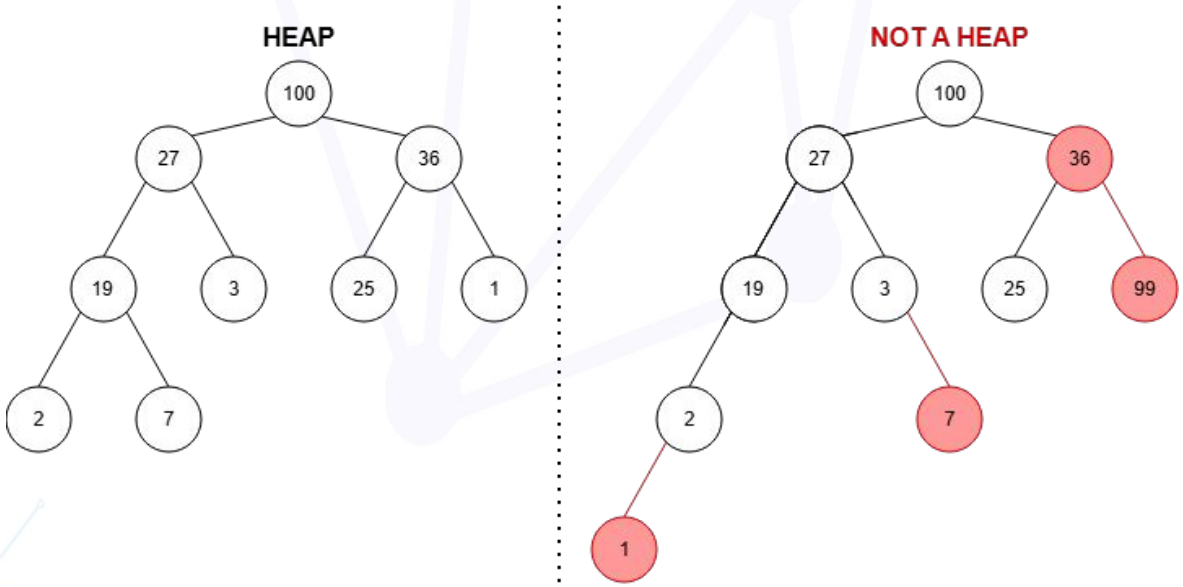
Teorija Algoritama

Heap strukture, Prioritetni redovi, Skip liste i Hibridni algoritmi

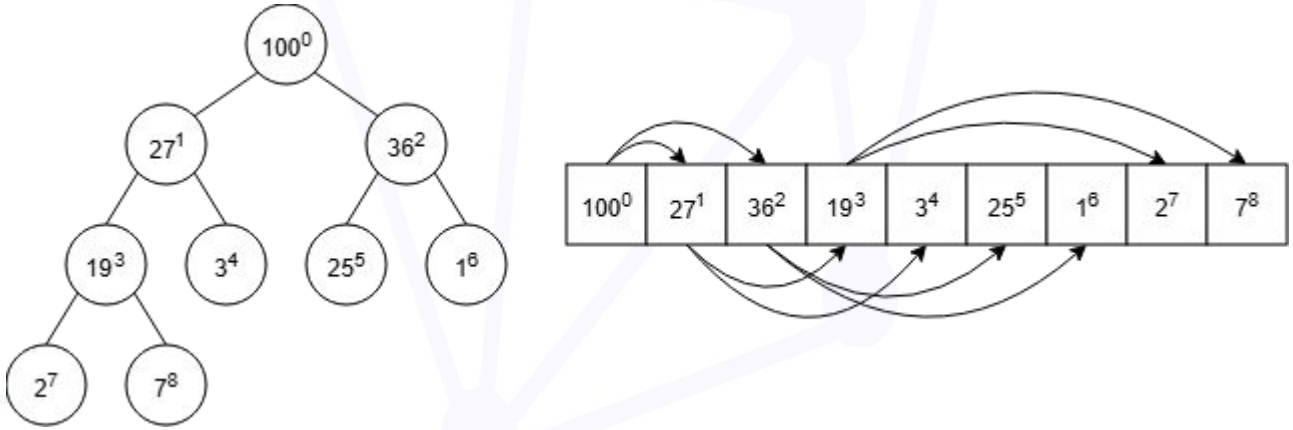
Struktura gomile - *Heap*

- Za prezentaciju *Heap* strukture, koristimo kompletno binarno stablo.
- Kompletno binarno stablo podrazumeva:
  - Svi elementi *Heap*-a imaju oba potomka, osim onih na poslednja dva nivoa;
  - Elementi u poslednjem nivou su poravnati ulevo.
- Dve varijacije:
  - Max-heap - Ključ roditelja je veći ili jednak od ključeva dece;
  - Min-heap - Ključ roditelja je manji ili jednak od ključeva dece.



Memorijska reprezentacija *Heap*-a

- Umesto spregnute strukture, koristi se jednodimenzionalni niz:
  - Prvi element je koren stabla Heap-a;
  - Ako se neki čvor nalazi na indeksu *i*, onda je njegovo levo dete na indeksu *2i + 1*, a desno na indeksu *2i + 2*;
  - Za svaki element na indeksu *i* (osim *i = 0*) ima roditelja i on se nalazi na indeksu *(i - 1) // 2*.



Operacije nad *Heap*-om

- Kreiranje gomile;
- Održavanje gomile;
- Dodavanje elementa;
- Brisanje najvećeg/najmanjeg elementa (korena)

Održavanje strukture gomile

- Polazi se od elementa sa indeksom *i* sa pretpostavkom da oba njegova podstabla već imaju strukturu gomile.

- Ispitamo koji od elemenata (roditelj i deca) je najveći:
  - Roditelj;
  - Levo dete (ako postoji);
  - Desno dete (ako postoji).

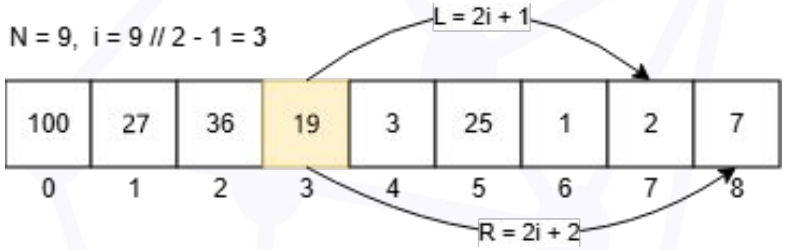
- Ukoliko je najveći element bilo dete, zamenimo ga sa roditeljom i rekurzivno nastavljamo postupak za:
  - Levo podstablo (i = 2i + 1) ukoliko je najveći element bilo levo dete;
  - Desno podstablo (i = 2i + 2) ukoliko je najveći element bilo desno dete.

Održavanje strukture gomile - *pseudo-kod*

```
function heapify(arr, i) // maxHeap example
    l = 2*i + 1
    r = 2*i + 2
    if l < arr.size and arr[l] > arr[i]
        largest = l
    else
        largest = i
    if r < arr.size and arr[r] > arr[largest]
        largest = r
    if largest != i
        swap(arr[i], arr[largest])
        heapify(arr, largest)
```

Kreiranje strukture gomile

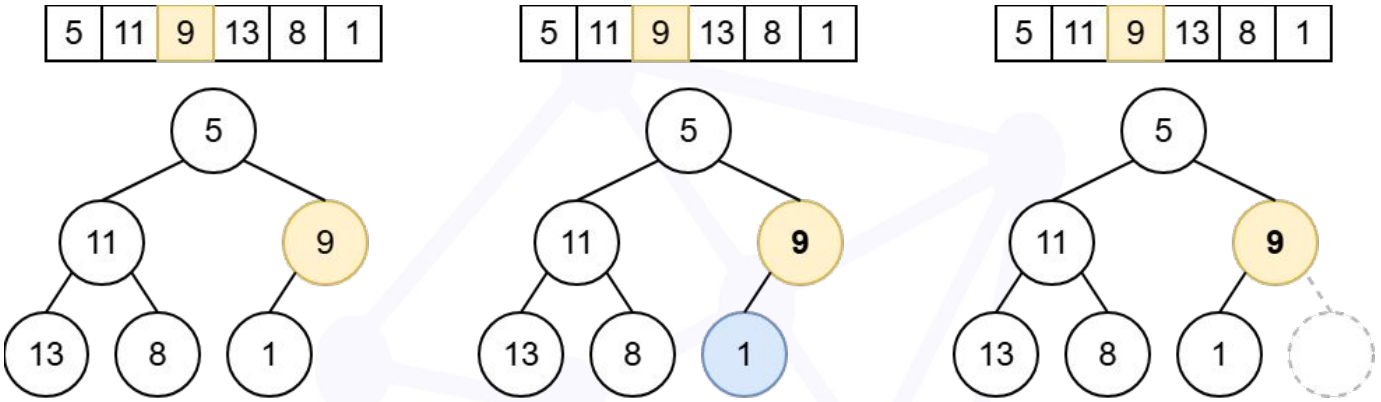
- Polazi se od proizvoljnog niza dužine *N*, koji nema strukturu gomile.
- Funkcija *heapify* se primenjuje za sve elemente koji imaju potomke, tj. od indeksa *i = N // 2 - 1* do korenskog elementa.



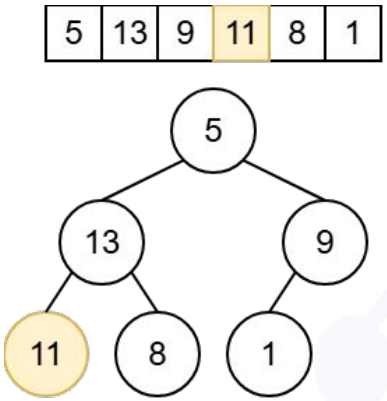
- Na ovaj način se postepeno, *bottom-up* pristupom formira struktura gomile.
- *Pseudo-kod:*

```
function buildHeap(arr)
    arr.heapSize = arr.size
    for i = (arr.size // 2 - 1) downto 0
        heapify(arr, i)
```

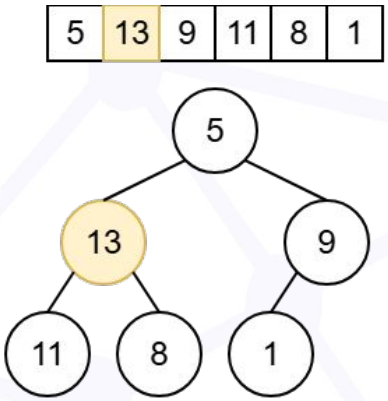
Kreiranje strukture gomile - primer



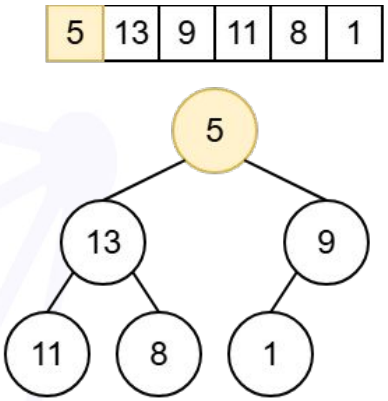
Kreiranje strukture gomile - primer



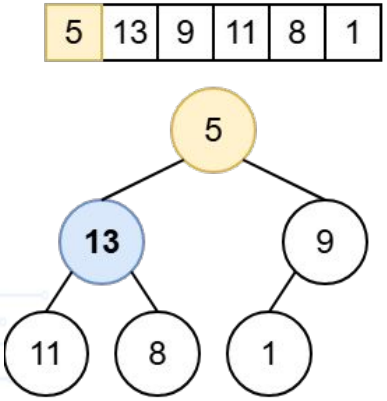
Pošto se desila zamena, nastavljamo rekurzivno za levo podstablo.



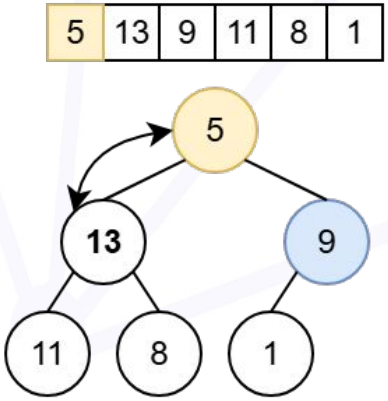
Element sa vrednošću 11 je list, tako da se vraćamo nazad na roditelja.



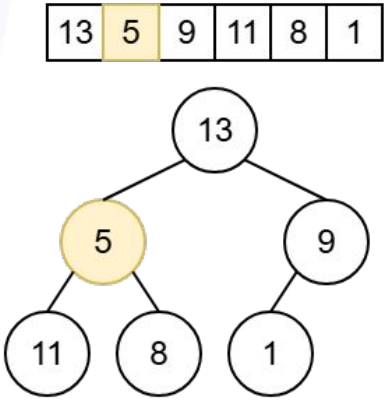
Nastavljamo za i = i - 1



Levo dete je veće od roditelja

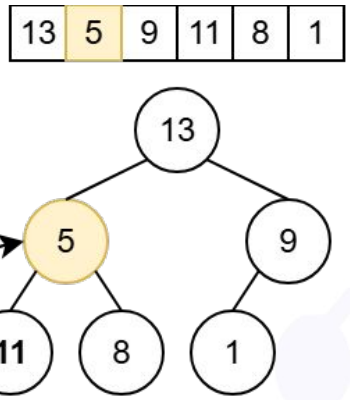


Levo dete je veće od desnog deteta - levo dete se menja sa roditeljem

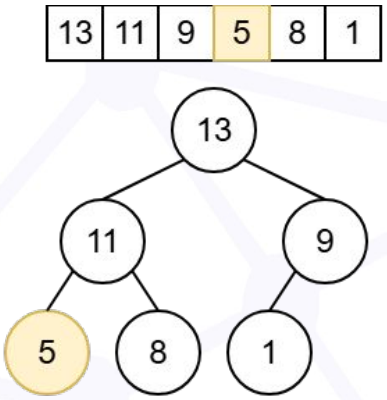


Desila se zamena roditelja i levog deteta, nastavljamo rekurzivno za levo podstablo

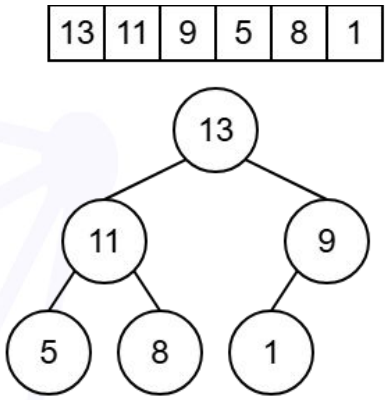
Kreiranje strukture gomile - primer



Od ova tri elementa, levo dete je najveće i menja se sa roditeljem



Element sa vrednošću 5 je list, tako da se vraćamo unazad rekurzivno.



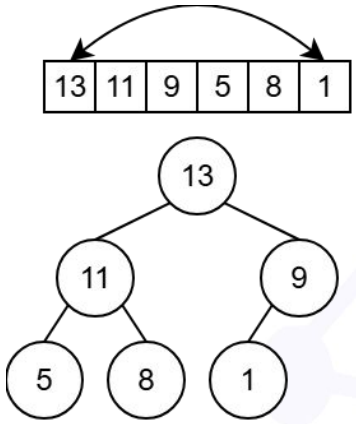
Završili smo algoritam i imamo Heap strukturu.

Heapsort algoritam

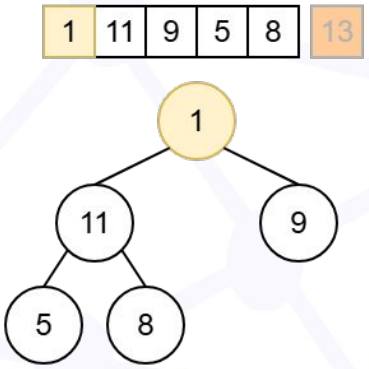
- Od proizvoljnog niza dužine *N* formiramo strukturu gomile.
- Nakon kreiranja gomile, prvi element (koren) je najveći/najmanji.
- Korenski element zamenimo sa poslednjim elementom u nizu.
- Pozivamo *heapify* za novi koren, pri čemu dužinu Heap strukture umanjujemo za jedan, kako bismo ponovo dobili strukturu gomile i samim tim sledeći najveći/najmanji element.
- Ovaj postupak ponavljamo dok veličina gomile ne postane 1 - u tom momentu imamo sortiran niz.
- *Pseudo-kod*:

```
function heapsort(arr)
    buildHeap(arr)
    for i = arr.size - 1 downto 1
        swap(arr[0], arr[i])
        arr.heapSize = arr.heapSize - 1
        heapify(arr, 0)
```

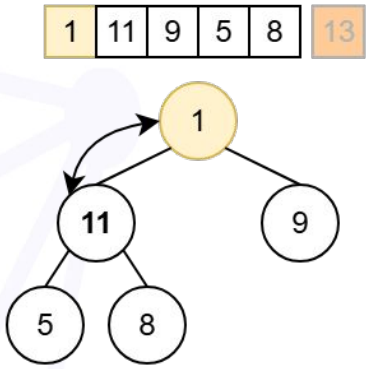
Heapsort - primer



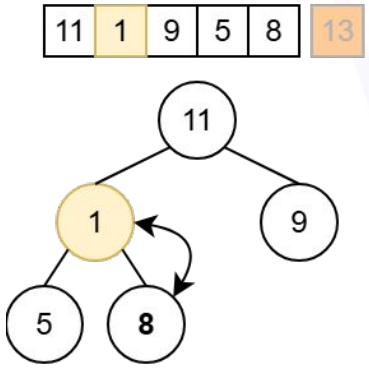
Za prethodno formiranu Heap strukturu, korenski element menjamo sa poslednjim



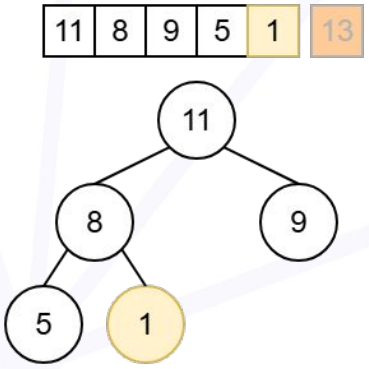
Pozivamo *heapify* nad novim korenom, ignorišući najveći element koji smo postavili na kraj



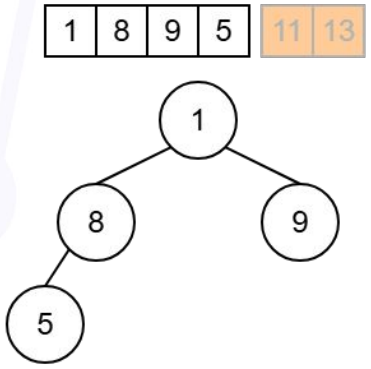
Element sa vrednošću 11 je najveći od ova tri i menja se sa roditeljom



Element sa vrednošću 8 je najveći od ova tri i menja se sa roditeljem



Došli smo do elementa koji je list, i sada ponovo imamo Heap strukturu



Korenski element menjamo sa poslednjim elementom, i ponavljamo postupak dok dužina Heap-a ne postane 1

Dodavanje elementa u gomilu

- Novi element se dodaje na kraj niza.
- Veličina gomile se poveća za 1.
- Ukoliko je dodati element veći od svog roditelja sledi zamena.
- Ponavljamo zamenu sve dok novododati element nije manji od roditelja ili postane korenski element.
- *Pseudo-kod*:

```
function insert(arr, x)
    arr.heapSize = arr.heapSize + 1
    arr[arr.heapSize] = x
    i = arr.heapSize - 1
    while i > 0 and arr[i] > arr[(i - 1) // 2]
        swap(arr[i], arr[(i - 1) // 2])
        i = (i - 1) // 2
```

Heapsort - performanse

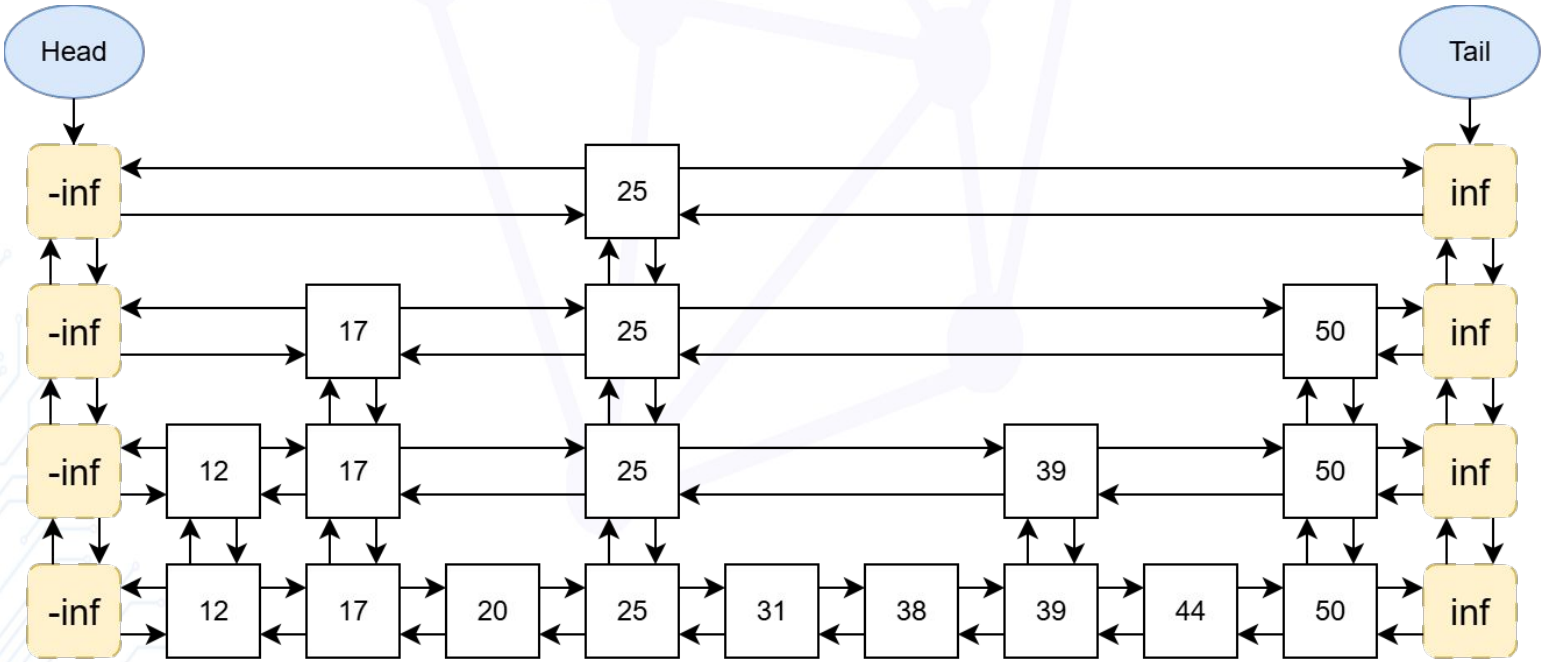
- Vremenska složenost: $O(n \cdot \log n)$
- Prostorna složenost: $O(1)$
- *Heapsort* je iterativni algoritam, pa samim tim nema preciznu rekurzivnu formulu.
- Stabilnost: *Nije stabilan*
- Primer: *heap.c*

Prioritetni redovi

- Prioritetni red je apstraktni tip podataka sličan običnom redu, pri čemu svaki element ima pridružen prioritet.
- Elementi sa većim prioritetom se uslužuju pre onih sa manjim, bez obzira kad su dodati u strukturu.
- Najefikasniji način za implementaciju je binarni *Heap*, jer omogućava operacije dodavanja/brisanja u logaritamskom vremenu.
- Osnovne operacije:
  - Dodavanje;
  - Uklanjanje elementa sa najvećim prioritetom.
  - Uvid u element sa najvećim prioritetom.
- Ideja je da svaki element, koji se dodaje u prioritetni red, pored neke svoje vrednosti ima i svoj prioritet.
- Dodavanje elementa u prioritetni red je ekvivalentno dodavanju novog elementa u *Heap*, gde je kriterijum za poređenje prioritet.

Skip liste

- Skip liste su uređena struktura koja nudi sve operacije u prosečnoj logaritamskoj vremenskoj složenosti.
- Zasnovana je na sortiranoj dvostruko spregnutoj listi.
- Od osnovne liste, formiraju se dodatne liste (nivoi) propagacijom elemenata sa verovatnoćom p (najčešće $p = 0.5$)

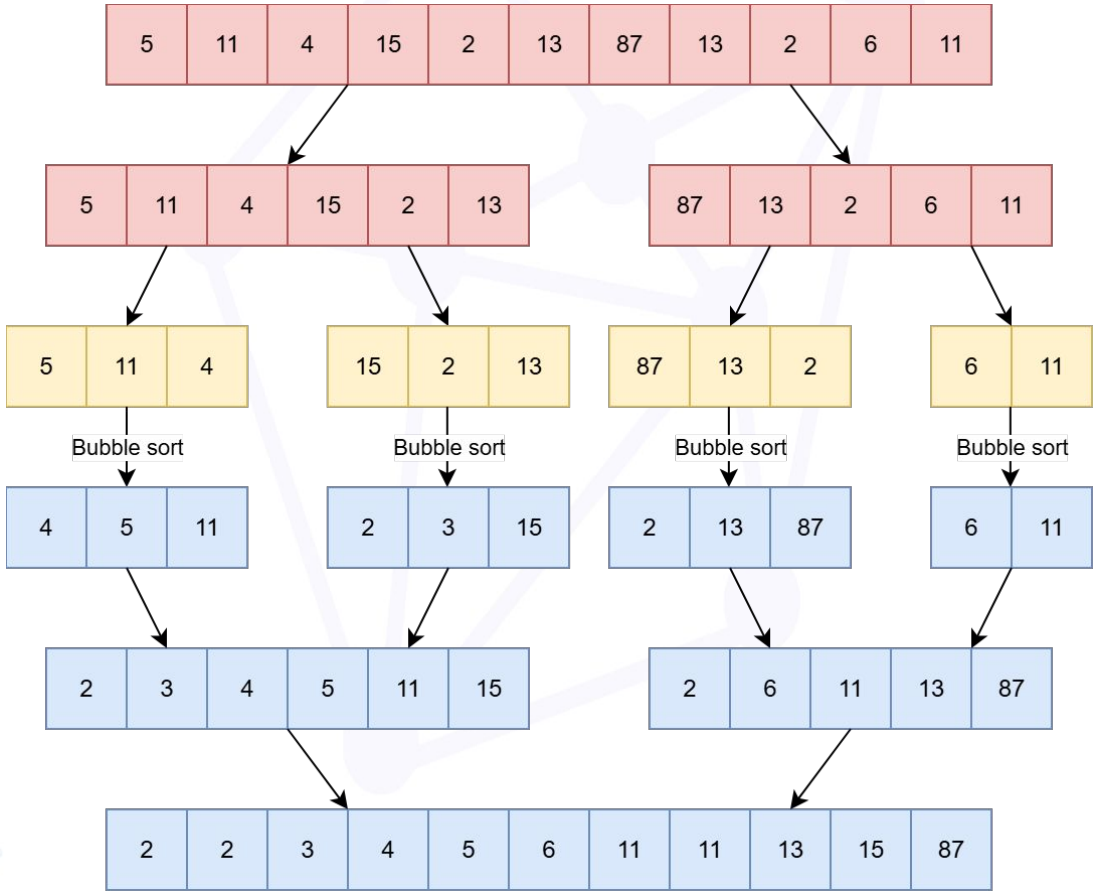


Hibridni algoritmi

- Hibridni algoritmi se zasnivaju na ideji da neki jednostavni algoritmi, poput Selection ili Bubble sort-a, rade brže na malim nizovima u odnosu na algoritme koji su dosta efikasniji na velikim nizovima, na primer Quicksort i Merge sort.
- Ideja je, koristiti algoritam, koji se oslanja na strategiju *zavadi pa vladaj*, sve dok potproblem ne postane manji od neke zadate granice.
- Kada potproblem postane manji od neke zadate granice, umesto da nastavimo koriscenje kompleksnog algoritma, potproblem resavamo nekim jednostavnijim algoritmom.

Hibridni algoritam - primer

- Merge sort + Bubble sort (za granicu n = 3)



Pripremni zadatak za K1

- U datotekama `pz1.c` i `utils.h` nalazi se postavka zadatka.
- Datoteku `utils.h` **ne menjate**, ona služi za testiranje.
- U datoteci `pz1.c` nalazi se implementirana `main` funkcija, koju takođe **ne menjate**, ona poziva testiranja.
- U datoteci `pz1.c`, se takođe nalaze 3 prazne funkcije, namenjene za tri sorta koja treba da implementirate:
  - Selection sort;
  - Merge sort;
  - Hybrid sort - koji koristi Merge sort dok su potproblemi veći od 15 elemenata, nakon čega koristi Selection sort.
- **Jako je bitno** da ne menjate potpis 3 date funkcije, jer u suprotnom testiranje neće raditi. Po potrebi možete (čitajte trebate) implementirati dodatne funkcije koje pozivate iz 3 date funkcije.
- Parametri za testiranje definisani na početku datoteke `pz1.c`:
  - `TEST_SORT_EL` i `TEST_SORT_ITERS` - dužina niza i broj pokretanja prilikom testiranja da li vaš sort uspešno sortira elemente.
  - TEST_TIME_MIN_EL i TEST_TIME_MAX_EL - određuje broj elemenata za testiranje (merenje) vremena izvršavanje sorta (npr. za 10 i 10000 > 10, 100, 1000, 10000).
  - TEST_TIME_ITERS - broj pokretanja merenja za svaki broj elemenata, nakon čega će biti izračunato prosečno vreme.