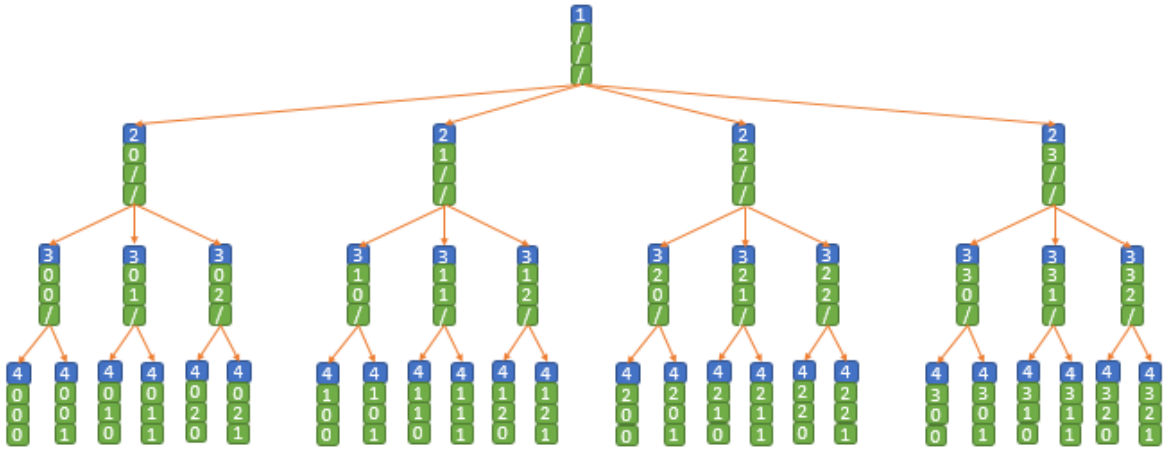
Backtracking

Bactracking ili „pretraživanje sa vraćanjem“ se odnosi na prolaženje kroz sva stanja (najčešće rekurzivno), pri čemu se prekida sa pretraživanjem pre kraja ukoliko se ustanovi da nema svrhe pretraživati dalje.

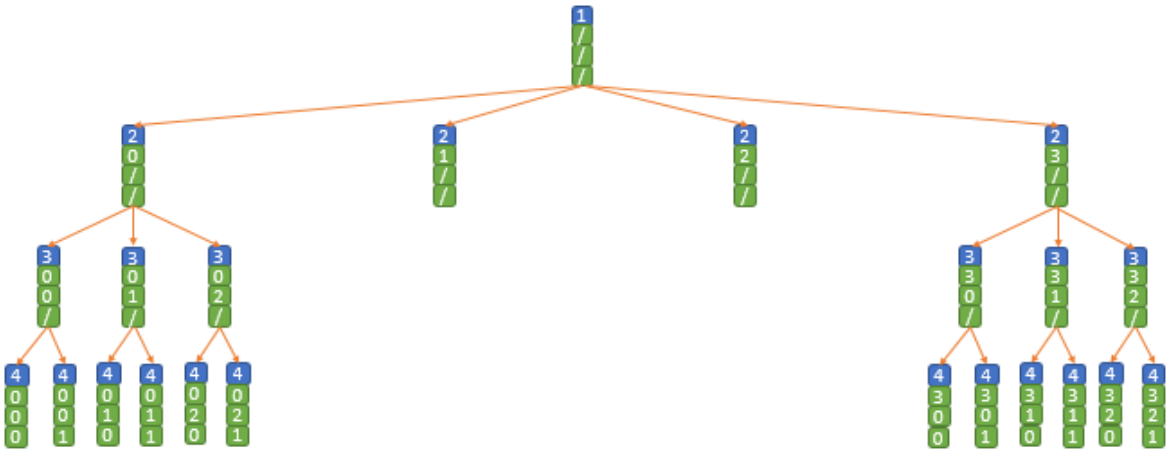
Primer backtracking-a ste mogli videti u lekciji „Rekurzija“ u zadatku gde se 8 dama postavlja na šahovsku tablu. U ovoj lekciji ćemo pre svega videti kako neke optimizacije mogu da utiču na vremensko izvršavanje programa.

Već ste radili u lekciji „Rekurzija“ primer uglježdenih petlji, gde prva for petlja ide od 0 do 2, druga od 0 do 3, a treća od 0 do 4. Na slici 1 možete da videti vizuelno kako radi rekurzija na ovom primeru.



Slika 1

Možemo primetiti da dobijamo 24 trocifrena broja, gde su vodeće nule dozvoljene. Recimo da nam od tih 24 broja trebaju samo oni gde je prva cifra 0 ili 3. Jedna mogućnost je da ponovo prođemo kroz svih 24 brojeva i da posle proverimo da li je prva cifra 0 ili 3, međutim efikasnije je da prekinemo rekurziju u trenutku kada shvatimo da nema svrhe da pretražujemo dalje. U ovom primeru kada postavimo prvu cifru i ukoliko ta cifra nije 0 ili 3, nema potrebe da ulazimo dalje u rekurziju, jer već u tom trenutku znamo da nam to ne predstavlja rešenje. Na slici 2 možemo videti da se broj stanja kroz koja prolazimo u rekurziji smanjio za 18. Ovde smo upotrebili jednu optimizaciju, tj. prekinuli smo sa pretraživanjem u trenutku kada smo shvatili da nema svrhe da nastavljamo iz tog stanja, te smo ubrzali našu pretragu.



Slika 2

Osvnovna dva pravila backtracking-a su:

1. Ne radi ništa dva puta
2. Ne radi ništa od čega nećeš imati koristi

Ta unapređenja koja nam ubrzavaju program, tj. smanjuju broj pretraženih stanja zovemo optimizacije. Treba se držati i pravila da što pre se može prekinuti sa pretraživanjem to bolje, tj. bolje je prekinuti sa pretraživanjem čim se može uočiti da nema svrhe pretraživati iz tog stanja nadalje.

Pseudo kod za backtracking bi mogao izgledati ovako:

```
============================================================
01     function Backtracking( currentState )
02            if NotWorthSearching( currentState ) then
03                   return
04            end if
05
06            if IsSolution( currentState ) then
07                   OutputSolution( currentState )
08            end if
09
10            while ExistNextState( currentState ) do
11                   nextState = NextState( currentState )
12                   Backtracking( nextState )
13            end while
14     end function
============================================================
```

U ovoj lekciji ćemo proći detaljno kroz jedan zadatak i videti kako optimizacije utiču na brzinu rada programa.

Zadatak. Potrebno je postaviti n kraljica na šahovsku tablu dimenzija $n \cdot n$, tako da se nikoje 2 ne napadaju.

Rešenje. U lekciji “Rekurzija” ste mogli da uočite te optimizacije, ovde ćemo videti koliko stvarno svaka od tih optimizacija ubrzava naš program. Vremena se računaju na računaru sa procesorom „Intel B960

2.2GHz, 2MB L3 cache“ i 6 GB DDR3 memorija. Svaki put kad vreme izvšavanja postane manje od 0.1 sekundu povećaćemo n, kako bismo bolje uvideli ubrzanje koje dobijamo nekom optimizacijom.

Osnovno rešenje. Rekurzivno postavljamo kraljice, svaku novu kraljicu pokušamo da postavimo na prvo slobodno polje, pa na sledeće slobodno polje i tako dalje, te kada postavimo sve kraljice onda proverimo da li se neke 2 međusobno napadaju.

Vreme izvšavanja za primer $n = 6$ je 73.462 sekundi

Optimizacija 1. Primetimo da ovde svaku postavku kraljica brojimo $n!$ puta, a prvo pravilo backtracking-a kaže da ništa ne radimo više od jednom. Sada ćemo sledeću kraljicu u rekurziji postavljati samo na polja posle prethodne kraljice, tj. u istom redu ali u narednim kolonama, ili u sledećim redovima.

Vreme izvršavanja za primer $n = 6$ je 0.151 sekundi, a za $n = 7$ je 4.734 sekundi

Primetimo da smo sa ovom jednostavnom optimizacijom dobili ubrzanje od skoro 500 puta, na primeru gde je $n = 6$.

Optimizacija 2. Prvom optimizacijom smo izbegli problem brojanja istog rešenja više puta, međutim radimo stvari koje nam ne mogu biti od koristi, tako dešava se da u neki red ne postavimo ni jednu kraljicu, ili stavljamo vise kraljica u jedan red. U svakom redu mora da bude tačno jedna kraljica, te u rekurziji kada postavljamo kraljicu, moramo da je postavimo tačno u sledeći red od prethodne.

Vreme izvšavanja kada je $n = 7$ je 0.142 sekundi, dok za $n = 8$ program radi 1.331 sekundi

Optimizacija 3. Do sada smo proveravali pozicije kraljica samo na kraju, tj. kad već postavimo svih n kraljica, međutim jedno od pravila je da treba prekinuti pretragu čim se može zaključiti da nam trenutno rešenje ne odgovara. Posle svake postavljene kraljice, treba proveriti da li se neke 2 kraljice međusobno napadaju.

Vreme izvšavanja programa za primer $n = 8$ je 0.036 sekundi, a za $n = 12$ je 4.949 sekundi

Optimizacija 4. Primetimo da kada postavimo neku kraljicu, dovoljno je da proverimo samo da li ona napada neku od prethodno postavljenih kraljica.

Vreme izvšavanja programa za primer $n = 12$ je 0.977 sekundi

Optimizacija 5. Ukoliko je jedna kraljica na polju (r_1, c_1), a druga na polju (r_2, c_2), jasno je da ne sme da bude $r_1 = r_2$ jer se kraljice nalaze u istom redu, međutim ovo ne moramo da proveravamo jer se ne može desiti uzimajući u obzir kako postavljamo kraljice rekurzivno. Dve kraljice su u istoj koloni ukoliko važi $c_1 = c_2$. Kraljice se napadaju dijagonalno ukoliko važi ili $r_1 - c_1 = r_2 - c_2$ ili $r_1 + c_1 = r_2 + c_2$. Iz pomenutog možemo da zaključimo da proveru da li postavljena kraljica napada neku od prethodno postavljenih možemo da uradimo samo proverom postojanja vrednosti $c, r - c$ i $r + c$, što možemo čuvati u jednom nizu logičkih promenljivih.

Vreme izvršavanja programa kada je $n = 12$ je sada 0.261 sekundi, a u primeru kada je $n = 13$ program radi 1.137 sekundi.

Uz 5 optimizacija smo uspeli da od programa kojem bi trebali vekovi da završi izvšavanje na primeru kada je $n = 13$, napravimo program koji to rešava za svega 1.137 sekunde.

Uz pametno kucanje programa i simuliranje listi, o kojima će biti reči u nekoj od narednih lekcija, preko nizova, moguće je optimizovati program da za primer kad je $n = 12$ vremensko izvršavanje bude 0.146 sekundi, dok za $n = 13$ vreme izvršavanje je 0.493 sekundi. Probajte da vidite koliko vaš program može brzo da radi na ovim primerima.

Važno je napomenuti da kod problema gde postoji algoritam bolje složenosti koji rešava taj problem, treba uvek kucati taj algoritam, a ne pokušavati sa optimizacijama da se ubrza algoritam lošije složenosti.