Oblikovanje algoritama

- Iskustveno je nađeno nekoliko općenitih strategija za oblikovanje algoritama
- Razmotrit ćemo, na nekoliko primjera, najvažnije strategije:
  - Podijeli-pa-vladaj
  - Dinamičko programiranje
  - Pohlepni pristup
  - Backtracking (strategija odustajanja/povlačenja)
- Postupak traženja algoritma za rješavanje novog problema: ispitati kakav algoritam daju najčešće strategije i onda ih usporediti
- Nema garancije da će neka strategija dati točno rješenje
- Rješenje može biti i samo približno, nema garancije da je egzaktno
- Dobiveni algoritam ne mora biti ni efikasan
- Ove strategije su se pokazale uspješne za mnoge probleme, pa je to razlog zašto se najčešće koriste u rješavanju novih problema

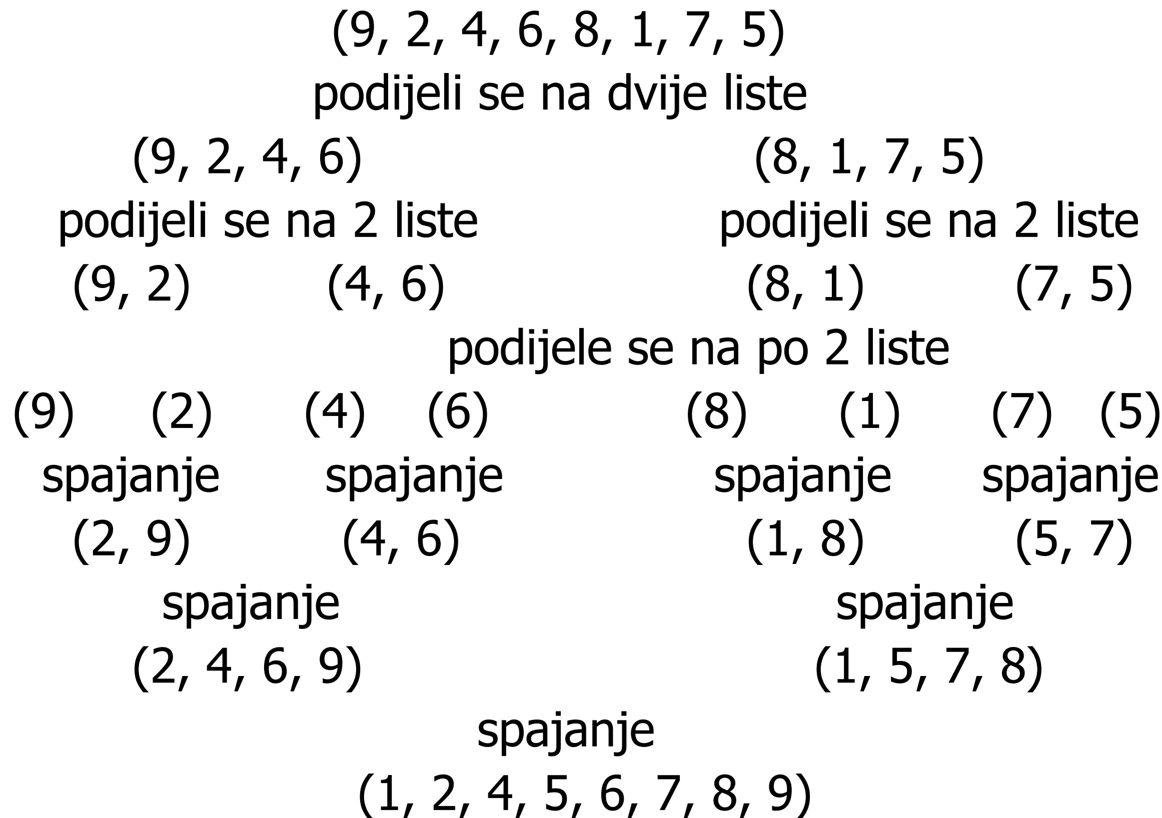
Strategija podijeli-pa-vladaj

- Najprimjenljivija strategija za oblikovanje algoritama
- Ideja je da se zadani problem razbije u nekoliko manjih istovrsnih problema, tako da se rješenje polaznog problema može relativno lako konstruirati iz rješenja manjih problema
- Dobiveni algoritam je rekurzivan, jer se svaki od manjih problema i dalje razbija u još manje probleme
- Nužna pretpostavka je da postoji direktno rješenje za dovoljno mali problem, tj. mora postojati neka relativno jednostavna tehnika rješavanja dovoljno malog problema
- Ovu strategiju smo već koristili u rješavanjima raznih problema u ovom kolegiju (npr. binarna stabla, hrpa itd)

Sortiranje spajanjem

- Objašnjenje algoritma sortiranja spajanjem (Merge Sort):
  - što je lista dulja, teže ju je sortirati, pa se zato početna lista razbije u dvije manje i svaka od njih se sortira zasebno
  - velika sortirana lista se dobiva relativno jednostavnim postupkom spajanja dvije male sortirane liste

Primjer:



- Algoritam sortiranja spajanjem:

  ulaz je lista $a_1, \ldots, a_n$, izlaz sortirana lista

  1. podijeli listu na dva jednaka dijela
  2. sortiraj listu $a_1, \ldots, a_{n/2}$
  3. sortiraj listu $a_{n/2+1}, \ldots, a_n$
  4. spoji liste $a_1, \ldots, a_{n/2}$ i $a_{n/2+1}, \ldots, a_n$

- Algoritam spajanja:

  ulaz su dvije sortirane liste $b_1, \ldots, b_k$ i $c_1, \ldots, c_l$, izlaz je sortirana lista $a_1, \ldots, a_{k+l}$

  1. $i = 1, j = 1$
  2. ponavljaj korak 3 sve dok je $i \leq k$ i $j \leq l$
  3. ako je $b_i < c_j$ onda $a_{i+j-1} = b_i$, $i=i+1$, inače $a_{i+j-1} = c_j$, $j=j+1$
  4. ponavljaj korak 5 sve dok je $i \leq k$
  5. $a_{i+j-1} = b_i$, $i=i+1$
  6. ponavljaj korak 7 sve dok je $j \leq l$
  7. $a_{i+j-1} = c_j$, $j=j+1$

- Složenost ovog algoritma je $O(n \lg n)$
- Nedostatak: potrebno pomoćno polje

```
// MergeSort – sortiranje spajanjem
void MergeSort (tip A [], int N) {
  tip *PomPolje;
  PomPolje = malloc (N * sizeof (tip));
  if (PomPolje != NULL) {
    MSort (A, PomPolje, 0, N - 1);
    free (PomPolje);
  } else
        printf ("Nema mjesta za PomPolje!\n");
}

// MergeSort - rekurzivno sortiranje podpolja
void MSort (tip A [], tip PomPolje[], int lijevo, int desno ) {
  int sredina;
  if (lijevo < desno) {
    sredina = (lijevo + desno) / 2;
    MSort (A, PomPolje, lijevo, sredina);
    MSort (A, PomPolje, sredina + 1, desno);
    Merge (A, PomPolje, lijevo, sredina + 1, desno);
  }
}
```

```
// udruzivanje LPoz:LijeviKraj i DPoz:DesniKraj
void Merge (tip A [], tip PomPolje [], int LPoz, int DPoz, int DesniKraj) {
  int i, LijeviKraj, BrojClanova, PomPoz;
  LijeviKraj = DPoz - 1;
  PomPoz = LPoz;
  BrojClanova = DesniKraj - LPoz + 1;
  // glavna pelja
  while (LPoz <= LijeviKraj && DPoz <= DesniKraj) {
    if (A [LPoz] <= A [DPoz])
      PomPolje [PomPoz++] = A [LPoz++];
    else
                        PomPolje [PomPoz++] = A [DPoz++];
  }
  while (LPoz <= LijeviKraj)
    // Kopiraj ostatak prve polovice
    PomPolje [PomPoz++] = A [LPoz++];
  while (DPoz <= DesniKraj)
    // Kopiraj ostatak druge polovice
    PomPolje [PomPoz++] = A [DPoz++];
  for (i = 0; i < BrojClanova; i++, DesniKraj--)
    // Kopiraj PomPolje natrag
    A [DesniKraj] = PomPolje [DesniKraj];
}
```

Traženje elementa u listi

- Problem: utvrditi da li u zadanoj listi postoji element s zadanom vrijednošću
- Strategija podijeli-pa-vladaj vodi na sljedeći algoritam:

  - što je lista dulja, teže ju je pretraživati, pa se početna lista podijeli u npr. 3 manje liste i u svakoj se zasebno traži zadana vrijednost
  - tražena vrijednost se javlja u početnoj listi ako se javlja u bar jednoj manjoj listi, znači konačno rješenje se dobije primjenom logičke "ili" operacije na parcijalna rješenja

Primjer: 4 ? (8, 3, 1, 4, 5, 2, 9, 6, 4, 3, 5)

podijeli se na tri liste

(8, 3, 1, 4) (5, 2, 9) (6, 4, 3, 5)

pretraživanje listi

DA NE DA

logička operacija "ili"

DA

- Ne dovodi do efikasnijeg algoritma

- Ako je početna list sortirana, ovaj algoritam se može bitno pojednostaviti:
  - podjela na manje liste se izvodi tako da srednja lista ima samo jedan element, a preostale dvije su podjednake duljine
  - jedna operacija uspoređivanja (traženi element s vrijednošću u sredini) omogućuje da se vrijednost odmah pronađe ili se odbace dvije liste
- To je algoritam binarnog pretraživanja

Primjer: 4 ? (1, 3, 3, 4, 6, 6, 7, 8, 9, 9)

podijeli se na tri liste

(1, 3, 3, 4, 6) (6) (7, 8, 9, 9)

uspoređivanje

? NE NE

podijeli se na tri liste

(1, 3) (3) (4, 6)

uspoređivanje

NE NE ?

podijeli se na tri liste

(4) (4) (6)

DA

- Algoritam binarnog pretraživanja:

  ulaz je kursor f na početak liste, kursor l na kraj liste, tražena vrijednost x, lista $a_1, \ldots,$ a_n, izlaz je DA ako je x u listi, inače NE

  1. p = (f + l)/2
  2. ako je $a_p = x$ vrati DA i stani
  3. ako je $a_p > x$ i $p > f$ onda ponovno na korak 1 za kursore f, l = p-1
  4. ako je $a_p < x$ i $p < l$ onda ponovno na korak 1 za kursore f = p+1, l
  5. vrati NE i stani

- **Složenost algoritma**: duljina liste koja se promatra n
- **Najgori slučaj**: x se ne nalazi u listi

izvršavaju se koraci 1, 2, 3 i 4 koji se svi obavljaju u konstantnom vremenu:

$$T_{max}(n) \le c_1 + T_{max}(n/2)$$

potrebno je riješiti rekurziju

$$t_n = t_{n/2} + c_1$$

To je **podijeli-pa-vladaj rekurzija**

- Rješava se uvođenjem supstitucije $a_n = t_{2^n}$ što vodi na izraz

$$a_n = t_{2^n} = t_{2^n/2} + c_1 = t_{2^{n-1}} + c_1 = a_{n-1} + c_1$$

$$a_n = a_{n-1} + c_1$$
$$a_{n-1} = a_{n-2} + c_1$$
---------------------- / oduzimanje
$$a_n - a_{n-1} = a_{n-1} - a_{n-2}$$

Što daje homogenu rekurziju

$$a_n = 2\, a_{n-1} - a_{n-2}$$

Rješavanje rekurzija – karakteristična jednadžba:

Definicija: $a_n = c_{k-1}\, a_{n-1} + \ldots + c_1\, a_{n-k+1} + c_0\, a_{n-k}$ se zove linearna homogena rekurzivna jednadžba k-tog reda s konstantnim koeficijentima

$x^k - c_{k-1}\, x^{k-1} + \ldots + c_1\, x + c_0 = 0$ se zove karakteristična jednadžba linearne homogene rekurzivne jednadžbe.

Teorem: ako ova jdba ima sve jednostruke i realne korijene $x_1, \ldots, x_k$ tada je opće rješenje linearne homogene rekurzivne jdbe oblika

$$a_n = C_1\, x_1^n + \ldots + C_k\, x_k^n$$

Ukoliko karakteristična jdba ima višestruke korijene, npr. x je njen t-struki korijen, tada x tvori t pribrojnika općeg rješenja rekurzije $C_1 x^n$, $C_2 n x^n$, …, $C_t n^{t-1} x^n$

- Rekurzivna formula koju rješavamo je: $a_n = 2 a_{n-1} - a_{n-2}$
- Karakteristična jdba je: $x^2 - 2x + 1 = 0$, njen dvostruki korijen je 1, pa je opće rješenje rekurzije

$$a_n = C_1 1^n + C_2 n 1^n$$

$$t_{2^n} = C_1 + C_2 n \quad \text{iz čega slijedi}$$

$$T_{max}(n) = t_n = C_1 + C_2 \log_2 n$$

$$T_{max}(n) = O(\lg n)$$

- **Prosječan slučaj**, kada je x u listi: lista ima n elemenata, možemo uzeti (bez smanjenja općenitosti) da je $n = 2^k - 1$ (time u svakom koraku lista ima srednji element)
- U 1. koraku se provjeri srednji element, u 2. koraku će algoritam stati ako je traženi element na pozicijama ¼ ili ¾ duljine liste, u 3. koraku se nalaze 4 elementa, … u zadnjem, k-tom koraku se provjeri 2^{k-1} elemenata, svaka od operacija traje neko konstantno vrijeme

- Po definiciji, matematičko očekivanje za slučajnu varijablu X(q) čija je vjerojatnost P({q}) je:

$$EX = \sum_{q} X(q)P(\{q\})$$

Pa je matematičko očekivanje za broj koraka potrebnih da nađemo traženi broj

$$EX = \sum_{i=1}^{k} i\, 2^{i-1}/(2^{k} -1)$$

A prosječno vrijeme izvršavanja algoritma je:

$$T_{pros}(k) = c_1 \,/\, (2^{k} -1) \sum_{i=1}^{k} i\, 2^{i-1}$$

Ovaj red se može riješiti znajući da je

$$\sum_{i=1}^{k} x^{i} = (x^{k+1} -1) \,/\, (x -1)$$

deriviranjem gornjeg reda izlazi

$$\sum_{i=1}^{k} i\, x^{i-1} = (k\, x^{k+1} - (k+1)\, x^{k} + 1) / (x - 1)^2$$

Uvrštavanjem slijedi:

$$T_{pros}(k) = c_1\, ((k - 1)\, 2^k + 1) / (2^k - 1)$$
$$T_{pros}(k) = c_1\, [(k - 1) + k / (2^k - 1)]$$
$$T_{pros}(n) = c_1\, [\lg(n - 1) + \lg n / (n - 1)]$$

Što konačno daje:

$$T_{pros}(n) = \Theta(\lg n)$$

Najbolji slučaj: kada je traženi element u sredini i nađe se u prvom koraku

$$T_{min}(n) = \Theta(1)$$

Množenje dugačkih cijelih brojeva

- Problem množenja dvaju n-bitnih cijelih brojeva X i Y
- Klasični algoritam: množenje X sa svakom znamenkom iz Y i zbrajanje parcijalnih rezultata, zahtijeva računanje n produkata veličine n, pa je složenost algoritma $O(n^2)$ (osnovna operacija je s 1 bitom)

- Primjer:

```
  11010010 * 10001101          210 * 141
 ------------------------------
  11010010
      11010010
       11010010
+        11010010
 ----------------------------
   111001110101010             29610
```

- Strategija podijeli-pa-vladaj vodi na algoritam:

  - svaki od brojeva X i Y se podijeli na dva dijela koji su duljine n/2 bitova, zbog jednostavnosti se uzima da je n potencija od 2

X ->	A | B	$X = A\ 2^{n/2} + B$
Y ->	C | D	$Y = C\ 2^{n/2} + D$

  - produkt je tada

$$X * Y = A * C * 2^{n} + (A * D + B * C) * 2^{n/2} + B * D$$

- Ovim algoritmom je jedno množenje velikih n-bitnih brojeva zamijenjeno s četiri množenja malih n/2-bitnih brojeva, tri zbrajanja brojeva duljine najviše 2n bitova te dva pomicanja bitova (množenje s 2^n i $2^{n/2}$)
- Zbrajanje i pomicanje bitova zahtijevaju O(n) koraka, pa je ukupan broj koraka ovog algoritma (opet se javlja rekurzija):

$$T(1) = 1 ; \qquad T(n) = 4\ T(n/2) + c\ n$$

Rješavamo rekurziju:

$$a_n = 4\ a_{n/2} + c\ n$$

Uvodi se supstitucija: $s_n = a_{2^n}$

$$s_n = a_{2^n} = 4\ a_{2^{n-1}} + c\ 2^n$$

$$s_n = 4\, s_{n-1} + c\, 2^n$$

Tražimo opći član rekurzije:

$$s_n = 4\, s_{n-1} + c\, 2^n$$

$$s_{n-1} = 4\, s_{n-2} + c\, 2^{n-1} \quad /*2$$

--------------------------------- /oduzimanje

$$s_n = 6\, s_{n-1} - 8\, s_{n-2}$$

Karakteristična jdba: $x^2 - 6x + 8 = 0$ ima korijene $x_1 = 4$, $x_2 = 2$ pa je rješenje

$$s_n = C_1\, 4^n + C_2\, 2^n$$

Konstante se mogu izračunati iz početnih uvjeta:

$$s_0 = a_1 = C_1 + C_2 = 1$$

$$s_1 = a_2 = 4\, C_1 + 2\, C_2 = 4 + 2c$$

što daje: $C_2 = -c\ ,\ C_1 = 1 + c$

uvrštavanjem: $s_n = (1 + c)\, 4^n - c\, 2^n = (1 + c)\, 2^{2n} - c\, 2^n$

vraćanjem iz supstitucije

$$a_n = (1 + c)\, n^2 - c\, n$$

Pa je dakle

$$T(n) = O(n^2)$$

strategija podijeli-pa-vladaj daje algoritam iste složenosti kao klasični algoritam !

- Moguće je postići poboljšanje strategijom podijeli-pa-vladaj ako se izraz

$$X * Y = A * C * 2^n + (A * D + B * C) * 2^{n/2} + B * D$$

preinači u

$$X * Y = A*C * 2^n + [(A–B) * (D-C) + A*C + B*D] * 2^{n/2} + B*D$$

Ovdje se javljaju samo tri množenja n/2-bitnih brojeva, te 6 zbrajanja/oduzimanja i dva pomicanja bitova (ima više zbrajanja, ali njihova složenost je linearna)

- Rekurzija za T(n) je sada $T(n) = 3\ T(n/2) + k\ n$, početni uvjet $T(1) = 1$
- Postupak rješavanja je identičan prethodnom:

$a_n = 3\ a_{n/2} + k\ n$, supstitucija $s_n = a_{2^n}$ daje izraz

$$s_n = 3\ s_{n-1} + k\ 2^n$$

Oduzimanjem izraza za s_{n-1} pomnoženim s 2 od gornjeg izraza i sređivanjem slijedi

$$s_n = 5\ s_{n-1} - 6\ s_{n-2}$$

Karakteristična jdba $x^2 – 5\ x + 6 = 0$ ima korijene $x_1 = 3$ i $x_2 = 2$, pa je rješenje

$$s_n = K_1\ 3^n + K_2\ 2^n$$

Konstante se mogu izračunati iz početnih uvjeta:

$$s_0 = a_1 = K_1 + K_2 = 1$$

$$s_1 = a_2 = 3\ K_1 + 2\ K_2 = 3 + 2k$$

što daje: $K_2 = -2\ k$, $K_1 = 1 + 2\ k$

$$s_n = (1 + 2k)\, 3^n - 2k\, 2^n$$

upotrebom izraza $3^n = 2^{n * \log_2 3}$

$$s_n = (1 + 2k)\, (2^n)^{\log_2 3} - 2k\, 2^n$$

Vraćanjem iz supstitucije $s_n = a_{2^n}$ dobije se konačan izraz za opći član rekurzije

$$a_n = (1 + 2k)\, n^{\log_2 3} - 2\, k\, n$$

Pa je složenost algoritma

$$T(n) = O(n^{\log_2 3}) = O(n^{1.59})$$

- Poboljšana verzija algoritma je asimptotski brža od klasičnog algoritma, no ubrzanje se ostvaruje kod velikih vrijednosti n (par stotina bitova)
- Za ilustraciju prikazat će se pseudokod koji koristi ovaj algoritam, za slučaj dugih cijelih brojeva u jeziku C, tip long long (što znači da zapravo vrijedi samo za male n, do 64 bita)
- Ispravna izvedba za zaista duge cijele brojeve zahtjeva razvijanje posebnog načina pohrane dugih brojeva (dugačko polje bitova), ta izvedbu koda za aritmetičke operacije zbrajanja i oduzimanja i operaciju pomicanja bitova za takve brojeve

```
long long mult (long long X, long long Y, int n) {
 /* X i Y mogu imati predznak; pretpostavlja se da je n potencija od 2 */
 int s;  /* predznak produkta */
 long long m1, m2 m3;
 long long A, B, C, D; /* polovice od X i Y */
 s = (X >= 0 ? 1 : -1) * (Y >= 0 ? 1 : -1);
 X = abs(X); Y = abs(Y);
 if ( n == 1)
    if ( (X == 1) && (Y == 1) ) return s;
    else return 0;
 else {
      A = lijevih n/2 bitova od X; B = desnih n/2 bitova od X;
      C = lijevih n/2 bitova od Y; D = desnih n/2 bitova od Y;
      m1 = mult(A, C, n/2);
      m2 = mult(A-B, D-C, n/2);
      m3= mult(B, D, n/2);
      return (s*(m1<<n + (m1+m2+m3)<<(n/2) + m3);
    /* operator << pomiče za n bitova ulijevo */
 }
}
```

Završna napomena

- Za dobar i efikasan algoritam dobiven strategijom podijeli-pa-vladaj jako je važno da podproblemi budu sličnih veličina
- Npr. sortiranje umetanjem se također može smatrati algoritmom tipa podijeli-pa-vladaj u kojem je polazna lista duljine n podijeljena u dvije liste nejednake duljine 1 i n-1; ovdje je vrijeme izvršavanja dano s $O(n^2)$
- Sortiranje spajanjem je algoritam tipa podijeli-pa-vladaj u kojem je polazna lista podijeljena u dvije liste jednake duljine pa je vrijeme izvršavanja ovdje $O(n \lg n)$
- Uspješan algoritam tipa podijeli-pa-vladaj treba voditi na vrijeme izvršavanja proporcionalno s lg n

Strategija dinamičkog programiranja

- Strategija podijeli-pa-vladaj može proizvesti neefikasan algoritam (npr. računanje Fibonaccijevih brojeva)
- Broj podproblema koje treba riješiti može rasti eksponencijalno s veličinom zadanog problema, isti podproblem se mora rješavati mnogo puta (rekurzijom)
- Kod takvih problema bolja je strategija dinamičkog programiranja, gdje se svi podproblemi redom riješe i rješenja se spreme u odgovarajuću tabelu; kad je potrebno neko rješenje uzima se iz tabele, a ne računa se ponovno
- Naziv dinamičko programiranje potječe iz teorije upravljanja i izgubio je svoje prvobitno značenje
- Algoritmi tipa podijeli-pa-vladaj idu od vrha prema dolje, a algoritmi dinamičkog programiranja od dna prema gore
- Prvo se rješavaju problemi najmanje veličine, pa nešto veći, itd. sve dok se ne dosegne potpuna veličina zadanog problema; ovdje je važan redoslijed ispunjavanja tabele
- Ovakva strategija ponekad zahtjeva i rješavanje podproblema koji nisu potrebni za rješavanje zadanog problema, no to je još uvijek isplatljivije od rješavanja istih podproblema mnogo puta

Računanje Fibonaccijevih brojeva

- Rekurzija za računanje: $F_0 = F_1 = 1$; $F_n = F_{n-2} + F_{n-1}$, $n > 2$
- Strategija podijeli-pa-vladaj daje rekurzivnu funkciju za određivanje Fibonaccijevih brojeva:

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

- Složenost algoritma: označimo s t_n vrijeme potrebno za izračun F_n, tada vrijedi:

$$t_n = t_{n-1} + t_{n-2}\text{, početni uvjet: } t_1 = t_2 = c$$

Karakteristična jdba ove rekurzije: $x^2 - x - 1 = 0$, korijeni su $x_{1,2} = (1 \pm \sqrt{5})/2$

Opće rješenje rekurzije:

$$F_n = C_1 \{(1+\sqrt{5})/2\}^n + C_2 \{(1-\sqrt{5})/2\}^n$$

Konstante C_1 i C_2 se odrede iz početnih uvjeta: $C_1 = - C_2 = 1/\sqrt{5}$, pa je rješenje:

$$F_n = 1/\sqrt{5} \{(1+\sqrt{5})/2\}^n - 1/\sqrt{5} \{(1-\sqrt{5})/2\}^n$$

$$t_n = O(\{(1+\sqrt{5})/2\}^n) = O(1.618^n)$$

- Dakle vrijeme izvršavanja raste eksponencijalno s veličinom ulaza

- Strategija dinamičkog programiranja vodi na iterativno rješenje
- Prvo se riješi (trivijalan) problem za F_0 i F_1, iz njih se izračuna F_2, pa F_3 iz F_2 i F_1, itd. Time se prvo riješe najjednostavniji problemi i spreme se njihovi rezultati, te se koriste za rješavanje malo složenijeg problema čiji se rezultat opet spremi, pa se koristi za rješavanje još složenijeg problema itd.

```
int fib( int n ) {
  int k, f1, f2;
  if ( n < 2 ) return 1;
  else {
     f1 = f2 = 1;
     for(k=2;k<n;k++) {
        f = f1 + f2;
        f2 = f1; f1 = f; }
  return f; }
```

- Složenost ovog algoritma je O(n) (ima jednu petlju koja ide od 2 do n)
- Ovaj algoritam je bitno brži, što je postignuto spremanjem prethodnih rezultata u dvije cjelobrojne varijable f1 i f2, dakle uz minimalno povećanje prostora postignuto je znatno smanjenje vremena izvršavanja

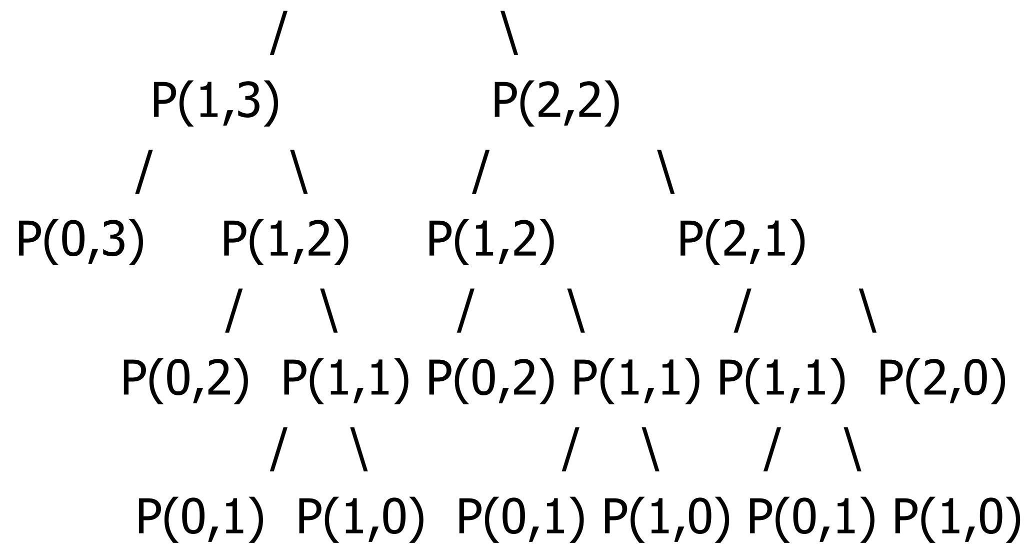
Problem određivanja šanse za pobjedu u nadmetanju

- Problem: dva natjecatelja (ili ekipe natjecatelja) A i B se nadmeću u nekoj (sportskoj) igri koja je podijeljena u dijelove (setove, runde, partije …). U svakom dijelu igre točno jedan igrač (ekipa) bilježi 1 bod. Igra traje sve dok jedan igrač (ekipa) ne sakupi n bodova (unaprijed utvrđena vrijednost) i pobjeđuje. Pretpostavlja se da su igrači podjednako jaki, tako da svaki od njih ima 50% šanse da dobije bod u bilo kojem dijelu igre. Označimo s $P(i,j)$ vjerojatnost da će A biti konačni pobjednik u situaciji kada A treba još i bodova za pobjedu, a B treba još j bodova. Npr. za n = 4, ako je A dobio 2 boda, a B je dobio 1 bod, tada je $i = 2$, $j = 3$ i traži se vjerojatnost P(2,3). Očigledno je da A ima veće šanse za pobjedu. Traži se algoritam koji za zadane i,j računa $P(i,j)$.
- Vrijedi relacija:

$$P(i,j) = \begin{cases} 1 & \text{za } i = 0,\ j > 0 \\ 0 & \text{za } i > 0,\ j = 0 \\ \frac{1}{2} P(i-1,j) + \frac{1}{2} P(i,j-1) & \text{za } i > 0,\ j > 0 \end{cases}$$

- Prva dva reda odgovaraju situaciji kad je jedan od igrača sakupio n bodova, treći red opisuje situaciju kad se igra bar još jedan dio igre u kojem A ima 50% šanse da dobije bod: ako dobije vjerojatnost konačne pobjede mu je $P(i-1,j)$, ako izgubi vjerojatnost je $P(i,j-1)$

- Ova relacija se može iskoristiti za rekurzivno računanje P(*i*,*j*) na koje vodi strategija podijeli-pa-vladaj, no tako dobiveni algoritam je neefikasan, jer se iste vrijednosti računaju mnogo puta
- Npr. da se izračuna P(2,3)



Ukupno se računa: P(0,1) 3 puta, P(1,0) 3 puta, P(0,2) 2 puta, P(1,1) 3 puta, P(2,0) 1 put, P(0,3) 1 put, P(1,2) 2 puta, P(2,1) 1 put, P(1,3) 1 put, P(2,2) 1 put, P(2,3) 1 put: to je 19 računanja

- Složenost podijeli-pa-vladaj algoritma:
- vrijeme izvršavanja za $P(0,j)$ i $P(i,0)$ je konstantno (izjednačavanje)
- za i,j različite od 0 vrijedi $P(i, j) = ½ P(i-1,j)+ ½ P(i,j-1)$, ako je t_n vrijeme za izračunavanje $P(i, j)$ tada je n proporcionalno zbroju $i + j$, jer je ukupan broj operacija (zbrajanja) koje traju konstantno vrijeme C jednak zbroju $i + j$
- Dakle: $t_n = t_{n-1} + t_{n-1} = 2 t_{n-1}$
- Karakteristična jdba $x – 2 = 0$ ima korijen $x = 2$, pa je opći član

$$t_n = C 2^n$$

odnosno vrijeme izvršavanja računa je $t_n = O(2^n)$

- Strategija dinamičkog programiranja vodi na ispunjavanje tabele pri računanju $P(i, j)$ u kojoj i odgovara stupcima a j redcima tabele

Bilo koji unutrašnji element se može dobiti kao aritmetička sredina elemenata ispod i udesno Popunjavanje tabele po dijagonalama od dolje desno

i

½	21/32	13/16	15/16	1	4	j
11/32	½	11/16	7/8	1	3	
3/16	5/16	½	¾	1	2	
1/16	1/8	1/4	½	1	1	
0	0	0	0		0	
4	3	2	1	0		

- Algoritam se može zapisati sljedećom C funkcijom koja radi nad globalnim poljem P[][]:

```
float vjerojatnost_pobjede(int i, int j) {
    int s, k ;
    for (s=1; s<=(i+j); s++) { /* računa dijagonalu elemenata sa zbrojem indeksa s */
        P[0][s] = 1.0; P[s][0] = 0.0;
        for (k=1; k<s; k++) {
            P[k][s-k] = (P[k-1][s-k] + P[k][s-k-1])/2.0; }
    return P[i][j];
}
```

- Složenost algoritma: unutrašnja petlja (u kojoj izvršavanje traje neko konstantno vrijeme) se vrti od 1 do s pa joj je vrijeme izvršavanja O(s), a vanjska se vrti za s=1, 2, …, i+j (u njoj je dodatno konstantno vrijeme za izjednačavanje)

- Vrijeme izvršavanja obje petlje: $\sum_{s=1}^{i+j} (c_1 * s + c_2) = c_1 * (i+j)(i+j+1)/2 + c_2 * (i+j)$

- Vrijeme računanja P(*i,j*) je dakle $O((i+j)^2)$

Rješavanje 0/1 problema ranca

- Problem: zadano je n predmeta $O_1, O_2, \ldots, O_n$ i ranac. Predmet O_i ima težinu w_i i vrijednost p_i. Kapacitet ranca je c težinskih jedinica. Koje predmete treba staviti u ranac tako da ukupna težina ne bude veća od c, a da ukupna vrijednost bude maksimalna ? Uzima se da su w_i i c pozitivni cijeli brojevi. 0/1 znači da se predmeti ne mogu podijeliti, ili se uzme predmet ili ne.
- Problem optimizacije.
- Problem bi se mogao riješiti algoritmom tipa podijeli-pa-vladaj koji bi generirao sve moguće podskupove skupa $\{O_1, O_2, \ldots, O_n\}$ i izabrao onaj s najvećom vrijednošću uz dopustivu težinu (takav algoritam je ustvari algoritam grube sile).
- Složenost takvog algoritma: trebaju se naći i provjeriti svi mogući podskupovi. Broj podskupova s k elemenata iz skupa s n elemenata (broj kombinacija k-tog razreda od n elemenata bez ponavljanja – binomni koeficijenti) je

$$n! \,/\, k!\,(n-k)! = \binom{n}{k}$$

- Za sve moguće podskupove k ide od 1, …, n pa je ukupna složenost dana s:

$$\sum_{k=0}^{n} \binom{n}{k} = 2^n$$

- Dakle, taj algoritam ima složenost $O(2^n)$.

- Strategija dinamičkog programiranja: označimo s M_{ij} maksimalnu vrijednost koja se može dobiti izborom predmeta iz skupa $\{O_1, O_2, \ldots, O_i\}$ uz kapacitet j. Pri dobivanju M_{ij} predmet O_i je stavljen u ranac ili nije. Ukoliko nije, vrijedi $M_{ij} = M_{i-1j}$. Ukoliko je stavljen, tada prije izabrani predmeti predstavljaju optimalan izbor iz skupa $\{O_1, O_2, \ldots, O_{i-1}\}$ uz kapacitet $j-w_i$. Znači, vrijedi relacija:

$$M_{ij} = \begin{cases} 0 & \text{za } i=0 \text{ ili } j\leq 0 \\ M_{i-1j} & \text{za } j<w_i \text{ i } i>0 \\ \max\{M_{i-1j}, (M_{i-1j-wi} + p_i)\} & \text{inače} \end{cases}$$

- Algoritam se sastoji od ispunjavanja tabele s vrijednostima M_{ij}. Vrijednost koja se traži je M_{nc}, to je maksimalna moguća vrijednost predmeta u rancu. Znači, tabela treba sadržavati M_{ij} za $0\leq i\leq n$, $0\leq j\leq c$, pa je njena dimenzija $(n+1)*(c+1)$. Redoslijed računanja je redak-po-redak (j se brže mijenja od i).
- Vrijeme izvršavanja ovog algoritma je $O(n*c)$, jer se u obavljanju računanja tablice izvrši ukupno $n*c+n+c$ operacija koje zahtijevaju konstantno vrijeme izvršavanja, a vrijeme izvršavanja pretrage tablice u određivanju rješenja je $O(n)$, jer se kreće iz n-tog reda tablice i pomiče po 1 red u svakom koraku.

- Algoritam ispunjavanja tablice prikazan C funkcijom koja koristi sljedeća globalna polja:

```
float M[D1][D2];      /* tablica Mij */
int w[D1];            /*  težine predmeta */
float p[D1];          /* vrijednosti predmeta */
```

- Funkcija vraća maksimalnu vrijednost koja se može prenijeti u rancu kapaciteta c ako se ograničimo na prvih n predmeta

```
float zero_one_knapsack(int n, int c) {
   int i,j;
   for (i=0; i<=n; i++) M[i][0]=0.0;
   for (j=0; j<=c; j++) M[0][j]=0.0;
   for (i=1; i<=n; i++)
     for (j=1; j<=c; j++) {
          M[i][j]=M[i-1][j];
          if (j>=w[i])
             if(M[i-1][j-w[i]]+p[i] > M[i-1][j])
                M[i][j]=M[i-1][j-w[i]]+p[i]; }
  return M[n][c]; }
```

- Primjer: zadani ulazni podaci n=3, $(w_1,w_2,w_3)=(2,3,4)$, $(p_1,p_2,p_3)=(1,7,8)$, c=6
- Algoritam izračuna ovu tabelu:

i \ j	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	1	1	1	1	1
2	0	0	1	7	7	8	8
3	0	0	1	7	8	8	9

- Rješenje: maksimalna vrijednost koja se može nositi u rancu je $M_{3,6}=9$ koja se postiže izborom prvog i trećeg predmeta, a ukupna težina je 6.
- Prikazana verzija algoritma daje samo maksimalnu vrijednost koja se može ponijeti u rancu, a ne i optimalni izbor predmeta
- Optimalni izbor: usavršeni algoritam uz svaki element M_{ij} posprema i “zastavicu” (logičku vrijednost) koja označava da li je M_{ij} dobiven kao M_{i-1j} ili $M_{i-1j-wi}+p_i$. Zastavica uz M_{nc} označava da li je predmet O_n dobiven u optimalnom izboru, ovisno o njenoj vrijednosti se može odrediti da li je predmet O_{n-1} izabran u optimalnom izboru itd.

Problem množenja lanca matrica

- Ovaj problem uključuje pitanje određivanja optimalnog niza izvršavanja serije operacija, što je važno u dizajniranju kompajlera za optimizaciju koda i u bazama podataka za optimizaciju pretraživanja i upotrebe podataka
- želi se pomnožiti niz matrica $A_1A_2 \dots A_n$. Množenje matrica je asocijativna operacija ($(A_1A_2)A_3 = A_1(A_2A_3)$) ali nije komutativna operacija (A_1A_2 nije jednako A_2A_1). To znači da se redoslijed matrica ne smije mijenjati, ali se može mijenjati položaj zagrada u množenju, odnosno redoslijed množenja matrica
- Važno: kada se množe dvije nekvadratne matrice, postoje ograničenja na dimenzije njihovog produkta. Npr. Kada se množe matrice A dimenzije p x q (p redova i q stupaca) i matrica B dimenzije q x r (broj stupaca od A mora biti jednak broju redaka od B), njihov produkt matrica C je dimenzije p x r. Produkt se računa po izrazu (za $1 \leq i \leq p$, $1 \leq j \leq r$)

$$C[i,j] = \sum_{k=1}^{q} A[i,k]B[k,j]$$

Matrica C ima p*r članova, a za računanje svakog treba vrijeme O(q), pa je ukupno vrijeme računanja O(p*q*r)

- Ako se množi tri ili više matrica, svaki redoslijed zagrada vodi na ispravan rezultat, ali uključuje različiti broj operacija potrebnih u računanju
- Npr. množe se tri matrice dimenzija: A_1 5x4, A_2 4x6 i A_3 6x2
- mult[$((A_1 A_2)A_3)$] znači 5*4*6 + 5*6*2 = 120 + 60 = 180 operacija množenja
- mult[$(A_1(A_2 A_3))$] znači 4*6*2 + 5*4*2 = 48 + 40 = 88 operacija množenja
- Znači, čak i za tako kratki niz matrica može se postići značajno ubrzanje algoritma preslagavanjem redoslijeda njihovog množenja
- Problem množenja niza matrica: za dani niz matrica $A_1, A_2, \ldots, A_n$ i dimenzija $p_0, p_1, \ldots, p_n$ gdje je matrica A_i dimenzije p_{i-1} x p_i, odrediti redoslijed množenja koji minimalizira broj potrebnih operacija
- Strategija podijeli-pa-vladaj (vodi opet na algoritam grube sile) daje sljedeći algoritam: isprobati sve moguće načine postavljanja zagrada i upotrijebiti onaj koji je najefikasniji – no to vodi na ogroman broj mogućnosti
- Za n=2 postoji samo jedan raspored zagrada, za n=3 postoje dva načina, za n matrica postoji n-1 mjesta gdje se može podijeliti niz za najvanjskiji par zagrada (nakon prve matrice, pa nakon druge, itd. sve do n-1 matrice). Ako se podijeli niz nakon k-te matrice, postoje dva podniza koje treba opet razdijeliti zagradama, jedan duljine k i drugi duljine n-k i treba pronaći broj mogućih načina podjela ta dva podniza

- Ako se prvi niz može razdijeliti zagradama na L načina, a drugi na R načina, jer su to nezavisni izbori, ukupni broj podjela početnog niza je L*R
- To vodi na sljedeću rekurziju za broj različitih načina razdvajanja zagradama niza od n matrica P(n)

$$P(n) = \sum_{k=1}^{n-1} P(k)P(n-k)\ ,\ P(1) = 1$$

- Rješenje ove rekurzije vodi na Catalonove brojeve (nazvane prema belgijskom matematičaru Eugene Charles Catalanu) koji se često javljaju u kombinatorici

$$P(n) = C_{n-1}$$

gdje je $C_n = \binom{2n}{n}/(n+1) = (2n)!/[(n+1)!n!]$

Rekurzija: $C_{n+1} = 2(2n+1)/(n+2) * C_n$

- Može se pokazati da je $C_n = \Omega(4^n/n^{3/2})$, tj. da C_n ne raste sporije od $4^n/n^{3/2}$
- Za vrlo velike n dakle vrijeme izvršavanja raste eksponencijalno
- Vrlo neefikasni algoritam

- Strategija dinamičkog programiranja: razbiti problem u podprobleme čija će rješenja kombiniranjem dati rješenje početnog problema
- Oznaka: $A_{i..j}$ je produkt matrica A_i do A_j , dimenzija ove matrice je p_{i-1} x p_j
- Gledamo najviši nivo stavljanja zagrada, zadnji korak u kojem se množe dvije matrice. Tada za svaki k, 1≤k≤n-1 vrijedi

$$A_{1..n} = A_{1..k} A_{k+1..n}$$

- Problem određivanja optimalnog rasporeda zagrada je time podijeljen u dva pitanja: kako odlučiti gdje podijeliti niz matrica (kako naći k) i kako podijeliti podnizove matrica $A_{1..k}$ i $A_{k+1..n}$
- Odgovor na drugo pitanje je rekurzivna upotreba istog postupka
- Odgovor na prvo pitanje: jedna mogućnost je naći takav k koji minimalizira dimenziju produkta matrica, ali to nažalost ne radi. Najjednostavnije rješenje je provjeriti sve moguće vrijednosti k
- Ovaj postupak zadovoljava princip optimizacije: ako želimo naći optimalno rješenje za $A_{1..n}$, treba nam optimalno rješenje za $A_{1..k}$ i $A_{k+1..n}$ – podproblemi trebaju biti riješeni optimalno da bi rješenje općeg problema bilo optimalno
- Rješenja podproblema se pospremaju u tabelu i tabela se puni od "dna prema vrhu"
- Neka m[i,j] za 1≤i≤j≤n označava minimalni broj množenja potrebnih za računanje $A_{i..j}$

- Ako je i=j, tada niz sadrži samo jednu matricu, nema množenja i m[i,i]=0
- Za i<j tražimo produkt $A_{i..j}$, što se može podijeliti na $A_{i..k}$ $A_{k+1..j}$ za svaki $i \le k < j$
- Optimalno vrijeme za izračunati $A_{i..k}$ je m[i,k] i za $A_{k+1..j}$ je m[k+1,j], pretpostavlja se da su te vrijednosti već izračunate i spremljene u tabelu
- Matrica $A_{i..k}$ je dimenzije p_{i-1} x p_k , a matrica $A_{k+1..j}$ je dimenzije p_k x p_j , pa vrijeme potrebno za njihovo množenje ovisi o $p_{i-1} * p_k * p_j$. Iz toga slijedi rekurzija za računanje m[i,j]

$$m[i,i] = 0$$

$$m[i,j] = \min_{i \le k < j}(m[i,k]+m[k+1,j]+ p_{i-1} * p_k * p_j) \quad \text{za } i<j$$

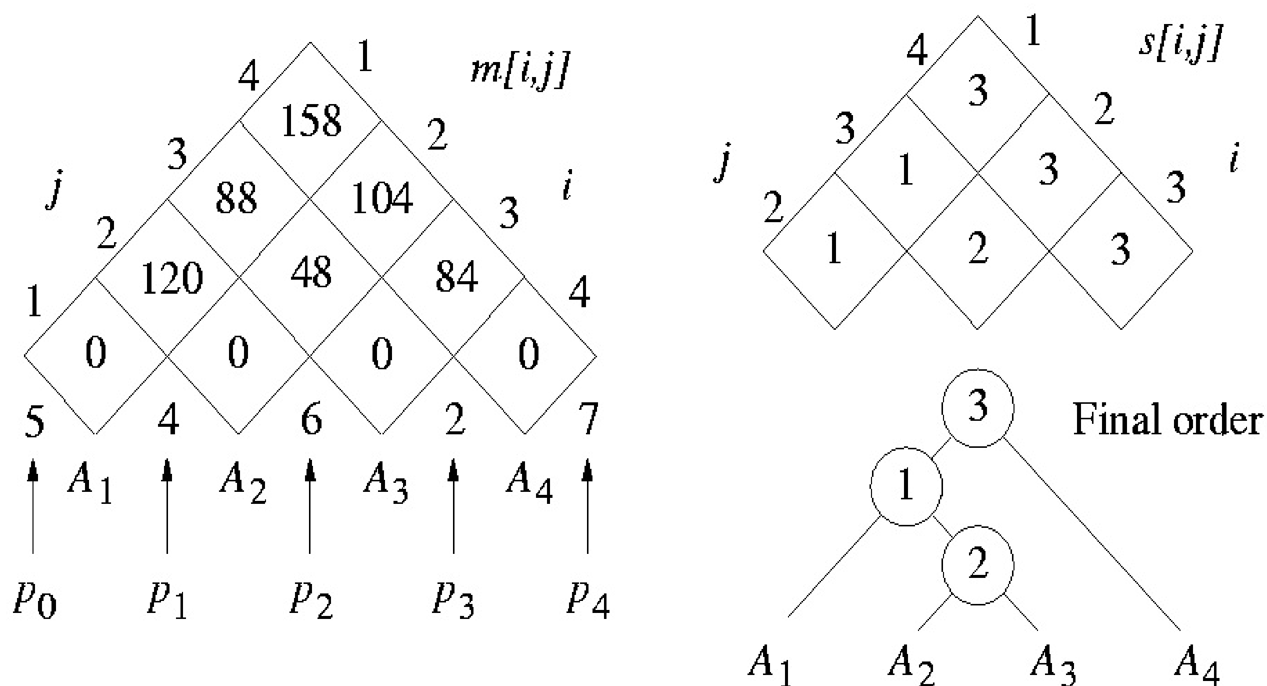
- redoslijed računanja gornjeg izraza: duljinu niza matrica koji se množi označimo s L=j-i+1. Za L=1 je i=j, trivijalan slučaj. Nadogradnja tabele se radi za L = 2, 3, ..., n, gdje je konačno rješenje m[1,n].
- Slaganje petlje: podniz duljine L počinje na poziciji i, pa je j=i+L-1, a mora biti $j \le n$, što vodi na $i \le n-L+1$, pa dakle petlja za i mora ići od 1 do n-L+1
- pseudokod koji obavlja ovaj račun, radi na globalnom polju m[n][n]:

```
Matrix-chain(array p[1..n], int n) {
 array s[1..n-1,2..n];
 for (i=1; i<=n; i++)
      m[i][i]=0;
 for (L=2; L<=n; L++) {
           for (i=1; i<=n-L+1;i++) {
                   j=i+L-1;
                   m[i][j] = INFINITY;
                   for (k=i; k<=j-1; k++) {
                         q=m[i][k]+m[k+1][j]+p[i-1]*p[k]*p[j];
                              if (q < m[i][j]) {
                                    m[i][j]=q;
                                    s[i][j]=k;
                              }
                        }
                 }
 return m[1][n] (konačni broj operacija) i s[][] (polje oznaka podjela)
}
```

- U ovom algoritmu se vrte tri petlje od kojih svaka može maksimalno poprimati vrijednosti u rasponu 1 … n, pa je vrijeme izvršavanja algoritma $\Theta(n^3)$
- Polje s[][] se koristi za određivanje završnog niza množenja matrica (rješenje)
- Osnovna ideja je postaviti oznaku podjele koja označava koja vrijednost k daje minimalnu vrijednost m[i,j], i ta vrijednost k se sprema u polje s[][]. Za s[i][j]=k tada znamo da je najbolji način računanja $A_{i..j}$ taj da prvo pomnožimo $A_{i..k}$ i A_{k+1j} i onda pomnožimo te dvije matrice. To znači da nam s[i][j] govori koje množenje da obavimo zadnje. Primjetimo da s[i][j] spremamo samo za j>i
- Stvarni algoritam množenja matrica tada koristi polje s[i][j] za određivanje podjele niza matrica
- Ako matrice držimo u polju matrica A[1..n] i s[i][j] je globalno polje, rekurzivna procedura koja vrši množenje je oblika (procedura vraća matricu) :

```
Mult(i,j) {
  if (i==j)
      return A[i];
  else {
      k=s[i][j]; X=Mult(i,k); Y=Mult(k+1,j);      /* X=A[i]..A[k], Y=A[k+1]..A[j]  */
      return X*Y; }                              /* vraća produkt matrica X i Y  */
}
```

- Primjer: zadan niz od četiri matrice dimenzija A_1 5x4, A_2 4x6, A_3 6x2 i A_4 2x7, pa je p_i=(5,4,6,2,7)



- Optimalni redoslijed množenja je $((A_1(A_2\,A_3))A_4)$.

Pohlepni pristup

- U mnogim algoritmima optimizacije potrebno je napraviti niz izbora
- Strategija dinamičkog programiranja se često koristi u rješavanju takvih problema, gdje se optimalno rješenje traži primjenom rekurzije računajući od najmanjih (najjednostavnijih) problema prema složenijim (od dna prema vrhu)
- No strategija dinamičkog programiranja može voditi na preduga vremena izvršavanja (neefikasne algoritme)
- Alternativna strategija (tehnika) koja se koristi u rješavanju takvih problema je strategija pohlepnog algoritma
- Ova tehnika obično vodi na jednostavne i brze algoritme, ali nije toliko moćna i primjenljiva kao dinamičko programiranje
- Dodatna prednost ove strategije je da čak kad ne proizvede optimalno rješenje, često vodi na novu strategiju razvoja algoritma koja rezultira efikasnijim rješenjem ili tehnikom koja brzo pronalazi dobru aproksimaciju rješenja (heuristika)
- Rješenje problema se konstruira u nizu koraka, gdje se u svakom koraku bira mogućnost koja je lokalno optimalna u nekom smislu. Ideja je da će takvi koraci dovesti do globalno optimalnog rješenja

- Primjer: trgovac vraća mušteriji 62 kune, na raspolaganju su mu novčanice od 50, 20, 10, 5 i 1 kune. Instinktivno će većina ljudi vratiti 1x50, 1x10 i 2x1 kunu. Takav algoritam vraća točan iznos uz najkraću moguću listu novčanica
- Algoritam: izabere se najveća novčanica koja ne prelazi ukupnu sumu, stavlja se na listu za vraćanje, oduzme se vraćena vrijednost od ukupne kako bi se izračunao ostatak za vraćanje, zatim se bira slijedeća novčanica koja ne prelazi ostatak koji treba vratiti, dodaje se na listu za vraćanje i računa novi ostatak za vraćanje itd, sve dok se ne vrati točan iznos
- Dakle strategija pohlepnog pristupa je dovela do najefikasnijeg rješenja problema vraćanja novca i to zahvaljujući specijalnim svojstvima raspoloživih novčanica
- Primjer gdje pohlepni pristup ne funkcionira: potrebno je vratiti 15 kuna pomoću novčanica od 11, 5 i 1 kune. Pohlepni algoritam prvo vraća 11 kuna i tada mu ostaje samo jedno rješenje: da vrati četiri puta po 1 kunu, znači ukupno 5 novčanica. Optimalno rješenje bi bilo vratiti tri puta po 5 kn.
- Primjer: 0/1 ranac: zadani ulazni podaci n=3, $(w_1,w_2,w_3) = (2,3,4)$, $(p_1,p_2,p_3) = (1,7,8)$, c=6. Strategija dinamičkog programiranja je dala optimalno rješenje: izborom prvog i trećeg predmeta maksimalna vrijednost je 9, a ukupna težina je 6. Pohlepni pristup: izračuna se profitabilnost (p_i/w_i), prvo se uzme najprofitabilniji 2 predmet, nakon čega slijedeći po profitabilnosti (3 predmet) ne stane u ranac, pa se može uzeti samo 1 predmet. Ukupna vrijednost je 8, a težina 5.

Huffmanov algoritam

- Predstavlja metodu efikasnog dekodiranja podataka za komprimiranje podataka bez gubitka informacije o podacima
- Uobičajeno kodiranje znakova poput ASCII koda: svaki znak je predstavljen kodnom riječi fiksne duljine (8 bitova/znak). Kodovi fiksne duljine su popularni jer omogućuju vrlo jednostavan postupak: niz znakova se podijeli u pojedinačne znakove i njihovo kodiranje/dekodiranje se vrši jednostavnim direktnim indeksiranjem
- No takvi kodovi ne minimaliziraju ukupnu količinu podataka
- Primjer: želi se (de)kodirati nizove znakova sastavljene od abecede s četiri znaka C={a, b, c, d}, koristi se slijedeći kod fiksne duljine:

  a = 00 ; b = 01 ; c = 10 ; d = 11

- Niz znakova "abacdaacac" se kodira zamjenom svakog znaka s odgovarajućom binarnom kodnom riječi

a	b	a	c	d	a	a	c	a	c
00	01	00	10	11	00	00	10	00	10

- Pa je konačni niz 20-bitni niz znakova "00010010110000100010"

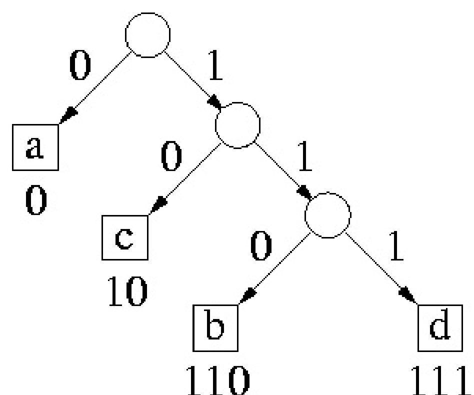
- Ukoliko su unaprijed poznate relativne vjerojatnosti pojavljivanja znakova (što se može izvesti analiziranjem velikog broja nizova tijekom duljeg vremena ili za komprimiranje jedne datoteke prebrojavanjem frekvencije pojavljivanja svih znakova) može se izvesti drugačije kodiranje: znakovi koji se češće pojavljuju se kodiraju koristeći manje bitova, a znakovi koji se javljaju rjeđe se kodiraju koristeći više bitova, čime se smanjuje ukupan broj bitova potrebnih za (de)kodiranje niza znakova
- Za gornji niz znakova, frekvencija njihovog pojavljivanja je: p(a) = 0.5, p(b) = 0.1, p(c) = 0.3, p(d) = 0.1
- Može se izvesti slijedeće kodiranje promjenljive duljine:

  a = 0 ; b = 110; c = 10; d = 111
- Isti niz od 10 znakova je sad kodiran 17-bitnim nizom “01100101110010010”
- Ušteđena su dakle 3 bita
- Općenito: koliko se može uštedjeti za niz znakova duljine n ? Ako se koristi 2-bitni kod fiksne duljine, duljina kodiranog niza je 2n bitova. Za kodiranje promjenljivom duljinom, očekivana duljina pojedinačnog kodiranog znaka je jednaka sumi umnožaka duljine kodne riječi s vjerojatnošću njenog pojavljivanja, a očekivana duljina kodiranog niza je n puta očekivana duljina pojedinačnog kodiranog znaka:

  $$n(0.5*1+0.1*3+0.3*2+0.1*3) = 1.7*n$$

- Problem koji se ovdje rješava je kako napraviti najbolji algoritam, koji će uštedjeti maksimalno prostora, ako su poznate vjerojatnosti pojavljivanja pojedinih znakova
- Važno pitanje u rješavanju ovog problema je mogućnost dekodiranja, i to brza i efikasna, kodiranog binarnog niza
- Iz tog razloga su i kodne riječi u gornjem primjeru pažljivo izabrane
- Npr. da se za znak a uzeo kod 1 (a ne 0), bilo bi nemoguće odrediti da li niz 111 odgovara znaku d ili nizu aaa
- Traži se da kodiranje bude takvo da se binarni niz može jednoznačno dekodirati
- To se može postići tako da niti jedna korištena kodna riječ nije prefiks ostalih kodnih riječi (kao u gornjem primjeru) – to je ključno svojstvo
- Znači da čim se nađe neka kodna riječ u nizu, može se jednoznačno odrediti kojem znaku ona odgovara
- Definicija: **prefiks kodovi** se dobivaju pridruživanjem kodnih riječi znakovima na takav način da niti jedna kodna riječ nije prefiks drugih kodnih riječi
- Svako binarno prefiksno kodiranje se može prikazati kao binarno stablo u kojem su kodne riječi u listovima stabla i u kojem lijeva grana znači 0 a desna znači 1
- Duljina kodne riječi je tada jednaka dubini odgovarajućeg lista u stablu

- Binarno stablo koje odgovara korištenom prefiksnom kodiranju

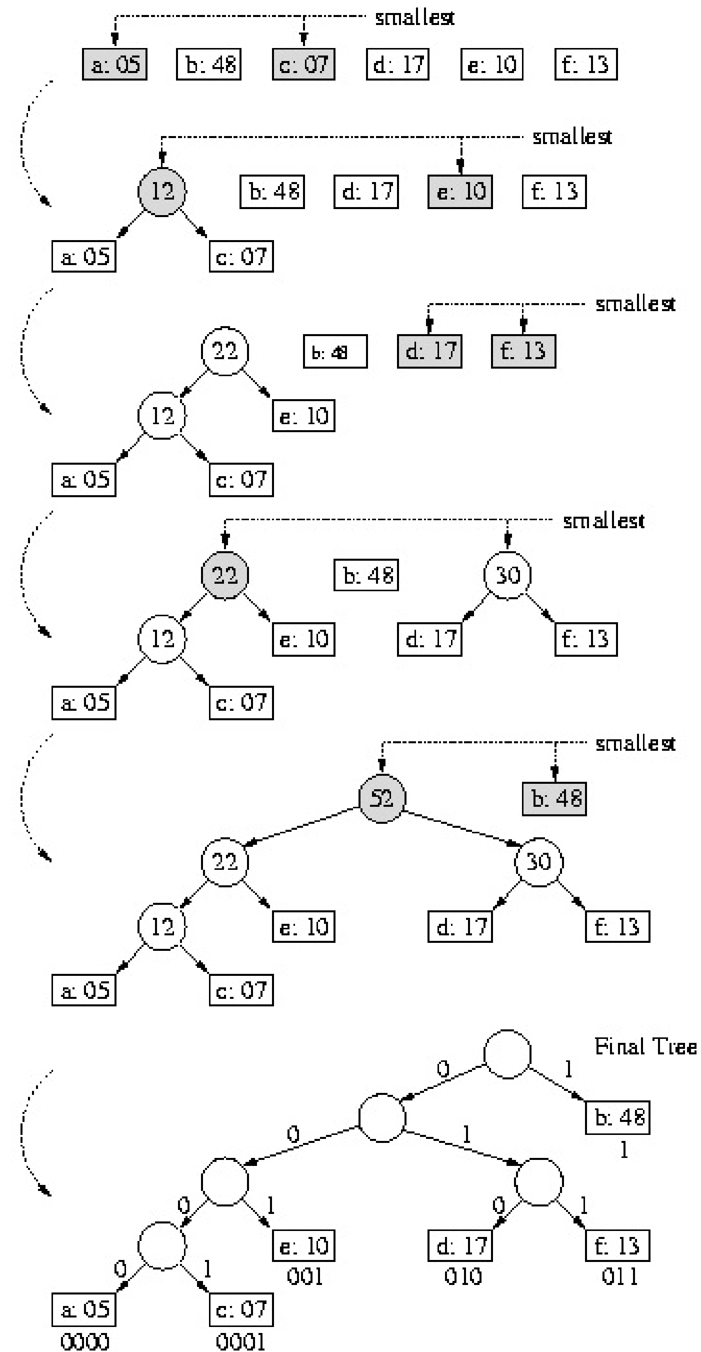


- Dekodiranje prefiksnog kodiranja je jednostavno: prolazi se binarnim stablom od korijena prema listovima i iz ulaznog niza binarnih znakova se odlučuje kojom granom treba ići. Kada se dosegne list, na izlaz se šalje odgovarajući znak, te se ponovno kreće od korijena stabla za nastavak dekodiranja ulaznog binarnog niza
- Očekivana duljina niza: ukupna duljina kodiranog niza se može odrediti iz poznate vjerojatnosti pojavljivanja pojedinih znakova. Ako se s $p(x)$ označi vjerojatnost pojavljivanja znaka x, a s $d_T(x)$ duljina kodne riječi tj. njena dubina u binarnom stablu T prema kojem se vrši prefiksno kodiranje, tada je očekivan broj bitova za kodiranje niza znakova duljine n

$$B(T) = \sum_{x} p(x) * d_T(x)$$

- što vodi na sljedeći problem: **generiranje optimalnog kodiranja**: zadana je abeceda C i vjerojatnosti pojavljivanja p(x) svih znakova x iz C. Odrediti prefiksno kodiranje T koje minimalizira očekivanu duljinu kodiranog binarnog niza B(T).
- Primijetimo da optimalno kodiranje nije jedinstveno, mogu se komplementirati svi bitovi (0 <-> 1) u svim kodnim riječima
- Postoji vrlo jednostavan algoritam za traženje takvog kodiranja: **Huffmanov algoritam**. Koristile su ga prve aplikacije za komprimiranje podataka (unix pack)
- Huffmanov algoritam: gradi se binarno stablo počevši od listova. Uzimaju se dva znaka x i y i spajaju se u "superznak" z, koji tada zamjenjuje x i y u abecedi. Vjerojatnost pojavljivanja znaka z je tada jednaka sumi vjerojatnosti za x i y. Rekurzivno se nastavlja algoritam na novoj abecedi koja ima jedan znak manje. Kada se proces završi, odredi se kodna riječ za znak z, recimo 010. Njoj se dodaju 0 i 1 da bi se dobile kodne riječi za x - 0100 i za y - 0101.
- Drugim riječima, spajaju se x i y kao lijevo i desno dijete korijena z, čime podstablo od z zamjenjuje x i y u listi znakova. Postupak se ponavlja dok ne ostane samo jedan "superznak". Rezultirajuće binarno stablo je tada prefiksno stablo
- Kako su x i y na dnu stabla, za njim se izabiru dva znaka s najmanjom vjerojatnošću pojavljivanja

- Ilustracija Huffmanovog algoritma:



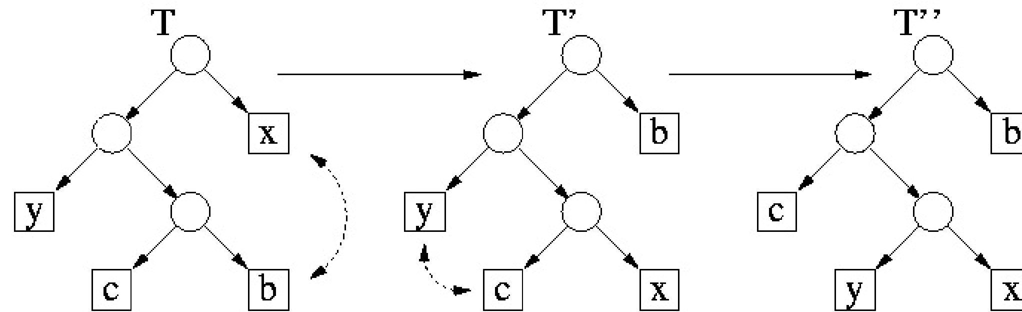
- Pseudokod za Huffmanov algoritam: imamo skup znakova C, svakom znaku x iz C je pridružena vjerojatnost pojavljivanja x.prob . Na početku su svi znakovi spremljeni u prioritetni red Q. Takva struktura podataka se izgrađuje u vremenu O(n), element s najmanjom oznakom se može pronaći u vremenu O(lg n) i novi element se dodaje u vremenu O(lg n). Objekti u Q su složeni po vjerojatnosti pojavljivanja.

```
Huffman(int n, character C[1..n]){
  Q = C;     /* prioritetni red složen po vjerojatnosti pojavljivanja */
  for (i=1; i<=n-1; i++) {
      z = novi interni čvor stabla;
      z.left = x = Q.extractMin();   /* pronađe dva znaka s najmanjim vjerojatnostima */
      z.right = y = Q.extractMin();
      z.prob = x.prob + y.prob; /* vjerojatnost za z je suma vjerojatnosti za x i y */
      Q.insert(z);  /* ubaci z u prioritetni red */
        }  return (zadnji element iz Q kao korijen stabla); }
```

- Svako izvršavanje petlje smanji broj elemenata u redu za jedan, pa nakon n-1 iteracije ostaje točno jedan element u redu i on je korijen konačnog prefiksnog binarnog stabla

- Pitanje: da li je taj algoritam ispravan i zašto ?
- Trošak za određivanje stabla kodiranja T je B(T), potrebno je dokazati da se svako stablo koje se razlikuje od onog konstruiranog Huffmanovim algoritmom može pretvoriti u stablo dobiveno HA bez povećanja troška
- Stablo iz HA je potpuno binarno stablo (svaki unutrašnji čvor ima točno dvoje djece), pa se ograničavamo na takav slučaj u razmatranju
- Tvrdnja: uzmimo dva znaka x i y koji imaju najmanje vjerojatnosti pojavljivanja. Tada postoji optimalno binarno stablo kodiranja u kojem su ta dva znaka braća na maksimalnoj dubini stabla.
- Dokaz: neka je T optimalno binarno stablo prefiksnog kodiranja i neka su b i c dva brata na maksimalnoj dubini stabla. Pretpostavimo, bez gubitka općenitosti, da je $p(b) \leq p(c)$ i $p(x) \leq p(y)$ (da bi se to postiglo, znakovi se mogu preimenovati). Sada, jer x i y imaju najmanje vjerojatnosti, slijedi da je $p(x) \leq p(b)$ i $p(y) \leq p(c)$. Jer su b i c na najdubljem nivou stabla, vrijedi $d(b) \geq d(x)$ i $d(c) \geq d(y)$. Dakle imamo $p(b) - p(x) \geq 0$ i $d(b) - d(x) \geq 0$, pa je produkt p*d nenegativan. Sada zamijenimo pozicije x i b u stablu, čime dobijemo novo stablo T'. Izračunajmo promjenu troška. Trošak stabla T' je

$$B(T') = B(T) - p(x)d(x) + p(x)d(b) - p(b)d(b) + p(b)d(x) =$$
$$= B(T) - (p(b)-p(x))(d(b)-d(x)) \leq B(T)$$



- Dakle, zamjenom pozicija x i b ne povećava se trošak, pa je dakle T′ optimalno stablo. Zamjenom pozicija y i c dobije se novo stablo T″, te identičnim razmatranjem se zaključuje da je i T″ optimalno stablo. Dakle konačno stablo T″ zadovoljava tvrdnju i time je ona dokazana.
- Ovaj teorem govori da je prvi korak Huffmanovog algoritma ispravan. Da bi dokazali ispravnost cijelog algoritma možemo koristiti matematičku indukciju, jer u svakom koraku izbacujemo točno jedan znak.
- Tvrdnja: Huffmanov algoritam proizvodi optimalno binarno stablo prefiksnog kodiranja.
- Dokaz: indukcijom po n, broju znakova. Osnovni slučaj, n=1, znači stablo s jednim listom, što je trivijalno optimalno binarno stablo.

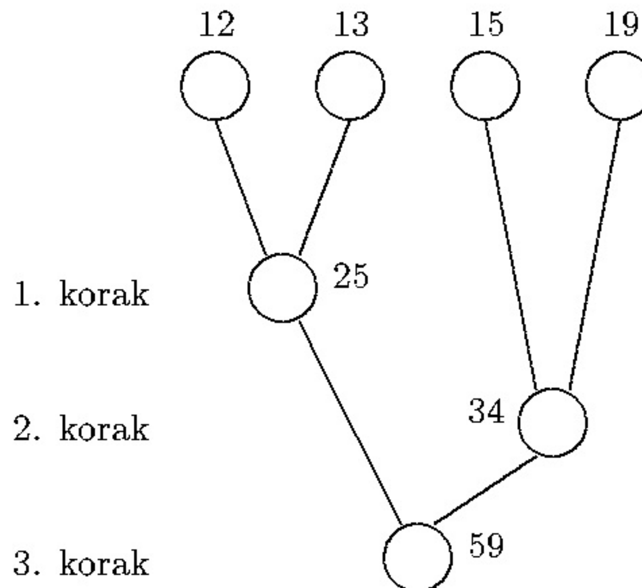
- Pretpostavlja se da Huffmanov algoritam proizvodi optimalno stablo za strogo manje od n znakova. Želi se pokazati da onda to vrijedi i za točno n znakova.
- Pretpostavimo da imamo točno n znakova. Prethodna dokazana tvrdnja govori da će u optimalnom stablu dva znaka s najmanjim vjerojatnostima, x i y, biti braća na najnižem nivou stabla. Zamijenimo x i y sa novim znakom z čija je vjerojatnost $p(z) = p(x) + p(y)$. Time nam ostaje samo n-1 znakova. Gledamo bilo koje binarno stablo prefiksnog kodiranja T napravljeno od novog skupa sa n-1 znakova. Ono se može pretvoriti u binarno stablo prefiksnog kodiranja T' za početni skup od n znakova operacijom koja je inverzna prijašnjoj operaciji, tj. zamjenom z sa x i y tako da se doda bit s vrijednošću 0 za x i bit s vrijednošću 1 za y. Trošak novog stabla T' je

$$B(T') = B(T) - p(z)d(z) + p(x)(d(z)+1) + p(y)(d(z)+1) =$$
$$= B(T) + p(x) + p(y)$$

- Promjena troška ne ovisi o strukturi stabla T, pa je za minimaliziranje troška konačnog stabla T' potrebno izgraditi stablo T od n-1 znakova tako da je ono optimalno. Po indukciji, to je upravo ono što Huffmanov algoritam radi, pa je dakle konačno stablo optimalno. Time je dokazana ispravnost Huffmanovog algortima.

Optimalno spajanje sortiranih lista

- n sortiranih listi duljine w_1, w_2, …, w_n se spaja u jednu veliku sortiranu listu. Spajanje se obavlja u n-1 koraka tako da se u svakom koraku spoje dvije odabrane liste algoritmom merge.Traži se optimalni plan sažimanja, takav odabir listi u pojedinom koraku koji vodi na najmanji ukupni broj operacija.
- Optimalni plan spajanja se može odrediti pohlepnim pristupom i može se pokazati da ga konstruira Huffmanov algoritam: u svakom koraku spajanja treba odabrati dvije najkraće liste (drugim riječima svaki korak se izvodi tako da imamo najmanje posla u tom koraku)



Problem planiranja aktivnosti

- Vrlo jednostavan problem planiranja: zadan je skup aktivnosti S = {1,2,…,n} za koje je potrebno neko sredstvo, gdje svaka aktivnost mora započeti u neko zadano vrijeme s_i i završiti u zadano vrijeme f_i (npr. planiranje predavanja u jednoj predavaonici za koja se određuju vremena odvijanja ili raspored prihvaćanja automobila za popravak kod auto-mehaničara).
- Postoji samo jedno sredstvo (predavaonica) a neka vremena odvijanja se mogu preklapati (nemoguće je imati dva predavanja istovremeno u istoj prostoriji), pa se neke aktivnosti (predavanja) moraju izbaciti
- Za dvije aktivnosti *i* i *j* se kaže da se ne preklapaju (smetaju jedna drugoj) ako se vremena njihovog odvijanja (interval između vremena početka i vremena kraja) ne preklapaju: $[s_i,f_i)$ presjek $[s_j,f_j)$ je prazan skup
- Problem planiranja aktivnosti je izbor najvećeg podskupa međusobno nemiješanih aktivnosti koje koriste dano sredstvo (druga mogućnost planiranja aktivnosti može biti recimo takav izbor aktivnosti da se maksimizira ukupno vrijeme iskoristivosti sredstva)
- Pohlepan pristup: prva ideja - da bi se dobio skup s najvećim brojem aktivnosti, neće se izabirati dugačke aktivnosti, dakle u svakom koraku će se izabirati najkraća aktivnost ukoliko se ne preklapa s do tada izabranim aktivnostima

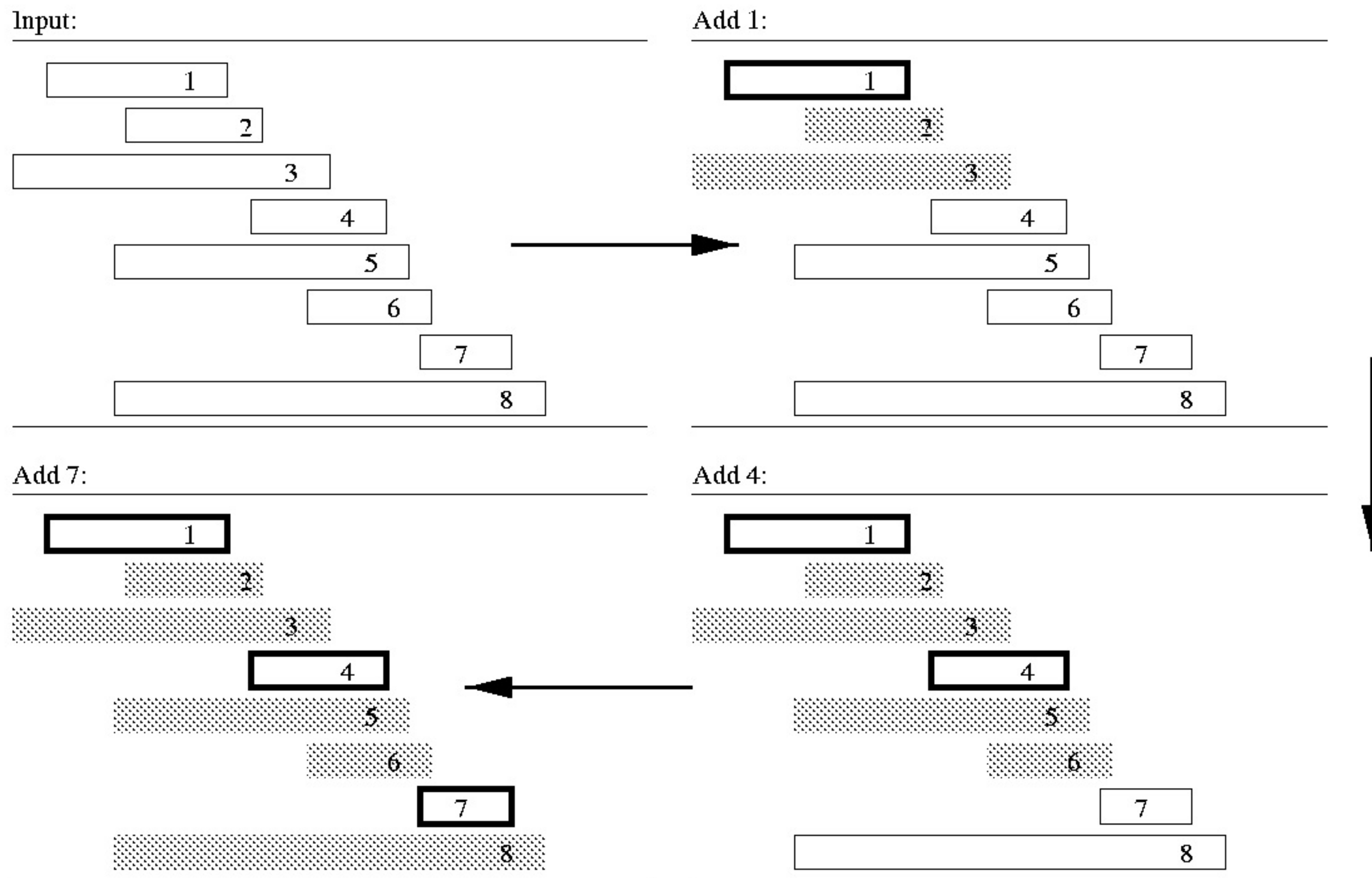
- No to vodi na neoptimalno iskorištenje sredstva
- Jednostavan pohlepni algoritam koji radi: slična ideja, opet se izbjegavaju dugačke aktivnosti, ali se prvo izabire aktivnost koja prva završava, a tada se od svih ostalih aktivnosti koje se ne preklapaju s prvom aktivnošću izabire ona koja završava prva i taj algoritam se ponavlja
- Počinje se s pretpostavkom da su aktivnosti sortirane po vremenima završavanja

$$f_1 \leq f_2 \leq \ldots \leq f_n$$

- Naravno s_i se također paralelno sortiraju
- Pseudokod za taj algoritam: pretpostavlja se da su sortiranja već obavljena, izlaz je lista A planiranih aktivnosti, varijabla *prev* čuva indeks zadnje planirane aktivnosti kako bi se odredilo da li se aktivnosti miješaju:

```
schedule(int n, int s[1..n], int f[1..n]) {  // pretpostavlja se da je f[1..n] sortiran
   A = <1>; prev = 1;   // prvo izabire prvu aktivnost iz sortirane liste
   for (i=2; i≤n; i++) {
      if (s[i] >= f[prev]) {   // provjera da li se aktivnosti preklapaju
         A = <1,…,prev,i>; prev = i;   // uzima se aktivnost i za slijedeću
      }}
   return A;}
```

- Algoritam je jednostavan i efikasan, vrijeme izvršavanja procedure je Θ(n), najviše vremena uzima sortiranje aktivnosti po vremenu završavanja, što se može obaviti u Θ(n*lg n), pa je ukupno vrijeme izvršavanja Θ(n*lg n)
- Primjer: slika, rješenje je {1,4,7}



- Ispravnost algoritma: dokaz ispravnosti algoritma se bazira na dokazivanju da je prvi izbor u algoritmu najbolji mogući i zatim treba pokazati upotrebom matematičke indukcije da je algoritam globalno optimalan. Način ovakvog dokazivanja se primjenjuje općenito za pohlepne algoritme, treba pokazati da se svako drugo rješenje može pretvoriti u ovakvo rješenje bez povećanja troška
- Tvrdnja: neka je S = {1,2,…,n} skup aktivnosti za koje treba složiti raspored sortiran po vremenu izvršavanja (prva aktivnost završava prva). Tada postoji optimalno rješenje u kojem je aktivnost 1 prva na rasporedu.
- Dokaz: neka je A optimalno rješenje. Neka je x aktivnost u A koja završi prva. Ako je x=1, tvrdnja vrijedi. Ako nije, složimo novi raspored A′ mijenjajući x sa aktivnošću 1. Tvrdi se da je A′ mogući raspored (aktivnosti se ne preklapaju). To je zato jer A-{x} ne može imati neke druge aktivnosti koje počinju prije nego x završi, jer bi se takve aktivnosti preklapale s x. Jer je aktivnost 1 po definiciji aktivnost koja prva završava, ona mora završiti prije x, pa se ne može preklapati s bilo kojom aktivnosti iz A-{x}. Slijedi da je A′ mogući raspored, i jasno je da A i A′ sadrže isti broj aktivnosti, iz čega slijedi da je A′ također optimalan raspored.
- Tvrdnja: pohlepni algoritam daje optimalno rješenje za problem planiranja aktivnosti
- Dokaz: indukcijom po broju aktivnosti. Osnovni slučaj: ako nema aktivnosti (S je prazan skup) algoritam trivijalno daje optimalno rješenje

- Korak indukcije: pretpostavimo da pohlepni algoritam daje optimalno rješenje za svaki skup aktivnosti čija je veličina striktno manja od S.
- Neka je S′ skup aktivnosti koji se ne preklapa s aktivnosti 1, tj. $S' = \{i \varepsilon S \mid s_i \geq f_1\}$.
- Primijetimo da se svako rješenje za S′ može pretvoriti u rješenje za S jednostavnim dodavanjem aktivnosti 1 (i obrnuto)
- Jer je, po gornjoj tvrdnji, aktivnost 1 u optimalnom rješenju za S (i to prva) slijedi da se optimalno rješenje početnog problema S može konstruirati tako da 1 bude prva aktivnost i tada se doda optimalno rješenje za S′.
- Po indukciji, takvo rješenje je optimalno rješenje za S (jer je S veći od S′), a pohlepni algoritam upravo tako i konstruira rješenje, pa je tvrdnja dokazana.

Kontinuirani problem ranca

- Ovaj problem je sličan 0/1 problemu ranca koji smo radili u razmatranjima strategije dinamičkog programiranja, s razlikom da se predmeti koji se stavljaju u ranac sad mogu rezati (ako ne stane cijeli predmet u ranac, može se staviti samo jedan njegov dio)
- Problem: zadan je prirodan broj n i pozitivni realni brojevi c, w_i i p_i (i=1,2,...,n). Traže se realni brojevi x_i (i=1,2,...,n) takvi da je $\sum_{i=1}^{n} p_i * x_i$ maksimalan uz ograničenja $\sum_{i=1}^{n} w_i * x_i \leq c$, $0 \leq x_i \leq 1$ (i=1,2,...,n). C je kapacitet ranca, w_i su težine predmeta, p_i vrijednosti predmeta i x_i označava koji se dio i-tog predmeta stavlja u ranac.
- Pohlepni pristup rješavanju problema zahtijeva da se za svaki predmet izračuna profitabilnost p_i/w_i (vrijednost po jedinici težine). Predmeti se sortiraju silazno po profitabilnosti te se u tom redoslijedu stavljaju u ranac dokle ima mjesta. Kod svakog stavljanja u ranac nastoji se staviti što veći dio predmeta.
- Pseudokod C funkcije koja izvodi gornji algoritam: radi nad globalnim poljima u kojima su spremljene težine, vrijednosti i dijelovi predmeta, pretpostavlja se da su podaci u poljima w[] i p[] već sortirani po profitabilnosti, tj. p[i]/w[i] ≥ p[i+1]/w[i+1]

```
#define MAXLEN
 float w[MAXLEN], p[MAXLEN], x[MAXLEN];
 void cont_knapsack (int n, float c) {
   float cu;
   int i;
   for (i=1; i<=n; i++) x[i] = 0.0;
   cu = c;
   for (i=1; i<=n; i++) {
     if (w[i] > cu) {
        x[i] = cu/w[i];
        return; }
     x[i]=1.0;
     cu -= w[i];}
   return; }
```

- Primjer: n=3, w_i=(2,3,4), p_i=(1,7,8), c=6. Profitabilnosti predmeta p_i/w_i=(1/2,7/3,2). Algoritam stavlja u ranac cijeli drugi predmet, pa ga popunjava s ¾ trećeg predmeta, pa je rješenje x_i=(0,1,3/4) i maksimalna vrijednost u rancu je 13.

- Tvrdnja: opisani pohlepni algoritam zaista daje korektno (optimalno) rješenje kontinuiranog problema ranca.
- Dokaz: po pretpostavci su predmeti sortirani po profitabilnosti, tj. $p_1/w_1 \geq p_2/w_2 \geq \ldots \geq p_n/w_n$. Neka je $X=(x_1, x_2,\ldots, x_n)$ rješenje dobiveno gornjim pohlepnim algoritmom. Ako su svi $x_i=1$, tada je vrijednost u rancu očigledno najveća. Ako nisu, uzima se da je j najmanji indeks takav da je $x_j<1$. Zbog sortiranosti po profitabilnosti slijedi da je $x_i=1$ za $1\leq i<j$ i $x_i=0$ za $j<i\leq n$. Također mora vrijediti $\sum_{i=1}^{n} w_i*x_i=c$.
- Neka je $Y=(y_1, y_2,\ldots, y_n)$ jedno optimalno rješenje. Tada vrijedi $\sum_{i=1}^{n} w_i*y_i=c$. Ako je X=Y tada je dokazano da je X optimalno rješenje. Ako je X različit od Y, može se naći najmanji indeks k takav da je x_k različit od y_k. Tvrdi se da je $k\leq j$ i da je $y_k<x_k$. Za provjeru te tvrdnje, promotre se tri mogućnosti:

  1) za k>j izlazi da je $\sum_{i=1}^{n} w_i*y_i>c$ što ne može biti

  2) za k<j izlazi da je $x_k=1$, no y_k mora biti različit od x_k pa mora biti $y_k<x_k$

  3) za k=j, zbog $\sum_{i=1}^{j} w_i*x_i=c$ i $y_i=x_i$ $(1\leq i<j)$ slijedi da je ili $y_k<x_k$ ili $\sum_{i=1}^{n} w_i*y_i>c$

Što znači da je tvrdnja $k\leq j$ i $y_k<x_k$ dokazana.

- Sada se y_k uveća tako da je jednak x_k, te se umanje vrijednosti $(y_{k+1},\ldots,y_n)$ koliko je potrebno da ukupna težina bude jednaka c. Time se dobiva novo rješenje $Z=(z_1, z_2,\ldots, z_n)$ sa svojstvima: $z_i=x_i$ $(1\leq i<k)$, $\sum_{i=k+1}^{n} w_i*(y_i-z_i)=w_k(z_k-y_k)$.

- Z također mora biti optimalno rješenje jer vrijedi:

$$\sum_{i=1}^{n} p_i*z_i = \sum_{i=1}^{n} p_i*y_i+(z_k-y_k)*p_k-\sum_{i=k+1}^{n}(y_i-z_i)*p_i=\sum_{i=1}^{n} p_i*y_i+(z_k-y_k)p_k/w_k*w_k-\sum_{i=k+1}^{n}(y_i-z_i)p_i/w_i*w_i =$$

= zbog sortiranosti predmeta po profitabilnosti ≥

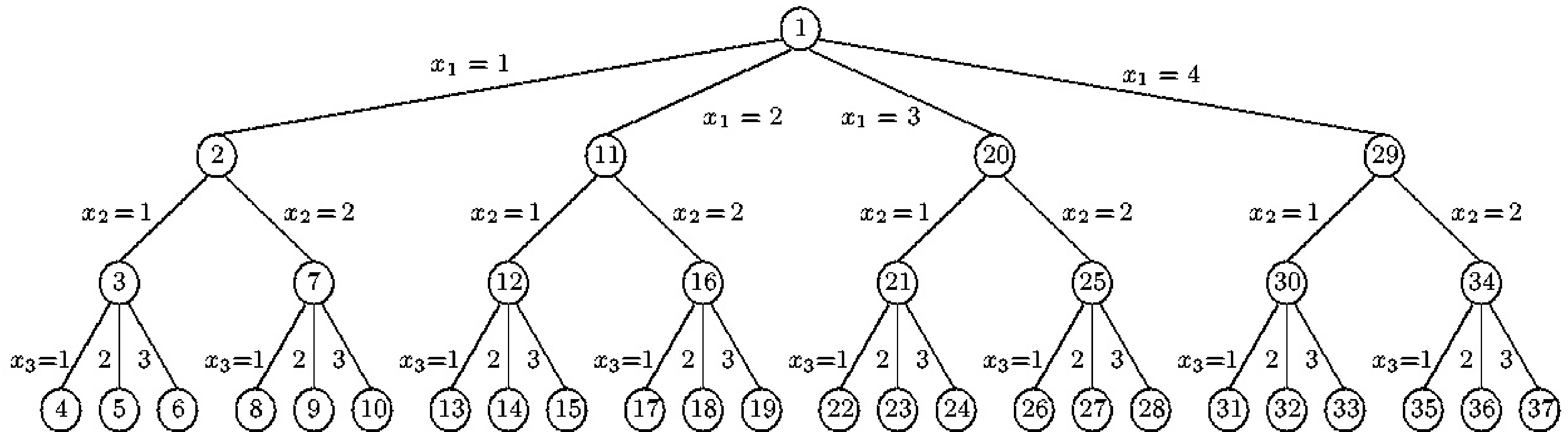
$$\geq \sum_{i=1}^{n} p_i*y_i + [(z_k-y_k)*w_k- \sum_{i=k+1}^{n}(y_i-z_i)*w_i]\ p_k/w_k \geq \sum_{i=1}^{n} p_i*y_i + 0*p_k/w_k =$$

$$= \sum_{i=1}^{n} p_i*y_i$$

- To znači da je nađeno novo rješenje Z koje se bolje poklapa s X nego što se Y poklapa s X. Iteriranjem ovog postupka konstrukcije rješenja doći će se do optimalnog rješenja koje se potpuno poklapa s X, što znači da je X optimalan.
- Time je teorem dokazan

Strategija odustajanja (backtracking)

- Ovo je vrlo općenita tehnika koja se primjenjuje za teške probleme u kombinatorici
- Rješenje problema se traži sistematskim ispitivanjem svih mogućnosti za konstrukciju rješenja
- Metoda zahtjeva da se traženo rješenje izrazi kao n-torka oblika $(x_1, x_2, \ldots, x_n)$ gdje je x_i element nekog konačnog skupa S_i. Kartezijev produkt $S_1 \times S_2 \times \ldots \times S_n$ se zove prostor rješenja
- Neka konkretna n-torka iz prostora rješenja je pravo rješenje problema ako zadovoljava neka dodatna ograničenja ovisna o samom problemu
- Prostor rješenja se prikazuje uređenim stablom rješenja gdje korijen stabla predstavlja sve moguće n-torke
- Dijete korijena predstavlja sve n-torke gdje prva komponenta x_1 ima neku određenu vrijednost, unuk korijena predstavlja sve n-torke gdje su prve dvije komponente x_1 i x_2 određene itd., a list stabla predstavlja jednu konkretnu n-torku u kojoj su sve vrijednosti x_i određene
- Rješenje problema odgovara jednom listu stabla rješenja



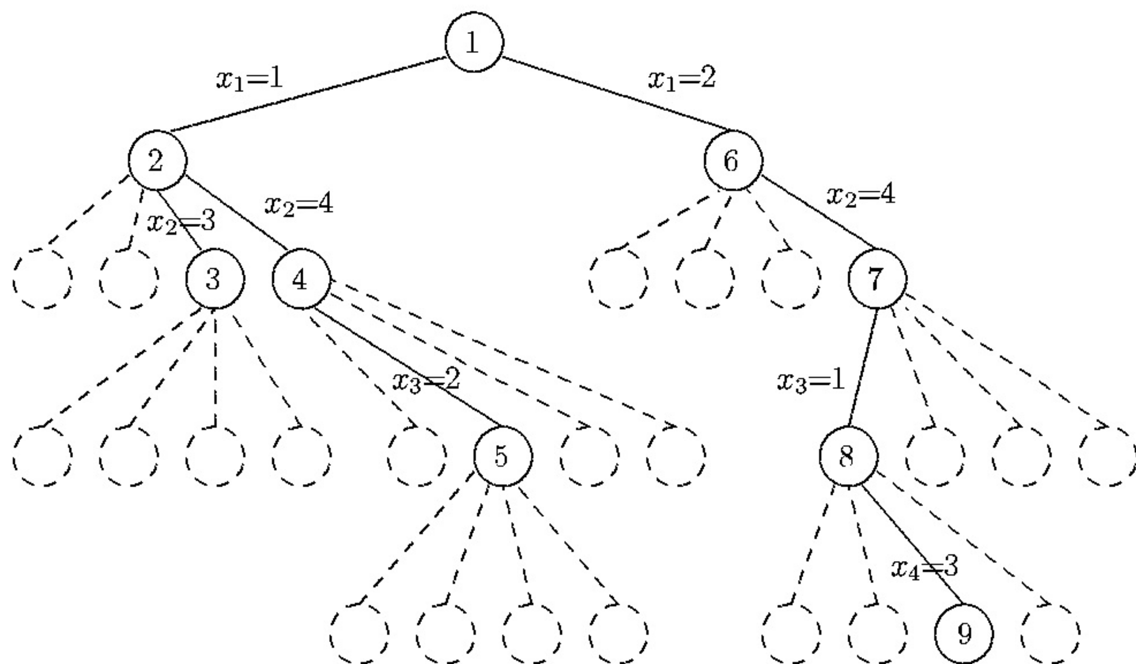
- Primjer: ako se rješenje problema može prikazati kao uređena trojka, n=3, gdje je x_1 iz skupa S_1 = {1,2,3,4}, x_2 iz S_2 = {1,2} i x_3 iz S_3 ={1,2,3}, prostor rješenja se prikazuje gornjim stablom
- To je u osnovi rekurzivni algoritam koji se sastoji od simultanog generiranja i ispitivanja čvorova u stablu rješenja
- Čvorovi se stavljaju na stog i jedan korak algoritma se sastoji od toga da se uzme čvor s vrha stoga i provjeri da li taj čvor predstavlja rješenje problema
- Ako čvor predstavlja rješenje poduzima se neka odgovarajuća akcija

- Ako čvor nije rješenje, pokušaju se generirati njegova djeca i stavljaju se na stog
- Na početku algoritma je na stogu samo korijen stabla, završetak je kad se nađe rješenje ili dobije prazan stog
- Redoslijed obrađivanja čvorova (skidanje sa stoga) je kao na slici
- Veličina prostora rješenja raste eksponencijalno s veličinom problema n
- Zbog toga dobar algoritam strategije povlačenja nikad ne generira cijelo stablo rješenja, nego odustaje od grana (podstabala) za koje uspije utvrditi da ne vode do rješenja
- U postupku generiranja čvorova provjeravaju se ograničenja koja n-torka mora zadovoljavati da bi zaista bila rješenje
- Čvor na i-tom nivou predstavlja n-torke u kojima je prvih *i* komponenti x_1, x_2, ..., x_i određeno; ukoliko se na osnovu vrijednosti tih *i* komponenti može utvrditi da ograničenja nisu zadovoljena, ne generira se ni taj čvor ni njegovi potomci jer ne vode na rješenje
- Npr. Ako se na gornji primjer stavi ograničenje da sve tri komponente rješenja moraju biti međusobno različite, tada je najlijevija generirana grana stabla rješenja ona za x_1=1, x_2=2, x_3=3, i ukupni broj generiranih čvorova je 19

- Strategija povlačenja se često rabi u rješavanju problema optimizacije, gdje se od svih mogućih rješenja traži ono koje je optimalno u nekom smislu
- Optimalnost se mjeri funkcijom cilja koju treba minimalizirati odnosno maksimalizirati; ovisno o problemu funkcija cilja se interpretira kao cijena, trošak, zarada i slično
- Ovakvi algoritmi odustaju od grana u stablu rješenja koja ne vode do rješenja, kod problema optimizacije odustaje se i od grana koje ne vode do boljeg rješenja od onog koje je već nađeno. Takva varijanta algoritma se obično naziva algoritam grananja i preskakanja (branch-and-bound)
- na primjer, ako se rješava problem minimalizacije, funkcija cilja se može interpretirati kao cijena koja se želi minimalizirati. Za realizaciju algoritma potreban je postupak određivanja donje ograde za cijenu po svim rješenjima iz zadanog podstabla
- Ukoliko je donja ograda veća od cijene najboljeg trenutno poznatog rješenja tada se razmatrano podstablo odbacuje, jer ne vodi do boljeg rješenja
- Za uspješnost takvog algoritma važno je što prije otkriti dobro rješenje (takvo koje ima dovoljno nisku donju ogradu da izbacuje veliki broj podstabala)
- Pri obradi čvorova braće prvo se obrađuje onaj koji ima najmanju donju ogradu

Problem n kraljica

- Problem: na šahovsku ploču veličine n x n polja treba postaviti n kraljica tako da se one međusobno ne napadaju
- Postupak rješavanja: očito svaka kraljica mora biti u posebnom retku ploče, pa se može uzeti da je i-ta kraljica u i-tom retku i rješenje problema se može prikazati kao n-torka (x_1, x_2, …, x_n), gdje je x_i indeks stupca u kojem se nalazi *i*-ta kraljica
- Slijedi da je skup rješenja S_i={1,2,…,n} za svaki *i*
- Broj n-torki u prostoru rješenja je n^n
- Ograničenja koja rješenje (x_1, x_2, …, x_n) mora zadovoljavati izvode se iz zahtjeva da se niti jedan par kraljica ne smije nalaziti u istom stupcu i istoj dijagonali
- Kao primjer razmotrimo n=4. Strategija povlačenja generira stablo rješenja prikazano na slici
- Crtkano su označeni čvorovi koje je algoritam odbacio odmah u postupku generiranja jer krše ograničenja
- Rad algoritma je prekinut čim je nađeno prvo rješenje



- Rješenje: (2, 4, 1, 3)

1			

1			
*	*	2	

1			
		2	
*	*	*	*

1			
			2
*	3		

1			
			2
	3		
*	*	*	*

	1		

	1		
*	*	*	2

	1		
			2
3			
*	*	4	

- Algoritam dobiven strategijom povlačenja (backtracking) za problem n kraljica zapisan u C, pretpostavlja se da postoji globalno polje za zapisivanje rješenja (x_1, x_2, …, x_n)

```
#define MAXLEN
int x[MAXLEN];  /* prvi element polja s indeksom 0 se ne koristi */
```

- Kod za pomoćnu logičku funkciju place() koja provjerava da li se k-ta kraljica može staviti u k-ti redak i x[k]-ti stupac tako da je već postavljenih k-1 kraljica ne napada (dakle elementi polja 1,…,k-1 su već određeni)

```
int place (int k) {
 int i;
 for (i=1; i<k; i++)
   if ( (x[i]==x[k]) || (abs(x[i]-x[k])==abs(i-k)) ) return 0;
 return 1;
}
```

- Funkcija queens() ispisuje sva rješenja problema:

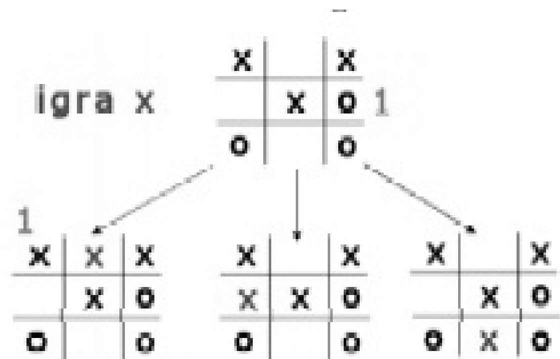
```
void queens(int n) {
int k, ind;
x[1]=0; k=1; /* k je trenutni redak, x[k] je trenutni stupac */
while (k > 0) { /* ponavlja za sve retke (kraljice) */
  x[k]++;
  while ( (x[k]<=n) && (place(k)==0) )  x[k]++; /* traži stupac */
   if (x[k]<=n)  /* nađen stupac */
     if (k == n) /* nađeno potpuno rješenje */
          for (ind=1; ind<=MAXLEN; ind++)
                         printf("x[%d]=%d ",ind,x[ind]);
          printf("\n"); }
     else {  /* traži sljedeći redak (kraljicu) */
        k++; x[k]=0;
      }
  else k--; /* vraća u prethodni redak */
 }
}
```

Igra križić - kružić

- Problem: zadana je tablica za igru križić – kružić na kojoj su već odigrani neki potezi. Treba utvrditi koji igrač ima strategiju koja vodi do pobjede, te koji potez je optimalan potez igrača koji je na redu – na koje polje treba staviti svoju oznaku da bi pobijedio ako je to moguće ili odigrao neriješeno ako je moguće
- Neka je na potezu igrač "x". Ako postoji način da on pobijedi, toj situaciji se dodijeli oznaka 1. Ako ne može pobijediti, ali može igrati neriješeno, situacija se označava s 0, a ako gubi kako god igrao situacija se označava s -1
- slično, ako je na potezu "o", situacija u kojoj on može pobijediti se označava s -1, kada može igrati najviše izjednačeno s 0 i kada gubi s 1
- Primjer: zadana je ovakva situacija:

- Na potezu je križić, ima tri polja na koja može igrati

x		x
	x	o
o		o

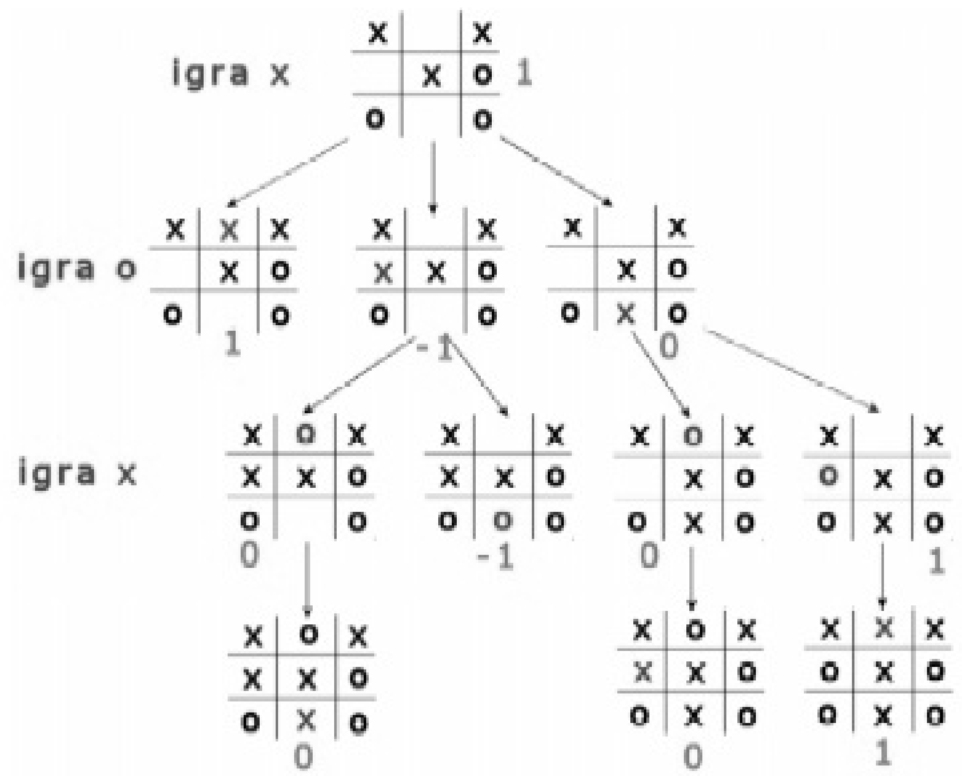


Dobije se stablo situacija, lijevo dijete trivijalno ima oznaku 1, jer je pobjeda za x. Za ostalu djecu se još ne znaju oznake, ali se zna da korijen ima oznaku 1, jer je x na potezu i može pobijediti

Ako x ne odigra svoj pobjednički potez, moguće su ove situacije:

Desno dijete je dobilo oznaku 0, jer je na potezu o i ako on odigra prvi potez, rezultat je neriješen, ako odigra drugi, pobjeđuje x

Srednje dijete dobija oznaku -1, jer ovisno o tome što igra o, x gubi ili igra neriješeno



- Iz ovog je jasno da je oznaka svakog čvora u kojem je na potezu “o” jednaka minimumu svih oznaka djece tog čvora i analogno ako je na potezu “x” oznaka čvora je maksimum svih oznaka djece čvora
- Dakle, algoritam rješavanja je: da bi se odredilo tko u zadanoj situaciji pobjeđuje treba konstruirati stablo svih situacija dostižljivih iz zadane situacije; korijen stabla je zadana situacija, a djeca svakog čvora su situacije do kojih se dolazi u jednom potezu iz tog čvora
- Težina utvrđivanja pobjednika pada što je nivo čvora u stablu veći
- Listovi stabla su trivijalne situacije – završetak igre
- Za utvrđivanje oznake u čvoru u kojem je na potezu “x” potrebno je naći maksimum svih oznaka djece tog čvora; za određivanje oznake kad je na potezu “o” potrebno je naći minimum svih oznaka djece tog čvora
- Konstrukcija takvog stabla radi se rekurzivnim algoritmom, pseudo-kod za rješavanje problema je slijedeći
- ulazne varijable su: tablica – djelomično popunjeno polje za igru i na_potezu – oznaka igrača koji je na potezu; izlazna varijabla je igrati – optimalan potez kojeg igrač na potezu može odigrati i funkcija vraća oznaku čvora u stablu koji odgovara tablici T

```
int krizic-kruzic(tablica T, char na_potezu, potez *igrati) {
 tablica dijete;
 potez ptemp;
 int vrijednost_djeteta, temp;
 if (T je list)                    /* ploča je puna, funkcija pobjednik() vraća 1 ako pobjeđuje x, */
   return pobjednik(T);     /*  0 za neriješeno, -1 za pobjedu o */
 else {
    if (na_potezu == ‘x’)
        vrijednost_djeteta = -beskonačno;
    else
       vrijednost_djeteta = +beskonačno;
  za svaki legalan potez move u tablici T radi  {
     if (na_potezu == ‘x’) {
        dijete = T + na mjestu move je stavljen ‘x’;
        temp = krizic-kruzic(dijete, ‘o’, &ptemp);
           if (temp > vrijednost_djeteta) {
             vrijednost_djeteta = temp;
             *igrati = move;
           }  }
```

```
         else {
          dijete = T + na mjestu move je stavljen ‘o’;
          temp = krizic-kruzic(dijete, ‘x’, &ptemp);
          if (temp < vrijednost_djeteta) {
             vrijednost_djeteta = temp;
             *igrati = move;
       }  }
   }
 return vrijednost_djeteta;
}
```

- Ako se ovaj algoritam pokrene na praznoj tablici dobije se da je oznaka korijena 0, tj. niti jedan igrač nema pobjedničku strategiju (veću šansu za pobjedu), nego igra završava neriješeno ako oba igrača igraju pametno