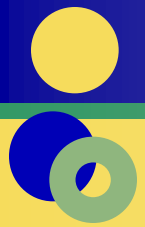
Sortiranje nizova
- napredna sortiranja -



Strukture podataka i algoritmi 2

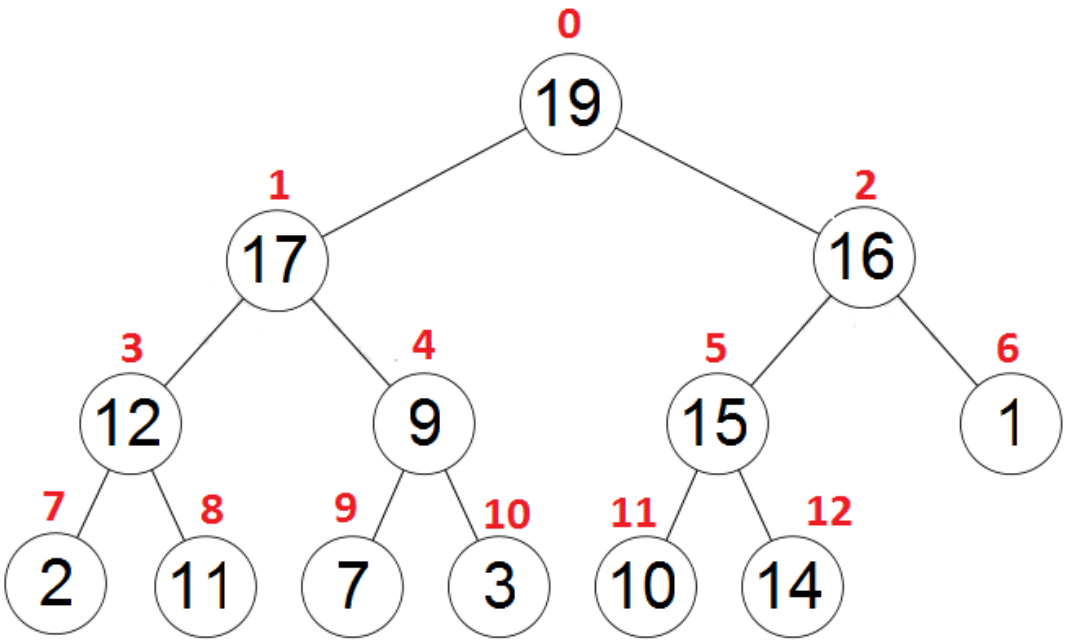
Heap sort

- Heap sort je unapređenje sortiranja izabiranjem linearitamske vremenske složenosti (O(n·log$n$))

- Opis algoritma:
  - Neka je A niz dužine n.
  - Transformišemo *A* tako da ima **strukturu hipa:** maksimalni element niza je tada prvi element u nizu.
  - Razmenimo maksimum sa elementom na kraju niza.
  - Uspostavimo strukturu hipa na podnizu A[0 .. n – 2] i maksimum zamenimo sa elementom na poziciji n - 2
  - Uspostavimo strukturu hipa na podnizu A[0 .. n – 3] i maksimum zamenimo sa elementom na poziciji n – 3
  - …
  - Uspostavimo strukturu hipa na podnizu A[0 .. 1] i maksimum zamenimo sa elementom na poziciji 1

Hip osobina (podsetnik)

- Niz ima strukturu hipa ako za svaki element niza važi da je veći ili jednak od svojih sinova
- Sinovi elementa sa indeksom *p* imaju indekse: 2*p* + 1 i 2*p*+ 2
- Roditelj elementa sa indeksom *s* ima indeks (s – 1)/2 (celobrojno deljenje)



19	17	16	12	9	15	1	2	11	7	3	10	14
0	1	2	3	4	5	6	7	8	9	10	11	12

Transformacija niza u hip

- Ako je niz A dužine n tada su elementi na pozicijama 0, 1, 2, ..., (n – 2) / 2 očevi.
  - Ako bi element na poziciji (n – 1) / 2 bio otac tada bi sinovi bili na pozicijama
    - 2 * **(n – 1) / 2** + 1 = n **(array index out of bounds exception)**
    - 2 * **(n – 1) / 2** + 2 = n + 1 **(-||-)**

- Počevši od poslednjeg ka prvom ocu uspostavljamo strukturu hipa tako što
  1. Proverimo da li je otac manji od većeg sina
  2. Ako jeste razmenimo oca sa većim sinom
  3. Ponavljamo 1. i 2. dokle god otac ne postane veći od sinova

```java
public static <T extends Comparable<T>> void sort(T[] arr) {
	int lastIndex = arr.length - 1;
	int lastParent = (lastIndex - 1) / 2;  // poslednji otac

	// uspostavi struktru heapa
	while (lastParent >= 0) {
		makeHeap(arr, lastParent, lastIndex);
		lastParent--;
	}

	// indeks poslednjeg elementa u nesortiranom delu niza
	int end = lastIndex;

	while (end > 0) {
		// razmeni prvi element sa poslednjim elementom
		// iz nesortiranog dela niza
		T tmp = arr[0];
		arr[0] = arr[end];
		arr[end] = tmp;

		// nesortirani deo je sada kraci za jedan element
		end--;

		// uspostavi strukturu hipa u nesortiranom delu
		makeHeap(arr, 0, end);
	}
}
```

```java
private static <T extends Comparable<T>>
void makeHeap(T[] arr, int start, int end) {
	int parentIndex = start;
	boolean heapRestored = false;

	while (!heapRestored) {
		int maxSonIndex = getMaxSon(arr, parentIndex, end);

		// ne postoji ni jedan od sinova
		if (maxSonIndex == -1) {
			heapRestored = true;
		} else {
			// uporedi oca sa vecim sinom
			if (arr[parentIndex].compareTo(arr[maxSonIndex]) < 0) {
				// razmeni oca sa vecim sinom
				T tmp = arr[maxSonIndex];
				arr[maxSonIndex] = arr[parentIndex];
				arr[parentIndex] = tmp;

				// otac je sada na poziciji vece sina
				parentIndex = maxSonIndex;
			} else {
				heapRestored = true;
			}
		}
	}
}
```

```java
/** Odredjuje indeks veceg sina za datog roditelja
 *  Vraca -1 ukoliko ne postoji ni jedan od sinova
 */
private static <T extends Comparable<T>>
int getMaxSon(T[] arr, int parentIndex, int end) {
	int son1Index = 2 * parentIndex + 1;
	int son2Index = 2 * parentIndex + 2;
	int maxSonIndex = -1;

	// postoji sin1?
	if (son1Index <= end) {
		maxSonIndex = son1Index;
	}

	// postoji sin2?
	if (son2Index <= end) {
		// postoje oba sina, uporedi ih
		if (arr[son2Index].compareTo(arr[son1Index]) > 0) {
			maxSonIndex = son2Index;
		}
	}

	return maxSonIndex;
}
```

Quicksort

- Quicksort je u praksi najčešće primenjivan postupak za sortiranje nizova.
- Osmislio ga je Tony Hoare 1959. godine.
- Quicksort je u proseku linearitamske, O($n \cdot \log n$), vremenske složenosti.
- Jedan od algoritama koji su zasnovani na principu zavadi pa vladaj (engl. *divide and conquer*)
- Ideja:
  - Selektujemo jedan element u nizu koga zovemo pivot.
  - Preuredimo niz tako da je oblika **(LE pivot QE)**
    - LE – elementi niza koji su manji ili jednaki od pivota
    - QE – elementi niza koji su veći ili jednaki od pivota
  - Sortiramo LE i QE quick sortom (stoga se quick sort najčešće realizuje rekurzivno)

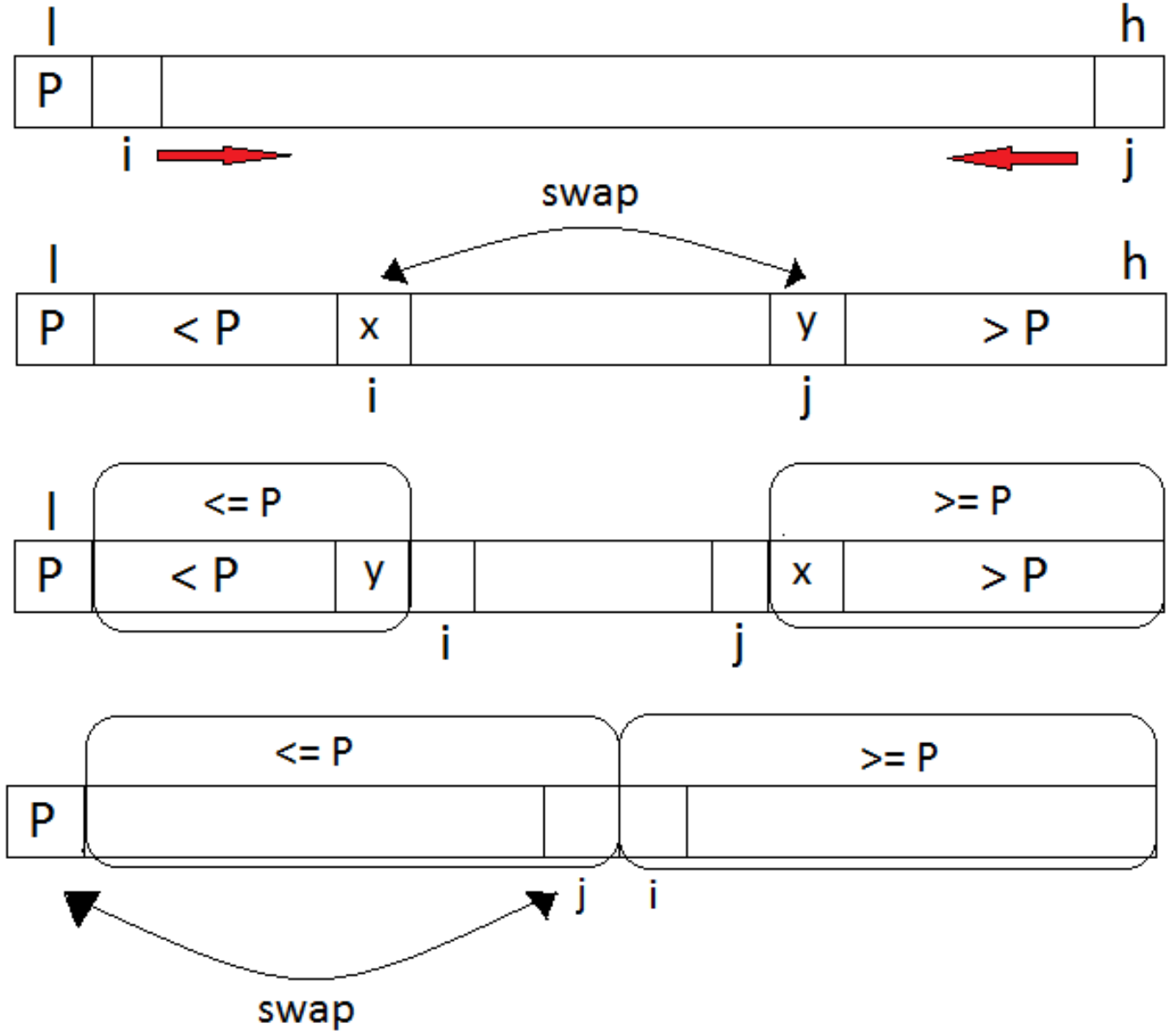
Quicksort

- Selektovanje pivota i transformacija u formu (LE pivot QE) se još naziva i particionisanjem niza.
- Postoji više šema particionisanja, od kojih su najpoznatije Hoarova šema, Lomutova šema i šema u kojoj biramo element pozicioniran na sredini (pod)niza koji se sortira.
- Opšti oblik quicksorta za Hoarovu i Lomutovu šemu je:

```
public static <T extends Comparable<T>> void sort(T[] arr) {
	sort(arr, 0, arr.length - 1);
}

private static <T extends Comparable<T>>
void sort(T[] arr, int l, int h) {
	if (l < h) {
		int j = partition(arr, l, h);
		sort(arr, l, j - 1);
		sort(arr, j + 1, h);
	}
}
```

Hoarova šema

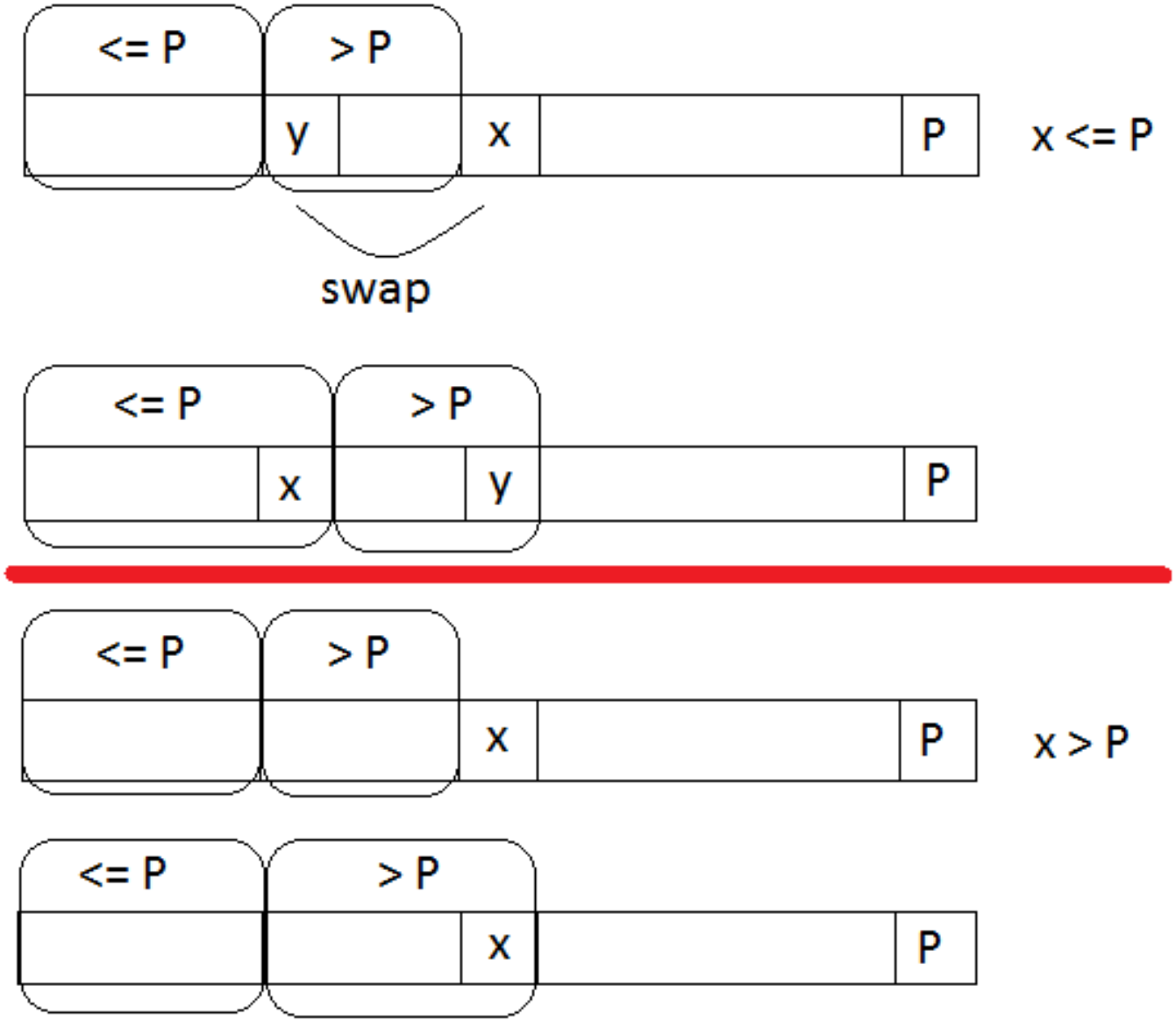
- Neka je A[l .. h] podniz koji se sortira, za pivota uzimamo A[l]



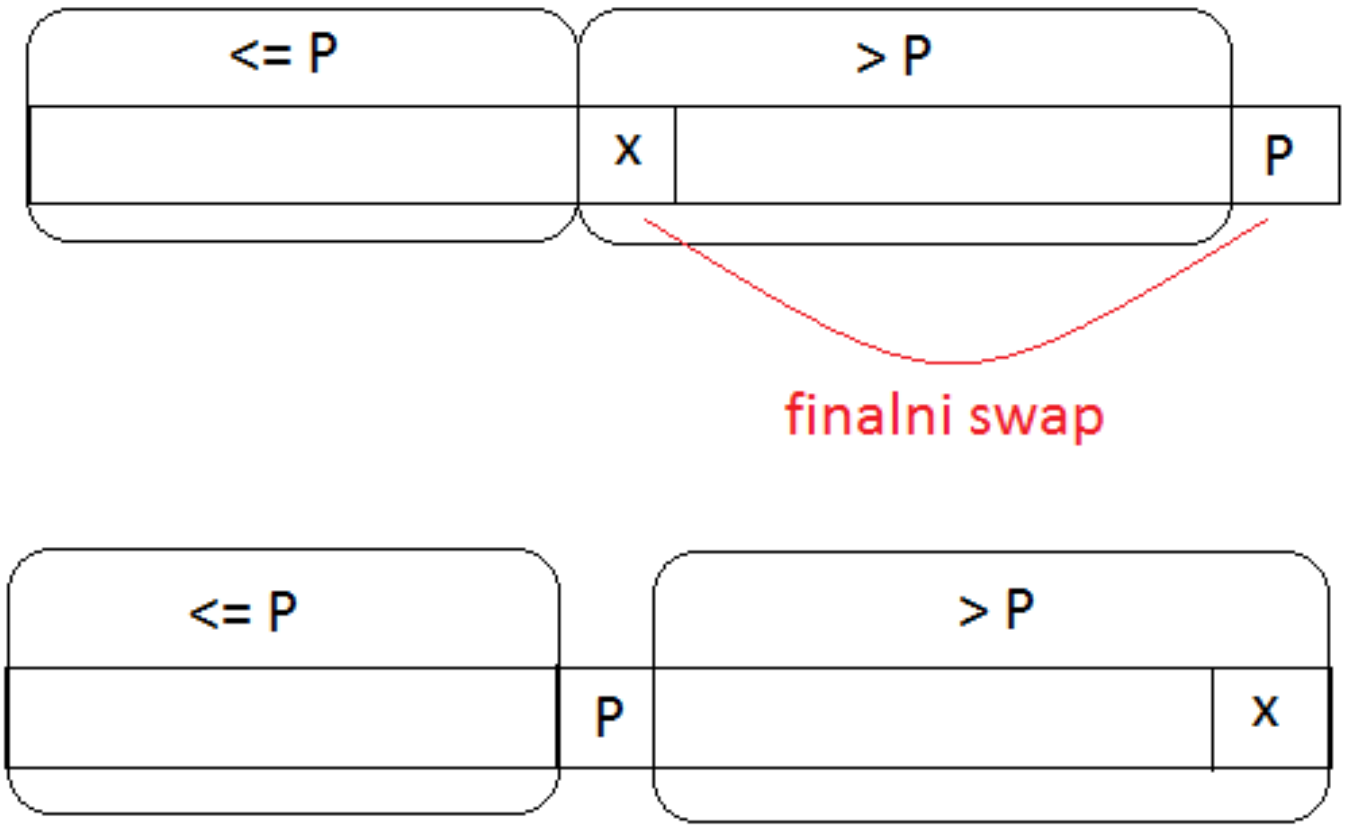
```java
private static <T extends Comparable<T>>
int partitionHoare(T[] arr, int l, int h) {
	T pivot = arr[l];
	int i = l + 1;
	int j = h;

	while (i <= j) {
		while (i <= h && arr[i].compareTo(pivot) < 0) i++;
		while (j >= l + 1 && arr[j].compareTo(pivot) > 0) j--;

		if (i <= j) {
			T tmp = arr[i];
			arr[i] = arr[j];
			arr[j] = tmp;
			i++;
			j--;
		}
	}

	T tmp = arr[l];
	arr[l] = arr[j];
	arr[j] = tmp;

	return j;
}
```

Lomutova šema

- Neka je A[l .. h] podniz koji se sortira, za pivota uzimamo A[h]



Lomutova šema – finalni korak

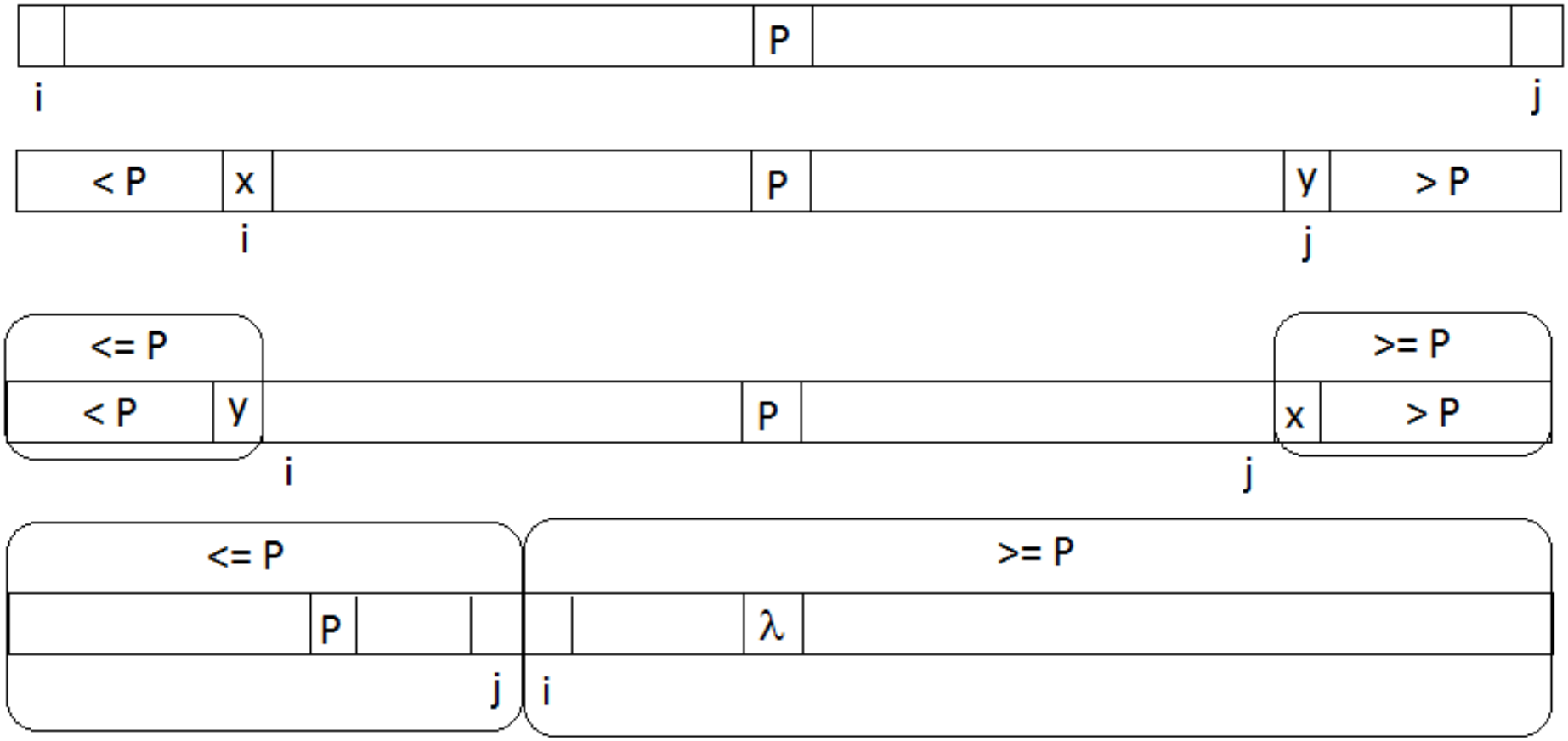


```java
private static <T extends Comparable<T>>
int partitionLomuto(T[] arr, int l, int h) {
      T pivot = arr[h];
      int ltePivot = l - 1;  // indeks elemenata koji su <= pivota

      for (int i = l; i < h; i++) {
            if (arr[i].compareTo(pivot) <= 0) {
                  ++ltePivot;
                  T tmp = arr[ltePivot];
                  arr[ltePivot] = arr[i];
                  arr[i] = tmp;
            }
      }

      int placeForPivot = ltePivot + 1;

      T tmp = arr[placeForPivot];
      arr[placeForPivot] = arr[h];
      arr[h] = tmp;

      return placeForPivot;
}
```

Pivot na sredini

- Šema koja ima sličnosti sa Hoarovom – koristimo dva indeksa koji se pomeraju, jedan sa leva na desno, a drugi sa desna na levo.
- Kod ovog particionisanja podniz A[l..h] transformišemo u oblik (LE QE) gde su
  - LE – elementi manji ili jednaki od pivota
  - QE – elementi veći ili jednaki od pivota
  - Pivot je poziciji (h + l) / 2
- Pivot može da završi ili u LE ili u QE (u zavisnosti od dinamike indeksa koji se pomeraju)
- Pomerajući indeksi će se “mimoići” na nekoj poziciji koja ne mora biti srednja pozicija u A.

Pivot na sredini



Quicksort – pivot na sredini

```
public static <T extends Comparable<T>> void sort(T[] arr) {
	sort(arr, 0, arr.length - 1);
}

private static <T extends Comparable<T>>
void sort(T[] arr, int l, int h) {
	T pivot = arr[(l + h) / 2];
	int i = l;
	int j = h;

	while (i <= j) {
		while (arr[i].compareTo(pivot) < 0) i++;
		while (arr[j].compareTo(pivot) > 0) j--;

		if (i <= j) {
			T tmp = arr[i];
			arr[i] = arr[j];
			arr[j] = tmp;
			i++;
			j--;
		}
	}

	if (l < j) sort(arr, l, j);
	if (i < h) sort(arr, i, h);
}
```

Dodatna unapređenja quicksort-a

- Podnizove A[l .. h] koji su male dužine (h – l < 17) je efikasnije sortirati elementarnim metodama

```
private static <T extends Comparable<T>>
void sort(T[] arr, int l, int h) {
  if (h - l < 17) ElementarySorts.insertionSort(arr, l, h);
  else {
    int j = partition(arr, l, h);
    sort(arr, l, j - 1);
    sort(arr, j + 1, h);
  }
}
```

- Performanse quicksorta drastično opadaju ako niz ima veliki stepen sortiranosti

```
public static <T extends Comparable<T>>
void sort(Comparable[] arr) {
     shuffle(arr); // randomizujemo niz (vidi bogosort)
     sort(arr, 0, arr.length - 1);
}
```

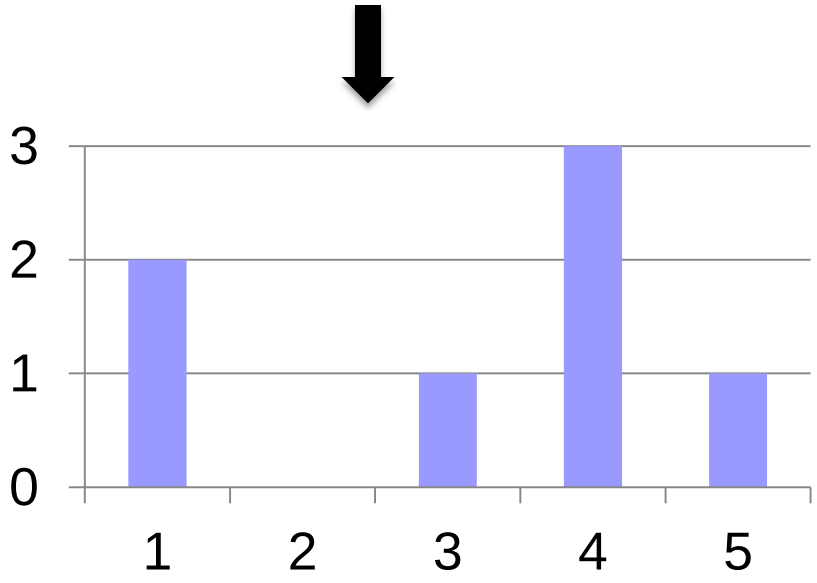
Sortiranje u linearnom vremenu

- Do sada smo videli postupke sortiranja koji imaju kvadratnu i linearitmičku vremensku složenost
- Nizove je moguće sortirati linearno uz određena ograničenja vezana za elemente niza i dodatni memorijski prostor
  - Ograničenja se odnose na tip elementa, opseg i raspodelu vrednosti.
- Linearni sortovi su esencijalno sortovi kod kojih ne poredimo elemente niza
  - Sortovi kod kojih se porede elementi niza i razmenjuju oni u inverziji su u najboljem slučaju linearitmični u proseku

Sortiranje brojanjem (counting sort)

- Pretpostavljamo da sortiramo ne-negativne cele brojeve.
- Ideja counting sorta je da
  - Od niza brojeva napravimo distribuciju brojeva
  - Na osnovu distribucije brojeva formiramo sortiran niz

4, 1, 1, 5, 3, 4, 4



1, 1, 3, 4, 4, 4, 5

Formiranje distribucije niza brojeva

- Distribuciju niza brojeva A možemo predstaviti nizom celih brojeva D koji
  - Ima dužinu m + 1, gde je m maksimum niza A
  - D[k] = broj ponavljanja broja k u nizu A.

- Formiranje distribucije:
  - Jedan prolaz kroz niz A da nađemo maksimum
  - Drugi prolaz kroz niz A da formiramo vrednosti niza D
    - D[A[i]]++ za svako i u intervalu [0, A.length – 1]

```java
public static void sort(int[] arr) {
	// pronalazimo maksimum niza
	int max = arr[0];
	for (int i = 0; i < arr.length; i++) {
		if (arr[i] > max)
			max = arr[i];
	}

	// formiramo distribuciju
	int[] distr = new int[max + 1];
	for (int i = 0; i < arr.length; i++) {
		distr[arr[i]]++;
	}

	// na osnovu distribucije formiramo sortiran niz
	int currentPosition = 0;
	for (int i = 0; i < distr.length; i++) {
		for (int j = 0; j < distr[i]; j++) {
			arr[currentPosition++] = i;
		}
	}
}
```