Uvod u algoritamske tehnike

Tema i nacrt predavanja:

Razmatraćemo različite pristupe u rešavanju programerskih problema. Fokusiraćemo se na tehnike za konstrukciju algoritama, na osobine, primenljivost i performanse.

UVOD

Pristup rešavanju problema:

1. Potrebna široka baza algoritama, poznavanje kako rade u cilju razumevanja i efikasne i tačne implementacije.
2. Poznavanje performansi i ponašanja nad različitim skupovima ulaznih podataka

Biblioteka algoritama i problemi koji se njome mogu rešiti – da li je problem samo drugačiji zapis već poznatog problema.

Prepoznavanje zajedničkih osobina sa nekim poznatim problemom – rešenje kroz modifikaciju rešenja poznatog problema.

Novi problem – nije sličan ni sa jednim poznatim. Sledi analiza i uočavanje strukture problema.

Da li se može podeliti na podprobleme različite po prirodi, a njhova rešenja uklopiti u celovito rešenje.

Da li se može podeliti na istorodne podprobleme manjeg reda veličine i da li je svrsishodno tražiti rešenja za istorodne probleme manjeg reda veličine, a kasnije različitim tehnikama ukomponovanti dobijena rešenja u celinu. Za dekompoziciju na probleme manjeg reda veličine i komponovanje rešenja podproblema u celovito rešenje postoje različite algoritamske tehnike. U principu možemo da govorimo o pristupima koji od funkcionišu

- odozgo naniže po dimenziji problema smanjujući dimenziju. Krećemo od velike slike ka detaljima, i nazad ka velikoj slici.
- odozdo naviše gde se polazi od jednostavnijih problema, čija se rešenja, detalji, koriste za formiranje velike slike.

Verifikacija - nadjeno rešenje je neophodno verifikovati u cilju potvrde njegove ispravnosti i celovitosti. Možemo postaviti pitanja:

- Da li su korišćeni algoritmi upotrebljeni za rešavanje problema kojima su namenjeni i pod uslovima pod kojima daju korektno rešenje.
- Da li su granični slučajevi korektno obradjeni

Evaluacija – nadjeno rešenje treba da zadovolji postavljene zahteve u pogledu efikasnosti. Algoritmi se primenjuju i čvrsto su povezani sa strukturama podataka. Mogu

se realizovati na različite načine i uz korišćenje odgovarajućih struktura podataka. Efikasno rešenje u sebi sadrži efikasan algoritam i efikasnu strukturu podataka.

Analiza algoritama

Algortam je jasno definisan set jednostavnih instrukcija koje trebasledeti u cilju rešavanja problema. Kad je dat algoritam za neki problem i poznato je da je korektan, važan korak je određivanje koliko resursa, vremena ili prostora algoritam zahteva. Analiza potrebna za procenjivanje korišćenja resursa nekog algoritma je uopšte uzev teorijski zahtev, te je potreban formalni okvir.

Počećemo sa nekoliko definicija:
definicija: $T(N) = O(f(N))$ ako postoje pozitivne konstante c i n_0 takve da $T(N) <= c\, f(N)$ kada $N >= n_0$.

definicija: $T(N) = \Omega(g(N))$ ako postoje pozitivne konstante c i n_0 takve da $T(N) => c\, g(N)$ kada $N >= n_0$.

definicija: $T(N) = \Theta(h(N))$ ako i samo ako $T(N) = O(h(N))$ i $T(N) = \Omega(h(N))$.

definicija: $T(N) = o(p(N))$ ako i samo ako $T(N) = O(p(N))$ i $T(N) \neq \Theta(p(N))$.

Ideja ovih definicija je uspostavljanje relativnog redosleda medju funkcijama u smislu relativnih brzina rasta. U tom smislu ako je $T(N) = O(f(N))$ kaže da je $f(N)$ gornja granica za $T(N)$. Takodje se za T(N) kaže da je donja granica za $f(N)$.

Izrazi koje dobijemo sa navedenim definicijama predstavljaju ponašanje algoritma zavisno od veličine ulaznog niza i ova analiza je poznata kao asimptotsko ponašanje.

Važno je znati i sledeće osobine:

1. Ako je $T_1(N) = O(f(N))$ i $T_2(N) = O(g(N))$ onda
   a) $T_1(N) + T_2(N) = \max(O(f(N)), O(g(N)))$,
   b) $T_1(N) * T_2(N) = O(f(N)) * O(g(N))$.

2. Ako je $T(N)$ polinom stepena k onda je $T(N) = \Theta(N^k)$

3. $\log^k N = O(N)$ za bilo koju konstantu k.

Ove su informacije dovoljne da uredjenje funkcija koje se najčešće javljaju u analizi algoritama:

Funkcija	Naziv
c	konstanta
$\log N$	logaritamska
$\log^2 N$	logaritam na kvadrat
N	linearna
$N \log N$	
N^2	kvadratna

N^3	kubna
2^N	eksponencijalna

Loš je stil uključivanje konstantni ili izraza nižeg reda unutar izraza Veliko O, kao u primerima: $O(2N^2)$ ili $O(2N^2+N)$.

Model

Radi analize algoritama formalnozasnovane potreban nam je model izračunavanja. Naš model je standardni računar u kome se instrukcije izvršavaju sekvencijalno. Naš model ima standardni repertoar jednostavnih instrukcija, sabiranje, množenje, poredjenje, dodela, ali za razliku od stvarnih računara troši se po tačno jedna jedinica vremena da se uradi biklo šta jednostavno. Ne postoje napredne operacije recimo sortiranje i inverzija matrica koje se rade u jednoj jedinici vremena. Takodje podrazumevamo beskonačnu memoriju.
Ovaj model ima nedostatke. U realnom životu ne uzimaju sve operacije tano jednu jedinicu vremena, kao što se i jedno čitanje sa diska računa isto kao sabiranje, mada je za više redova veličina duže. Takodje pretpostavka o beskonačnoj memoriji takodje nas oslobadja brige o promašaju stranice page faulting, što je realan problem posebno kod efikasnih algoritama.

Šta se analizira

Najvažnija tema za analizu je vreme izvršenja. Nekoliko faktora utiču na vreme izvršenja programa. Neki, kao što su prevodilac i računar koji se koriste su izvan bilo kog teorijskog modela, pa iako su važni ne možemo da radimo sa njima. Ostali glavni faktori su korišćeni algoritam i ulazni podaci.

Tipično su velična ulaznih podataka i njihov sadržaj glavne teme. Tako definišemo $T_{avg}(N)$ i $T_{worst}(N)$, kao srednji i najgori slučaj vremena izvršenja. Retko se razmatraju najbolje performanse. Srednji slučaj predstavlja tipično ponašanje, dok najgori slučaj predstavlja garantovane performanse.

Izračunavanje vremena izvršenja

Postoji nekoliko načina za procenu vremena izvršenja programa. Najtačnija je ona dobijena empirijski. Naime, ako očekujemo da dva programa imaju slična vremena verovatno je najbolji način da se oba kodiraju i onda izvrše.

Uopšte postoji nekoliko algoritamskih ideja i dobro je rano odbaciti loše, te je neophodna analiza algoritama. Štaviše, mogućnost davanja analize obično daje uvid u projektovanje efikasnih algoritama. Analiza takodje identifikuje i uska grla koje treba kodirati pažljivo.
primer:

```
int sum(int N)
{
      int delimicnaSuma;
      delimicnaSuma = 0;                  // 1
      for( int i = 1; i <= N;  i++)       // 2
            delimicnaSuma +=  i * i * i;  // 3
      return delimicnaSuma;        // 4
}
```

Analiza fragmenta je jednostavna. Deklaracija ne uzima vreme, a linije 1 i 4 jednu jedinicu vremena. Linija 3 zahteva 4 jedinice (2 množenja, jedno sabiranje i jedna dodela). Linija 2 nosi inicijalizaciju i, zatim test i <= N, inkrementiranje i. Ukupna cena je 1+ N+1 + N = 2N + 2. Ukupno ima 6N + 4 jedinica vremena. Dakle metod je *O(N)*.

Razmatranje nas vodi do nekoliko jednostavnnih pravila koje možemo koristiti:

1. for petlja: Vreme izvršenja for petlje najviše proizvodu vremena za iskaze unutar for petlje (uključujući testove) i broja iteracija.

2. ugnježdene petlje: Vreme izvršenja je jednako proizvodu iskaza unutar ugnježdene petlje i veličina svih petlji.

```
      for( i = 0; i < n;  i++)
            for( j = 0; i < n;  i++)
                  k++;
```

3. iskazi u sekvenci: Vremena se sabiraju

4. if else Vreme izvrsenja nije manje od vremena testa plus veće od vremena iz grane.

Analiza rekuruzije može biti složenija jer uključuje rekurentne ralcije koje moraju biti rešene. Početni i ilustrativni primer korišćenja rekurzije za računanje faktorijela:

```
long faktoriel( int n )
{
      if( n <= 1 )
            return 1;
      else
            return  n * faktoriel ( n - 1);
}
```

U slučaju da se rekurzija može da zameni for petljom lako je izračunati vreme izvršenja, mada ovo nije ispravno korišćenje rekurzije. Takodje je preporuka da se rekurzijom nikad ne rešava isti problem dva puta. Primer je izračunavanje fibonačijevih brojeva:

```
long fib ( int n )
{
      if ( n <= 1)
            return 1;
      else
            return  fib ( n - 1)  +  fib ( n - 2 );
}
```

Problem podniza sa maksimalnom sumom

Razmatraćemo četiri različita algoritma za rešavanje ovog problema i ilustrovati tehnike rešavanja kao i procenu performansi izračunavanja.

Prvo rešenje je reda $O(N^3)$ i iscrpljuje sve mogućnosti :

```
//seqStart i seqEnd predstavljaju nadjenu najbolju sekvencu
static private int seqStart = 0;
static private int seqEnd = -1;

//Kubni algoritam
public static int maxSubSum1( int [ ] a )
{
  int maxSum = 0;
  seqStart = 0; seqEnd = -1;

  for( int i = 0; i < a.length; i++ )
    for( int j = i; j < a.length; j++ )
    {
        int thisSum = 0;

        for( int k = i; k <= j; k++ )
            thisSum += a[ k ];
        if( thisSum > maxSum )
        {
            maxSum   = thisSum;
            seqStart = i;
            seqEnd   = j;
        }
      }
  return maxSum;
}
```

Drugo rešenje je reda $O(N^2)$.
Kubno vreme izračunavanja se može izbeći izbacivanjem ugnježdene for petlje. To nije uvek moguće učiniti, ali u ovom slučaju jeste. Naime $\sum_{k=i}^{j} A_k = A_j + \sum_{k=i}^{j-1} A_k$ pa se nepotrebno izračunavanje sume sa desne strane može izbeći:

```
//Kvadratni algoritam
//seqStart i seqEnd predstavljaju nadjenu najbolju sekvencu
public static int maxSubSum2( int [ ] a )
{
  int maxSum = 0;
  seqStart = 0; seqEnd = -1;
```

```
  for( int i = 0; i < a.length; i++ )
  {
    int thisSum = 0;
    for( int j = i; j < a.length; j++ )
    {
        thisSum += a[ j];
        if( thisSum > maxSum )
        {
            maxSum   = thisSum;
            seqStart = i;
            seqEnd   = j;
        }
    }
    return maxSum;
}
```

Treće rešenje je reda *O(N log N).*

Postoji i rekurzivno i relativno komplikovano rešenje problema koji razmatramo. Algoritam koristi podeli i pobedi strategiju. Ideja je da se problem podeli u dva približno jednaka podproblema, koji se onda rešavaju rekurzivno. To je podeli deo. Pobedi deo se sastoji od spajanja rešenja dva podproblema i malog dodatnog rada u cilju rešenja celog problema.

U našem slučaju maksimalna sekvenca može biti na jednom od tri mesta. Ili se javlja potpuno u levoj polovini ulazna ili potpuno u desnoj polovini ili seče sredinu i delovi se nalazi u obe polovine. Prva dva slučaja mogu se rešiti rekurzivno. Treći slučaj se može rešiti nalaženjem najveće sume u levoj polovini koja uključuje poslednji element u prvoj polovini i nejveće sume u drugoj polovini koja uključuje prvi element u desnoj polovini. Ove dve sume mogu biti dodate i spojene u jednu celinu.

Prmer niza:

leva polovina				desna polovina			
4	-3	5	-2	-1	2	6	-2

Vrednost maximalnog podniza potpuno sadržanog u levoj polovini je 6 (A_1 do A_3) , a maximum koji uključuje poslednji element u levoj polovini je 4 (A_1 do A_4).
Vrednost maximalnog podniza potpuno sadržanog u desnoj polovini je 8 (A_6 do A_7) , a maximum u desnoj polovini koji počinje od prvog elementa je 7 (A_5 do A_7). Tako je maximum podniza koji uključuje obe polovine 4+7 = 11. U ovom primeru je to ujedno i maximum i za ceo ulazni niz.

```
// Recurzivna verzija maximuma podniza
// Nalazi maximum u nizu delu niza a  izmedju [left..right].

private static int maxSumRec( int [ ] a, int left, int right )
{
    int maxLeftBorderSum = 0, maxRightBorderSum = 0;
    int leftBorderSum = 0, rightBorderSum = 0;
    int center = ( left + right ) / 2;
```

```
    if( left == right )  // Base case
        return a[ left ] > 0 ? a[ left ] : 0;

    int maxLeftSum  = maxSumRec( a, left, center );
    int maxRightSum = maxSumRec( a, center + 1, right );

    for( int i = center; i >= left; i-- )
    {
        leftBorderSum += a[ i ];
        if( leftBorderSum > maxLeftBorderSum )
            maxLeftBorderSum = leftBorderSum;
    }

    for( int i = center + 1; i <= right; i++ )
    {
        rightBorderSum += a[ i ];
        if( rightBorderSum > maxRightBorderSum )
        maxRightBorderSum = rightBorderSum;
    }

    return max3( maxLeftSum, maxRightSum,
                 maxLeftBorderSum + maxRightBorderSum );
}

//  rutina za nalazenje maximalnog medju tri argumenta
private static int max3( int a, int b, int c )
{
    return a > b ? a > c ? a : c : b > c ? b : c;
}
```

Analiza vremena izvršenja ide na sličan način kao za Fibbonacci brojeve. Uočavamo dve sukcesivne for petlje gde svaka ima po 3 aktivnosti i obe zajedno obrade ceo ulazni niz. Dodatni kod zahteva još nekoliko vremenskih jedinica, ali to ostaje u okviru *O(N)*. Treba izračunati koliko vremena treba za dva rekurzivna poziva za svaku polovinu niza. To nas vodi do relacije:

$T(1) = 1$
$T(N) = 2\,T(N/2) + O(N)$

Kasnije ćemo videti detaljno razmatranje diferencne jednačine koja se javlja u metodi podeli i pobedi, a ovde se može da uoči sledeći šablon:

$T(1) = 1$ $T(2) = 4 = 2 * 2$ $T(4) = 12 = 4 * 3$ $T(8) = 32 = 8 * 4$ i $T(16) = 80 = 16 * 5$

Ako je $N = 2^k$ onda je $T(N) = N * (k + 1) = N \log N =$ *O(N log N)*
Analiza polazi od pertpostavke da je N parno, štaviše da je N stepen 2. Ukoliko to nije slučaj potrebna je komplikovanija analiza, ali i ona ostaje pri istom zaključku *O(N log N)*

Četvrti algoritam je O(N)

Ovaj algoritam je jednostavniji za implementaciju nego prethodni rekurzivni i efikasniji je.

```
public static int maxSubSum3( int [ ] a )
{
    int maxSum = 0;
    int thisSum = 0;

    for( int i = 0, j = 0; j < a.length; j++ )
    {
        thisSum += a[ j ];

        if( thisSum > maxSum )
        {
            maxSum = thisSum;
            seqStart = i;
            seqEnd   = j;
        }
        else
            if( thisSum < 0 )
            {
                i = j + 1;
                thisSum = 0;
            }
    }

    return maxSum;
}
```

Radi razumevanja korektonsti četvrtog algoritma podsetimo se da su u prvom i drugom algoritmu promenljive `i` i `j` predstavljale početak i kraj podniza. Razmotrimo moguće vrednosti članova niza na mestu `i` . Ako je `A[i]` negativan on ne mo\e da predstavlja po;etak podsekvence, koja bi bez njega imala veću zbir. Slično bilo koja negativna podsekvenca ne može biti prefix optimalne podsekvence. Ključno je zapažanje da povećanje `i` ne mora da bude na `i+1` već na `j+1.` Da bismo se uverili da je ovo korektno pretpostavimo da je p neki index izmedju `i` i `j` . Bilo koja podsekvenca koja počinje od `p` nije veća od odgovarajuće sekvence koja počinje od `i` i uključuje podsekvencu od `A[i]` do `A[p-1]`, jer ona nije negativna. Sekvenca počinje da bude negativna od `j` .

Ekstra prednost ovog algoritma je što on radi samo jedan prolaz kroz podatke, što ne pamti u memoriji ni jedan deo ulaznog niza. Takodje je vrlo bitno i to što on za pročitani deo niza u svakom trenutku ima najbolje rešenje, koje nemeju ostali algoritmi. Algoritmmi kjoi imaju ovu osobinu se zovu *on-line* algoritmi. On-line algoritam koji zahteva konstantni prostor i izvršava se u linearnom vremenu jeako postoji najbolje moguće rešenje.

Tehnike projektovanja algoritama

Razmatraćemo pet opštih tipova algoritama koji se koriste za rešavanje problema. Za mnogo problema prilično je sizgledno da će najmanje jedan od ovih metoda da da rezultate. Svaku metodu ćemo obradjivati u redosledu upoznavanja sa opštim pristupom, pogledaćemo nekoliko primera i diskutovati vremensku i prostornu kompleksnost.

Grabljiivi (greedy) algoritmi

Grabljivi algoritmi ili algoritmi grabljivog izbora funkcionišu u koracima - fazama. U svakom koraku se razmatra trenutno stanje i donosi odluka koja izgleda dobro u tom trenutku, bez razmatranja budućih konsekvenci. Uopšte uzev, to znači da se bira neki lokalni optimum. Izvorni naziv "take what you can get now" sugeriše strategiju ovih algoritama. Kada se algoritam završi nadamo se da je lokalni optimum jednak globalnom optimumu. Ako je to slučaj dobili smo optimalno rešenje, inače je rešenje suboptimalno. Ako se ablsolutno najbolji odgovor ne zahteva onda jednostavni grabljivi algoritmi koriste za generisanje približnih odgovora, radije no korišćenje znatno komplikovanijih algoritama za generisanje tačnih odgovora.

Postoje primeri grabljivih algoritama iz svakodnevnog života. Svima je poznat primer isplate odredjene sume novca uz upotrebu minimalnog broja novčanica. Rešenje kroz usitnjavanje najveće novačnice koja prevazilazi traženi iznos je primer grabljivog ponašanja. Grabljivo ponašanje ne mora da da globalni optimum, elementarni primer je saobraćajna gužva, gde je u časovim asaobraćajnog špica bolje ići dužim okolnim putem izbegavajući centar grada. "Preko preče, naokolo bliže".

Dalje ćemo pogledati nekoliko primera problema rasporedjivanja, koji koriste greedy algoritme. Mnogo problema rasporedjivanja su ili NP-kompletni problemi ili slične kompleksnosti ili su rešivi grabljivim algoritmima. Prvi primer razmatra problem rasporedjivanja poslova, drugi se bavi generisanjem kodova Hufmann, a u trećem ćemo pogledati nalaženje greedy aproksimativnog rešenja.

Greedy - Jednostavni problem rasporeda poslova

Neka su dati poslovi $j_1, j_2, \ldots j_n$ svi sa poznatim vremenom izvršenja $t_1, t_2, \ldots t_n$. Imamo jedan procesor na raspolaganju. Zadatak je naći najbolji redosled poslova tako da srednje vreme kompletiranja bude minimalno. Pod vremenom kompletiranja podraumevamo vreme provedeno u redu čekanja i vreme izvršenja. Izvršenje poslova se ne može prekidati.

Rešenje problema je u redosledu po kome se bira prvo najkraći posao, a zatim medju preostalim opet najkraći i tako do kraja. Pogodno je sortirati u rastućem redosledu poslove po vremenu izvršenja i taj redosled je optimalni. Dokaz korektnosti algoritma se može izvesti iz sledećeg razmatranja vremena kompletiranja svih poslova. Neka su posle sortiranja poslovi poredjani u rastućem redosledu i označeni: $t_{i1}, t_{i2}, t_{i3}, \ldots. t_{in}$, Vremena kompletiranja su redom po poslovima:

t_{i1}
$t_{i1} + t_{i2}$
$t_{i1} + t_{i2} + t_{i3}$
$t_{i1} + t_{i2} + t_{i3} + \ldots + t_{in}$

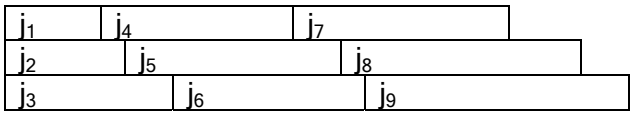
A ukupno vreme je $C = \sum_{k=1}^{N} (N - k + 1) t_{ik}$

$$C = (N+1) * \sum_{k=1}^{N} t_{ik} - \sum_{k=1}^{N} k * t_{ik}$$

U izrazu za C prva suma je nezavisna od redosleda izvršenja poslova i samo druga može da utiče na ukupni rezultat. Pretpostavimo da u redosledu postoje x > y tako da $t_{ix} < t_{iy}$. Zamenom j_{ix} i j_{iy} druga suma se povećava smanjujući ukupni rezultat. Tako je bilo koji redosled poslova u kojima su vremena neopadajuća optimalno rešenje.

Problem se može proširiti na više procesora. Ukoliko je na raspolaganju više procesora, a možemo pretpostaviti da su poslovi uredjeni u neopadajući niz po vremenu izvršenja. Algoritam za višeprocesorski slučaj startuje poslove po redosledu, ciklično ih rasporedjujući po procesorima.
Primer:

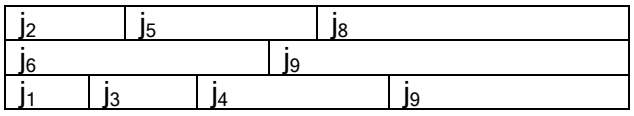
Posao	1	2	3	4	5	6	7	8	9
Vreme	3	5	6	10	11	14	15	18	20



0 3 5 6 13 16 20 28 34 40

Može se primetiti da procesori I i II svoje poslove završe pre III.

Proširenje problema može ići u pravcu zahteva da se minimizira vreme završetka poslednjeg posla. Primer redosleda za ovaj problem je:



0 3 5 9 14 16 19 34

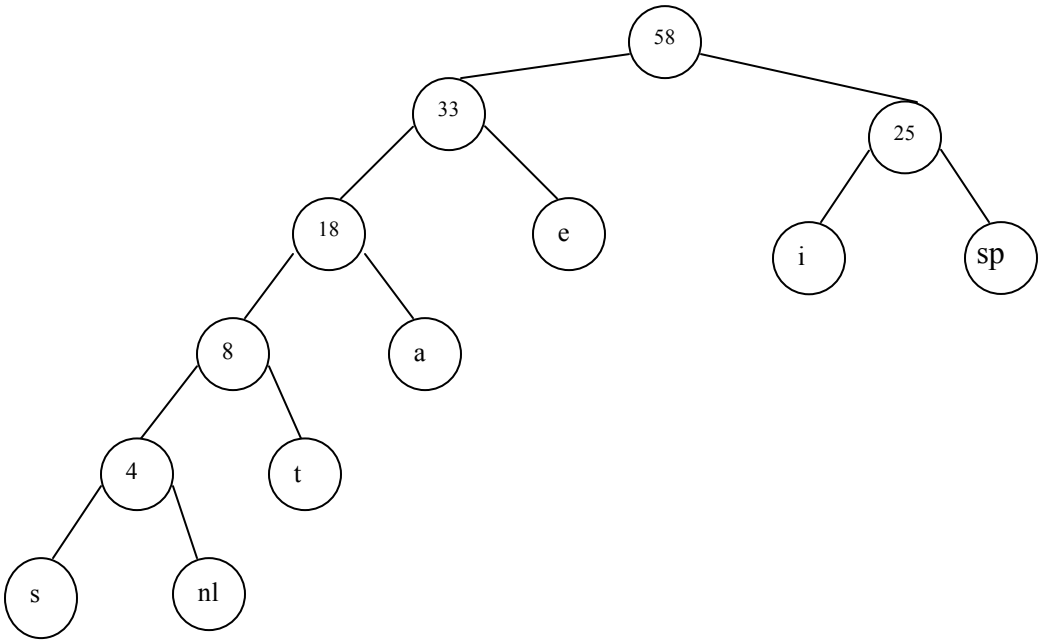
Ako posovi pripadaju različitim korisnicima izbor najpre najkraći posao je dobar. Ali ako svi ovi poslovi imaju istog vlasnika, za njega bi druga verzija rešenja bila bolja.
Iako ova dva problema sa rasporedom poslova izgledaju slično, poslednja varijanta je NP-kompletna.

Hufman kod

Uobičajeni ASCII set sadrži oko 100 karaktera koji se moguodštampati. Radi razlikovanja potrebno je $\lceil \log 100 \rceil = 7$ bitova. Osmi bit je dodat kao dopuna do parnosti. Pretpostavimo da imamo skup karaktera a, e, i s, t, sp i nl. Broj ponavljanja dat je u tabeli, kao i broj bitova potrebnih za predstavljanje ako koristimo minimalna 3 bita:

a	e	i	s	t	sp	nl
10	15	12	3	4	13	1
30	45	36	9	12	39	3

Ukupno je potrebno 174 bitova. U realnom životu fajlovi mogu biti veoma veliki, mogu biti rezultat nekih programa i može da postoji veliki disparitet izmedju najčešćih i najredjih karaktera. Zadatak je iskoristi ovaj disparitet i predstaviti fajlove na način koji redukuje potreban prostor. Ideja je da se koristi kod promenljive dužine i to tako da se češći karakteri kodiraju kraćim, a redji dužim kodom. Ako bismo kodove karaktera predstavili stablom može se usvojiti tumačenje koda kao puta od korena do lista gde se u kod recimo dodaje 0 ako se krećemo levo, a 1 ako se krećemo desno. Za primer jedno od stabala koje zadovoljava uslove je:



Generisani kodovi:

a	e	i	s	t	sp	nl	ukupno
10	15	12	3	4	13	1	
30	45	36	9	12	39	3	174
001	01	10	00000	0001	11	000011	
30	30	24	15	16	26	6	141

Hufmann algoritam generiše traženo stablo. On održava šumu težinskih stabala, čija se težina dobija kao zbir frekvencija karaktera u stablu. Inicijalno se za svaki karakter formmira stablo od po jednog čvora. Zatim se biraju dva stabla sa najmanjim težinama,

ona se pomešaju tako da se dobije zajedničko stablo čiji su oni levi i desni potomak. Na ovaj način se u svakom koraku broj stabala smanjuje za jedan te nam na kraju preostaje samo jedinstveno stablo iz koga čitamo kodove.

Problem pakovanja u kutije

Poznat je pod nazivom "bin packing" . Ovi algoritmi rade brzo ali nije sigurno da daju optimalno rešenje. Pokazaćemo da rešenja nisu daleko od optimalnih.

Dato je N brojeva s_1, s_2, . . . , s_N . Svi zadovoljavaju uslov $0 < s_i <= 1$.Zadatak je spakovati te brojeve u najmanji broj kutija, koje imaju jedinični kapacitet.

Za brojeve 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8 optimalno pakovanje je: 0.8 + 0.2; 0.7+0.3; 0.4 + 0.1 + 0.5.

Postoje dve verzije ovog problema: on-line i off-line. U on-line verziji svaki broj mora da bude spakovan u svoju kutiju, pre no što počne obrada narednog broja. Kod off-line verzije ne moramo da radimo ništa dok se svi brojevi sa ulaza ne pročitaju.

Za on-line verzije algoritama za problem koji razmatramo treba prvo da razmotrimo da li mogu da daju optimalno rešenje čak iako je neograničeno računanje dozvoljeno. Da bismo pokazali da oni ne mogu da uvek daju optimalno rešenje daćemo posebno teške ulazne podatke za obradu. Posmatrajmo ulaznu sekvencu I_1 od M brojeva oblika $0.5 - \varepsilon$ koj slede M brojeva oblika $0.5 + \varepsilon$, gde je $0 < \varepsilon < 0.01$. Očigledno je da oni mogu da budu optimalno spakovani u M kutija. Pretpostavimo da postoji optimalni on-line algoritam A . Posmatrajmo rezultat rada algoritma A na na sekvenci I_2 koja se sastoji samo od M brojeva $0.5 - \varepsilon$. Očigledno se oni mogu spakovati optimalno u $\lceil M / 2 \rceil$ kutija. Algoritam A će spakovati ove brojeve u M kutija kako bi kao optimalan takvo optimalno rešenje dao i za I_2 koje je prva polovina I_1 zato što je I_2 prva polovina I_1 . Odavde vidimo da takav algoritam A ne postoji.

Teorema: Postoji ulazni niz za koji bilo koji optimalni algoritam koristi najmanje 4/3 optimalnog broja kutija.

Next fit algoritam je možda najjednostavniji on-line algoritam. On pokuša da smesti novi broj u kutiju u koju je stavljen prethodni i ako može tu ga i ostavi. Ako nije moguće uzima se nova kutija. Algoritam je vrlo jednostavan za implementaciju i radi u lineranom vremenu.
Teorema: Neka je M optimalan broj kutija za pakovanje I brojeva. Next fit nikad ne koristi više od 2M kutija.

Iako next fit ima prihvatljive garantovane performanse on se slabo ponaša u praksi, jer uzima novu kutiju iako nije neophodno. First fit pristup pretražuje kutije od početka tražeći prvu u kojiu može da smesti novi broj. Ovde se nova kutija zauzima tek kad to nema alternativu. Jednostavno je implementirati ovaj algoritam tako da radi u $O(N^2)$, ali se može da impelentira i sa $O(N \log N)$ performansama.
Teorema: Neka je M optimalan broj kutija potrebnih za pakovanje I brojeva. First fit ne koristi više od $\lceil 17M / 10 \rceil$ kutija.
Praksa pokazuje da ako se first fit testira na velikom uzorku brbojeva uniformno rasporedjenih izmedju 0 i 1 on koristi oko 2% više kutija no što je optimalno, što je vrlo prihvatljivo.

Treći pristup je best fit. Umesto da se novi broj stavlja u prvu kutiju u kojoj je to moguće on to radi u kutiju čiji je preostali prostor najbliži novom broju. Može se impelmentirati kao *O(N log N)* . Garantovane performanse su iste kao za first fit iako se čini kao učeniji izbor, jer imaju iste najgore slučajeve ulaza.

Off-line algoritmi se mogu primeniti ako možemo videti ceo niz brojeva pre davanja odgovora. Očekujemo, s pravom, da se mogu postići bolja rešenja korišćenjem iscrpljujućeg pretraživanja i imamo teorijsko poboljšanje nad on-line algoritmima. Glavni problem sa on-line algoritmima je što ime je teško da spakuju velike brojeve naročito ako se javljaju kasnije u nizu. Jednostavan način da se sa tim izadje na kraj je da se oni sortiraju u opadajućem ili tačnije nerastućem redosledu i da se onda primeni first fit opadajući ili best fit opadajući. Oni mogu da daju optimalna rešenja, ali to nije u opštem slučaju tačno. Oni imaju vrlo slične izraze za garantovane performanse i za first fit je da nikad ne koristi više od $\lceil 11M / 9 \rceil + 4$ kutije.

Podeli i pobedi "divide and conquer"

Ovo je još jedna opšta tehnika korišćena za dizajn algoritama. Kak joj i sam naziv kaže sastoji se od dva dela:

- podeli – "divide" problem se deli na manje probleme koji se rešavaju rekukrzivno
- pobedi – "conquer" rešenja manjih problema se kombinuju u cilju rešavanja polaznog problema.

Tradicionalno se rutine u kojima text sadrži najmanje dva rekurzivna poziav zovu podeli i pobedi, što ne važi za rutine sa jednim rekurzivnim pozivom. Insistira se da se podproblemi ne preklapaju. Već smo u uvodu videli jedan primer ove koncepcije, a mogu se navesti i drugi primeri kao što su obilasci stabla, mergesort, quicksort algoritmi. Kao strašno neefikasan primer možemo navesti rekurzivno rešenje nalaženja Fibonacci-jevih brojeva, koje samo liči na ovu strategiju, jer u njemu nema stvarne deobe problema na dva nezavisna podproblema.

Procena vremena izvršenja podeli i pobedi algoritama se dobija iz jednačine:

$T(N) = aT(N/b) + \Theta(N^k)$ a >= 1 i b > 1

gde je smisao b broj podproblema, a $\Theta(N^k)$ vreme potrebno za dekomponovanje na podproblema i formiranje rešenja polaznog problema.
Rešenje ove diferencne jednačine je:

$$T(N) = \begin{cases} O(N^{\log_b a}) & a > b^k \\ O(N^k \log N) & a = b^k \\ O(N^k) & a < b^k \end{cases}$$

U zadacima se najčešće sreće oblik jednačine $T(N) = 2T(N/2) + \Theta(N)$
a u tom slučaju je *T(N) = O(N log N)* .

Problem najbližih tačaka

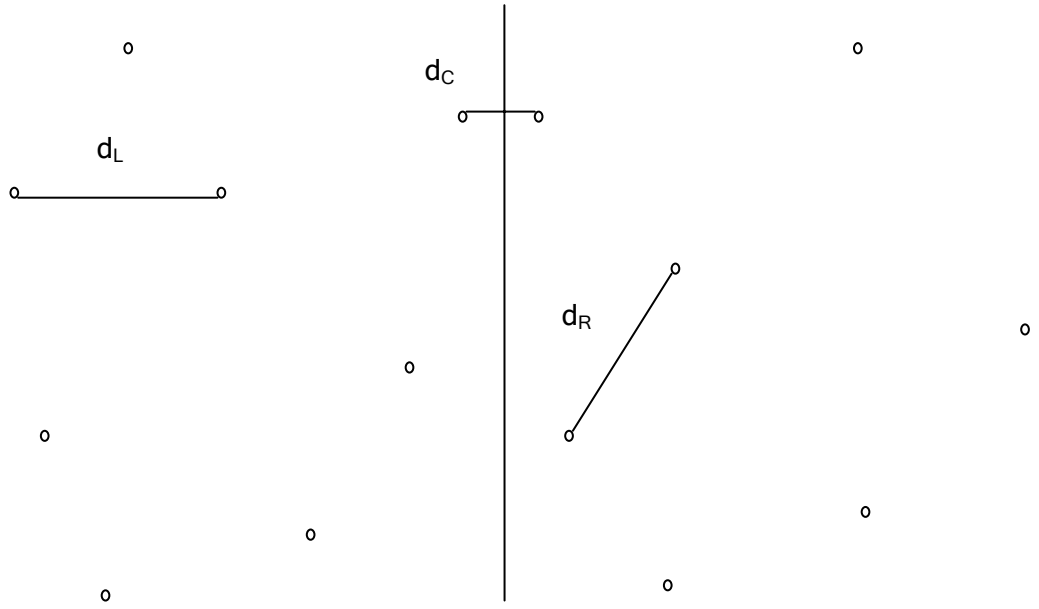
Dato je N tačaka u ravni i treba naći najbliži par tačaka.

Za N tačaka postoji N * (N - 1) / 2 rastojanja te je moguće proveriti sve njih i naći najkraće. Pošto je ovo u osnovi iscrpljujućeg pretraživanje m0ogu se očekivati bolji algoritmi i rezultati.

```
for( i = 0; i < numPt;  i++ )
   for( j = i + 1; j < numPt;  j++ )
      if(  dist( p[i], p[j] ) < delta )
         delta = dist( p[i], p[j] );
```

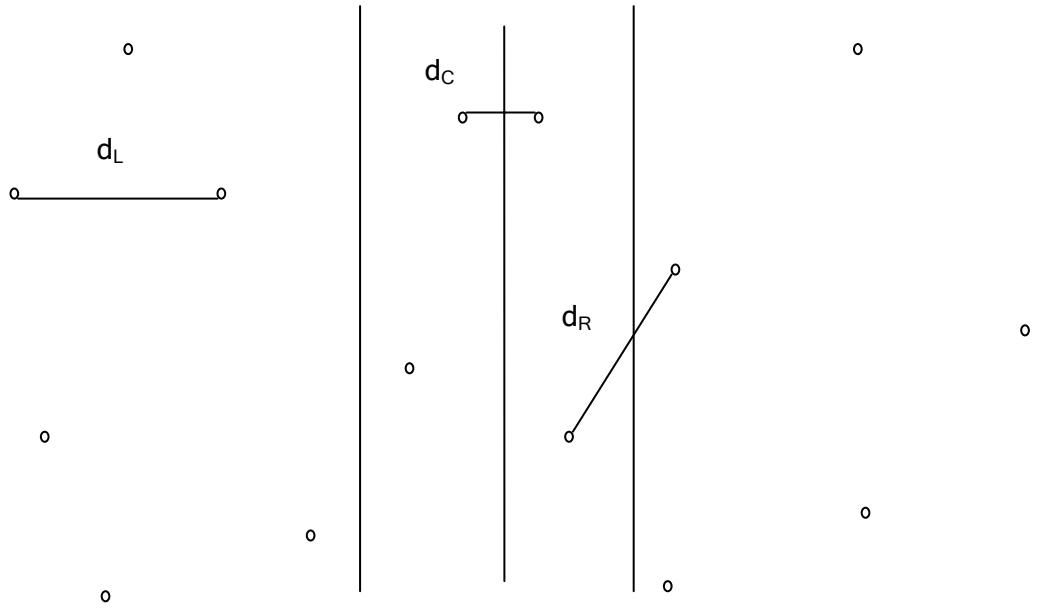
Razmišljanje i rešenje koje ćemo pokazati je reda *O(N log N)* te ćemo njega opteretiti sortiranjem po x koordinati, bez ugrožavanja performansi rezultata.

Slika pokazuje mali skup tačaka P. Pošto su tačke sortirane po x koordinati možemo iscrtati zamišljenu vertikalnu liniju koja deli skup tačaka u dve polovine P_L i P_R . Sada imamo skoro istovetnu situaciju kao u problemu sekvence sa maksimalnom sumom.



Dalje zaključujemo da ili su obe tačke u P_L ili u P_R ili je jedna u P_L , ajedna u P_R. Označimo te distance sa d_L d_R i d_C. Slika takodje pokazuje pokazuje te distance.

Možemo rekurzivno da izračunamo d_L i d_R. Da bi rešenje ostalo u okviru *O(N log N)* d_C treba izračunati u okviru O(N). Neka je $\delta = \min(d_L, d_R)$.




```
for( i = 0; i < numPtInStrip;  i++ )
   for( j = i + 1; j < numPtInStrip;  j++ )
      if(  dist( p[i], p[j] ) > delta )
         break;    // sledece je p[i+1]
      else
         if( dist( p[i], p[j] ) < delta )
            delta = dist( p[i], p[j] );
```

Odrediti koje su tačke u traci i naćo minimalno rastojanje medju njima je posao koji treba efikasno rešiti. Ako su tačke uniformno distribuirane dovoljno je za njih pozvati pretraživanje svih rastojanja. Ako ovo nije slučaj moramo da pribegnemo dodatnoj organizaciji ulaznih podataka. Naime, već smo uradili sortiranje po x, koordinati, a potrebno je da uradimo i sortiranje po y koordinati u niyz Q. Tako ćemo da kad formiramo levu i desnu polovinu P_L i P_R da jednim prolazom, dakle linearno formiramo i Q_L i Q_R koje su već sortirane po y koordinati i oba podniza ćemo prosledjivati u rekurzivne pozive. Na ovaj način ostajemo u okviru $O(N \log N)$ granice.

Dinamičko programiranje

Takmičenja

Poslednjih godina se često na takmičenjima iz informatike, prvo medjunarodnim, a zatim i domaćim javljaju zadaci iz ove oblasti, te se njeno poznavanje smatra potrebnim, čak neophodnim za ostvarenje dobrih rezultata. Rešenja zadataka se vrednuju na osnovu uspešnosti rešavanja test primera kojih ima više i graduirani su po težini od trivijalnih do kompleksnih. Može se zadati problem za čije potpuno i efikasno rešavanje je potrebno izvesno teorijsko predznanje, ili dobra ideja. Učenici koji su ovladali tim znanjem (odnosno imaju dobru ideju), mogu da reše zadatak u potpunosti i na efikasan način i time da osvoje sve poene. Ostali učenici mogu da smisle na licu mesta neki postupak rešavanja. Takav postupak često nije potpuno korektan, pa na nekim primerima nosi poene, dok na drugim primerima poena nema. Takodje se može dogoditi da korišćeni algoritam teorijski jeste korektan, ali nije dovoljno efikasan (brz), pa takmičar koji reši zadatak na ovakav način može dobiti poene na testovima sa malim ulaznim podacima, medjutim ostaje bez poena na većim primerima, jer program ne završava rad u predvidjenom vremenu. Zadaci su praćeni pažljivim izborom testova, koji proveravaju potpunost ideje, kao i vremensku i prostornu efikasnost rešenja. Jedan tip problema koji izuzetno dobro odgovara ovim zahtevima su problemi koji se rešavaju dinamičkim programiranjem.

Opis problema i pristupa rešavanju

Dinamičko programiranje se obično primenjuje u problemima optimizacije: problem može imati mnogo rešenja, svako rešenje ima vrednost, a traži se rešenje koje ima optimalnu (najveću ili najmanju) vrednost. U slučaju da postoji više rešenja koja imaju optimalnu vrednost obično se traži bilo koje od njih. Sama reč “programiranje” ovde se kao i u linearnom programiranju odnosi na popunjavanje tabele pri rešavanju problema, a ne na upotrebu kompjutera i programskih jezika. Tvorcem se smatra profesor Ričard E. Belman koji je sredinom pedesetih godina proučavao dinamičko programiranje i dao čvrstu matematičku osnovu za ovaj način rešavanja problema. Uopšteno govoreći, problem se rešava tako što se uoči hijerarhija problema istog tipa, sadržanih u glavnom problemu, i rešavanje se počne od najjednostavnijih problema. Vrednosti i delovi rešenja svih rešenih podproblema se pamte u tabeli, pa se dalje njihovim kombinovanjem dobijaju rešenja većih podproblema sve do rešenja glavnog problema.

Problem ranca – formulacija i ad hoc pristupi

Ilustrujmo ideju na jednom od najpoznatijih problema dinamičkog programiranja – problemu ranca (knapsack problem). Ima nekoliko poznatih varijanti, a svaka se može sresti u više formulacija: Provalnik sa rancem u koji može da stane N zapreminskih jedinica, upao je u prostoriju u kojoj se čuvaju vredni predmeti. U prostoriji ima ukupno M tipova predmeta, svakog u vrlo velikoj količini (više no što može da stane u ranac). Za svaki tip predmeta poznata je njegova vrednost $V(k)$ i njegova zapremina $Z(k)$ $k = 1,M$. Sve navedene veličine su celobrojne. Probvalnik želi da napuni ranac najvrednijim sadržajem. Potrebno je odrediti predmete koje treba staviti u ranac i njihovu zbirnu vrednost.

Napomene:

- Ranac ne mora biti napunjen do vrha, tj zbir zapremina predmeta stavljenih u ranac ne mora biti jednak zapremini ranca. Bitno je da taj zbir nije veći od zapremine ranca, a da u isto vreme zbir vrednosti tih predmeta bude maksimalan. Na primer za N=7, M=3, V=(3, 4, 8), Z=(3, 4, 5), ranac se može popuniti do vrha stavljajući prvi i drugi predmet, jer je $Z_1 + Z_2 = 3 + 4 = 7 = N$. Vrednost u rancu je 3 + 4 = 7, što nije optimalno jer postoji bolje: stavljanjem trećeg predmeta dobijamo ranac vrednijeg sadržaja $V_3 = 8$, a pri tome $Z_3 = 5 < 7$.
- Na osnovu prethodnog primera sti;e se utisak da se do rešenja dolazi tako što se odredi k za koje je $V(k) / Z(k)$ najveće pa se ranac popunjava samo predmetom k. ovaj pristup nije rešenje, što ćemo takođe pokazati na primeru: $N = 7$, $M = 3$, V= (3, 4, 6), Z = (3, 4, 5). Navedena ideja (garbljjivi izbor) nalaže da se odabere treći predmet kao najvredniji po jedinici zapremine. Time bi u ranac bio stavljen samo jedan od predmeta trećeg tipa, a vrednost plena bi bila jednaka 6. Lako je videti da izbor po jednog predmeta tipa 1 i 2 daje plen vrednosti 7, što je bolje (i optimalno rešenje). Napomenimo da je logika grabljivog izbora ispravna ako ne moraju da se uzimaju celi predmeti i da je vrednost dela predmeta srazmerna njegovoj veličini.

Dakle, potrebna je pažljivija analiza svojstava problema i rešenja.

Analiza problema ranca

Analizirajući problem uočavamo: ako je pri optimalnom popunjavanju ranca izabrani predmet x, onda prethodno izabrani predmeti na optimalan način popunjavaju ranac kapaciteta N-z. Svodjenjem na kontadikciju ova se tvrdnja lako dokazuje. Dakle:

Optimalno rešenje problema sadrži u sebi optimalna rešenja podproblema istog tipa sadržanih u glavnom problemu.

Navedeno svojstvo naziva se svojstvom Belmana, po autoru metoda dinamičkog programiranja.

Podsetimo se matematičke indukcije i pokažimo kako za svako celobrojno N i date tipove predmeta, umemo da nadjemo najbolje popunjavanje ranca kapaciteta N. Za N=0 rešenje je prazan ranac. Neka su poznata rešenja za sve rance kapaciteta q manjeg od N i neka je $B(q)$ zbirna vrednost svih predmeta, a $S(q)$ niz rednih brojeva predmeta stavljenih u ranac kapaciteta q pri optimalnom izboru. Svaki od M predmeta koji može da stane u prazan ranac kapaciteta N, isprobamo kao poslednji izabrani za ranac

kapaciteta *N*. Ostatak ranca u svakom od ovih slučajeva popunimo na optimalan način, što na osnovu induktivne hipoteze umemo. Najveća od svih dobijenih vrednosti ranca kapaciteta *N* biće optimalna. Ova činjenica sledi direktno na osnovu optimalnosti podstrukture, jer jedan predmet mora biti poslednji, a mi smo svaki isprobali kao poslednji. Prema svemu rečenom, rešenje za ranac kapaciteta *N* je:

$$B(N) = max\{V(x) + B(N - Z(x))\}, \quad S(N) = S(N - Z(X_N)) \cup \{X_N\}$$

gde je X_N ono x za koje ostvaruje maksimum u *B(N)*. Primetimo da ne moramo pamtiti ceo skup *S(q)* za svaki kapacitet ranca *q,* dovoljno je kao član niza *C(q)* zapamtiti poslednji dodati element X_N. Svi elementi se tada mogu naći redom (od poslednjeg ka prvom) i to su:

$$a=C(N), \quad b = C(N-Z(a)), \quad c = C(N - Z(a) - Z(b)), \quad d = C(N - Z(a) - Z(b) - Z(c)), \quad ...$$

Rekurzivno rešenje

Zadatak se može rešiti upotrebom rekurzivne procedure napuni(q, B, C), koja za ranac kapacietta *q* nalazi njegovu optimalnu vrednost *B* i redni broj poslednjeg dodatog predmeta *C*. Smatraćemo da su nizovi *V* i *Z,* kao i broj tipova predmeta *M* globalne veličine.

```
procedure napuni( q: integer, var B: integer, var C: integer);
var BB, CC : integer;
begin
    B := 0;
    C := 0;
    if( q > 0 ) then
    begin
        for i := 1 to M do
            if (Z[i] <= q) then
            begin
               napuni( q – Z[i], BB, CC);
                  if( BB + V[i] > B) then
                  begin
                   B := BB + V[i];
                   C := i;
                  end;
        end;
    end;
end;
```

Iz glavnog programa bi se pozivala procedura napuni(N, B, C), a svaki poziv ove procedure može prouzrokovati *M* novih poziva radi rešavanja generisanih podproblema. Na primer za *N*=1000, *M*=10 i predmete zapremine Z_{min}= 5 do Z_{max}=10, bilo bi izmedju $M^{N/Zmax} = 10^{100}$ i $M^{N/Zmin} = 10^{200}$ rekurzivnih poziva procedure napuni i to samo na poslednjem nivou dubine rekurzije (listova u drvetu rekurzivnih poziva). Ovo je previše i za starost vasione.

Nerekurzivno rešenje

Umesto rekurzivno, probleme možemo rešavati i i redom od q = 1 do q =N, popunjavajući pri tome nizove (tabele) B i C. Za svako q potrebno je M prolazaka kroz ciklus, da se odredi poslednji izabrani element i najveća vrednost, što znači da je ukupan broj operacija proporcionalan NM. U primeru bi to bilo 10000 ciklusa sa po nekoliko računskih operacija što je manje od jedne sekunde.

```
program ranac;
var
    B, C : array[0..1000] of longint;
    V, Z : array[1..10] of integer;
    M, N, q, i : integer;
begin
    readln(N, M);
    for i := 1 to M do readln( V[i], Z[i] );
   B[0] := 0;
   C[0] := 0;
   for q := 1 to N do
   begin
    B[q] := 0; C[q] := 0;
       for i := 1 to M do
    if( Z[i] <= q ) then
        if( B[q] – Z[i]] + V[i] > B[q] ) then
        begin
            B[q] := B[q – Z[i]] + V[i];
            C[q] := i;
        end;
   end;

   writeln(' Optimalna vrednost ranca je ', B[N] );
   writeln(' Izabrani predmeti su:');
         { ispis unazad, ali nebitno}
    q := N;
    while( C[q] > 0 ) do
    begin
        writeln( C[q] );
        q := q – Z[C[q]];
    end;
end.
```

Poredjenje efikasnosti rešenja

Postavlja se pitanje odakle ovakva drastična razlika u efikasnosti navedenih rešenja.
Kod rekurzivnog rešenja se problem ranca za kapaciteta q < N uvek iznova rešava, pri čemu broj ponovljenih rešenja raste ekponencijalno sa povećanjem dimenzije polaznog problema. Podproblemi se delimično preklapaju i to je druga osobina bitna za primenu dinamičkog programiranja. Ova osobina nije neophodna da bi se primenilo dinamičko programiranje, ali ako je problem nema onda ovaj način gubi svoju veliku rednost u brzini u odnosu na rekurziju, jer se tada i rekurzijom svaki podproblem kreira i rešava samo jednom.

Varijante osnovnog problema

Problem ranca se javlja u nekoliko različitih varijanti. Problem se može postaviti i tako da se za svaki predmet zada broj raspoloživih primeraka, ili tako da je od svake vrste predmeta na raspolaganju po tačno jedan primerak. Za ovu drugu varijantu gde svaki predmet učestvuje najviše jednom analiza problema dovodi do sledećeg:

Neka je poznato optimalno popunjavanje ranca veličine *N*. Ako u tom popunjavanju ne učestvuje M-ti predmet, onda je isto popunjavanje optimalno i za ranac veličine *N* i prvih *M-1* predmeta. Ako u optimalnom popunjavanju učestvuje i m-ti predmet, onda ostali predmeti iz ranca čine optimalno popunjavanje za ranac veličine *N-Z(M)* i prvih *M-1* predmeta. Označimo sa *B(X,Y)* najveću vrednost ranca kapaciteta *X*, popunjavanog nekima od prvih *Y* predmeta. Tada je:

$$B(X,0) = 0$$

$$B(X,Y) = \begin{cases} B(X,Y-1, & Z(Y) > X \\ \max\begin{cases} B(X,Y-1) \\ V(Y)+B(X-Z(Y),Y-1 \end{cases} & Z(Y) < X \end{cases}$$

Prema tome podproblemi se mogu rešavati sledećim redom: najpre za sve kapacitete rančeva i jedan predmet, pa za sve kapacitete rančeva i dva predmeta, itd..Nakon što rešimo *B(N,M)* imamo rešenje polaznog problema.
Ovde je kao i u prethodnoj varijanti radi rekonstrukcije rešenja dovoljan dodatni niz *C*, gde ćemo pamtiti poslednji stavljeni predmet pri optimalnom popunjavanju ranca kapaciteta *X*. Pri popunjavanju matrice *B* po kolonama koriste se samo vrednosti iz prethodne kolone *(Y-1).* Zahvaljujući tome, možemo umesto matrice *B* sa *M* kolona, koristiti matricu sa samo dve kolone, što je značajna ušteda prostora, koja može presudno uticati na primenljivost postupka (za neke vrednosti *M* i *N*). Posmatrajući još pažljivije relaciju po kojoj se računa *B(X,Y),* vidimo da potrebni elementi pethodne kolone svi imaju redni broj maji ili jednak *X*. Stoga pri popunjavanju z_te kolone matrice *B* unazad (od poslednje vrste ka prvoj) možemosve operacije izvesti u istoj koloni, pa je za čuvanje potrebnih podataka dovoljan niz *B* (ako se koristi na opisani način). Program:

```
program ranac2;
var
    B, C : array[0..1000] of longint;
    V, Z : array[1..10] of integer;
    N, M, x, y : integer;
begin
    write(' n,m =?'); readln(N,M);
   for y := 1 to M do readln(v[y], z[y]);
   for( x := 0 to N do
   begin
       B[x] := 0;
       C[x] := 0;
   end;
   for y := 1 to M do
    for x := N downto 1 do
       if( Z[y] <= x) then
       begin
          B[x] := V[y] + B[x – Z[y]];
          C[x] := y;
       end;
   writeln('Optomalna vrednost sadzaja ranca je ',B[N]);
   writeln('Izabrani predmeti su: ');
       { Ispisuje spisak unazad }
   x := N;
   while( C[x] > 0 ) do
   begin
    writeln(C[x]);
    x := x – Z[C[x]];
   end;
end.
```

Pri rešavanju ove varijante problema (svaki predmet najviše jednom) treba imati na umu još jednu važnu činjenicu: rešavanje problema dinamičkim programiranjem se isplati jedino ako je broj predmeta M dovoljno veliki, a kapacitet ranca N relativno mali, tako da se NM operacija (koliko je približno potrebno za ovaj način rešavanja) izvrši brže nego nalaženje svih 2^M mogućih podnizova i njihovih zbirova vrednosti, odnosno zapremina.

Na takmičenjima se javljaju razne varijante problema ranca, uz usložnjavanja. Pomenimo neke:

1. Iz datog niza prirodnih brojeva A, izdvojjti podniz čiji je zbir elemenata dati broj M, ili ispisati poruku da takav podniz ne postoji.
   Označimo sa S sumu, a sa N broj elemenata niza A. Možemo pretpostaviti da su nizom A zadati takvi predmeti kojima *je* $V_i = Z_i = A_i$, $i = 1,N$. Sada je jasno da se problem svodi na optimalno popunjavanje ranca kapaciteta N, pri čemu rešenja ima ako i samo ako je vrednost optimalnog sadržaja ranca jednaka njegovoj zapremini, tj ako je ranac napunjen do vrha.

2. Brojeve iz datog niza prirodnih brojeva podeliti u dve grupe, tako da je razlika zbirova elemenata u pojedinim grupama minimalna.
   U ovom primeru niz *A* ponovi ima ulogu i niza *Z* i niza *V*. Neka *S* i *N* imaju ista unačenja kao i u prethodnom primeru. Rešenje problema sastoji se u sledećem: Predmete koji čine optimalno popunjavanje ranca kapaciteta *S//2* treba svrstati u jednu, a sve ostale predmete u drugu grupu. Ako je ranac popunjen do vrha, grupe će za parno *S* imati jednake zbirove. U suprotnom prva grupa ima manji zbir, a druga veći, ali ti zbirovi su najbliži vrednosi *S/2*, a time i jedan drugom.