Algoritmi i strukture podataka

- predavanja -

8. Sortiranje

Algoritmi

- odabrani postupci za ilustraciju:
  - sortiranje biranjem (*selection sort*)
  - *bubble sort*
  - sortiranje umetanjem (*insertion sort*)
  - Shellov sort
  - *mergesort*
  - *quick sort*
  - sortiranje s pomoću gomile (*heap sort*) - kasnije!

Sorts.cpp

Sortiranje biranjem (*selection sort*)

- pronađi najmanji element niza i zamijeni ga s prvim elementom niza
- ponavljaj s ostatkom niza, smanjujući nesortirani dio

6	4	1	8	7	5	3	2
1	4	6	8	7	5	3	2
1	2	6	8	7	5	3	4
1	2	3	8	7	5	6	4
1	2	3	4	7	5	6	8
1	2	3	4	5	7	6	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8

Sortiranje biranjem - algoritam i složenost

- izvedba - 2 petlje:
  - vanjska određuje granice sortiranog dijela niza
  - unutarnja traži najmanji element u nizu

```
SelectionSort(T A[], size_t n) {
    size_t i, j, min;
    for (i = 0; i < n; i++) {
        min = i;
        for (j = i + 1; j < n; j++) {
            if (A[j] < A[min]) min = j;
        }
        Swap(&A[i], &A[min]);
    }
}
```

$$T(n) = \sum_{i=0}^{n-1}(t_i + \Theta(1)) = \sum_{i=0}^{n-1}(t_i) + \Theta(n) = \sum_{i=0}^{n-1}\sum_{j=i+1}^{n-1}\Theta(1) + \Theta(n) = \sum_{i=0}^{n-1}(n-i-1)\Theta(1) + \Theta(n) = \frac{n(n-1)}{2}\Theta(1) + \Theta(n) = \boldsymbol{\Theta(n^2)}$$

$$t_i = \sum_{j=i+1}^{n-1}\Theta(1) = (n-i-1)\Theta(1)$$

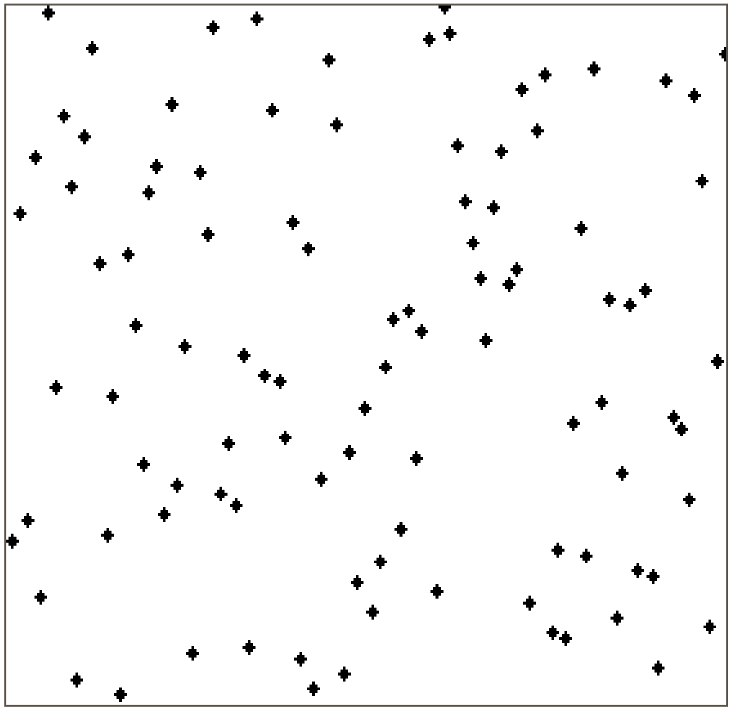
Sorts.cpp

Sortiranje biranjem – analiza vremena izvođenja

- dominira usporedba: $\Theta(n^2)$; zamjena ima manje: $\Theta(n)$
- vrijeme izvođenja ne ovisi o početnom redoslijedu, nego o broju elemenata
- vrijeme izvođenja: $\boldsymbol{\Theta(n^2)}$
  - otprilike `n`^{2}`/2` usporedbi i `n` zamjeni u prosječnom i u najlošijem slučaju
  - nije bitno brži kad su ulazni elementi blizu svojih mjesta
  - i u najboljem i u najlošijem slučaju (sortiran niz i naopako sortiran niz) vrijedi $\Theta(n^2)$
    - malo ubrzanje: ako bi se prije zamjene ispitivalo treba li obaviti zamjenu
- ubrzanje:
  - sortirati odjednom s obje strane
  - Pitanje: utječe li ovakvo ubrzanje na asimptotsko vrijeme izvođenja algoritma?

Bubble sort

- ideja: zamjena susjednih elemenata ako nisu u dobrom redoslijedu
  - kreni od početka niza prema kraju
  - zamijeni 2 elementa ako je prvi veći od drugog
  - ubrzanje: ako u prolasku kroz cijeli niz nije bilo zamjene, niz je sortiran



Bubble sort - primjer

1. prolaz

6 4 1 8 7 5 3 2
4 6 1 8 7 5 3 2
4 1 6 8 7 5 3 2
4 1 6 8 7 5 3 2
4 1 6 7 8 5 3 2
4 1 6 7 5 8 3 2
4 1 6 7 5 3 8 2
4 1 6 7 5 3 2 8

2. prolaz

4 1 6 7 5 3 2 8
1 4 6 7 5 3 2 8
1 4 6 7 5 3 2 8
1 4 6 7 5 3 2 8
1 4 6 5 7 3 2 8
1 4 6 5 3 7 2 8
1 4 6 5 3 2 7 8

3. prolaz

1 4 6 5 3 2 7 8
1 4 6 5 3 2 7 8
1 4 6 5 3 2 7 8
1 4 5 6 3 2 7 8
1 4 5 3 6 2 7 8
1 4 5 3 2 6 7 8

4. prolaz

1 4 5 3 2 6 7 8
1 4 5 3 2 6 7 8
1 4 5 3 2 6 7 8
1 4 3 5 2 6 7 8
1 4 3 2 5 6 7 8

5. prolaz

1 4 3 2 5 6 7 8
1 4 3 2 5 6 7 8
1 3 4 2 5 6 7 8
1 3 2 4 5 6 7 8

6. prolaz

1 3 2 4 5 6 7 8
1 3 2 4 5 6 7 8
1 2 3 4 5 6 7 8

7. prolaz

1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8

Bubble sort – algoritam i složenost

```
template <typename T>
static void BubbleSort(T A[], size_t n) {
 size_t i, j;
 for (i = 0; i < n - 1; i++) {
  for (j = 0; j < n - 1 - i; j++) {
     if (A[j + 1] < A[j])
        Swap(&A[j], &A[j + 1]);
  }
 }
}
```

$$T(n) = \sum_{i=0}^{n-2} t_i =$$
$$= \sum_{i=0}^{n-2} \sum_{j=0}^{n-1-i} \Theta(1)$$
$$= \sum_{i=0}^{n-2} (n-i-1)\Theta(1)$$
$$= \frac{n(n-1)}{2}\Theta(1) = \boldsymbol{\Theta(n^2)}$$

$$t_i = \sum_{j=0}^{n-i-2} \Theta(1) = (n-i-1)\,\Theta(1)$$

Sorts.cpp

Poboljšani *bubble sort*

- ako u nekom prolazu nije bilo zamjene, niz je sortiran

```
template <typename T>
static void BubbleSortEnhanced(T A[], size_t n) {
      size_t i, j;
      bool swapHappened = 1;
      for (i = 0; swapHappened == 1; i++) {
         swapHappened = 0;
         for (j = 0; j < n - 1 - i; j++) {
            if (A[j + 1] < A[j]) {
               Swap(&A[j], &A[j + 1]);
               swapHappened = 1;
            }
         }
      }
   }
```

Sorts.cpp

Bubble sort - analiza vremena izvođenja (1)

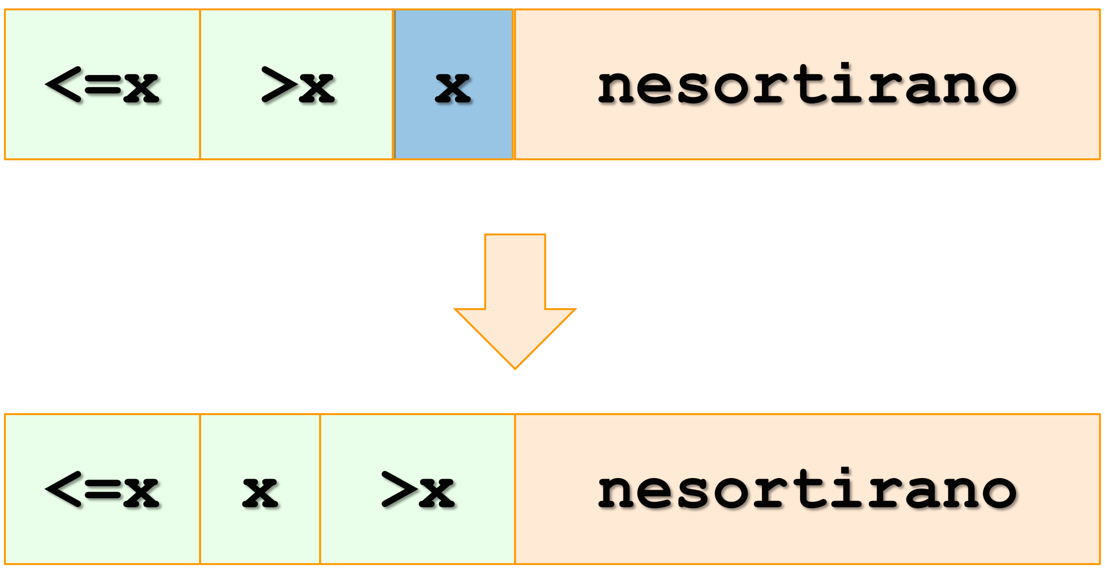
- poboljšani algoritam:
  - najbolji slučaj: niz je sortiran
  - najlošiji slučaj: naopako sortiran niz
  - kad su ulazni elementi blizu svojih mjesta, poboljšani algoritam može brzo završiti
  - za poboljšani *bubble sort* vrijedi: $\Omega(n)$ i $O(n^2)$
- prosječan i najlošiji slučaj za obje varijante *bubble sort*-a: $\boldsymbol{\Theta(n^2)}$
  - otprilike **$n^2/2$** usporedbi i **$n^2/2$** zamjena u prosječnom i u najlošijem slučaju
- razlika u vremenu izvođenja između osnovnog i poboljšanog *bubble sort*-a postoji samo za najbolji slučaj
  - poboljšani *bubble sort* za najbolji slučaj: $\Theta(n)$
  - za običan *bubble sort* vrijedi: $\Theta(n^2)$

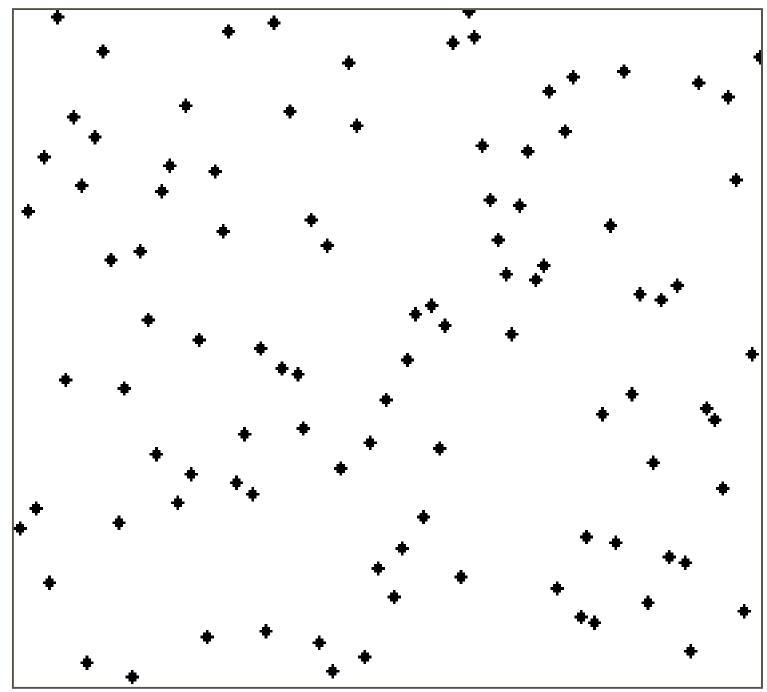
Bubble sort - analiza vremena izvođenja (2)

- položaj elemenata u polju bitan za učinkovitost
  - veliki elementi na početku niza nisu problem – brzo idu na kraj („zečevi”)
  - mali elementi na kraju niza su problem – polako idu prema vrhu („kornjače”)

Sortiranje umetanjem (*insertion sort*)

- ideja: postoje dva dijela niza: sortirani i nesortirani
  - u svakom koraku sortirani dio se proširuje tako da se u njega na ispravno mjesto ubaci prvi element iz nesortiranog dijela niza
- tako se (obično) sortiraju karte u kartaškim igrama





Sortiranje umetanjem - primjer

6	4	1	8	7	5	3	2
6	4	1	8	7	5	3	2
4	6	1	8	7	5	3	2
1	4	6	8	7	5	3	2
1	4	6	8	7	5	3	2
1	4	6	7	8	5	3	2
1	4	5	6	7	8	3	2
1	3	4	5	6	7	8	2
1	2	3	4	5	6	7	8

Sortiranje umetanjem - algoritam i složenost

- izvedba - 2 petlje:
  - vanjska služi za određivanje granice sortiranog dijela
  - unutarnja ubacuje element u sortirani niz i pomiče ostale elemente

```
template <typename T>
static void InsertionSort(T A[], size_t n) {
  size_t i, j;
  T temp;
  for (i = 1; i < n; i++) {
     temp = A[i];
     for (j = i; j >= 1 && A[j - 1] > temp; j--)
        A[j] = A[j - 1];
     A[j] = temp;
  }
}
```

$$T(n) = \sum_{i=1}^{n-1} t_i$$

t_i

Sorts.cpp

Sortiranje umetanjem - analiza vremena izvođenja (1)

- razmatramo tri slučaja:
  - najbolji slučaj: sortiran niz
  - najlošiji slučaj: naopako sortiran niz
  - prosječan slučaj: slučajno poredani podatci
  - općenito: što su ulazni elementi bliže svojim mjestima, sortiranje brže završava
- za najbolji slučaj: $t_i = 1$, pa slijedi: $T(n) = \sum_{i=1}^{n-1} t_i = \Theta(n)$
- za najlošiji slučaj: $t_i = i$, pa slijedi: $T(n) = \sum_{i=1}^{n-1} i = \Theta(n^2)$
- za prosječan slučaj: $t_i \approx i / 2$
  - možemo pretpostaviti da kad uspoređujemo i-ti element sa svojim prethodnicima, prosječno polovica prethodnika će biti manja, a polovica će biti veća od njega
  - slijedi: $T(n) = \sum_{i=1}^{n-1} i/2 = \Theta(n^2)$

Sortiranje umetanjem - analiza vremena izvođenja (2)

- razmatramo tri slučaja:
  - najbolji slučaj: $T(n) = \Theta(n)$
  - najlošiji slučaj: $T(n) = \Theta(n^2)$
  - prosječan slučaj: $T(n) = \Theta(n^2)$

- vrijeme izvođenja sortiranja umetanjem: $\Omega(n)$ i $O(n^2)$

Sortiranje umetanjem – stabilnost sortiranja

- sortiranje umetanjem je stabilno
  - ne dolazi do zamjene relativnih pozicija elemenata koji imaju istu vrijednost ključa
  - ako *a* i *b* imaju isti ključ i *a* je bilo prije *b*, nakon stabilnog sorta *a* će i dalje biti ispred *b*

Shellov sort (*Shell sort*)

- najstariji brzi algoritam, modificirano sortiranje umetanjem
  - autor: Donald Shell (1959.)
- ideja:
  - za k-sortirano polje A vrijedi $A[i] \leq A[i + k], \forall i, i+k$ indeksi
  - ako je polje k-sortirano i dodatno se t-sortira ($t<k$), ostaje i dalje k-sortirano
  - potpuno sortirano polje je 1-sortirano
- općenito, koristi se inkrementalni slijed brojeva $h_1, h_2, h_3, \ldots, h_t$
  - izbor slijeda izuzetno je bitan za učinkovitost algoritma

Shellov sort – primjer

korak = 4	korak = 2	korak = 1
6 4 1 8 7 5 3 2	1 4 6 2 7 5 3 8	1 2 3 4 6 5 7 8
6 4 1 8 7 5 3 2	1 2 6 4 7 5 3 8	1 2 3 4 6 5 7 8
6 4 1 8 7 5 3 2	1 2 6 4 7 5 3 8	1 2 3 4 6 5 7 8
6 4 1 2 7 5 3 8	1 2 6 4 7 5 3 8	1 2 3 4 6 5 7 8
	1 2 3 4 6 5 7 8	1 2 3 4 5 6 7 8
	1 2 3 4 6 5 7 8	1 2 3 4 5 6 7 8
		1 2 3 4 5 6 7 8

Koristimo slijed { 4, 2, 1 }

Shellov sort – algoritam

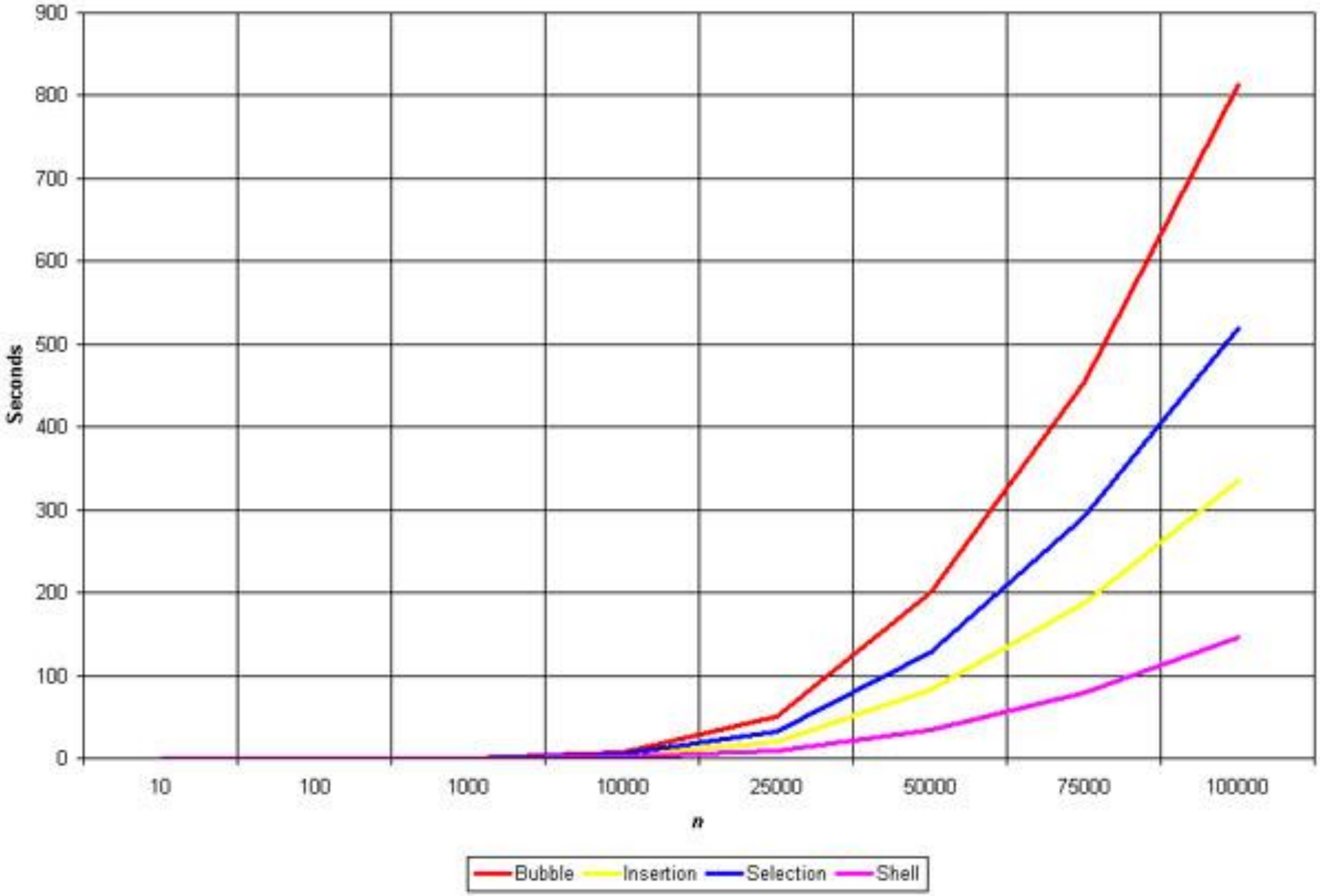
```
template <typename T>
static void ShellSort(T A[], size_t n) {
  size_t i, j, step;
  T temp;
  for (step = n / 2; step > 0; step /= 2) {
   for (i = step; i < n; i++) {
      temp = A[i];
      for (j = i; j >= step && A[j - step] > temp; j -= step) {
         A[j] = A[j - step];
      }
      A[j] = temp;
   }
  }
}
```

Sorts.cpp

Shellov sort - analiza složenosti

- prosječno vrijeme izvođenja je već dugo otvoreni (neriješeni) problem
  - najlošiji slučaj $O(n^2)$
- Hibbardov slijed (1963.): {1, 3, 7, ..., 2^k -1} rezultira najlošijim slučajem $O(n^{3/2})$
  - prosječno $O(n^{5/4})$ utvrđeno je simulacijom; nitko to nije uspio dokazati
- Sedgwickov slijed (1986.): {1, 5, 19, 41, 109,...}, odnosno $9*4^i - 9*2^i + 1$ alternira s $4^i - 3*2^i + 1$
  - najlošiji slučaj $O(n^{4/3})$, a prosječno $O(n^{7/6})$
- ne zna se postoji li bolji slijed
- jednostavan algoritam, a krajnje složena analiza složenosti

Usporedba sortova sa složenošću $O(n^2)$



Bubble

Selection

Insertion

Shell

Mergesort

- koristi se strategija "podijeli, pa vladaj" uz rekurziju
- autor: John von Neumann, 1945. godine
- ideja algoritma:
  - nesortirani niz podijeli se na dva niza podjednake veličine
  - svaki podniz sortira se rekurzivno, dok se ne dobije niz od 1 elementa
    - taj niz od jednog elementa je sortiran!
  - spoje se dva sortirana podniza u sortirani niz
    - na temelju dva sortirana polja (A i B) puni se treće (C)
- grananjem nastane $\log_2 n$ razina, a u svakoj od razina obavlja se $O(n)$ posla → trajanje je $O(n \log_2 n)$

Primjer sortiranja

| 31 | 24 | 47 | 1 | 6 | 78 | 12 | 65 |

| 31 | 24 | 47 | 1 | | 6 | 78 | 12 | 65 |

| 31 | 24 | | 47 | 1 | | 6 | 78 | | 12 | 65 |

| 31 | | 24 | | 47 | | 1 | | 6 | | 78 | | 12 | | 65 |

| 24 | 31 | | 1 | 47 | | 6 | 78 | | 12 | 65 |

| 1 | 24 | 31 | 47 | | 6 | 12 | 65 | 78 |

| 1 | 6 | 12 | 24 | 31 | 47 | 65 | 78 |

Mergesort - algoritam

- vježba: napišite funkciju **Merge**
  - funkcija spaja lijevu i desnu sortiranu polovicu u sortirani niz

```
template <typename T>
static void MergeRecursive(T A[], T helperArray[], size_t left, size_t right){
      size_t middle;
      if (left < right) {
         middle = (left + right) / 2;
         MergeRecursive(A, helperArray, left, middle);
         MergeRecursive(A, helperArray, middle + 1, right);
         MergeArrays(A, helperArray, left, middle + 1, right);
      }
   }
```

Sorts.cpp

Mergesort - napomene

- cijena bržeg sortiranja: memorija
  - stvara se pomoćno polje!
- rijetko se koristi za sortiranje u središnjoj memoriji
  - povećani su zahtjevi za dodatnom memorijom i kopiranjem
- to je ključni algoritam za sortiranje na vanjskoj memoriji
- ponašanje u prosječnom i najlošijem slučaju: $O(n\log_2 n)$
- ne radi ništa brže ako je ulazni niz već sortiran!

Quicksort (1)

- često se koristi zbog svoje brzine za prosječne skupove podataka
- ima mnoštvo varijanti (wiki)
- rekurzija: „podijeli, pa vladaj“
- sortiranje korištenjem dijeljenja i zamjena (engl. *partition-exchange sort*)

QuickSort.cpp

Quicksort (2)

- Algoritam (za ulazno polje S): $quicksort\ (S)$
  - ako je broj članova polja S jednak 0 ili 1, povratak u pozivni program
  - odabrati član v u polju S kao stožer (*pivot*)
    - odabir stožera: str. 31
  - podijeli preostale članove polja S, $S \setminus \{v\}$ u dva odvojena skupa:
    - $S_1 = \{\, x \in S \setminus \{v\} \,/\, x \leq v \}$ (sve što je manje od stožera, preseli lijevo)
    - $S_2 = \{\, x \in S \setminus \{v\} \,/\, x \geq v \}$ (sve što je veće od stožera, preseli desno)
  - vrati niz sastavljen od $\{quicksort\ (S_1),\ v,\ quicksort\ (S_2)\}$

Izbor stožera

- nije jednoznačno određen
- nije jednoznačno određeno niti što učiniti s članovima polja jednakim stožeru
  - to postaje pitanje realizacije algoritma
  - dio dobre izvedbe je učinkovito rješenje ovog pitanja
    - Weiss: "Data Structures and Algorithm Analysis in C"
- moguće metode izbora stožera:
  - procjena medijana temeljem 3 elementa (prvi element, zadnji element, element na polovici)
    - pri procjeni medijana, ti se elementi odmah sortiraju
  - druge mogućnosti: slučajni element, prvi element, zadnji element
  - npr. niz: 8 1 4 9 6 3 5 2 7 0
    - pivot = med3 (8, 6, 0) = 6
  - Koji je najlošiji mogući odabir stožera?
  - Što je stvarni medijan?

Primjer sortiranja quicksortom

izbor stožera

```
8 1 4 9 6 3 5 2 7 0
^       ^         ^
0 1 4 9 6 3 5 2 7 8
```

```
0 1 4 9 7 3 5 2 6 8
i->           <-j
0 1 4 9 7 3 5 2 6 8
      i       j
0 1 4 2 7 3 5 9 6 8
        i   j
0 1 4 2 5 3 7 9 6 8
```

i i j se mimoilaze

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

vraćamo stožer na i

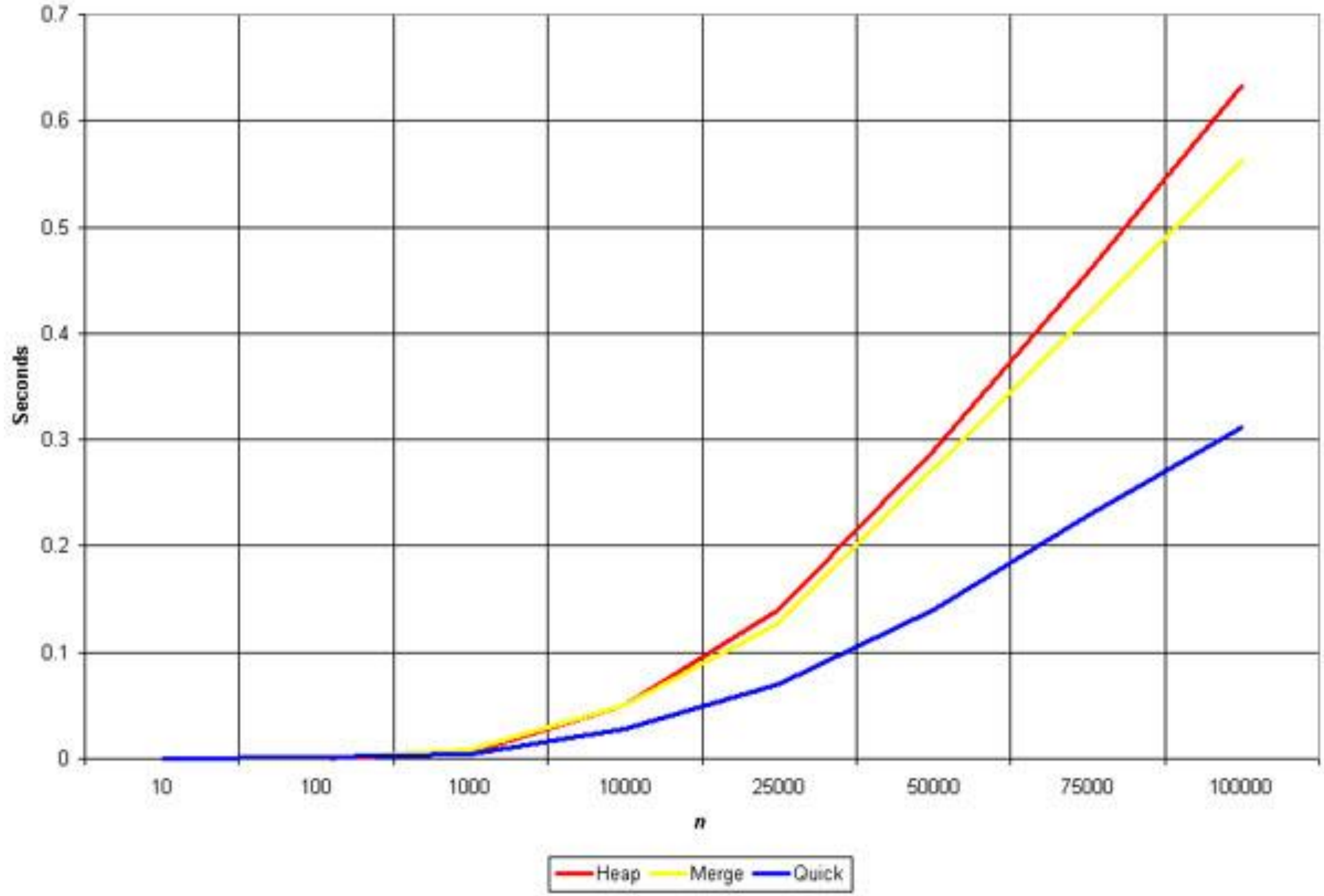
```
0 1 4 2 5 3 7 9 6 8
            i
0 1 4 2 5 3 6 9 7 8
```

izbor stožera		izbor stožera
0 1 4 2 5 3	6	9 7 8

Složenost algoritma

- prosječno vrijeme izvođenja je ***O(n log n)***
  - sortiranje je vrlo brzo, uglavnom zbog optimirane unutarnje petlje
- najlošiji slučaj je ***O(n²)***
  - uz krivi odabir stožera (min ili max član), dobiva se **`n`** particija i za svaku je vrijeme izvođenja ***O(n)***
  - može se postići da vjerojatnost da takav slučaj nastupi eksponencijalno pada

Usporedba sortova sa složenošću *O(nlogn)*



Heap

Merge

Quick

Postupci sortiranja

- sortiranje oko milijun podataka nije u praksi rijetko
- ako jedno obavljanje programske petlje traje 1 μs:
  - klasično sortiranje trajalo bi reda veličine 10^6 s (odnosno više od 11 dana)
  - *quicksort* traje reda veličine 20 s
- ne treba uvijek tražiti rješenje u kupnji bržih i skupljih računala
  - može se isplatiti investicija u razvoj i primjenu boljih algoritama

Indirektno sortiranje

- za sortiranje velikih struktura nema smisla obavljati mnogo zamjena velikog broja podataka
  - primjeri takvih struktura
    - matični broj studenta, prezime, ime, adresa, upisani predmeti i ocjene
  - ako se podatci sortiraju npr. po matičnom broju, tada se izdvoje u posebno polje matični brojevi s pripadnim pokazivačima na ostale podatke
  - sortira se (bilo kojim od postupaka) samo takvo izdvojeno polje

Usporedba

naziv	najbolje	prosječno	najlošije	stabilan	metoda
selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	ne	biranje
insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$	da	umetanje
bubble sort	$O(n)$	-	$O(n^2)$	da	zamjena
shell sort	-	-	$O(n^{3/2})$	ne	umetanje
merge sort	O(nlogn)	O(nlogn)	O(nlogn)	da	spajanje
quick sort	O(nlogn)	O(nlogn)	$O(n^2)$	ne	podjela
heap sort	O(nlogn)	O(nlogn)	O(nlogn)	ne	biranje

Animacije algoritama

- http://www.cs.usfca.edu/~galles/visualization/Algorithms.html
- http://www.sorting-algorithms.com/
- http://cg.scs.carleton.ca/~morin/misc/sortalg/

Zadatak za vježbu (1)

- Napisati program koji će u cjelobrojnom polju od `n` članova pronaći k-ti najveći član polja.
  a) Učitano polje sortirati po padajućim vrijednostima i ispisati član s indeksom $k\text{-}1$.
  b) Učitati k članova polja, sortirati ih po padajućim vrijednostima. Učitavati preostale članove polja. Ako je pojedini član manji od onoga s indeksom $k\text{-}1$, ignorirati ga, ako je veći umetnuti ga na pravo mjesto, a izbaciti član polja koji bi sad imao indeks k.
  c) Varirati postupke sortiranja te odrediti pripadna apriorna vremena i izmjeriti aposteriorna vremena izvođenja.
- Odrediti apriorna vremena trajanja, a izmjeriti aposteriorna vremena. Varirati postupak sortiranja.

Zadatak za vježbu (2)

- U programskom jeziku C++ napisati klasu `SortableArray` koja će modelirati ponašanje polja koje se po potrebi sortira traženim algoritmom. Klasa treba:
  - Koristiti već postojeće sortove implementirane u sklopu ovog predmeta
  - Biti parametrizirana tipom `T`
  - Definirati `enum` u kojem će biti pobrojani svi sortovi
  - Definirati javnu metodu `Sort` koja će primati vrstu sorta parametrom i koja će sortirati polje pozivom algoritama sortiranja
  - Definirati metodu ispisa elemenata polja