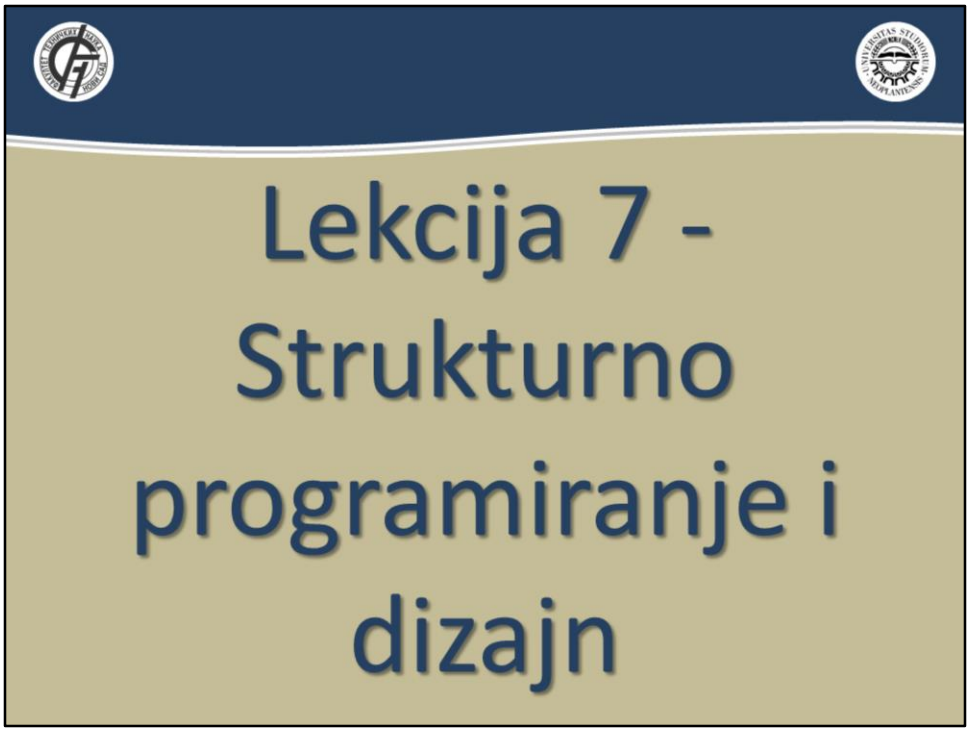
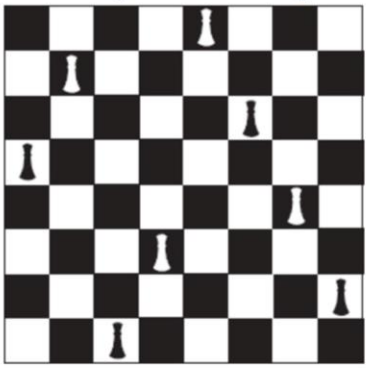

Lekcija 7 -
Strukturno
programiranje i
dizajn

Strukturno programiranje

- Dijkstrin čuveni rad o štetnosti korišćenja **goto** naredbe.
- Većina savremenih programskih jezika ima bogat skup konstrukcija (`if-then-else`, `while` petlju, `do` petlju, `switch` iskaz, itd.) za kontrolu toka izvršenja programa bez bezuslovnih skokova.
- Sistematsko razlaganje složenog problema na manje sastavne delove (*odozgo na dole* pristup) i oslanjanje na jezike koje demotivišu bezuslovne skokove predstavlja strukturno programiranje.
- Formalizovan od strane Wirta u obliku tzv. *stepwise refinement* metode. Može biti primenjen po principu *odozgo na dole* i/ili *odozdo na gore*. Prilikom svake iteracije se detaljnije opisuju podaci i interfejsi, a polazni problem razbija na manje delove. Koristi se uglavnom pseudo-kôd za opise. Kriterijum za prestanak raščlanjivanja na manje delove može biti i Milerovo 7±2 pravilo za podatke.

Primer *stepwise refinement* metode

- Problem 8 dama na šahovskoj tabli. Jedno moguće rešenje je:



- Ne postoji za sada analitičko rešenje problema

Rešenje 1

- *Brute-force* metoda (sirova snaga) – ukupan broj kombinacija je 2^{32} što je za današnje računare ok. Algoritam je sledeći (pseudo-kôd na engleskom):

```
1. Generate the set A of all board configurations
2. while there are still untested configurations in A do
3.     x = the next configuration from A
4.     if (q(x) == true) then print x and exit
5. end
```

Napomena: funkcija `q(x)` testira da li je konfiguracija `x` validna. Ako se proces ne zaustavi nakon ispisa prvog rešenja tada će naći i sve ostale validne konfiguracije.

Iako je *brute-force* metoda u većini slučajeva neadekvatna, često se i za veoma složene probleme koristi kao test. Naime, na jednom smanjenom skupu ulaza se pokrene *brute-force* koncipiran program da bi dobili rešenje. Kasnije se to rešenje koristi za testove optimizovane metode. Na taj način, iako metoda sirove snage nije direktno upotrebljiva, može se koristiti kao katalizator za naprednije varijante.

Sa aspekta *top-down* pristupa vidimo da je originalni problem razložen na dva suštinska potproblema: generisanje svih mogućih konfiguracija i testiranje validnosti pojedinačne konfiguracije. Petlja kojom prolazimo kroz skup nije od značaja.

Rešenje 2

- Poboljšana *brute-force* metoda sa redukovanim skupom A
- Moguće je uočiti pravilo da unutar jedne kolone možemo imati samo jednu damu. To omogućuje da smanjimo broj potencijalnih konfiguracija na 2^{24}
- Ostatak algoritma je apsolutno isti kao kod prvog rešenja.

Neko bi rekao da smanjivanje sa 2^{32} na 2^{24} nije zanemarljivo, pa se stoga rešenje 2 ne može smatrati *brute-force*-om. Međutim, pošto nije primenjena nikakva sofisticirana tehnika (nema kvalitativno novog pristupa) u odnosu na prvobitno rešenje, autor i ovakva rešenja smatra za *brute-force*.

Međutim, čak i ovakva poboljšanja mogu biti od velike koristi. Naime, imaju kapacitet da ukažu u kom pravcu treba dalje tragati. Rešenje 2 nam je nagovestilo da ako uložimo više truda u smanjivanje broja konfiguracija možemo postići značajna ubrzanja.

Rešenje 3/1

- Suštinsko poboljšanje je rad sa parcijalnim konfiguracijama umesto da se prvo izgeneriše čitav skup kandidata. Ovo je osnova cele jedne klase algoritama koje se naziva *trial-and-error + backtracking*.

```
do {
    while ((row < 8) && (col < 8))  {
        if (the current queen is safe) then
            trial: keep the queen on the board and advance to the next col.
        else
            error: the queen is not safe, so move up to the next row.
    }
    if (we've exhausted all the rows in this column) then
        backtrack: retreat a column,
                   move its queen up a row, and start again.
} while ((col < 8) && (col >= 0));

if (we've reached column 8) then
    we have a solution, print it.
```


Ovde vidimo iterativno rešenje. Ovaj isti algoritam se može još lepše iskazati korišćenjem rekurzije, ali to predstavlja ozbiljan problem sa aspekta performanse (ovde mislimo na nefunkcionalne programske jezike).

Iako se algoritam u ovoj formi može ručno verifikovati, dosta detalja je potrebno podrobnije definisati za prevođenje u neki konkretni programski jezik. Drugim rečima, mnogo detalja je još na suviše visokom nivou. U sledećoj rundi će se neki detalji iskristalisati. Ovo je suština *stepwise refinement* metode.

Rešenje 3/2

- Opisati način provere validnosti dame na datoj poziciji. Treba samo pregledati da li ima dame u tekućem redu u prethodnim kolonama i proveriti dijagonale. Treba primetiti da je razlika reda i kolone za polja po dijagonali sleva udesno je konstanta, kao i suma reda i kolone za suprotnu dijagonalu.
- Razmisliti o strukturama podataka, prvenstveno kako predstaviti tablu.

```
public boolean isSafe (int[ ] board) {
    boolean safe = true;
    for (int i = 0; i < col; i++) {
        if ( ( ( board[i] + i) == (row + col) ) ||  // down diagonal test
             ( ( board[i] - i) == (row - col) ) ||  // up diagonal test
               ( board[i]  == row) )                // row test
            safe = false;
    }
    return safe;
}
```

Elektroenergetski softverski inženjering – Razvoj EE softvera - 2016

7

Tabla je predstavljena celobrojnim nizom **`int board[8]`**, gde indeks predstavlja kolonu a vrednost red postavljene dame. Pošto koristimo pracijalne konfiguracije i počinjemo od prve kolone (sa indeksom nula) onda ne treba brinuti kako predstaviti prazna polja.

Vidimo da je metoda isSafe opisana u Java jeziku (ovde je odabran trenutak kada se sa pseduo-kôda prešlo na ciljni jezik).

Ostali detalji iz prethodnog opisa su veoma jednostavni i mogu se direktno prevesti u ciljni jezik. Treba primetiti samo da kod backtracking-a pomeranje unazad može biti na više nivoa, tj. ne samo do prethodne kolone već skroz do početne kolone.

NAPOMENA:

U izvornom kôdu knjige NQueens.java postoji ozbiljna greška. Šta će se desiti ako se unese veličina polja 3? Takođe, da li je korektno uvećavati totalcount promenljivu unutar metode isSafe? Koji princip je ovde povređen?

Modularna dekompozicija

- Komplementarna tehnika razlaganja inicijalnog problema na sastavne delove. Umesto da se isključivo prati kontrola toka izvršenja može se koristiti i kriterijum skrivanja informacija i enkapsulacija. Na taj način se inicijalni problem razlaže na module, gde je svaki zadužen za određene podatke i operacije nad tim podacima.
- Tri ključne karakteristike modularnog dizajna su:
  - enkapsulacija
  - slaba sprega i čvrsta kohezija
  - skrivanje informacija (čak i dizajn odluka vezanih za modul)

Kod modulranog dizajna tzv. *separation of concerns* se primenjuje ne samo na funkcionalnost već i na podatke.

Vrste sprega između modula

- Jednostavna sprega preko nestrukturiranih podataka
- Sprega preko strukturiranih podataka
- Sprega preko kontrole
- Sprega preko globalnih podataka

Primer: Keyword in Context (KWIC) programa

- Opis problema: na osnovu dve ulazne datoteke (jedna sadrži reči koje treba ignorisati, a druga tekst koji treba indeksirati) kreirati permutovani indeks (KWIC). Za svaku liniju teksta cirkularno odabrati reči indeksa, i sortirano ih prikazati, tako da se svaka reč indeksa prikaže samo jednom za datu liniju. Ako se isti indeks koristi u više linija, tada linije poređati po hronološkom redosledu (onako kako su zapisani u ulaznom fajlu).
- *Top-down* dekompozicija bi mogla biti sledeća:
  1. Uneti reči koje treba ignorisati i linije teksta koje treba indeksirati
  2. Kreirati strukturu podataka za smeštaj cirkularno pomerenih linija teksta sa podatkom koji indeks referencira koju liniju
  3. Sortirati sadržaj prethodne strukture podataka po indeksu
  4. Formatirati izlaz
  5. Ispisati permutovani indeks

Ovde treba primetiti da se mora koristiti stabilna sort metoda.

Modularna dekompozicija KWIC programa

- Sledeći moduli (klase u našem slučaju) se mogu identifikovati na bazi principa skrivanja informacija i enkapsulacije:
  - `Line` klasa za smeštaj jedne cirkularno šiftovane verzije ulazne linije
  - `Index` klasa za uređeno skladištenje Line instanci uzimajući u obzir reči koje treba ignorisati
  - `Print` klasa za formatirani ispis permutovanog indeksa
  - `MControl` klase za kontrolu izvršenja celog programa (praktično orkestrira instance ostalih klasa)

NAPOMENA:

Koja se implicitna pretpostavka krije u implementaciji Index klase (videti metodu add i promenljivu words)? Šta je osnovni problem kod ovakvog pristupa? Koliko je intuitivna upotreba klase PriorityQueue za stabilno sortiranje linija na osnovu ključne reči (indeksa)?

Pogotovo je instruktivna (u negativnom smislu) analiza Print klase i načina kako se osigurava da se destruktivno iščitavanje elemenata iz prioritetnog reda nekako restaurira na kraju. Ovako nešto se ne sme raditi u praksi.

Šta se dešava sa linijama koje sadrže istu reč više puta? Da li je korektno kreirati više puta indeks za jednu te istu reč unutar iste linije?