijevog broja dinamičkim programiranjem od dna prema vrhu.

Rješenje u ispisu 1.3 podsjeća na naivni algoritam uz zamjenu rekurzivnih poziva funkcije pristupima u niz `Fib`. Nakon izvođenja petlje u retcima 4 i 5, traženi n-ti član Fibonaccijevog slijeda nalazi se u `Fib[n]`.

Prikazani algoritam ima istu vremensku i prostornu složenost kao i memoizirani algoritam — $\Theta(n)$. Kako Fibonaccijevi brojevi rastu eksponencijalno brzo, u praksi bi za računanje trebalo koristiti tip podataka koji predstavlja cijele brojeve proizvoljnog broja znamenaka ili n-ti Fibonaccijev broj računati modulo nekog relativno malog broja. Preostalo ograničenje predstavlja prostorna složenost, tj. maksimalni n za koji je ovaj algoritam primjenjiv ovisi o dostupnoj količini spremničkog prostora. Ako je cilj isključivo odrediti n-ti član Fibonaccijevog slijeda za neki određeni n, a prethodni članovi nisu potrebni, moguće je prostornu složenost smanjiti na $\Theta(1)$ zamjenom niza `Fib` sa dvije varijable koje čuvaju samo zadnja dva izračunata člana[4].

1.2 Problem ruksaka

Naziv *problem ruksaka* odnosi se na cijeli razred problema koji se unutar sličnog modela razlikuju u ograničenjima. Gotovo sve zanimljive inačice problema ruksaka su *NP-potpune* [3]; drugim riječima, za njih nije moguće naći

[4]U poglavlju 2.4 je pokazano kako se sličan postupak može primijeniti i na mnogo složenije probleme i znatno smanjiti potrebnu količinu spremničkog prostora.

polinomni algoritam u okviru RAM modela računala ako je $P \neq NP$[5]. U ostatku poglavlja definiran je *0-1 problem ruksaka* i prikazani su različiti algoritmi koji problem rješavaju.

0-1 problem ruksaka

Neka je zadano n predmeta i dva cjelobrojna niza V i M, pri čemu je $V[i]$ *vrijednost*, a $M[i]$ *masa* i-tog predmeta. Dodatno, neka je zadan cijeli broj C koji predstavlja maksimalnu ukupnu masu predmeta koje je moguće pohraniti u ruksak. 0-1 problem ruksaka traži maksimalan zbroj vrijednosti predmeta u ruksaku, pri čemu njihova ukupna masa ne smije biti veća od C. Predmeti su nedjeljivi, tj. moguće je predmet ili uzeti u cijelosti ili ne uzeti uopće.

Pri prvom susretu s problemom ruksaka, nameće se ideja pohlepnog rješenja, na primjer punjenje ruksaka najvrjednijim predmetima ili predmetima s najvećim omjerom $V[i]/M[i]$. Za obje pohlepne strategije, protuprimjer je:

$$\begin{aligned} n &= 3 \\ V &= [5, 3, 3] \\ M &= [3, 2, 2] \\ C &= 4. \end{aligned}$$

Obje strategije za dani primjer uzimaju nulti predmet i daju vrijednost 5 uz masu 3 dok je optimalno rješenje uzeti preostala dva predmeta za ukupnu vrijednost 6 i masu 4.

Rekurzivno rješenje

Za formulaciju rekurzivnog rješenja definira se funkcija $R(i, m)$ koja daje najveću moguću vrijednost ruksaka koristeći neke od prvih $i + 1$ predmeta ukupne mase ne veće od m. Tražena maksimalna vrijednost ruksaka onda je $R(n - 1, C)$.

Preostaje definirati početne vrijednosti i opću formulu koja povezuje veću instancu problema s rješenjima manjih instanci. Početne vrijednosti lako je

[5]Preciznije rečeno, ako je $P \neq NP$, nije moguće konstruirati deterministički Turingov stroj koji u polinomnom broju koraka odgovara na pitanje je li zadana instanca problema ruksaka rješiva ili ne.

definirati za jednostavne slučajeve; ako je $i = 0$ tj. dozvoljeno je koristiti samo nulti predmet, očito je da će za sve ukupne mase m manje od mase nultog predmeta $M[0]$ vrijednost funkcije R biti nula. Za sve vrijednosti parametra m koje su veće ili jednake masi predmeta $M[0]$, vrijednost funkcije R bit će jednaka vrijednosti nultog predmeta $V[0]$ jer je optimalno rješenje danog potproblema staviti nulti predmet u ruksak.

Matematički zapisano:

$$R(0, m) = \begin{cases} 0, & \text{ako je } m < M[0] \\ V[0], & \text{ako je } m \geq M[0]. \end{cases}$$

Način razmišljanja potreban za opću formulu zasniva se na matematičkoj indukciji: uz pretpostavku da je poznato rješenje za sve manje vrijednosti parametara i i m, potrebno je definirati vrijednost funkcije $R(i, m)$. Za svaki novi predmet, na raspolaganju su dvije opcije: predmet se preskače i ne ulazi u ruksak ili se predmet stavlja u ruksak.

U svakoj instanci problema, i-ti predmet je moguće preskočiti. U tom slučaju, vrijednost funkcije $R(i, m)$ jednaka je vrijednosti $R(i-1, m)$, tj. maksimalnoj vrijednosti ruksaka maksimalne mase m u kojem se nalaze neki od prvih $i - 1$ predmeta.

Predmet je moguće dodati u ruksak samo ako njegova masa nije veća od maksimalne dozvoljene mase za danu instancu problema tj. vrijedi $M[i] \leq m$. U tom slučaju, moguće je odabrati hoće li se predmet dodati u ruksak ili ne, a odabire se ona opcija koja daje veću vrijednost ruksaka. Ako se predmet ne dodaje u ruksak, vrijednost funkcije je $R(i - 1, m)$. U suprotnom, ako se predmet dodaje u ruksak, s nekima od prethodnih $i - 1$ predmeta mogla se zauzeti maksimalna ukupna masa $m - M[i]$ pa je nova vrijednost ruksaka nakon dodavanja novog predmeta $R(i-1, m-M[i])+V[i]$. Ovdje je pohlepni izbor potpuno opravdan jer je ukupna masa ruksaka u potproblemu fiksno postavljena na m, bez obzira uzmemo li predmet u ruksak ili ne.

Konačno, funkcija R definirana je na sljedeći način:

$$R(i, m) = \begin{cases} 0, & \text{ako je } i = 0 \text{ i } m < M[0] \\ V[0], & \text{ako je } i = 0 \text{ i } m \geq M[0] \\ R(i-1, m), & \text{ako je } i > 0 \text{ i } m < M[i] \\ max\{R(i-1, m), & \\ \quad R(i-1, m - M[i]) + V[i]\}, & \text{ako je } i > 0 \text{ i } m \geq M[i]. \end{cases}$$

U skladu s razmatranjima u uvodu, funkcija R može poslužiti za definiranje dvoparametarskog slijeda cijelih brojeva. Nadalje, dvoparametarski slijed

mogao bi se svesti na jednoparametarski slijed nešto složenije definicije, ali u praksi to najčešće nimalo ne olakšava rješavanje problema.

```
1  R(i, m):
2      ako je i = 0:
3          ako je m < M[0]:
4              vrati 0
5          inace:
6              vrati V[0]
7      inace:
8          pom := R(i-1, m)
9          ako je m >= M[i]:
10             pom := max(pom, R(i-1, m-M[i]) + V[i])
11         vrati pom
```

Ispis 1.4: Rekurzivno rješenje 0-1 problema ruksaka.

Rekurzivno rješenje 0-1 problema ruksaka, prikazano u ispisu 1.4, izravni je prijevod definicije funkcije R u pseudokod; retci 2–4 odnose se na prvi slučaj, retci 5 i 6 na drugi slučaj, redak 8 na treći slučaj i retci 9 i 10 na četvrti slučaj.

Lako je vidjeti da je vremenska složenost ovog algoritma u najgorem slučaju $\Omega(2^n)$. Na primjer, ako se uzmu predmeti mase jedan tako da se u svakom pozivu za $i > 0$ funkcija poziva dvaput s argumentima $i - 1$ i m odnosno $m-1$. Složenijom analizom može se vidjeti da je vremenska složenost u najgorem slučaju i $\Theta(2^n)$. Prostorna složenost je $\Theta(n)$ zbog stoga na koji se pohranjuju okviri funkcija pri rekurzivnim pozivima.

Memoizirano rekurzivno rješenje

Slično kao u računanju Fibonaccijevih brojeva, rekurzivno rješenje iz ispisa 1.4 može se znatno ubrzati memoizacijom.

```
1  Rmemo je matrica n x (C+1) cijelih brojeva
2  inicijaliziraj Rmemo[i][m] := -1 za sve i < n i m <= C
3  R(i, m):
4      ako je Rmemo[i][m] = -1:
5          ako je i = 0:
6              ako je m < M[0]:
```

```
7                Rmemo[i][m] := 0
8            inace:
9                Rmemo[i][m] := V[0]
10       inace:
11           Rmemo[i][m] := R(i-1, m)
12           ako je m >= M[i]:
13               Rmemo[i][m] := max(pom, R(i-1, m-M[i]) + V[i])
14   vrati Rmemo[i][m]
```

Ispis 1.5: Memoizirano rekurzivno rješenje 0-1 problema ruksaka.

Memoizirano rješenje iz ispisa 1.5 umjesto vraćanja optimalne vrijednosti kada je ona izračunata, vrijednost pohranjuje u matricu `Rmemo`. Vrijednosti u matrici se prije prvog poziva funkcije inicijaliziraju na -1 što je za problem ruksaka vrijednost koju je nemoguće ostvariti i zbog toga može poslužiti kao indikator da vrijednost u toj ćeliji još nije izračunata.

Vremenska složenost memoiziranog algoritma iz ispisa 1.5 je $\Theta(nC)$ u najgorem slučaju[6]. Prostorna složenost je zbroj veličine matrice `Rmemo` i prostorne složenosti stoga koji podržava rekurzivno izvođenje $\Theta(nC) + \Theta(n) = \Theta(nC)$. U slučaju 0-1 problema ruksaka, prostorna složenost memoiziranog rekurzivnog algoritma je za faktor C veća od prostorne složenosti naivnog rekurzivnog algoritma. S obzirom na to da je C tipično mnogo veći od n, prostorna složenost je osnovno ograničenje primjenjivosti ovog algoritma.

Rješenje dinamičkim programiranjem od dna prema vrhu

Vrijednost funkcije R moguće je izračunati i od dna prema vrhu, od manjih vrijednosti argumenata prema većima. Preciznije rečeno, vrijednosti u $n \times (C + 1)$ matrici računaju se red po red, slijeva nadesno.

Retci 2 i 3 u ispisu 1.6 osiguravaju ispravne početne vrijednosti za slučaj $i = 0$. Petlja u retcima 4–8 računa funkciju R za preostale vrijednosti argumenta `i`. Tražena vrijednost za početnu instancu problema nalazi se nakon izvođenja petlje u `R[n-1][C]`.

```
1 R je matrica n x (C+1) cijelih brojeva
2 inicijaliziraj R[0][m] := 0 za sve m < M[0]
3 inicijaliziraj R[0][m] := V[0] za M[0] <= m <= C
```

[6]Vidi *Određivanje vremenske i prostorne složenosti* u poglavlju 2.3.

```
4 za i od 1 do n-1:
5     za m od 0 do min{M[i]-1, C}:
6         R[i][m] := R[i-1][m]
7     za m od M[i] do C:
8         R[i][m] := max{R[i-1][m], R[i-1][m-M[i]] + V[i]}
```

Ispis 1.6: Rješenje 0-1 problema ruksaka dinamičkim programiranjem od dna prema vrhu.

Kako se računanje svake vrijednosti u matrici `R` obavlja u konstantnom broju operacija, vremenska složenost algoritma je $\Theta(nC)$. Prostorna složenost je određena veličinom matrice `R` i iznosi $\Theta(nC)$.

U nastavku je prikazano kako algoritam rješava navedeni protuprimjer za pohlepne algoritme

$$\begin{aligned} n &= 3 \\ V &= [5, 3, 3] \\ M &= [3, 2, 2] \\ C &= 4. \end{aligned}$$

	0	1	2	3	4
0	0	0	0	5	5
1					
2					

Slika 1.3: Vrijednosti u matrici `R` nakon inicijalizacije.

Matrica `R` nakon inicijalizacije u retcima 2 i 3 prikazana je na slici 1.3. Kako je $n = 3$ i $C = 4$, matrica ima tri retka i pet stupaca. Ćelija u i-tom retku i m-tom stupcu sadrži najveću ukupnu vrijednost ruksaka koristeći neke od prvih i predmeta sa masom manjom ili jednakom m.

Na slici 1.4 je prikazana matrica `R` nakon prve iteracije petlje u retcima 4–8. U prva dva stupca nalazi se vrijednost nula jer dozvoljena masa nije dovoljna za uzimanje prvog predmeta u ruksak. Crvenom bojom su označene vrijednosti u sljedeća tri stupca. Računanje ovih vrijednosti odvija se u retku 8 algoritma jer je dozvoljena masa u ovim instancama dovoljno velika da se u ruksak doda prvi predmet.

	0	1	2	3	4
0	0	0	0	5	5
1	0	0	3	5	5
2					

Slika 1.4: Vrijednosti u matrici `R` nakon prve iteracije petlje u retcima 4–8.

	0	1	2	3	4
0	0	0	0	5	5
1	0	0	3	5	5
2	0	0	3	5	6

Slika 1.5: Vrijednosti u matrici `R` na kraju izvođenja algoritma.

Na slici 1.5 je prikazano konačno stanje matrice `R` nakon završetka rada algoritma. Pohlepno rješenje vrijednosti pet nalazi se u predzadnjem stupcu. Optimalno rješenje u kojem se u ruksak uzimaju posljednja dva predmeta nalazi se u zadnjem stupcu.

Rekonstrukcija rješenja

U optimizacijskim problemima često je osim optimalne vrijednosti potrebno odrediti i kako tu optimalnu vrijednost postići. Na primjer, za 0-1 problem ruksaka osim maksimalne vrijednosti ruksaka može nas zanimati koje predmete treba pohraniti u ruksak. Ovo proširenje problema naziva se *rekonstrukcija rješenja*.

Osnovna ideja rekonstrukcije rješenja je proširiti algoritam tako da se pri rješavanju svakog potproblema, osim određivanja maksimalne vrijednosti, zapiše i korak koji je odabran kako bi se dani potproblem povezao s nekim manjim potproblemom.

```
1 R je matrica n x (C+1) cijelih brojeva
2 odMase je matrica n x (C+1) cijelih brojeva
```

```
3  inicijaliziraj R[0][m] := 0 za sve m < M[0]
4  inicijaliziraj R[0][m] := V[0] za M[0] <= m <= C
5  inicijaliziraj odMase[0][m] := m za sve m < M[0]
6  inicijaliziraj odMase[0][m] := m - M[0] za M[0] <= m <= C
7  za i od 1 do n-1:
8      za m od 0 do min{M[i]-1, C}:
9          R[i][m] := R[i-1][m]
10         odMase[i][m] := m
11     za m od M[i] do C:
12         ako je R[i-1][m] >= R[i-1][m-M[i]] + V[i]:
13             R[i][m] := R[i-1][m]
14             odMase[i][m] := m
15         inace:
16             R[i][m] := R[i-1][m-M[i]] + V[i]
17             odMase[i][m] := m - M[i]
```

Ispis 1.7: Proširenje algoritma iz ispisa 1.6 za rekonstrukciju rješenja.

U ispisu 1.7 prikazano je proširenje rješenja 0-1 problema ruksaka za rekonstrukciju rješenja. Algoritam koristi dodatnu $n \times (C+1)$ matricu `odMase`. U i-ti redak i m-ti stupac matrice `odMase` pohranjuje se gornja granica za ukupnu masu ruksaka prije odluke o dodavanju i-tog predmeta u ruksak maksimalne mase m. U retcima 9 i 13, i-ti predmet se ne dodaje u ruksak pa se u matricu `odMase` pohranjuje vrijednost `m`, dok se u 16. retku pohranjuje vrijednost `m - M[i]` jer se predmet dodaje u ruksak[7].

```
1  Rekonstruiraj(i, m):
2      ako je i > 0:
3          Rekonstruiraj(i-1, odMase[i][m])
4      ako je odMase[i][m] = m:
5          ispisi("Preskoci predmet " + i)
6      inace:
7          ispisi("Dodaj predmet " + i)
```

Ispis 1.8: Rekurzivna rekonstrukcija rješenja koristeći matricu `odMase`.

Rekurzivna rekonstrukcija rješenja 0-1 problema ruksaka koristeći matricu `odMase` prikazana je u ispisu 1.8. S obzirom na to da se rekonstrukcija

[7]Zbog jednostavnosti 0-1 problema ruksaka, očito je da bi za rekonstrukciju rješenja bilo dovoljno pamtiti samo je li optimalno u određenom potproblemu uzeti i-ti predmet ili ne. Ipak, prikazano rješenje bliže je tipičnom slučaju gdje se svaki potproblem razlaže na veći broj potproblema uz složeniju lokalnu odluku.

nužno mora obavljati od najopćenitije instance problema, u retcima 2 i 3 se rekurzivno rekonstruira rješenje za manju instancu koja je dovela do optimalnog rješenja. Konačno, u retcima 4–7 ispisuje se odluka o i-tom predmetu. Rekonstrukcija rješenja se dobiva pozivom `Rekonstruiraj(n-1, C)`. Odluke o predmetima će se zbog rekurzivnog poziva ispisati redom, od nultog predmeta do predmeta $n-1$.

Ako je potrebno izbjeći rekurzivnu rekonstrukciju rješenja, moguće je predmete prije računanja matrica `R` i `odMase` okrenuti ili promijeniti algoritam tako da predmete obrađuje u obrnutom smjeru.

	0	1	2	3	4
0	0	1	2	0	1
1	0	1	0	0	1
2	0	1	0	0	2

Slika 1.6: Vrijednosti u matrici `odMase` na kraju izvođenja algoritma.

Na slici 1.6 prikazana je matrica `odMase` za primjer koji je rješavan u ovom poglavlju. Crvenom bojom su označene vrijednosti koje utječu na ispis poziva `Rekonstruiraj(2, 4)`:

```
Preskoci predmet 0
Dodaj predmet 1
Dodaj predmet 2
```

Poglavlje 2

Primjenjivost i tehnike dinamičkog programiranja

Kao nastavak na uvodne probleme računanja n-tog Fibonaccijevog broja i rješavanja problema ruksaka, u ovom poglavlju je dan općeniti pregled dinamičkog programiranja kao tehnike za dizajn algoritama. U poglavlju 2.1 formalno je definirana klasa NP-potpunih problema koji se mogu riješiti dinamičkim programiranjem. U poglavlju 2.2 opisani su kriteriji točnosti i primjenjivosti dinamičkog programiranja. Kriterij točnosti izravno se može primijeniti i na rekurzivno rješenje nekog problema, dok kriterij primjenjivosti ukazuje ima li smisla neko rekurzivno rješenje pretvoriti u rješenje dinamičkim programiranjem. Konačno, neka ograničenja i prednosti memoizacije i dinamičkog programiranja od dna prema vrhu opisana su u poglavljima 2.3 i 2.4.

2.1 Rješavanje *NP-potpunih* problema u "polinomnom vremenu"

Kao što je navedeno u poglavlju 1.2, mnoge su inačice problema ruksaka, pa tako i 0-1 problem ruksaka, NP-potpuni problemi. Drugim riječima, kada bismo pronašli polinomni algoritam za rješavanje 0-1 problema ruksaka, to bi bio valjan dokaz da je $NP = P$. Algoritam dinamičkim programiranjem od dna prema vrhu iz ispisa 1.6 ima vremensku složenost $\Theta(nC)$, gdje ne n broj predmeta, a C kapacitet ruksaka.

Zbog neformalnog i nedovoljno preciznog preslikavanja teorije složenosti

jezika u intiutivnu složenost algoritama, čini se da je prikazani algoritam polinomno rješenje NP-potpunog 0-1 problema ruksaka. Ipak, pokazuje se da je rješenje samo polinomno u nC, ali ne i u duljini opisa instance problema, tj. ulaznih parametara. Duljina opisa instance problema iznosi $\Theta(\log n + n \log Vmax + n \log Mmax + \log C)$, gdje su $Vmax$ i $Mmax$ redom maksimalna vrijednost i masa nekog od n predmeta. Funkcija nC nije polinom nad prikazanim izrazom. Ova činjenica može se jasnije vidjeti ako pretpostavimo instancu u kojoj vrijedi $Vmax = O(C)$ i $Mmax = O(C)$. Tada je duljina opisa instance $O(n \log C)$.

U skladu s prethodnim razmatranjem, algoritam za rješavanje 0-1 problema ruksaka dinamičkim programiranjem naziva se *pseudopolinomni algoritam* [3]. Svi NP-potpuni problemi koji se mogu riješiti pseudopolinomnim algoritmom su *brojevni problemi*. Brojevni problemi u opisu instanci sadrže broj čiju vrijednost nije moguće ograničiti niti jednim polinomom nad duljinom opisa instance. Iz same definicije je očito da ako vremenska složenost nekog algoritma polinomno ovisi o tom broju, nikako ne može polinomno ovisiti o duljini opisa instance pa algoritam ne može biti polinomno rješenje danog problema. Konačno, NP-potpuni problemi koje nije moguće riješiti pseudopolinomnim algoritmom nazivaju se NP-potpuni *u punom smislu* (engl. *NP-complete in the strong sense*). Primjer NP-potpunog problema u punom smislu je problem trgovačkog putnika.

2.2 Kriterij točnosti i primjenjivosti dinamičkog programiranja

Najvažnije svojstvo svakog algoritma bez sumnje je njegova točnost. Za dinamičko programiranje, kriterij točnosti naziva se *optimalna podstruktura* (engl. *optimal substructure*) [2]. Optimalna podstruktura zahtjeva da se optimalno rješenje neke instance problema može izračunati iz optimalnih rješenja drugih, u nekom smislu "manjih", instanci istog problema. Dokaz da rješenje zadovoljava ovaj kriterij često se provodi zaključivanjem iz kontradikcije.

Ključna posljedica kriterija optimalne podstrukture je zahtjev za nezavisnost odluka u svakom stanju u *prostoru stanja*. Prostor stanja je skup svih instanci problema koje se rješavaju primjenom dinamičkog programiranja. Na primjer, za 0-1 problem ruksaka, prostor stanja je dimenzija $n \times (C + 1)$, a rješenja svih instanci pohranjena su u matrici istih dimenzija. Dimenzije prostora stanja moraju u potpunosti opisati potproblem koji se rješava tako

da se ispitivanjem mogućih odluka koje ovise isključivo o opisu trenutnog potproblema može doći do jednostavnijih potproblema koji su prethodno riješeni. Na primjer, ako 0-1 problem ruksaka proširimo zahtjevom da se u ruksak nikada ne smiju dodati susjedni elementi, prostor stanja treba proširiti logičkom vrijednošću koja u donošenju odluke o i-tom predmetu pokazuje je li predmet $i-1$ u ruksaku.

Iz rješenja problema iz poglavlja 1, čini se da je dinamičko programiranje u općenitom smislu jednostavna transformacija rekurzivnog rješenja u iterativno rješenje ili čak jednostavna primjena memoizacije nad argumentima funkcije. Ipak, lako je pokazati da takvom transformacijom mnogih rekurzivnih rješenja neće biti ostvareno nikakvo ubrzanje algoritma.

```
1 R(i, m, v):
2     ako je i = n:
3         vrati v
4     inace:
5         pom := R(i+1, m, v)
6         ako je m + M[i] <= C:
7             pom := max{pom, R(i+1, m+M[i], v+V[i])}
8         vrati pom
```

Ispis 2.1: Rekurzivno rješenje 0-1 problema ruksaka kroz iscrpno pretraživanje mogućih rješenja

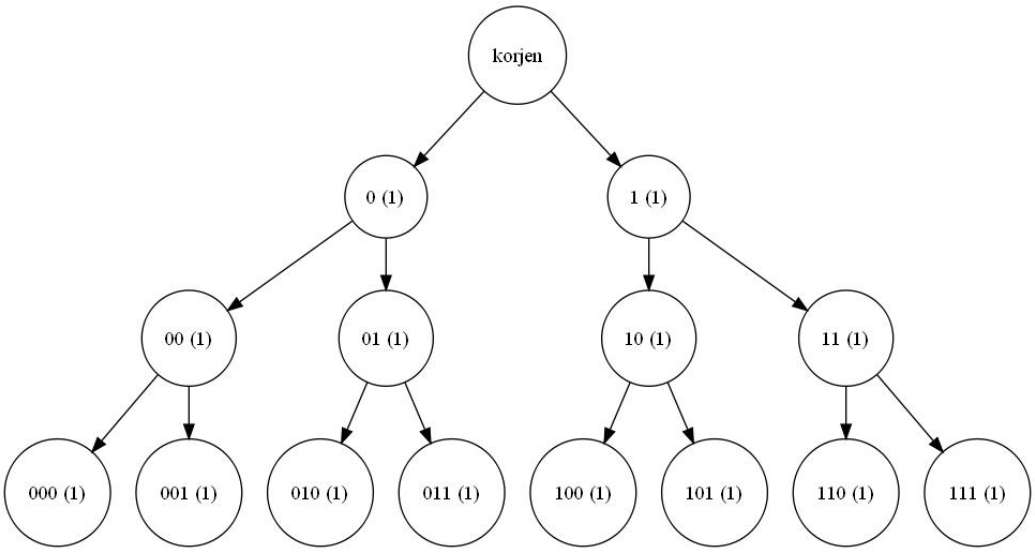
Na primjer, u ispisu 2.1 je prikazano rekurzivno rješenje 0-1 problema ruksaka koje generira sve moguće skupove predmeta čija ukupna masa ulazi u kapacitet ruksaka i vraća maksimalnu vrijednost koju je među tim skupovima moguće postići. Preciznije, funkcija `R(i, m, v)` vraća maksimalnu vrijednost koju je moguće staviti u ruksak ako je do sada odabran neki broj od prvih `i-1` predmeta ukupne mase `m` i vrijednosti `v`. Traženo rješenje dobiva se onda jednostavno pozivom `R(0, 0, 0)`.

U najgorem slučaju, vremenska složenost ovog algoritma je $\Theta(2^n)$, na primjer ako je moguće sve predmete staviti u ruksak. Prostorna složenost je jednaka maksimalnoj visini stoga i iznosi $\Theta(n)$.

Kao uvid u izvođenje ovog algoritma, ilustrativno je promotriti sljedeći razred instanci 0-1 problema ruksaka:

$$M = [1, 2, 4, \ldots, 2^{n-1}]$$
$$C = 2^n.$$

Rečeno riječima, za svaki n i bilo kakve vrijednosti predmeta V, mase predmeta su redom različite potencije broja dva, a kapacitet ruksaka je dovoljno velik da u njega stanu svi predmeti. Za instancu iz ovog razreda, argument `m` u svakom pozivu funkcije `R` predstavlja bit-masku skupa predmeta koji su prethodno dodani u ruksak.



Slika 2.1: Stablo aktiviranja procedura za ispis 2.1 i $n = 3$.

Čvorovi u stablu aktiviranja sa slike 2.1 označeni su samo vrijednostima parametra `m` u binarnom zapisu. Kao što je i očekivano, stablo ima 2^n listova i ukupno $2^{n+1} - 1$ čvorova. U zagradama pokraj vrijednosti parametra `m`, prikazan je broj poziva funkcije `R` s tim istim argumentom. S obzirom na to da se funkcija poziva točno jednom sa svakim argumentom `m`, primjenom memoizacije algoritam se ne bi nimalo ubrzao — štoviše, iako bi njegova asimptotska složenost ostala $\Theta(2^n)$, konstante u vremenu izvođenja bi se nepotrebno povećala i algoritam bi bio sporiji. Dodatno, memoizirano rješenje imalo bi prostornu složenost $\Theta(nCV_u)$, gdje je V_u ukupna vrijednost svih predmeta.

Kriterij primjenjivosti dinamičkog programiranja je *preklapanje potproblema* (engl. *overlapping subproblems*). Drugim riječima, ako se potproblemi ne preklapaju, dinamičko programiranje neće nimalo ubrzati algoritam. Preklapanje potproblema podrazumijeva da se pri rješavanju jednog potproblema rješavaju instance čija se rješenja koriste i za rješavanje drugih potproblema koji se s prvim preklapaju. Svi algoritmi do sada prikazani u radu osim algoritma iz ispisa 2.1 zadovoljavaju ovaj kriterij.

Kroz raspravu o kriterijima točnosti i primjenjivosti dinamičkog programiranja, očito je da je ključni problem kod dinamičkog programiranja osmisliti prikladno rekurzivno rješenje. Transformacija rekurzivnog rješenja u memoizirano rješenje ili rješenje dinamičkim programiranje od dna prema vrhu je u najvećoj mjeri mehanička. Pri osmišljavanju rekurzivnog rješenja, osnovni cilj je ispravan odabir prostora stanja i transformacija među stanjima.

2.3 Memoizacija

Memoizacija je jednostavniji oblik dinamičkog programiranja i u pravilu se može provesti nad bilo kojom funkcijom bez popratnih efekata (engl. *side effects*).

```
1 Memoiziraj(F):
2     memo je asocijativno polje
3     F'(argumenti):
4         ako argumenti nisu u memo:
5             memo[argumenti] = F(argumenti)
6         vrati memo[argumenti]
7     vrati F'
```

Ispis 2.2: Memoizacija proizvoljne funkcije F.

Funkcija `Memoiziraj` u ispisu 2.2 za danu funkciju `F` vraća memoiziranu verziju iste funkcije. U retku 2 priprema se asocijativno polje `memo` koje može biti, na primjer, tablica raspršenog adresiranja ili samobalansirajuće stablo pretraživanja. U retcima 3–6 definira se nova funkcija `F'` koja vraća vrijednost iz strukture `memo` ako je ona već izračunata ili poziva originalnu funkciju `F` i rezultat pohranjuje u strukturu. Primjenjivost ovakvog generičkog rješenja na rekurzivne funkcije uglavnom je ograničena na interpretirane programske jezike[1].

Osnovna prednost memoizacije nad dinamičkim programiranjem od dna prema vrhu je u rješavanju problema gdje je teško unaprijed odrediti ispravan redoslijed rješavanja potproblema. Rješenje memoizacijom može biti značajno brže od rješenja dinamičkim programiranjem od dna prema vrhu

[1]Na primjer Python, Ruby, JavaScript i mnoge druge. Naime, ako se neka rekurzivna funkcija `f` memoizira naredbom `f = Memoiziraj(f)`, nova funkcija tijekom svog izvođenja poziva staru funkciju `f`. Ako je funkcija `f` prevedena u strojni oblik, rekurzivni pozivi i dalje će se odnositi na nememoiziranu inačicu funkcije i neće se postići nikakvo ubrzanje.

ako se prilikom izračunavanja funkcije za zadane argumente izračunaju vrijednosti za samo dio prostora stanja. S druge strane, ako se i u rekurzivnom rješenju obradi veći dio prostora stanja, memoizirano rješenje će načelno biti sporije zbog lošijeg korištenja priručnog spremnika i velikog broja funkcijskih poziva. Dodatno, kao i svako rekurzivno rješenje, memoizacijsko rješenje može biti neprimjenjivo ako je maksimalna veličina stoga prevelika. Naime, uobičajeno je da je ograničenje na veličinu stoga znatno manje od ograničenja na veličinu gomile.

Memoizacija nije primjenjiva na neke probleme kod kojih u grafu ovisnosti prostora stanja postoje ciklusi, kao što je, na primjer, problem najkraćeg puta u grafu. Ciklusi se mogu razriješiti u dinamičkom programiranju od dna prema vrhu, a točan način ovisi o semantici problema. Na primjer, za problem najkraćeg puta u grafu se redoslijed računanja potproblema određuje pomoću prioritetnog reda.

Određivanje vremenske i prostorne složenosti

Određivanje vremenske složenosti memoizirane rekurzije se na prvi pogled čini složeno. Vrijednost neke ćelije u memoizacijskoj strukturi podataka računa se samo pri prvom pozivu funkcije s određenim argumentima. Svaki sljedeći poziv funkcije pohranjenu vrijednost dohvaća iz strukture. Zbog toga je očito da vremenska složenost ovisi o izboru memoizacijske strukture. U općem slučaju, vremenska složenost samo prvih poziva funkcije iznosi $O(N \times P \times F)$, gdje je N kardinalitet prostora stanja, P vremenska složenost pretraživanja memoizacijske strukture i F vremenska složenost razlaganja problema na potprobleme i povezivanja rješenja potproblema u rješenje početne instance.

Svi preostali pozivi funkcije imaju vremensku složenost $O(P)$, pa preostaje samo izbrojiti koliko takvih poziva ima. Ako se uzme u obzir da svaki poziv funkcije dolazi iz nekog prvog poziva funkcije s određenim argumentima, onda je očito da je ukupni broj poziva funkcije ograničen s $O(N \times F)$ pa je ukupna vremenska složenost memoiziranog algoritma $O(N \times P \times F)$.

Na primjer, za rješenje 0-1 problema ruksaka iz odjeljka 1.2 vrijedi $N = \Theta(nC)$ jer su dimenzije memoizacijske matrice $n \times C$. Nadalje, kako je memoizacijska struktura matrica s konstantnim vremenom pristupa vrijedi $P = \Theta(1)$. Konačno, vrijeme razlaganja problema na potprobleme i povezivanje rješenja potproblema iznosi $F = \Theta(1)$ pa je vremenska složenost algoritma $O(N \times P \times F) = O(nC)$. U odjeljku 1.2 je pokazano i da je

vremensla složenost $\Theta(nC)$ u najgorem slučaju.

Prostorna složenost je jednostavno zbroj maksimalne veličine memoizacijske strukture i maksimalne veličine stoga. Tipično veličina memoizacijske strukture dominira, ali je često upravo maksimalna veličina stoga ograničavajući faktor primjene memoizacije.

Na primjer, za isto rješenja 0-1 problema ruksaka, veličina memoizacijske strukture je $\Theta(nC)$, a maksimalna veličina stoga je $\Theta(n)$. Prostorna složenost algoritma je stoga $\Theta(nC) + \Theta(n) = \Theta(nC)$.

2.4 Dinamičko programiranje od dna prema vrhu

Dinamičko programiranje od dna prema vrhu nudi neke značajne prednosti u odnosu na memoizaciju. Za mnoge algoritme moguće je značajno smanjiti prostornu složenost, a ponekad čak i vremensku složenost za polinomni faktor. Dodatno, sustavni pristup matrici redak po redak u pravilu bolje iskorištava priručni spremnik od mnogo nepravilnijeg pristupa u memoiziranoj rekurziji. Upravo zbog ovog razloga, važno je dimenzije matrice koja se koristi za dinamičko programiranje poredati onim redom kojim se vrijednosti računaju.

S druge strane, dinamičkim programiranjem se izračunavaju rješenja za sve manje instance problema, čak i za one koje nije nužno izračunati za rješavanje zadane početne instance. Ako se dimenzije prostora stanja ne mogu prikazati cijelim brojevima ili je pretvorba u cijele brojeve složena, memoizacija uz korištenje asocijativnog polja je jednostavnije rješenje od dinamičkog programiranja od dna prema vrhu.

Određivanje vremenske i prostorne složenosti

Vremenska složenost rješenja dinamičkim programiranjem od dna prema vrhu je $\Theta(N \times F)$, gdje je N veličina prostora stanja, a F vrijeme potrebno da se izračuna jedna ćelija u matrici. Ako se za memoizaciju isto koristi matrica, vremenska složenost obje tehnike je jednaka.

Prostorna složenost jednaka je veličini prostora stanja, tj. $\Theta(N)$. S obzirom na to da je maksimalna veličina stoga tipično dominirana veličinom prostora stanja, prostorne složenosti memoiziranog algoritma i algoritma od

dna prema vrhu su asimptotski jednake.

Smanjenje prostorne složenosti algoritma

Ako nije potrebno rekonstruirati rješenje, često je moguće prvu dimenziju matrice smanjiti na dva i time ostvariti značajno smanjenje potrebnog spremničkog prostora. Dodatno, takvo smanjenje prostorne složenosti može uzrokovati i ubrzanje algoritma zbog boljeg korištenja priručnog spremnika. Preduvjet za ovakvo smanjenje prostorne složenosti je da svi potproblemi na koje se razlaže neka instanca problema imaju jednaku vrijednost nekog parametra.

Na primjer, za rekurzivnu definiciju rješenja 0-1 problema ruksaka

$$R(i,m) = \begin{cases} 0, & \text{ako je } i = 0 \text{ i } m < M[0] \\ V[0], & \text{ako je } i = 0 \text{ i } m \geq M[0] \\ R(i-1,m), & \text{ako je } i > 0 \text{ i } m < M[i] \\ max\{R(i-1,m), & \\ \quad R(i-1,m-M[i]) + V[i]\}, & \text{ako je } i > 0 \text{ i } m \geq M[i], \end{cases}$$

svi potproblemi čija rješenja se koriste nalaze se u prethodnom retku. Zbog toga je dovoljno umjesto n redaka koristiti dva retka matrice.

```
1 R je matrica 2 x (C+1) cijelih brojeva
2 inicijaliziraj R[0][m] := 0 za sve m < M[0]
3 inicijaliziraj R[0][m] := V[0] za M[0] <= m <= C
4 za i od 1 do n-1:
5     za m od 0 do min{M[i]-1, C}:
6         R[i&1][m] := R[(i-1)&1][m]
7     za m od M[i] do C:
8         R[i&1][m] := max{R[(i-1)&1][m],
9                          R[(i-1)&1][m-M[i]] + V[i]}
```

Ispis 2.3: Rješenje 0-1 problema ruksaka dinamičkim programiranjem od dna prema vrhu uz smanjenje prostorne složenosti za faktor n.

Rješenje u ispisu 2.3 razlikuje se od početnog rješenja dinamičkim programiranjem od dna prema vrhu iz ispisa 1.6 u dva detalja. Prvo, u retku 1 je matrica `R` definirana kao matrica od dva retka. Drugo, u svakom pristupu matrici u retcima 6–9 umjesto cijelog indeksa retka, koristi se samo njegov posljednji bit. Do posljednjeg bita dolazi se bitovnom-i operacijom sa brojem jedan. Nakon izvođenja algoritma, konačno rješenje nalazi se u ćeliji `R[(n-1)&1]`.

Korištenjem ove tehnike, prostorna složenost algoritma smanjena je sa $\Theta(nC)$ na $\Theta(C)$, a vremenska složenost ostala je ista. Iako se asimptotski prostorna složenost ne može dalje smanjiti, umjesto dva retka za 0-1 problem ruksaka je dovoljno koristiti jedan redak.

```
1 R je niz C+1 cijelih brojeva
2 inicijaliziraj R[m] := 0 za sve m < M[0]
3 inicijaliziraj R[m] := V[0] za M[0] <= m <= C
4 za i od 1 do n-1:
5     za m od C silazno do M[i]:
6         R[m] := max{R[m], R[m-M[i]] + V[i]}
```

Ispis 2.4: Rješenje 0-1 problema ruksaka dinamičkim programiranjem od dna prema vrhu koristeći niz $C + 1$ brojeva.

Rješenje u ispisu 2.4 koristi *niz* `R` za potrebe dinamičkog programiranja. Inicijalizacija niza u retku 2 jednaka je inicijalizaciji prvog retka matrice `R` u prije prikazanim rješenjima. Ključna razlika u odnosu na prethodna rješenja je silazna petlja u retku 5. U ovom rješenju očuvana je sljedeća invarijanta koja ujedno dokazuje točnost rješenja: nakon izvođenja pridruživanja u retku 6, `R[m]`, `R[m+1]`, ..., `R[C]` sadrže optimalnu vrijednost ruksaka koristeći prvih `i` predmeta uz masu ne veću od indeksa niza, a `R[0]`, `R[1]`, ..., `R[m-1]` sadrže optimalnu vrijednost ruksaka koristeći prvih `i-1` predmeta uz masu ne veću od indeksa niza. Drugim riječima, niz `R` istovremeno sadrži sve bitne vrijednosti iz dva retka matrice `R` iz ispisa 2.3.

Smanjenje vremenske složenosti algoritma

Ponekad je moguće i vremensku složenost algoritma smanjiti za polinomni faktor. Ova tehnika može se primijeniti na višedimenzionalne prostore stanja gdje odluka u nekom stanju vodi do skupa potproblema koji su u nekom smislu ravnopravni i do optimalnog rješenja instance dolazi se ili izborom najboljeg ili zbrajanjem rješenja tih potproblema.

Tehnika smanjenja vremenske složenosti ilustrirana je na sljedećem problemu. Neka je zadana matrica A cijelih brojeva dimenzija $m \times n$. Potrebno je odrediti maksimalan zbroj koji je moguće postići krećući se od prvog do zadnjeg retka, pri čemu se u svakom retku mora odabrati točno jedan broj, a pri prelaženju iz retka u redak indeks stupca se može promijeniti najviše za k. Rješenje problema za danu matricu ovisit će o parametru k. Na primjer,

za $k = 0$ problem se svodi na traženje stupca s najvećim zbrojem. Za $k = n$, u svakom retku se odabire najveći broj.

	0	1	2	3	4
0	1	3	5	1	1
1	-3	5	4	8	1
2	1	-1	1	1	1

Slika 2.2: Primjer matrice za $m = 3$ i $n = 5$.

Na slici 2.2 prikazan je primjer matrice za $m = 3$ i $n = 5$. Za $k = 0$, rješenje je zbroj elemenata drugog ili trećeg stupca $5+4+1 = 1+8+1 = 10$. Za $k = 5$, rješenje je $5 + 8 + 1 = 14$.

Općenito, neka je $S(i, j)$ najbolja vrijednost koju je moguće postići u prvih i redaka, pri čemu je u i-tom retku uzet broj iz j-tog stupca. Preostaje definirati početne uvjete i općenitu rekurzivnu definiciju funkcije. Iz definicije problema i funkcije S, očito je $S(0, j) = A[0][j]$. Prilikom prelaska u novi redak, dozvoljeno je promijeniti stupac za maksimalno k. Zbog toga za $i > 0$ vrijedi $S(i, j) = A[i][j] + max\{S(i-1, s_0), \ldots, S(i-1, s_p)\}$, pri čemu je $s_0 = max\{0, j-k\}$ i $s_p = min\{n-1, j+k\}$. Potpuna definicija funkcije dakle glasi

$$S(i,j) = \begin{cases} A[0][j], & \text{ako je } i = 0 \\ A[i][j] + max\{S(i-1, s_0), \ldots, S(i-1, s_p)\}, & \text{ako je } i > 0. \end{cases}$$

```
1  S je mxn matrica cijelih brojeva
2  inicijaliziraj S[0][j] := A[0][j] za sve j < n
3  za i od 1 do m-1:
4      za j od 0 do n-1:
5          najbolji := S[i-1][j]
6          lijevo := max{0, j-k}
7          desno := min{n-1, j+k}
8          za s od lijevo do desno:
9              najbolji := max{najbolji, S[i-1][s]}
10         S[i][j] := A[i][j] + najbolji
```

Ispis 2.5: Računanje funkcije S dinamičkim programiranjem.

Algoritam u ispisu 2.5 izravna je implementacija računanja funkcije S dinamičkim programiranjem. Petlje u retcima 2 i 3 redom prolaze kroz prostor stanja, redak po redak, slijeva nadesno. Varijabla `najbolji` služi za odabir najboljeg dohvatljivog rješenja iz prethodnog retka matrice i u retku 5 se inicijalizira na vrijednost koja je sigurno dohvatljiva, bez obzira na vrijednost parametra `k`. Varijable `lijevo` i `desno` sadrže vrijednosti s_0 i s_p iz definicije funkcije. Konačno, petlja u retcima 8 i 9 pronalazi najbolje rješenje iz prethodnog retka i to rješenje se u retku 10 uvećava za vrijednost trenutne ćelije u matrici.

Prostor stanja danog algoritma je dimenzija $\Theta(mn)$. Pri računanju rješenja svake instance, obavlja se $O(k)$ operacija, pa je ukupna vremenska složenost $O(mnk)$. S obzirom na to da je $k = O(n)$, vremenska složenost je u najgorem slučaju $\Theta(mn^2)$.

```
1  M je niz 2 max-intervalna stabla sa n listova
2  za j od 0 do n-1:
3      M[0].dodaj(j, A[0][j])
4  za i od 1 do m-1:
5      M[i&1].isprazni()
6      za j od 0 do n-1:
7          lijevo := max{0, j-k}
8          desno := min{n-1, j+k}
9          zbroj := A[i][j]
10             + M[(i-1)&1].dohvatiMaksimum(lijevo, desno)
11         M[i&1].dodaj(j, zbroj)
```

Ispis 2.6: Računanje funkcije S dinamičkim programiranjem uz smanjenje vremenske složenosti.

Algoritam prikazan u ispisu 2.6 umjesto matrice S koristi dva intervalna stabla `M[0]` i `M[1]`. Intervalno stablo je jednostavna struktura podataka koja omogućuje logaritamsko dodavanje vrijednosti i dohvaćanje neke informacije o bilo kojem intervalu. U prikazanom algoritmu koristi se intervalno stablo koje kroz metodu `dohvatiMaksimum(lijevo, desno)` vraća najveću vrijednost dodanu u stablo između indeksa stupaca `lijevo` i `desno`. Dva intervalna stabla naizmjence se koriste kroz indeksiranje posljednjim bitom indeksa retka. U svakoj iteraciji petlje u retku 6 vrijedi da intervalno stablo `M[(i-1)&1]` sadrži optimalne vrijednosti za redak $i - 1$, dok se vrijednosti izračunate za i-ti redak pohranjuju u stablo `M[i&1]`. Konačno rješenje dobiva se pozivom `M[(m-1)&1].dohvatiMaksimum(0, n-1)`.

S obzirom da je vrijeme dodavanja vrijednosti u intervalno stablo i dohvaćanja vrijednosti iz intervalnog stabla $\Theta(\log n)$, ukupna vremenska složenost algoritma je $\Theta(mn \log n)$. Dok je jednostavni algoritam iz ispisa 2.5 primjenjiv za vrijednosti m i n do nekoliko stotina, ovaj ubrzani algoritam primjenjiv je za m i n do nekoliko tisuća[2].

Umjesto maksimuma rješenja nekih potproblema, često je u rješavanju instance problema potrebno izračunati zbroj rješenja nekih potproblema. U takvom slučaju je ponekad moguće zbrajanje u petlji zamijeniti pristupom nizu djelomičnih zbrojeva (engl. *partial sums*) iz kojeg se u konstantnom vremenu može izračunati traženi zbroj.

[2]Prostorna složenost je također smanjena sa $\Theta(mn)$ na $\Theta(n)$, ali se ista složenost može postići i u jednostavnom algoritmu kao što je prikazano u prošlom poglavlju.

Poglavlje 3

Problemi raspoređivanja

Među problemima raspoređivanja postoje značajne razlike koje određuju u kojoj mjeri je problem rješiv. Za neke inačice poznati su polinomni algoritmi, dok su neke inačice NP-potpune ili NP-teške i rješavaju se heurističkim algoritmima[1]. U nastavku poglavlja ukratko je prikazano nazivlje koje se koristi u radu i prikazana su rješenja dinamičkim programiranjem za neke inačice problema raspoređivanja.

3.1 Nazivlje

U svim inačicama problema raspoređivanja, određeni broj *zadataka* treba rasporediti u vremenu na određenu konfiguraciju *strojeva*. Broj dostupnih strojeva za neku inačicu problema može biti konstantan ili može biti parametar instance problema. Vrijeme izvođenja i-tog zadatka označava se s p_i (engl. *processing time*) i može se razlikovati među zadacima pa čak i ovisiti o stroju na kojem se zadatak izvodi.

Rasporedi zadataka mogu biti ograničeni raznim uvjetima nad zadacima. Na primjer, zadaci mogu biti *djeljivi* ili *nedjeljivi* što znači da se jednom započeti zadatak mora u cijelosti i završiti odnosno da se zadatak može podijeliti na manje dijelove koji se mogu slijedno izvoditi i na različitim strojevima i u različitim vremenskim intervalima. Nadalje, među zadacima može biti zadana *relacija prednosti* P koja određuje da se određeni zadaci moraju završiti prije nego može početi izvođenje nekih drugih zadataka. Za zadatke mogu

[1]Opsežan pregled poznatih rezultata dostupan je na adresi `http://www.informatik.uni-osnabrueck.de/knust/class/`.

biti zadani jednostrano ili dvostrano omeđeni intervali vremena u kojima se zadaci mogu izvršavati. Trenutak nakon kojeg može započeti izvođenje i-tog zadatka označava se sa r_i (engl. *ready time*, *release time*), a trenutak do kojeg se zadatak treba izvršiti sa d_i (engl. *due time*).

Konačno, inačice se bitno razlikuju u funkciji cilja po kojoj se raspored zadataka optimira. Kvaliteta rasporeda tako se može ocijeniti maksimalnim vremenom završetka zadatka $max\{C_i\}$ (engl. *completion time*), maksimalnim kašnjenjem nekog zadatka $max\{L_i\}$ (engl. *lateness*), gdje je kašnjenje definirano kao $L_i = C_i - d_i$, minimalnim ukupnim zaostajanjem zadataka $\sum T_i$ (engl. *tardiness*), gdje je zaostajanje zadatka definirano kao $T_i = max\{0, L_i\}$ ili ukupnim brojem zakašnjelih zadataka $\sum U_i$, gdje U_i poprima vrijednost nula ili jedan ovisno o tome je li zadatak završio na vrijeme ili ne.

Inačica problema raspoređivanja opisuje se poredanom trojkom; prvi element opisuje konfiguraciju strojeva, drugi element ograničenja rasporeda i treći element funkciju cilja. Na primjer, problem $1|r_i; p_i = p|C_{max}$ je problem raspoređivanja zadataka na jedan stroj pri čemu zadaci imaju zadano najranije vrijeme početka izvođenja r_i. Trajanje izvođenja svih zadataka jednako je i iznosi p, a optimira se ukupno vrijeme izvođenja svih zadataka.

3.2 Problem $1||\sum w_iU_i$

Problem $1||\sum w_iU_i$ je problem raspoređivanja n zadataka na jedan stroj. Svakom zadatku pridruženo je vrijeme izvođenja p_i, težina w_i i trenutak do kojeg bi zadatak trebao biti izvrđen d_i. Cilj problema je pronaći raspored zadataka koji će minimizirati zbroj težina zakašnjelih zadataka.

Problem $1||\sum w_iU_i$ je NP-težak, što je dokazano u klasičnom Karpovom radu [4]. Ipak, problem se može riješiti jednostavnim algoritmom zasnovanom na dinamičkom programiranju u pseudopolinomnom vremenu.

Kako za zadatke nisu zadana ograničenja na početak obrade, očito je da će u optimalnom rasporedu stroj obrađivati zadatke u svim trenucima od trenutka nula do završetka izvođenja posljednjeg zadatka. U suprotnom se izvođenje svih zadataka koji se izvode nakon nekog trenutka neiskorištenosti stroja može pomaknuti prema ranijim vremenskim trenucima, pri čemu vrijednost funkcije cilja ostaje jednaka ili se smanjuje. Drugim riječima, problem je ekvivalentan pronalaženju optimalne permutacije zadataka π.

U nastavku razmatranja pretpostavlja se da su zadaci poredani po nerastućem d_i, tj. vrijedi $d_1 \leq d_2 \leq \ldots \leq d_n$. U osmišljavanju algoritma za

rješavanje ovog problema, važno je uočiti da vremena završetka izvođenja zadataka nisu važna, osim u cilju određivanja je li izvođenje zadatka zakašnjelo ili nije. Koristeći ovu činjenicu, lako je dokazati da postoji optimalan poredak π oblika $i_1, i_2, \ldots, i_s, i_{s+1}, \ldots, i_n$ pri čemu su zadaci $i_1 < i_2 < \ldots < i_s$ završeni na vrijeme, a zadaci $i_{s+1} < i_{s+2} < \ldots < i_n$ kasne. Naime, jednom kada neki zadatak kasni, nije važno koliko on kasni pa ga je svakako moguće preseliti u drugi dio poretka, a i zadatke koji kasne moguće je proizvoljno poredati, pa tako i po rastućim indeksima. Nadalje, pretpostavimo da se u optimalnom poretku j-ti zadatak izvodi prije i-tog zadatka i oba zadatka se izvode na vrijeme, a pritom vrijedi $i < j$ tj. $d_i \leq d_j$. Zamjenom mjesta i-tog i j-tog zadatka uz prikladno pomicanje svih zadataka koji su u poretku između j-tog i i-tog, opet se dobiva poredak u kojemu se oba zadatka izvode na vrijeme i koji je optimalan.

Rješenje dinamičkim programiranjem prikazano je u ispisu 3.1. U ćeliju matrice `opt[i][t]` pohranjuje se vrijednost funkcije cilja za optimalni poredak prvih $i + 1$ zadataka u kojem zadnji zadatak koji završava izvođenje na vrijeme završava izvođenje najkasnije u trenutku `t`. U pripadnoj ćeliji matrice `naVrijeme` pohranjuje se hoće li se u tom poretku i-ti zadatak izvršiti na vrijeme, za potrebe rekonstrukcije poretka.

```
1  p je niz n cijelih brojeva // trajanje zadataka
2  d je niz n cijelih brojeva // rokovi zavrsetka zadataka
3  w je niz n cijelih brojeva // tezine kasnjenja zadataka
4  opt je n x (d[n-1]+1) matrica cijelih brojeva
5  naVrijeme je n x (d[n-1]+1) matrica logickih vrijednosti
6
7  za i od 0 do p[0]-1:
8      opt[0][i] := w[0]
9      naVrijeme[0][i] := laz
10 za i od p[0] do d[n-1]:
11     opt[0][i] := (p[0] > d[0]) * w[i]
12     naVrijeme[0][i] := (p[0] <= d[0])
13 za i od 1 do n-1:
14     za t od 0 do p[i]-1:
15         opt[i][t] := opt[i-1][t]
16         naVrijeme[i][t] := laz
17     za t od p[i] do d[i]:
18         ako je opt[i-1][t-p[i]] < opt[i-1][t] + w[i]:
19             opt[i][t] := opt[i-1][t-p[i]]
20             naVrijeme[i][t] := istina
```

```
21          inace:
22              opt[i][t] := opt[i-1][t] + w[i]
23              naVrijeme[i][t] := laz
24      za to od d[i]+1 do d[n-1]:
25          opt[i][t] := opt[i][d[i]]
26          naVrijeme[i][t] := naVrijeme[i][d[i]]
```

Ispis 3.1: Rješenje problema $1||\sum w_i U_i$.

U retcima 7–12 inicijaliziraju se vrijednosti za prvi zadatak. Zadatak se nikako ne može izvesti u vremenu manjem od `p[0]`, pa se u retcima 8 i 9 označava da zadatak u tim potproblemima kasni. Za sve preostale vremenske trenutke, zadatak se može izvesti na vrijeme ako mu je trajanje izvođenja manje ili jednako roku završetka izvođenja.

Pri dodavanju novog zadatka u raspored u petlji u retcima 13–26, treba donijeti odluku hoće li novi zadatak kasniti ili će se izvršiti na vrijeme. Slično kao u inicijalizacijskom koraku, zadatak se ne može izvršiti na vrijeme ako izvršavanje svih zadataka koji ne kasne traje manje od `p[i]`. Za sve trenutke od `p[i]` do `d[i]` u petlji u retcima 17–23 se provjeravaju dvije opcije: i-ti zadatak se može izvršiti na vrijeme tako da se od prethodnih zadataka svi zadaci koji završavaju na vrijeme izvrše do najkasnije `t-p[i]` ili može kasniti pri čemu se najboljem rješenju za prethodne zadatke dodaje `w[i]`. Konačno, za sve trenutke nakon `d[i]` odabire se ista opcija kao za trenutak `d[i]`. Točnost ovog postupka posljedica je dvije činjenice. Prvo, nakon trenutka `d[i]` i-ti zadatak se ne može izvršiti na vrijeme. Drugo, s obzirom na pretpostavku da su zadaci poredani po nepadajućim vrijednostima roka izvršavanja, vrijednosti `opt[i-1][t]` za `t > d[i]` su konstantne pa tako niti `opt[i][t]` ne može biti manja vrijednost od `opt[i][d[i]]`.

Prostorna složenost rješenja je $\Theta(nd_n)$. Kako se rješenje za svaki potproblem računa u konstantnom vremenu, vremenska složenost je $\Theta(nd_n)$.

```
1 Rekonstruiraj(i, t, jeNaVrijeme):
2     jeNaVrijeme[i] := naVrijeme[i][t]
3     ako je i > 0:
4         ako je naVrijeme[i][t]:
5             Rekonstruiraj(i-1, t-p[i], jeNaVrijeme)
6         inace:
7             Rekonstruiraj(i-1, t, jeNaVrijeme)
8
9 jeNaVrijeme je niz n logickih vrijednosti
```

```
10 IspisiOptimalniPoredak():
11     Rekonstruiraj(n-1, d[n-1], jeNaVrijeme)
12     a = []
13     b = []
14     za i od 0 do n-1:
15         ako je jeNaVrijeme[i]:
16             a.dodaj(i)
17         inace:
18             b.dodaj(i)
19     t = 0
20     za i od 0 do a.duljina():
21         ispisi("zapocni zadatak " + a[i] + " u trenutku " + t)
22         t += p[a[i]]
23     za i od 0 do b.duljina():
24         ispisi("zapocni zadatak " + b[i] + " u trenutku " + t)
25         t += p[b[i]]
```

Ispis 3.2: Rekonstrukcija optimalnog poretka za problem $1||\sum w_i U_i$.

Postupak rekonstrukcije optimalnog poretka na osnovi matrice `naVrijeme` prikazan je u ispisu 3.2. Koristi se pomoćna procedura `Rekonstruiraj` koja u nizu `jeNaVrijeme` za svaki zadatak označava je li taj zadatak u optimalnom poretku izvršen na vrijeme ili ne. Koristeći vrijednosti u nizu `jeNaVrijeme`, u retcima 12–18 se stvaraju nizovi zadataka koji se izvode na vrijeme i zadataka koji kasne. Konačno, u recima 19–25 se ispisuje optimalan poredak.

Vremenska i prostorna složenost rekonstrukcije rješenja su linearne s n tako da ukupna vremenska i prostorna složenost pronalaženja optimalnog poretka za $1||\sum w_i U_i$ problem iznose $\Theta(nd_n)$.

3.3 Problem $1||\sum T_i$

Problem $1||\sum T_i$ je problem raspoređivanja n zadataka na jedan stroj s ciljem minimiziranja ukupnog zaostajanja zadataka. Trajanje izvođenja zadataka je proizvoljno i za i-ti zadatak iznosi p_i. Kako bi zaostajanje zadatka bilo definirano, za svaki zadatak definirano je i vrijeme do kad bi zadatak trebao biti izvršen d_i.

Definirani problem je NP-težak [5]. U nastavku odjeljka je pokazano da za zadanu inačicu problema raspoređivanja postoji pseudopolinomno rješenje

dinamičkim programiranjem i da problem *nije* NP-težak u punom smislu. Ova činjenica davno je dokazana u [6].

Isto kao kod problema $1||\sum w_i U_i$, lako je vidjeti da se zapravo traži permutacija zadataka jer uvijek postoji optimalan raspored u kojem stroj u svakom trenutku izvršava neki zadatak.

Tipično se dinamičkim programiranjem optimalna permutacija može pronaći eksponencijalnom vremenskom i prostornom složenošću, ali pri rješavanju ovog problema se značajan dio permutacija može unaprijed eliminirati koristeći strukturne teoreme prikazane u nastavku.

Lema 3.3.1. *Ako vrijedi $d_i \leq d_j$ i $p_i \leq p_j$ onda postoji optimalan poredak π' u kojemu se i-ti zadatak izvodi prije j-tog zadatka.*

Dokaz. Neka je π neki optimalan poredak zadataka u kojemu vrijedi suprotno, tj. j-ti zadatak se izvodi prije i-tog zadatka i vrijedi $C_j < C_i$. Doprinos ovih dvaju zadataka ukupnoj zaostalosti zadataka iznosi

$$T = T_i + T_j = max\{0, C_i - d_i\} + max\{0, C_j - d_j\}.$$

Konstruirajmo iz poretka π poredak π' tako da zamijenimo mjesta i-tom i j-tom zadatku. Pritom, ako je $p_i < p_j$, zadatke koji u π slijede j-ti zadatak a prethode i-tom zadatku pomičemo prema ranijim vremenskim trenutcima tako da stroj i u π' uvijek obrađuje zadatak. Cilj je dokazati da je ukupna zaostalost u poretku π' manja ili jednaka zaostalosti u poretku π.

Kako će i-ti zadatak u poretku π' započeti izvođenje u trenutku u kojem je u poretku π izvođenje započeo j-ti zadatak, vrijedi

$$C'_i = C_j - (p_j - p_i) \leq C_j.$$

Dodatno, kako je skup svih zadataka prije i uključujući i-ti zadatak u π jednak skupu svih zadataka prije i uključujući j-ti zadatak u π', vrijedi

$$C'_j = C_i.$$

Doprinos zaostalosti i-tog i j-tog zadatka ukupnoj zaostalosti u π' iznosi

$$\begin{aligned} T' &= T'_i + T'_j \\ &= max\{0, C'_i - d_i\} + max\{0, C'_j - d_j\} \\ &= max\{0, C_j - (p_j - p_i) - d_i\} + max\{0, C_i - d_j\}. \end{aligned}$$

Doprinos ukupnoj zaostalosti poretka π' zadataka koji se izvode između i-tog i j-tog zadatka u odnosu na doprinos u π nije se mogao povećati jer se

vremena izvođenja tih zadataka pomiču u ranije trenutke ili ostaju ista ako je $p_i = p_j$ pa se taj doprinos u nastavku ne razmatra.

Za dokaz tvrdnje pokazuje se da vrijedi $T' \leq T$ u svih šest mogućih relativnih odnosa vremena C_i, C_j, d_i i d_j pri čemu po uvjetima teorema i odabiru poretka π uvijek vrijedi $C_j < C_i$ i $d_i \leq d_j$.

Slučaj $C_j < d_i$, $C_i \leq d_i$, $T = 0$

$$T' = max\{0, C_j - (p_j - p_i) - d_i\} + max\{0, C_i - d_j\} = 0 + 0 \leq T$$

Slučaj $C_j \leq d_i$, $d_i \leq C_i \leq d_j$, $T = C_i - d_i \geq 0$

$$T' = max\{0, C_j - (p_j - p_i) - d_i\} + max\{0, C_i - d_j\} = 0 + 0 \leq T$$

Slučaj $C_j \leq d_i$, $d_j \leq C_i$, $T = C_i - d_i \geq 0$

$$\begin{aligned} T' &= max\{0, C_j - (p_j - p_i) - d_i\} + max\{0, C_i - d_j\} \\ &= 0 + (C_i - d_j) \\ &\leq C_i - d_i \\ &= T \end{aligned}$$

Slučaj $d_i < C_j < d_j$, $C_i \leq d_j$, $T = C_i - d_i \geq 0$

$$\begin{aligned} T' &= max\{0, C_j - (p_j - p_i) - d_i\} + max\{0, C_i - d_j\} \\ &\leq (C_j - d_i) + 0 \\ &< C_i - d_i \\ &= T \end{aligned}$$

Slučaj $d_i \leq C_j \leq d_j$, $d_j \leq C_i$, $T = C_i - d_i \geq 0$

$$\begin{aligned} T' &= max\{0, C_j - (p_j - p_i) - d_i\} + max\{0, C_i - d_j\} \\ &\leq (C_j - d_i) + (C_i - d_j) \\ &= (C_i - d_i) + (C_j - d_j) \\ &= T + (C_j - d_j) \\ &\leq T \end{aligned}$$

Slučaj $d_j \leq C_j$, $d_j < C_i$, $T = (C_i - d_i) + (C_j - d_j) > 0$

$$\begin{aligned} T' &= max\{0, C_j - (p_j - p_i) - d_i\} + max\{0, C_i - d_j\} \\ &\leq (C_j - d_i) + (C_i - d_j) \\ &= (C_i - d_i) + (C_j - d_j) \\ &= T \end{aligned}$$

□

Korolar 3.3.2. *Neka su zadaci nesilazno poredani po vremenu d_i i neka je $p_k = max\{p_i\}$ najdulje vrijeme izvođenja nekog zadatka. Tada postoji optimalan poredak π u kojemu vrijedi $\{1, 2, ..., k-1\} \to k$ što označava da su zadaci iz skupa $\{1, 2, ..., k-1\}$ poredani prije zadatka k.*

Lema 3.3.3. *Neka je π neki optimalan poredak uz rokove završetka zadataka $d_1, d_2, \ldots, d_n$ i neka je C_i vrijeme završetka i-tog zadatka u π. Tada je za bilo koji d'_i takav da je*

$$min\{d_i, C_i\} \leq d'_i \leq max\{d_i, C_i\}$$

bilo koji optimalan poredak π' uz rokove završetka $d'_1, d'_2, \ldots, d'_n$ optimalan i uz originalne rokove završetka $d_1, d_2, \ldots, d_n$.

Dokaz leme 3.3.3 zasniva se na sljedećim izrazima. Neka je T ukupna zaostalost zadataka uz rokove završetka $d_1, d_2, \ldots, d_n$, a T' ukupna zaostalost zadataka uz rokove završetka $d'_1, d'_2, \ldots, d'_n$. Vrijedi

$$T(\pi) = T'(\pi) + \sum_{i=1}^{n} A_i$$

$$T(\pi') = T'(\pi') + \sum_{i=1}^{n} B_i$$

$$A_i = \begin{cases} 0, & \text{ako je } C_i \leq d_i \\ d'_i - d_i, & \text{ako je } C_i > d_i \end{cases}$$

$$B_i = \begin{cases} -max\{0, min\{C'_i, d_i\} - d'_i\}, & \text{ako je } C_i \leq d_i \\ max\{0, min\{C'_i, d'_i\} - d_i\}, & \text{ako je } C_i > d_i \end{cases}$$

što se može dokazati analizom slučajeva kako bi se uklonili maksimumi i minimumi iz definicije B_i. Očito za svaki i vrijedi $A_i \geq B_i$ pa vrijedi i $\sum_{i=1}^{n} A_i = \sum_{i=1}^{n} B_i$. Nadalje, zbog optimalnosti π' vrijedi $T'(\pi) \geq T'(\pi')$ i konačno $T(\pi) \geq T(\pi')$. Kako smo pretpostavili da je π optimalan poredak uz originalne rokove završetka $d_1, d_2, \ldots, d_n$, onda je i π' također optimalan, kao što lema 3.3.3 tvrdi.

Teorem 3.3.4. *Neka su zadaci nesilazno poredani tako da je $d_1 \leq d_2 \leq \ldots \leq d_n$ i neka k-ti zadatak ima najdulje vrijeme izvođenja, tj. $p_k = max\{p_i\}$. Tada postoji $0 \leq \delta \leq n - k$ takav da u nekom optimalnom poretku vrijedi $\{1, 2, \ldots, k-1, k+1, \ldots, k+\delta\} \to k \to \{k+\delta+1, \ldots, n\}$.*

Dokaz. Neka je C'_k najkasnije moguće vrijeme završetka k-tog zadatka u bilo kojem optimalnom poretku uz rokove završetka $d_1, d_2, \ldots, d_n$. Neka je nadalje π optimalan poredak uz rokove završetka

$$d_1, d_2, \ldots, d_{k-1}, d'_k = max\{C'_k, d_k\}, d_{k+1}, \ldots, d_n$$

koji zadovoljava uvjet korolara 3.3.2 i neka je C_k vrijeme završetka k-tog zadatka u π.

Po lemi 3.3.3, π je optimalan poredak i uz originalne rokove završetka. Neka je $\delta = \#\{i \,|\, d_k < d_i \leq d'_k\}$. Tada po korolaru 3.3.2 i izboru poretka π vrijedi $\{1, 2, \ldots, k-1, k+1, \ldots, k+\delta\} \to k$. Nadalje, po definiciji vrijednosti C'_k vrijedi $C_k \leq C'_k \leq max\{C'_k, d_k\} = d'_k$, tj. k-ti zadatak u π ne kasni u odnosu na d'_k. Zbog toga postoji takav poredak π u kojem dodatno vrijedi i $k \to \{k + \delta + 1, \ldots, n\}$. Naime, svi zadaci u tom skupu imaju $d_i > d'_k$ pa je sve takve zadatke za koje je $C_i < C_k$ moguće prebaciti neposredno iza k-tog zadatka. Pritom njihova zaostalost ostaje nula jer je $C'_i \leq C_k \leq d'_k < d_i$. □

Važno je uočiti da dokaz teorema 3.3.4 nije konstruktivan za δ jer je δ definiran u ovisnosti o nepoznatoj vrijednosti C'_k. Teorem samo tvrdi da takav δ postoji, a njegova vrijednost se pronalazi dinamičkim programiranjem.

```
p je niz n cijelih brojeva // trajanje zadataka
d je niz n cijelih brojeva // rokovi zavrsetka zadataka
memo je matrica n x n x n x P parova (cijeli broj, lista)
inicijaliziraj sve celije matrice memo na NIL
Poredaj(izbacen, n_izb, l, r, tpoc):
    ako je l>r || n_izb=r-l+1:
        vrati (0, [])
    ako je memo[l][r][n_izb][tpoc] != NIL
        vrati memo[l][r][n_izb][tpoc]
    k := -1
    maxp := -1
    za i od l do r:
        ako je izbacen[i]:
            nastavi
        ako je p[i] >= maxp:
            maxp := p[i]
            k := i

    l_trajanje := 0
```

```
20      l_izbacen := niz(n, laz)
21      l_n_izb := 0
22      za i od l do k-1:
23          ako je izbacen[i]:
24              l_izbacen[i] := istina
25              l_n_izb += 1
26          inace:
27              l_trajanje += p[i]
28      l_izbacen[k] := istina
29      l_n_izb += 1
30
31      r_izbacen := niz(n, laz)
32      r_n_izb := 0
33      za i od k+1 do r:
34          ako je izbacen[i]:
35              r_izbacen[i] := istina
36              r_n_izb += 1
37
38      opt := +beskonacno
39      opt_poredak = []
40      za i od k+1 do r+1:
41          ako je i<=r && izbacen[i]:
42              l_izbacen[i] := istina
43              l_n_izb += 1
44              r_izbacen[i] := laz
45              r_n_izb -= 1
46          inace:
47              l_rjes := Poredaj(l_izbacen, l_n_izb, l, i, tpoc)
48              r_rjes := Poredaj(r_izbacen, r_n_izb, i+1, r, tpoc +
                    l_trajanje + p[k])
49              val := l_rjes[0] + r_rjes[0] + max(0, tpoc +
                   l_trajanje + p[k] - d[k])
50              ako je val < opt:
51                  opt := val
52                  opt_poredak := l_rjes[1] + [k] + r_rjes[1]
53              l_trajanje += p[i]
54
55      memo[l][r][n_izb][tpoc] := (opt, opt_poredak)
56      vrati memo[l][r][n_izb][tpoc]
```

Ispis 3.3: Rješenje problema $1||\sum T_i$.

Algoritam je prikazan u ispisu 3.3. Prikazani algoritam pretpostavlja da su zadaci poredani po nepadajućem roku završetka. Funkcija `Poredaj` vraća uređene parove ukupne zaostalosti zadataka i poretka kojim je ta zaostalost ostvarena. Parametar `izbacen` je bit-vektor koji služi za označavanje zadataka koje ne treba rasporediti u intervalu `[l, r]`, a parametar `n_izb` sadrži broj takvih zadataka. Konačno, parametar `tpoc` prikazuje početno vrijeme od kojeg raspored treba ostvariti. Vrijednost `tpoc` uvijek je manja od $\sum p_i$ što je u algoritmu označeno sa `P`.

U retcima 6 i 7 rješava se slučaj praznog intervala zadataka, a u retcima 8 i 9 se koristi memoizirano rješenje potproblema. Ključno je uočiti da nije potrebno memoizirati nad parametrom `izbacen` zato jer je njegova vrijednost jedinstveno određena parametrima `l`, `r` i `n_izb`. Po načinu rada algoritma, uvijek je izbačeno `n_izb` zadataka s najduljim trajanjem i indeksima između `l` i `r`.

U retcima 10–17 određuje se indeks zadatka s najduljim trajanjem izvođenja i taj indeks se postavlja u varijablu `k`.

Kako se u nastavku problem dijeli na lijevi i desni potproblem, u retcima 19–29 i 31–36 određuju se početne vrijednosti potrebnih argumenata rekurzivnim pozivima funkcije `Poredaj`. Konačno, u retcima 38–53 određuje se optimalan poredak u skladu s teoremom 3.3.4. Važno je uočiti da se uz pretpostavku da su poretci u drugom elementu para povratne vrijednosti funkcije `Poredaj` dvostruko povezane liste operacija nadovezivanja listi u retku 52 može izvršiti u konstantnom vremenu pa je vrijeme izračunavanja svakog poziva funkcije `Poredaj` $\Theta(n)$. Kako se funkcija `Poredaj` u najgorem slučaju može pozvati s $\Theta(n^3 * P)$ različitih argumenata, ukupno vrijeme izvođenja algoritma je $O(n^4 * P)$. U literaturi se ova složenost navodi bez dokaza jer implementacije algoritma na visokoj razini apstrakcije imaju višu složenost, barem za logaritamski faktor. Ako je zadatke prethodno potrebno sortirati po d_i, složenost očito ostaje asimptotski ista jer se sortiranje može u najgorem slučaju obaviti u $\Theta(n \log n)$ vremenu. Prostorna složenost jednaka je veličini memoizacijske strukture i iznosi $\Theta(n^3 * P)$.

Zaključak

Dinamičkim programiranjem se učinkovito rješavaju mnogi problemi, većinom optimizacijskog tipa. Ključna ideja dinamičkog programiranja je izbjegavanje višestrukog izračunavanja iste vrijednosti kroz korištenje dodatnog spremničkog prostora za pohranu međurezultata u rješavanju instance nekog problema. U radu su načelno i na konkretnim primjerima prikazana dva vida dinamičkog programiranja: memoizacija rekurzivnih funkcija i dinamičko programiranje od dna prema vrhu. Iako se oba vida zasnivaju na istoj ideji, za neke probleme je primjenjiva samo jedna. Na primjer, smanjiti vremensku ili prostornu složenost za memoizirani algoritam kao što je prikazano za dinamičko programiranje od dna prema vrhu može biti iznimno složeno ili nemoguće. S druge strane, memoiziranom rekurzijom se mogu riješiti problemi sa složenijim prostorima stanja koje nije jednostavno ili uopće moguće prikazati matricom.

Iako se dinamičko programiranje može jednostavno primijeniti za dobivanje eksponencijalnog rješenja mnogih inačica problema raspoređivanja, takva rješenja upitne su primjenjivosti. Za konstrukciju pseudopolinomnog rješenja tipično je potrebno matematički istražiti strukturu problema. Ipak, takva pseudopolinomna rješenja mogu biti primjenjiva kada je optimalnost rješenja visoke važnosti.

U radu je prikazan postupak konstrukcije pseudopolinomnih rješenja za probleme $1||\sum w_iU_i$ i $1||\sum T_i$. Primjenjivost jednostavnijeg rješenja za problem $1||\sum w_iU_i$ ovisi najviše o krajnjem roku d_n izvođenja zadnjeg zadatka jer je za očekivati da je to vrijeme značajno veće od broja zadataka. Kod složenijeg rješenja problema $1||\sum T_i$ slična ovisnost postoji o najduljem vremenu izvođenja pojedinog zadatka ili ukupnom vremenu izvođenja zadataka P. Ako se d_n odnosno P može u instancama ograničiti na neki polinom nad n, prikazana rješenja postaju polinomna, a i sam problem uz takva ograničenja ulazi u klasu P. U suprotnom će primjenjivost rješenja ovisiti o vremenskim zahtjevima za pronalaženje poretka i dostupnim računalnim resursima.

Dodatak A

Definicije korištenih oznaka za asimptotsku složenost

Neka su zadane funkcije $f(n)$ i $g(n)$. Tada vrijedi

$f(n) = O(g(n)) \Leftrightarrow \exists n_0 > 0, c > 0$ takvi da $0 \leq f(n) \leq c * g(n), \forall n \geq n_0$

$f(n) = \Omega(g(n)) \Leftrightarrow \exists n_0 > 0, c > 0$ takvi da $0 \leq c * g(n) \leq f(n), \forall n \geq n_0$

$f(n) = \Theta(g(n)) \Leftrightarrow f(n) = O(g(n))$ i $f(n) = \Omega(g(n))$.

Neformalno, ako vrijedi $f(n) = O(g(n))$, funkcija $f(n)$ za dovoljno velike n nije veća od funkcije $g(n)$ zanemarujući konstantne faktore. Na primjer, vrijedi $2n+3 = O(n)$, ali i $2n+3 = O(2^n)$. Kada je vremenska složenost nekog algoritma $O(g(n))$, to znači da vrijeme izvođenja algoritma s n ne raste brže od $g(n)$. Zbog toga nema smisla reći da je vremenska složenost algoritma *u najgorem slučaju* $O(g(n))$ jer je tada vremenska složenost u svakom slučaju $O(g(n))$.

Ponekad je važno iskazati i donju asimptotsku granicu, a za to se koristi oznaka Ω. Na primjer, može se dokazati da je vremenska složenost svakog algoritma za sortiranje ako se elementi mogu samo uspoređivati $\Omega(n \log n)$. Drugim riječima, niti jedan algoritam za sortiranje u takvom modelu ne može imati asimptotski manju složenost od $n \log n$.

Ako se želi istovremeno iskazati obje granice, koristi se Θ. Neformalno govoreći, koristeći Θ dobiva se u asimptotskom smislu relacija jednakosti. Iako Θ u sebi sadrži više informacija od O, u praksi se često koristi O čak i kada vrijedi i Θ.

Na primjer, ako je vremenska složenost nekog algoritma $O(2^n)$, moguće je da uistinu ne vrijedi da je vremenska složenost i $\Theta(2^n)$. To će sigurno

biti slučaj ako je vremenska složenost algoritma i $O(n)$ kao što je slučaj za funkciju $2n + 3$. Čak i ako nije moguće asimptotski smanjiti funkciju u O-oznaci, svejedno je moguće da se ne može izraziti Θ-složenost. Poznati primjer ovog slučaja je algoritam *quicksort*. Vremenska složenost algoritma *quicksort* je $O(n^2)$, ali *nije* $O(n \log n)$[1]. Prosječno vrijeme izvođenja iznosi $\Theta(n \log n)$ pa je neispravno reći da je složenost algoritma i $\Theta(n^2)$. S druge strane, ispravno je reći da je složenost algoritma *quicksort* $\Theta(n^2)$ *u najgorem slučaju* jer takvi slučajevi stvarno postoje.

[1]Ovo vrijedi za naivnu implementaciju, kao i za sve jednostavne modifikacije kao što su *medijan-od-K* particioniranje. Štoviše, i uz slučajan odabir elementa za particioniranje moguć je slučaj u kojem je vrijeme izvođenja kvadratno, ali je vjerojatnost tog slučaja vrlo malena. Složenost $\Theta(n \log n)$ može se postići tek primjenom linearnog algoritma za traženje medijana, ali takva inačica algoritma *quicksort* nije praktično značajna zbog visokih konstanti.

Literatura

[1] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1994.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

[3] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[4] R. Karp. Reducibility among combinatorial problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[5] J. Du and Joseph Y.T. Leung. Minimizing total tardiness on one machine is NP-hard. *Math. Oper. Res.*, 15:483–495, July 1990.

[6] E. L. Lawler. A pseudo-polynomial algorithm for sequencing jobs to minimize total tardiness. 1977.