UVOD U DIZAJN I ANALIZU ALGORITAMA

UVOD U DIZAJN I ANALIZU ALGORITAMA …

- Prilikom pisanja algoritama posebna pažnja se posvećuje njihovoj ispravnosti i efikasnosti
- Ispravnosti (korektnosti) algoritma se posebna pažnja posvećuje tokom njegovog dizajna
- Ovo je naročito istaknuto kod složenih algoritama kod kojih korektnost nije lako uočljiva
- Cilj analize algoritma je kvantitativna ocena efikasnosti algoritma, najčešće spram veličine računarskih resursa (vreme i memorija kao najznačajniji)
- Efikasnost algoritma umnogome zavisi od količine i strukture ulaznih podataka za algoritam

… UVOD U DIZAJN I ANALIZU ALGORITAMA …

- Postupak dobijanja dobrih algoritama, u opštem slučaju, za većinu problema obuhvata

1. Analizu problema – traženje odgovora na pitanje šta i sa čim, identifikacija šta je to što algoritma treba da uradi ili reši.
   Uključuje identifikaciju ulaznih, izlaznih podataka i promenljivih, kao i podataka/promenljivih koje se koriste u među-koracima.

2. Dekompozicija problema. Eliminisanje svih nevažnih i remetilačkih detalja praktičnog problema koji se rešava i dobijanje problema u što čistijem matematičkom obliku.
   Problem se predstavlja u terminima prostih matematičkih pojmova, kao što su skupovi, funkcije, grafovi i slično

… UVOD U DIZAJN I ANALIZU ALGORITAMA

- Postupak dobijanja dobrih algoritama, u opštem slučaju, za većinu problema obuhvata (nastavak)

3. Piše se algoritam koji rešava jasno definisan matematički problem. Konstantno tokom pisanja vrši se provera ispravnosti pojedinačnih delova algoritma

4. Dokazuje se njegova ispravnost upotrebom poznatih algoritamskih paradigmi i matematičkih alata

5. Sprovođenje analize algoritma kako bi se odredila njegova efikasnost. Primenjuju se tehnike za pojednostavljivanje problema čime se olakšava analiza

- Ovi koraci se međusobno prepliću i nisu nezavisni jedni od drugih
- Takođe, u nekim slučajevima nisu potrebni svi navedeni koraci

UVOD U DIZAJN ALGORITAMA

UVOD U DIZAJN ALGORITAMA

- Dizajn složenih problema zahteva od programera visok stepen obrazovanja, intuicije i iskustva. Takođe, važno je još dobro poznavati domen problema što nekada zahteva i dobro poznavanje matematike
- Međutim, ni svo znanje sveta ne garantuje uspeh, jer je rešavanje problema i dizajniranje algoritama vrlo kreativan posao za koji ne postoji propisan šablon
- Pomoć se ogleda u tome da za različite klase problema postoje standardne metode čijom primenom se može doći do rešenja, ali i do dobrih, efikasnih, algoritama
- U ove opšte metode spadaju
  - inkrementalni pristup
  - rekurzivni metod
  - pohlepni metod
  - linearno programiranje
  - randomizacija
  - dinamičko programiranje

INSERTION SORTIRANJE

PROBLEM SORTIRANJA …

- **Šta je to sortiranje?**
- Jedan od najšire proučavanih problema u računarstvu
- Sortiranje je osnova za mnoge algoritme i troši veliki deo računarskog vremena u mnogim tipičnim aplikacijama
- Postoje mnoge verzije problema sortiranja, kao i desetine algoritama za sortiranje
- Radićemo to detaljnije kasnije
- Retko kad je to samo niz brojeva već deo neke veće strukture
- Algoritmi za sortiranje su primer kako apstrakcijom svodimo problem na jednostavniju (matematičku) formu koju rešavamo algoritmom

… PROBLEM SORTIRANJA …

- Ulaz je nesortiran niz (brojeva) od n brojeva:

$$\langle a_1, a_2, \dots, a_n \rangle$$

- Izlaz je permutacija ulaznog niza

$$\langle a'_1, a'_2, \dots, a'_n \rangle$$

takva da je

$$a'_1 \leq a'_2 \leq \cdots \leq a'_n$$

… PROBLEM SORTIRANJA

- Primer:
- Ne sortiran niz:

$$[5\ 2\ 4\ 6\ 1\ 3]$$

- Sortiran niz:

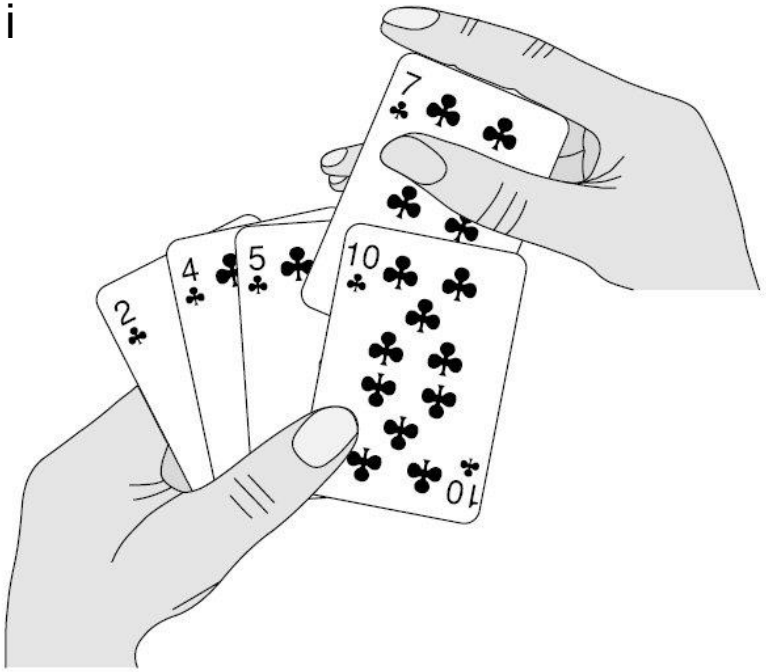
$$[1\ 2\ 3\ 4\ 5\ 6]$$

INSERTION SORTIRANJE …

- Sortiranje umetanjem
- Efikasan za male skupove podataka
- Radi po principu sortiranja karata u ruci

  1. Kreće se sa praznom levom rukom i kartama na stolu
  2. Uzima se jedna po jedna karta sa stola i stavlja se u levu ruku
  3. Kako se broj karata u ruci povećava, svaka nova karta se dodaje između karata sa neposredno manjom i neposredno većom vrednošću

- Kako opisani algoritam preneti na algoritam koji radi na računaru



… INSERTION SORTIRANJE …

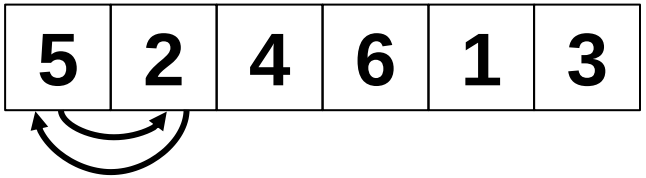
5 2 4 6 1 3

5	2	4	6	1	3

… INSERTION SORTIRANJE …

5 2 4 6 1 3

5 2 4 6 1 3

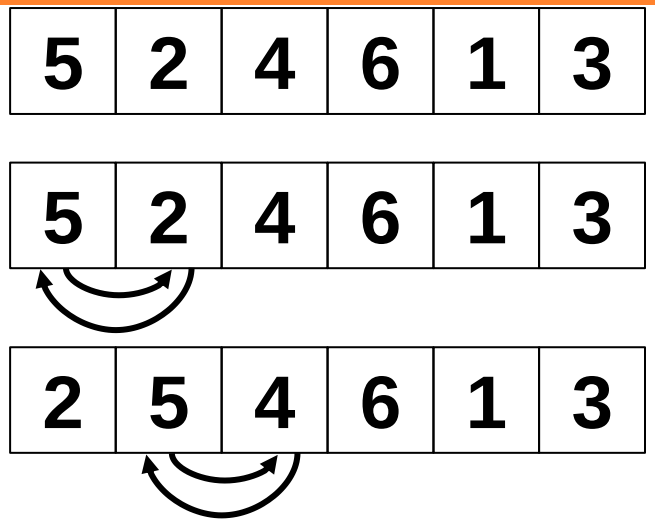


… INSERTION SORTIRANJE …

5 2 4 6 1 3

5 2 4 6 1 3

2 5 4 6 1 3



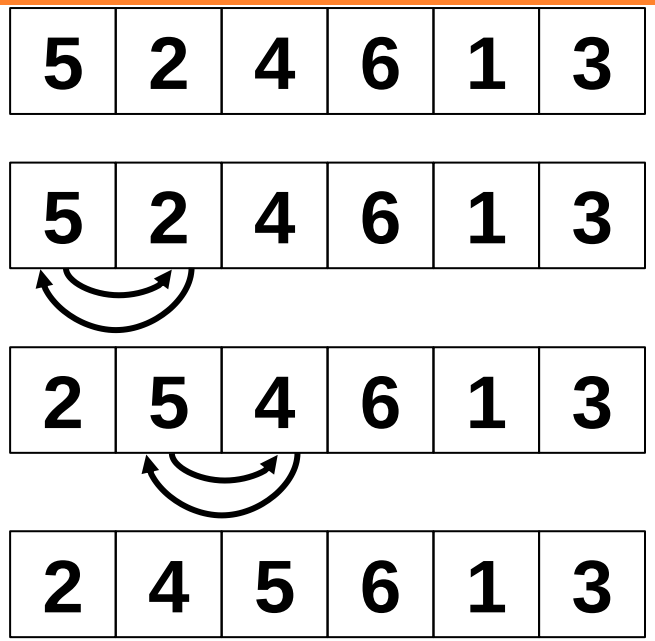
… INSERTION SORTIRANJE …

5 2 4 6 1 3

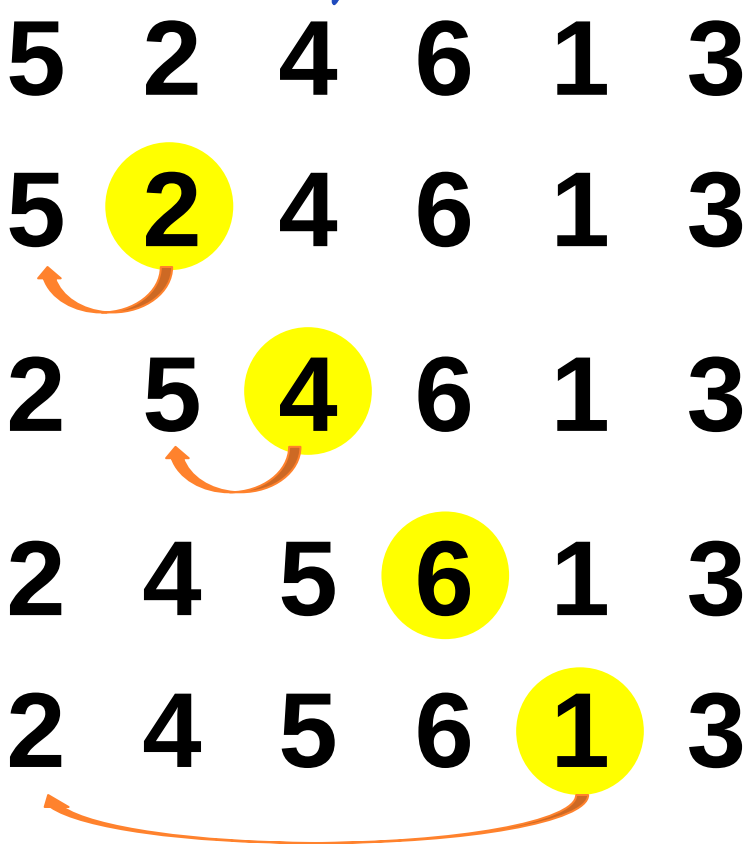
5 2 4 6 1 3

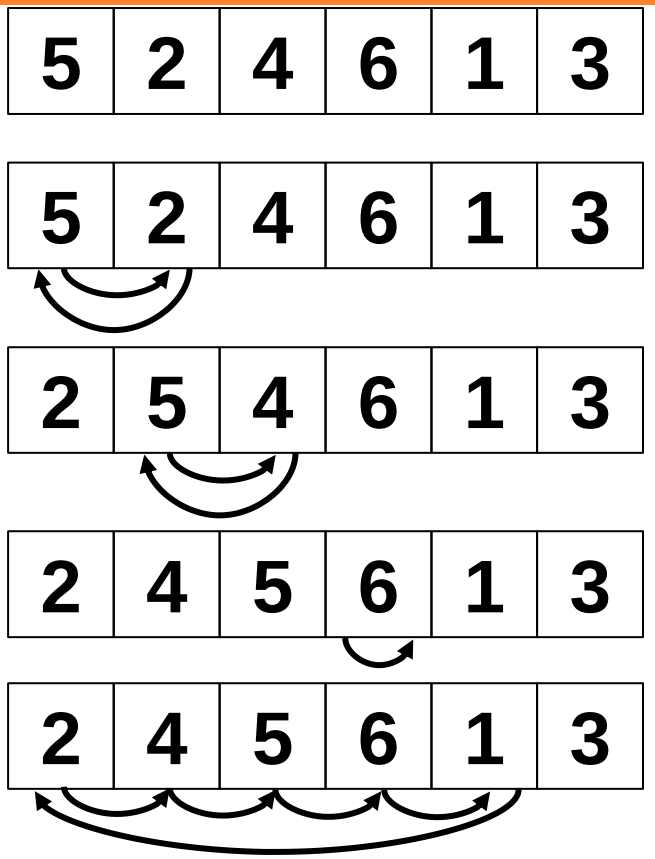
2 5 4 6 1 3

2 4 5 6 1 3

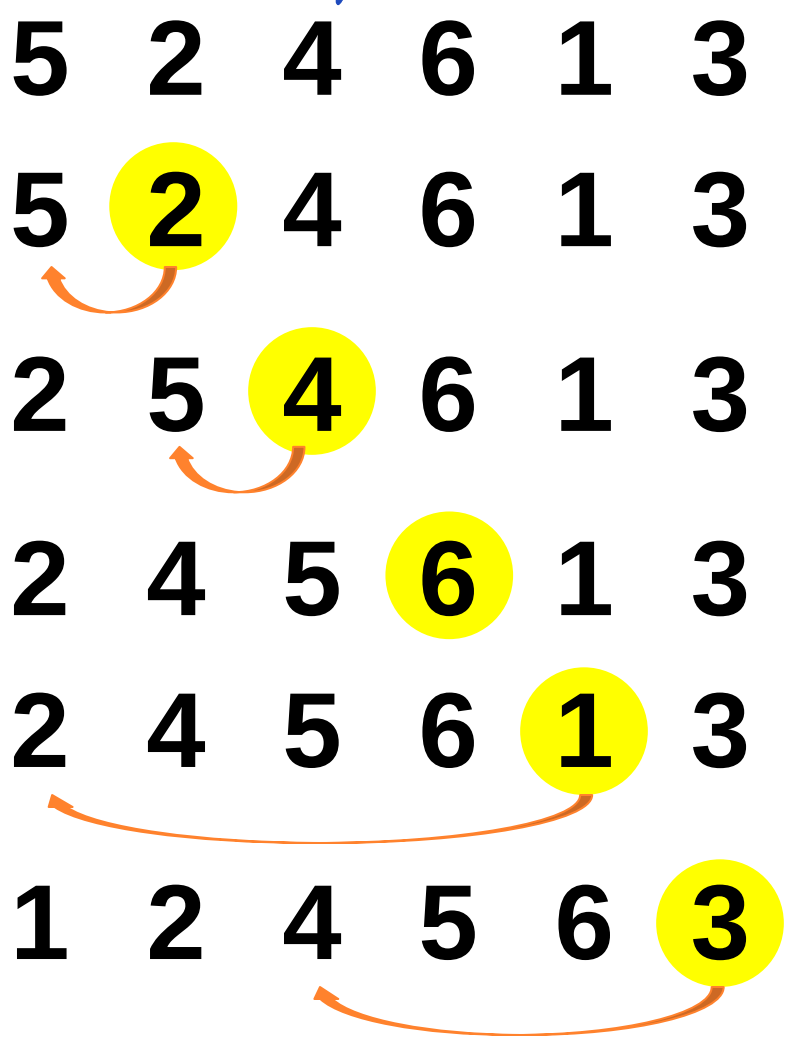


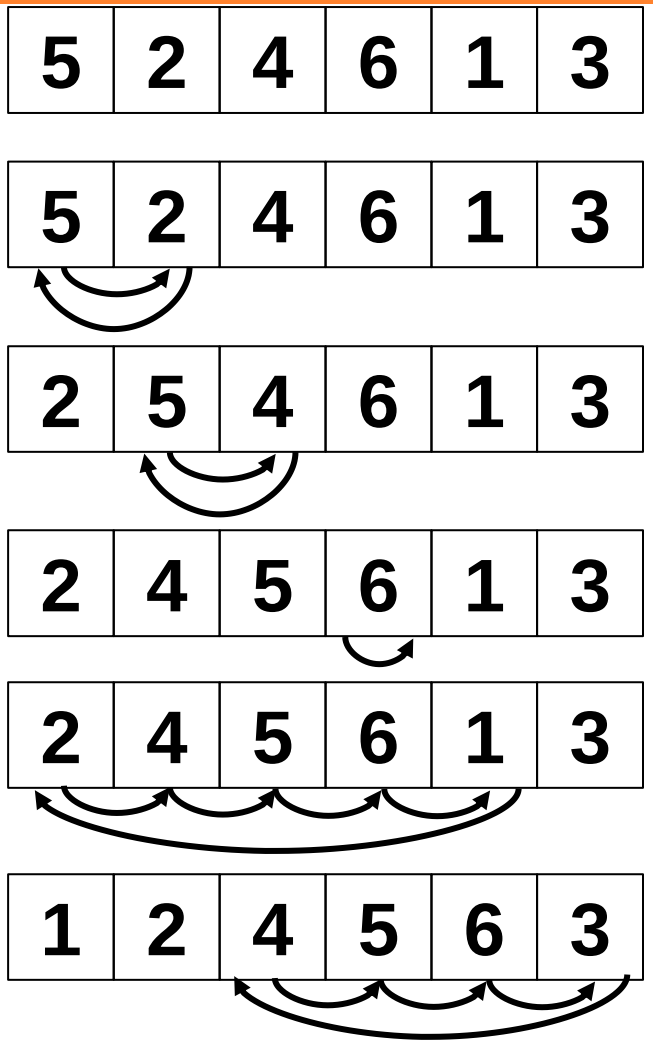
… INSERTION SORTIRANJE …



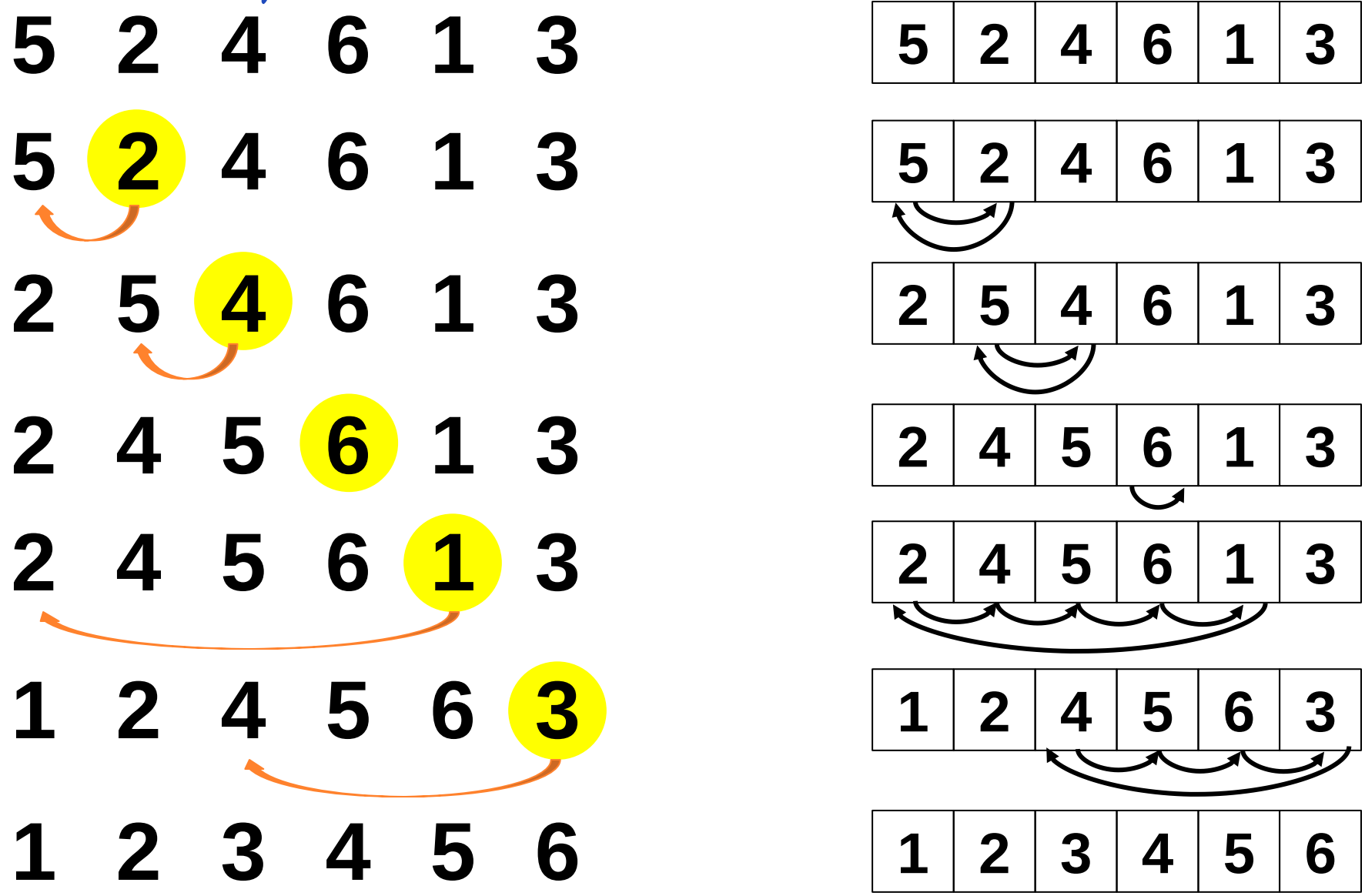


… INSERTION SORTIRANJE …

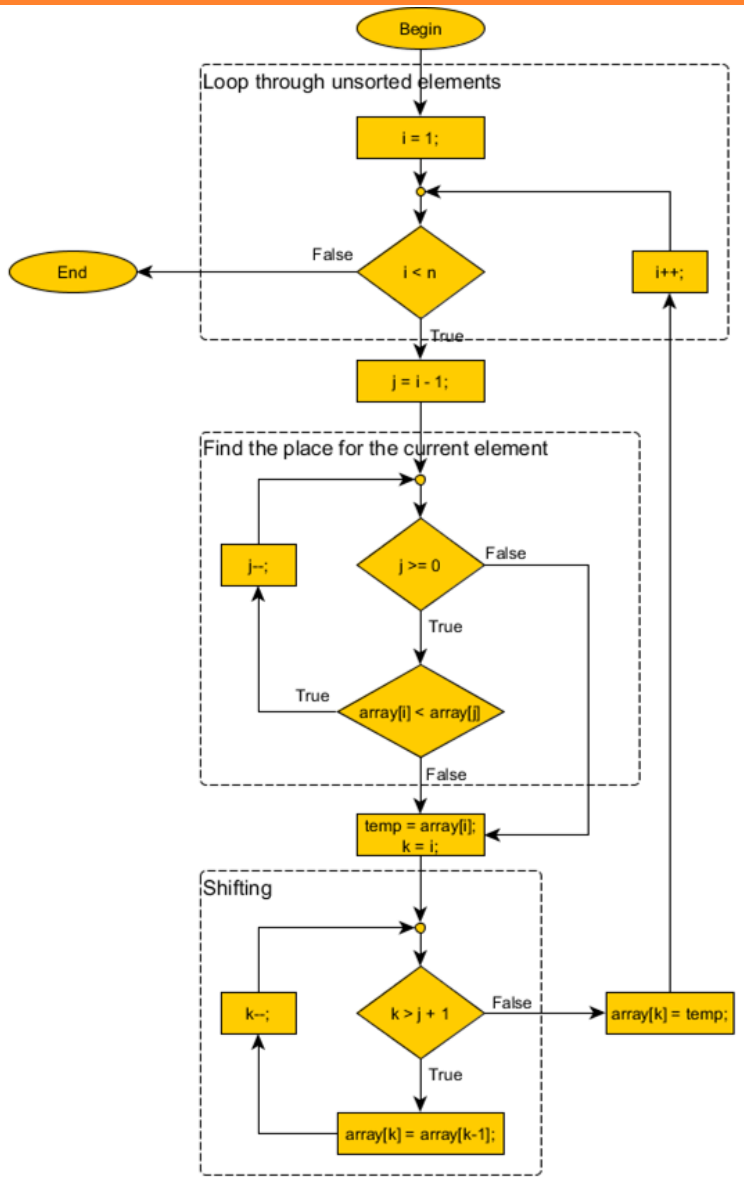




… INSERTION SORTIRANJE …



… INSERTION SORTIRANJE



PREDSTAVLJANJE ALGORITAMA

- Kako dizajnirati algoritam?
- Kako ga opisati?
  - Govornim jezikom?
  - Grafički?
  - Pseudokodom?
  - Strukturogram
- Na ovom kursu se koristi pseudukod iz knjige Introduction to Algorithms

PSEUDOKOD ZA INSERTION SORTIRANJE

- Algoritam za insertion sortiranje u pseudokodu iz knjige (detaljniji opis sintakse na narednim slajdovima)

```
INSERTION-SORT(A)
1: for j = 2 to A.length
2:     key = A[j]
3:     // Insert A[j] to the sorted
           sequence A[1..j − 1]
4:     i = j − 1
5:     while i > 0 and A[i] > key
6:         A[i + 1] = A[i]
7:         i = i − 1
8:     A[i + 1] = key
```

PSEUDOKOD KONVENCIJE

PSEUDOKOD KONVENCIJE …

- Uvlačenje predstavlja blok strukturu (npr. ***for*** petlja iz prvog reda obuhvata redove 2-8)
  - Ovo se odnosi i na ***if-else***
- Petlje:
  - **while**
  - **for**
  - **repeat-until**

  i selekcija:
  - ***if-else***
- imaju isto značenje i upotrebu kao u C, C++, Java, Phyton i Paskalu programskim jezicima

INSERTION-SORT(A)

1: **for** $j = 2$ **to** $A.length$
2: $key = A[j]$
3: // Insert $A[j]$ to the sorted sequence $A[1..j-1]$
4: $i = j - 1$
5: **while** $i > 0$ **and** $A[i] > key$
6: $A[i+1] = A[i]$
7: $i = i - 1$
8: $A[i+1] = key$

… PSEUDOKOD KONVENCIJE …

- Kod **for** petlje uobičajeno ja da brojanje krene od 1
  - Prva vrednost iteratora i posle petlje je ***n+1***, gde je ***n*** vrednost do koje se broji
  - Rezervisana reč ***to*** koristi se da označi da se vrednost iteratora povećava do ***n*** , dok ***downto*** označava da se vrednost iteratora umanjuje
  - Kada se vrednost iteratora povećava/smanjuje za više od faktora 1 onda se koristi rezervisana reč ***by***

```
INSERTION-SORT(A)
1: for j = 2 to A.length
2:     key = A[j]
3:     // Insert A[j] to the sorted
           sequence A[1..j − 1]
4:     i = j − 1
5:     while i > 0 and A[i] > key
6:         A[i + 1] = A[i]
7:         i = i − 1
8:     A[i + 1] = key
```

- Za komentar se koristi **//** oznaka

… PSEUDOKOD KONVENCIJE …

- Dozvoljena su višestruka dodavanja tipa ***i = j = e***, što znači da obe promenljive dobijaju istu vrednost ***e***
- Promenljive poput ***key***, ***i***, ***j*** su lokalne za datu proceduru
- Članovima niza se pristupa preko indeksa njihove pozicije, npr. ***A[i]***
  - ***..*** se koristi da označi opseg vrednosti, npr. ***A[1 .. j]***
- Strukturirani podaci smeštaju se u objekte sastavljene od atributa
  - Atributima se pristupa preko operatora ***.*** , npr. ***A.length***
  - Dozvoljeno je izjednačavanje tipa ***x = y***, tako da je posle toga ***x.a == y.a***
  - Za prazne objekte se koristi ***NIL***

```
INSERTION-SORT(A)
1: for j = 2 to A.length
2:     key = A[j]
3:     // Insert A[j] to the sorted
          sequence A[1..j − 1]
4:     i = j − 1
5:     while i > 0 and A[i] > key
6:         A[i + 1] = A[i]
7:         i = i − 1
8:     A[i + 1] = key
```

… PSEUDOKOD KONVENCIJE …

- Parametri se u funkciju/proceduru prenose po vrednosti
  - Jedino se objekti prenose po referenci (kao pokazivači u C-u)
  - I nizovi se prenose po referenci
- Vrednosti se vraćaju preko ***return*** naredbe
  - Kontrola se odmah vraća pozivaocu
  - Moguće je vratiti više vrednosti sa jednom ***return*** naredbom

```
INSERTION-SORT(A)
1: for j = 2 to A.length
2:     key = A[j]
3:     // Insert A[j] to the sorted
          sequence A[1..j − 1]
4:     i = j − 1
5:     while i > 0 and A[i] > key
6:         A[i + 1] = A[i]
7:         i = i − 1
8:     A[i + 1] = key
```

… PSEUDOKOD KONVENCIJE

- Boolean operatori, poput „and“ i „or“ rade po principu gledanja prve vrednosti, pa tek posle i druge
  - Tako u izrazu ***x and y***, prvo se evaluira ***x***
  - Ako je ***x false***, onda je ceo izraz netačan i ***y*** se više ne evaluira
  - Ako je ***x true***, onda se mora i ***y*** evaluirati

```
INSERTION-SORT(A)
1: for j = 2 to A.length
2:     key = A[j]
3:     // Insert A[j] to the sorted
          sequence A[1..j − 1]
4:     i = j − 1
5:     while i > 0 and A[i] > key
6:         A[i + 1] = A[i]
7:         i = i − 1
8:     A[i + 1] = key
```

- Rezervisana reč ***error*** koristi se za označavanje da je došlo do nekakve greške, jer su uslovi prilikom poziva procedure bili pogrešni i nisu trebali rezultovati pozivom
  - Pozivalac procedure je odgovoran za rukovanje greškom, te algoritmi neće implementirati njeno rukovanje

LOOP INVARIANT

LOOP INVARIANT

- Osobina koju ima algoritam za insertion sortiranje sa naziva **loop invariant** (nepromenljivost petlje),
  - Algoritam ima neku osobinu uvek istu, nezavisno od trenutka u kojem se nalazi
  - U ovom konkretnom slučaju, to znači da algoritam ne zavisi pozicije elementa u nizu koji se trenutno obrađuje
- Koju osobinu ima insertion sortiranje nezavisno od koraka u petlji? Koja je to osobina koja uvek veži bez obzira gde se nalazi u izvršavanju petlje?
  - Ako je ***j*** pozicija elementa niza koji se trenutno sortira, onda se sortirani podniz ***A[1 .. j-1]*** uvek sastoji od originalnih elemenata podniza ***A[1 .. j-1]*** , samo što su oni sada sortirani, bez obzira koja je vrednost indeksa ***j***
- Invarijantnost petlje se koristi za razumevanje i/ili dokazivanje ispravnosti algoritma

LOOP INVARIANT – DOKAZIVANJE

- Invarijantost petlje ima tri važne osobine
  - **Inicijalizacija (Initialization)** – da je invarijantnost tačna pre prvog izvršavanja petlje
  - **Održivost (Maintenance)** – da, ako je invarijantnost tačna tokom izvršavanja trenutnog ciklusa petlje, ostaje tačna i pre početka izvršavanja sledećeg ciklusa petlje
  - **Završavanje (Termination)** – da invarijantnost važi i po okončanju izvršavanja petlje, što je važno, jer pomaže u dokazivanju korektnosti algoritma (zašto?)
- Dokazivanje invarijantnosti liči veoma na matematičku indukciju
- Šta smo rekli, koja je pretpostavka za insertion sortiranje?
  - Ako je ***j*** pozicija el. niza koji se trenutno sortira, onda se sortirani podniz ***A[1 .. j-1]*** uvek sastoji od originalnih elemenata podniza ***A[1 .. j-1]*** , samo što su oni sada sortirani, bez obzira koja je vrednost indeksa ***j***

LOOP INVARIANT – INSERTION SORTIRANJE …

- **INICIJALIZACIJA (INITIALIZATION)**
  - Da li je invarijantnost tačna pre prvog izvršavanja petlje?
  - U prvom izvršavanju ***j=2***, ***A[1 .. j-1]*** se sastoji od samog jednog elementa
  - Kako se podniz ***A[1]*** sastoji samo od jednog elementa, on je po svojoj prirodi sortiran
  - To je ujedno i prvi element originalnog niza.
  - Ovaj uslov je, dakle, ispunjen

```
INSERTION-SORT(A)
1: for j = 2 to A.length
2:     key = A[j]
3:     // Insert A[j] to the sorted
          sequence A[1..j − 1]
4:     i = j − 1
5:     while i > 0 and A[i] > key
6:         A[i + 1] = A[i]
7:         i = i − 1
8:     A[i + 1] = key
```

… LOOP INVARIANT – INSERTION SORTIRANJE …

- **ODRŽIVOST (MAINTENANCE)**
  - Ako je invarijantnost tačna tokom izvršavanja trenutnog ciklusa petlje, da li ostaje tačna i pre početka izvršavanja sledećeg ciklusa petlje?
  - Telo ***for*** petlje funkcioniše tako što ide u nazad od ***j***, preko ***A[j – 1]***, ***A[j – 2]***, ***A[j - 3]***, i tako dalje (linije 4-7) sve dok ne pronađe pravo mesto za ***A[j]*** (linija 8)
  - Podniz ***A[1 .. j]*** se zbog toga sastoji od prvih ***j*** elemenata originalnog niza
  - Ali su oni sada sortirani
  - Povećanjem vrednosti iteratora ***j*** za 1 invarijantnost se održava (ovo formalno zahteva i dokaz za invarijantnost ***while*** petlje – **domaći!**)

```
INSERTION-SORT(A)
1: for j = 2 to A.length
2:     key = A[j]
3:     // Insert A[j] to the sorted
          sequence A[1..j − 1]
4:     i = j − 1
5:     while i > 0 and A[i] > key
6:         A[i + 1] = A[i]
7:         i = i − 1
8:     A[i + 1] = key
```

… LOOP INVARIANT – INSERTION SORTIRANJE

- **ZAVRŠAVANJE (TERMINATION)**
  - Šta se dešava kada se petlja završi?
  - ***for*** petlja prestaje sa izvršavanjem kada je ispunjen uslov ***j > A.length***, gde je ***A.legth = n***,
  - Kako se posle svake iteracije ***j*** uvećava za ***1***, onda i na kraju izvršavanja petlje ***j = n + 1***
  - Ako se ***j*** zameni sa ***n + 1***, onda u izrazu invarijantnosti petlje dobijemo podniz ***A[1 .. n]***, koji se sastoji od originalnih elemenata niza ***A[1 .. n]***, ali koji su sortirani
  - Kako je podniz ***A[1 .. n]*** zapravo ceo niz, zaključujemo da je ceo niz sortiran
  - DAKLE, ALGORITAM JE KOREKTAN

```
INSERTION-SORT(A)
1: for j = 2 to A.length
2:     key = A[j]
3:     // Insert A[j] to the sorted
          sequence A[1..j − 1]
4:     i = j − 1
5:     while i > 0 and A[i] > key
6:         A[i + 1] = A[i]
7:         i = i − 1
8:     A[i + 1] = key
```

LOOP INVARIANT – VEŽBANJE …

- Hajde da dokažemo zajedno invarijantnost petlje za dva veoma poznata algoritma koje ste već radili kada ste učili osnove programiranja
  - Traženje maksimalne vrednosti niza
  - Selection sortiranje
- A za domaći probajte:
  - Sumu elemenata niza

… LOOP INVARIANT – VEŽBANJE …

- **Traženje maksimuma**
  - Kako izgleda njegova implementacija?

```
MAX-ARRAY(A)
1:max = A[1]
2:for i = 2 to A.length
3:   if A[i] > max
4:      max = A[i]
```

  - Koja bi bila pretpostavka za traženje maksimuma?
    - u ***i***-oj iteraciji vrednost promenljive ***max*** je jednaka najvećoj vrednosti iz podniza ***A[1 .. i – 1]***

… LOOP INVARIANT – VEŽBANJE …

- **Traženje maksimuma – inicijalizacija**
  - Za ***i*** = 1, invarijantnost petlje je zadovoljena
  - podniz ***A[1]*** sadrži samo jedan element
  - Vrednost promenljive ***max*** jednaka je najvećoj vrednosti podniza ***A[1]***, tj. jednaka je vrednosti ***A[1]***
  - Dokaz je trivijalan, a invarijantost petlje je ispunjena

```
MAX-ARRAY(A)
1:max = A[1]
2:for i = 2 to A.length
3:   if A[i] > max
4:      max = A[i]
```

… LOOP INVARIANT – VEŽBANJE …

- **Traženje maksimuma – održivost**
  - Za ***i*** = ***n***, invarijantnost petlje je zadovoljena na sledeći način
  - Vrednost promenljive ***max*** na početku iteracije jednaka je najvećoj vrednosti podniza ***A[1 .. n-1]***
  - Postoje 2 slučaja:
    1. ***A[n] > max***; u tom slučaju dobijamo da je ***A[n]*** veća od svih vrednosti iz podniza ***A[1 .. n-1]***, što znači da je to ujedno i najveća od svih vrednosti iz podniza ***A[1 .. n]*** , samim tim promenljiva ***max*** dobija vrednost ***A[n]***, pa je invarijantnost petlje očuvana i na kraju ***n***-te iteracije, odnosno pre sledeće iteracija
    2. ***A[n] ≤ max***; u tom slučaju najveća vrednost iz podniza ***A[1 .. n-1]*** je makar jednaka ***A[n]*** , što znači da je maksimum podniza ***A[1 .. n]*** jednak maksimumu podnizna ***A[1 .. n-1]***, odnosno jednak promenljivoj ***max***, pa je invarijantnost petlje očuvana i na kraju ***n***-te iteracije, odnosno pre sledeće iteracija

```
MAX-ARRAY(A)
1:max = A[1]
2:for i = 2 to A.length
3:   if A[i] > max
4:      max = A[i]
```

… LOOP INVARIANT – VEŽBANJE …

- **Traženje maksimuma – završavanje**
  - U koracima inicijalizacije i održivosti pokazano je da je na kraju svake iteracija vrednost promenljive ***max*** jednaka najvećoj vrednosti odgovarajućeg podniza

```
MAX-ARRAY(A)
1:max = A[1]
2:for i = 2 to A.length
3:   if A[i] > max
4:      max = A[i]
```

  - Na kraju poslednje iteracije, podniz je zapravo sam niz, jer je podniz ***A[1 .. A.length]*** niz ***A***
  - Samim tim vrednost promenljive ***max*** jednaka je maksimumu niza ***A***, čime je dokazana korektnost algoritma

… LOOP INVARIANT – VEŽBANJE …

- **Selection sort**
  - Kako izgleda njegova implementacija?

```
SELECTION-SORT (A)
1:for i = 1 to A.length-1
2:   for j = i+1 to A.length
3:      if A[i] > A[j]
4:         tmp = A[i]
5:         A[i] = A[j]
6:         A[j] = tmp
```

  - Koja bi bila pretpostavka za selection sortiranje?
    - ako je ***i*** pozicija elementa niza koji se trenutno sortira, onda je podniz ***A[1 .. i-1]*** sortiran i bilo koji element podniza ***A[i .. A.length]*** *je jednak ili veći od bilo kog elementa podniza* ***A[1 .. i-1]***

… LOOP INVARIANT – VEŽBANJE …

- **Selection sort – inicijalizacija**
  - Za ***i*** = 1, invarijantnost petlje je zadovoljena
  - podniz ***A[1 .. 0]*** je sortiran, jer je on prazan
  - svi elementi niza ***A[i .. A.length]*** su veći od elementa podniza ***A[1 .. 0]***, jer je on prazan
  - dokaz je trivijalan

SELECTION-SORT (A)

```
1:for i = 1 to A.length-1
2:   for j = i+1 to A.length
3:     if A[i] > A[j]
4:       tmp = A[i]
5:       A[i] = A[j]
6:       A[j] = tmp
```

… LOOP INVARIANT – VEŽBANJE …

- **Selection sort – održivost**

SELECTION-SORT (A)

```
1: for i = 1 to A.length-1
2:   for j = i+1 to A.length
3:     if A[i] > A[j]
4:       tmp = A[i]
5:       A[i] = A[j]
6:       A[j] = tmp
```

  - Za ***i*** = ***n***, invarijantnost petlje je zadovoljena na sledeći način
  - Podniz ***A[1 .. n-1]*** je sortiran
  - Unutar ***j*** petlje se u ***n***-toj iteraciji ***i*** petlje na ***n***-tu poziciju umeće najmanja vrednost iz ne-sortiranog podniza ***A[n .. A.length]***
  - Sve vrednosti unutar podniza ***A[1 .. n-1]*** su jednake ili manje od vrednosti iz podniza ***A[n .. A.length]***
  - Takođe, vrednost ***A[n]*** je jednaka ili manja od vrednosti iz podniza ***A[n+1 .. A.length]***
  - Tako da je podniz ***A[1 .. n]*** sortiran i sve vrednosti unutar njega su jednake ili manje od vrednosti iz podniza ***A[n+1 .. A.length]***
  - Čime je invarijantnost petlje očuvana

… LOOP INVARIANT – VEŽBANJE

- **Selection sort – završavanje**
  - Na kraju poslednje iteracije podniz ***A[1 .. A.length-1]*** je sortiran
  - Sve vrednosti u podnizu ***A[A.length-1 .. A.length]*** jednake su ili veće od svih vrednosti iz podniza ***A[1 .. A.length-1]*** (zapravo jedna vrednost)

SELECTION-SORT (A)

```
1: for i = 1 to A.length-1
2:   for j = i+1 to A.length
3:     if A[i] > A[j]
4:       tmp = A[i]
5:       A[i] = A[j]
6:       A[j] = tmp
```

  - Čime je invarijantnost petlje očuvana, tj. niz ***A*** je na kraju sortiran

LOOP INVARIANT – STRATEGIJE …

- Iz primera sa može videti da je pretpostavka ili tvrđene, odnosno da se invarijantnost petlje iskazuje u obliku:
  - Na početku iteracija sa indeksom ***j***, … podniz ***A[1 .. j]*** …
- Kako doći do tvrđenja koje se dokazuje?
  - **Razmisliti o specifičnoj iteraciji**
  - **Šta se zna na kraju petlje?**
  - **Razmisliti o algoritamskoj tehnici koja se koristi**
  - **Ima i drugih pristupa …**

… LOOP INVARIANT – STRATEGIJE …

- **Kako doći do tvrđenja koje se dokazuje?**
- **Razmišljanjem o specifičnoj iteraciji**
- Koju informaciju (podatke) će algoritam koji se već neko vreme izvršava imati ako se prekine neposredno pre ***j***-e iteracije?
- Koje vrednosti promenljiva ***max*** ima u 0 koraku, pa u prvom, drugom, i tako dalje.
- Ili recimo kod algoritma za sumu elemenata niza, kako se menja suma kroz korake algoritma?
  - 0,
  - ***A[1]***,
  - ***A[1]*** + ***A[2]***,
  - ***A[1]*** + ***A[2]*** + ***A[3]***,
  - …

… LOOP INVARIANT – STRATEGIJE …

- **Kako doći do tvrđenja koje se dokazuje?**
- **Šta se zna na kraju petlje?**
- Najčešće, ono što želimo da ostvarimo: sortiran niz, maksimum niza, suma niza i sl.
- Iz tog poslednjeg koraka ide su unazad i gradi strategija
- Ako se zna da promenljiva ***max*** u poslednjem, ***n***-tom koraku sadrži maksimum niza, šta onda ***max*** sadrži u pretposlednjome koraku, ***n-1***?
  - Sadrži naravno maksimum podniza ***A[1 .. A.length-1]***
- Ovo, nažalost, ne radi uvek!
- Informacija na kraju petlje se koristi da bi se došlo do rešenja, nije nužno i rešenje

… LOOP INVARIANT – STRATEGIJE

- **Kako doći do tvrđenja koje se dokazuje?**
- **Razmišljanjem o algoritamskoj tehnici koja se koristi**
- Kada se koriste petlje? – kada inkrementalno gradimo rešenje
- Tako da kada su petlje u pitanju, najčešće stoji tvrđenje:
  - Rešenje određeno do sada (do ove iteracije) je korektno rešenje za one podatke (stvari) koje sam video do sada (do ove iteracije), ili
  - Na početku iteracija sa indeksom ***j***, promenljiva ***x*** je korektno rešenje za podniz ***A[1 .. j]*** …
- U primeru za određivanje maksimuma:
  - u ***j***-oj iteraciji vrednost promenljive ***max*** je jednaka najvećoj vrednosti iz podniza ***A[1 .. j+1]***

UVOD U ANALIZU ALGORITAMA

UVOD U ANALIZU ALGORITAMA …

- Analiziranje algoritma podrazumeva procenu koje resurse algoritam zahteva
- Često to znači procenu potrebne memorije, procesorske moći, širine komunikacionog kanala ili potrebnog kapaciteta hardvera
- Ali u većini slučajeva, to je vreme
- Analizom nekoliko kandidata (algoritama) moramo proceniti i identifikovati koji je najefikasniji
- To može rezultovati sa više kandidata, ali u većini slučajeva dovodi do odbacivanja onih inferiornih
- **Kako izbeći zavisnost od hardvera i implementacije?**
- PRIMER: kompresija podataka

… UVOD U ANALIZU ALGORITAMA …

- U suštini, pre analize vremena izvršavanja algoritma, mora se napraviti model tehnologije na kojem se on implementira i hardvera na kojem se on izvršava, tzv. RAM model
- Mašina koja ima jedan procesor i RAM, kao i da će algoritmi generalno biti implementirani kao računarski programi
- Striktno govoreći, mora se tačno proračunati vreme izvršavanja svakog koraka spram vremena mašine
- Striktno govoreći, mora se u obzir uzeti i veličina tipova podataka, ograničenje veličine mašinske reči
- I tako dalje u nedogled
- Analiza algoritma kroz RAM model je TEŠKA
- Gde je tu prednost analize u odnosu na merenje implementacije u stvarnom hardveru?

… UVOD U ANALIZU ALGORITAMA

- Koje je onda rešenje?
- **SIMPLIFIKACIJA!**
- Ne treba nam egzaktno vreme, već nešto na osnovu čega da procenimo kompleksnost algoritama jedno spram drugog
- Ako su svi dosledno i na isti način analiziraju, dobija se odnos koji uvek važi
- Šta to generalno znači?
- U globalu na primer nešto u smislu:
- Nije važno da li je brže za 20 s, već samo da je duplo brže
- Uvodimo pretpostavku da postoje jedinične instrukcije/koraci algoritma (čije izvršavanje je konstantno u vremenu), kao i da se kompleksnije instrukcije sastoje od jediničnih, pa se do vremena kompleksnih instrukcija dolazi sabiranjem vremena izvršavanja jediničnih instrukcija

VREME IZVRŠAVANJA …

- Vreme izvršavanja zavisi od veličine ulaza
  - već sortiran niz je lakše sortirati
  - brže je sortirati niz od 3 elementa od niza koji sadrži 100 000 elemenata
- Opis veličine ulaza zavisi od problema (nekad se opisuje brojem elemenata u ulazu, nekada sa brojem bita, nekada kao 2 broja, …)
- Vreme izvršavanja bi trebalo parametrizovati tako da se vidi njegova zavisnost od veličine ulaza
- U suštini kada analiziramo bilo šta, uvek gledamo koje su gornje granice, tzv. **worst-case** scenario
- Zašto **worst-case** scenario?
  - Jer svi (inženjeri, ne i menadžeri), volimo da budemo sigurno da sporije (gore) od toga ne može!

… VREME IZVRŠAVANJA …

- Koje vrste analize vremena postoje?
  - **worst-case** (najčešće):

    ***T(n)* =** najduže vreme potrebno za izvršavanja algoritma nad ulazom veličine n
    - Gore od ovoga ne može
    - Za neke algoritme ovo se dešava često
    - **avarage-case** je često isti **worst-case** vremenu izvršavanja
  - **avarage-case** (ponekad):

    ***T(n)* =** očekivano (prosečno) vreme potrebno za izvršavanja algoritma nad ulazom prosečne veličine n
    - Zahteva nekakvu procenu statističke raspodela ulaza
  - **best-case** (retko):

    ***T(n)* =** najbrže moguće vreme izvršavanja algoritma koje radi samo na određenim ulazima
    - Varanje

… VREME IZVRŠAVANJA …

- Vreme izvršavanja algoritma nad datim ulazom je zapravo jednako broju primitivnih operacija ili koraka koje treba da se izvrše
- Šta su koraci?
- Kako označiti korake?
- Idealno bi bilo tretirati korake što je moguće više nezavisnim od implementacije (machine-independent)
- Pretpostavimo da se svaki korak realizuje u konstantnom vremenu, tako da je vreme izvršavanja i-tog reda c_i
- Kada se to uzme u razmatranje, onda se za svaki korak **insertion** sortiranja može odrediti njegova vreme izvršavanja koje se dobija kombinacijom koliko puta se izvršava taj korak i koliko vremena je potrebno da bi se taj korak izvršio

… VREME IZVRŠAVANJA …

INSERTION-SORT(A)

1: **for** $j = 2$ **to** $A.length$
2: $key = A[j]$
3: // Insert $A[j]$ to the sorted sequence $A[1..j-1]$
4: $i = j - 1$
5: **while** $i > 0$ **and** $A[i] > key$
6: $A[i+1] = A[i]$
7: $i = i - 1$
8: $A[i+1] = key$

… VREME IZVRŠAVANJA …

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j = 2$ **to** $A.length$	c_1	n
2 $key = A[j]$	c_2	$n-1$
3 // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
4 $i = j-1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 $A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7 $i = i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8 $A[i+1] = key$	c_8	$n-1$

… VREME IZVRŠAVANJA …

- ***while*** petlja u redu 5 se za svako ***j*** izvršava određen broj puta, što se može iskazati t_j vremenom za svaku vrednost ***j = 2, 3, …, n***, gde je ***n = A.length***
- Kada se ***for*** ili ***while*** završavaju na uobičajeni način oni se izvrše jedan put više nego telo petlje

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j = 2$ **to** $A.length$	c_1	n
2 $key = A[j]$	c_2	$n-1$
3 // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
4 $i = j-1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 $A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7 $i = i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8 $A[i+1] = key$	c_8	$n-1$

… VREME IZVRŠAVANJA …

- Kada se sva vremena iz koraka algoritma saberu dobija se:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) .$$

- U najboljem slučaju, kada niz već sortiran, ***while*** se izvršava samo jednom za svako ***j***, pa je vreme sortiranja:

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) . \end{aligned}$$

- Očigledno je vreme izvršavanja linearna funkcija od ***n***

… VREME IZVRŠAVANJA …

- U najgorem slučaju, kada je niz sortiran u obrnutom redosledu, svaki element niz ***A[j]*** mora da se poredi sa svakim elementom podniza ***A[1 .. j–1]***, tako da je $t_j = j$: tako da je

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1 \qquad \sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

- Što dovodi do **worst-case** vremena izvršavanja insertion sortiranja:

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n \\ &\quad - (c_2 + c_4 + c_5 + c_8) \, . \end{aligned}$$

… VREME IZVRŠAVANJA …

- Jasno je da se formula:

$$T(n) = \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n - (c_2 + c_4 + c_5 + c_8) .$$

- može svesti na:

$$an^2 + bn + c$$

- te je:

Očigledno je vreme izvršavanja kvadratna funkcija od ***n***

… VREME IZVRŠAVANJA

- Kao što je već rečeno, vreme izvršavanja kompleksnih koraka algoritma se može dobiti sabiranjem vremena izvršavanja jediničnih koraka algoritma

Konstrukcija	**Vreme izvršavanja**
Naredba serije ***S***: ***P; Q;***	$T_S = T_P + T_Q$
Naredba grananja ***S***: ***if C then P; else Q;***	$T_S = T_C + max\{T_P, T_Q\}$
Naredba petlje ***S***: (1) ***while C do P;*** (2) ***repeat P; until C;*** (3) ***for i = j to k do P;***	$T_S = n \cdot T_P$ n – najveći broj iteracija petlje

POJEDNOSTAVLJENA ANALIZA …

- Napravili smo nekoliko pojednostavljivanja
  - Konstantno vreme izvršavanja svakog koraka
  - Pojednostavljivanje i ignorisanje konstanti: $\boldsymbol{an^2 + bn + c}$
- Sledeće pojednostavljene ogleda se u uvođenu stepena rasta (**order of growth**)
- Šta to generalno znači?
- To znači da se posmatra ***T(n)*** za $\boldsymbol{n \rightarrow \infty}$
- U inženjerskom smislu:
  - Odbaciti sve elemente nižeg reda (npr. $\boldsymbol{bn + c}$ u $\boldsymbol{an^2 + bn + c}$)
  - Ignorisati vodeću konstantu (npr. $\boldsymbol{a}$ u $\boldsymbol{an^2}$)
  - Tako da se dobija procena vremena od $\boldsymbol{n^2}$

… POJEDNOSTAVLJENA ANALIZA

- Opisani postupak naziva asimptotska analiza
- Procena za **worst-case** vreme izvršavanja se onda svodi ***Θ(n²)***
- Ukratko ***Θ***-notacija se koristi da označi **worst-case** vreme izvršavanja
- Matemitika (za sada):

$$\Theta(g(n)) = \{f(n): postoje\ pozitivne\ konstante\ c_1, c_2\ i\ n_0\ takve\ da \\ 0 \le c_1 g(n) \le f(n) \le c_2 g(n)\ za\ svako\ n \ge n_0\}$$

- Kada n postane dovoljno veliko algoritam sa ***Θ(n²)*** složenošću će uvek biti bolji od ***Θ(n³)*** algoritam
- U praksi se asimptotski sporiji algoritmi neće uvek odbacivati
- Inženjerski problemi zahtevaju balans
- Asimptotska analiza alat za usmeravanje razmišljanje