Sorting using heaps

- We can first build heap, then repeat: remove max.
- In place:

  BuildHeap
  for i=n down to 2
  exch A(1), A(i)
  Heapify(A,1,i-1)

- Essentially same as the first approach.
  We use the fact that:
  - heap becomes smaller after "remove max",
  - last array entry becomes free.

Example

Sort:

```
5 7
3 2 1 7 3 2 8 7 5 1
```

Variations on Heaps

- Min instead of max.
- K-ary instead of binary
  - Time for Insert: \( \log_k n \) (n = #elems. in heap).
  - Time for extract-max: \( kn \log_k n \)
  - Best value of k to use is determined by application:
    - Mostly inserts: use big k (e.g. \( k = \sqrt{n} \))
    - Mostly extract-max: use small k (i.e. \( k=2 \) or \( 3 \))

Lower bound for sorting

- All sorting algs that we saw: comparison-sorts
- only operation allowed on data is comparison.
- Is \( O(n \log n) \) the best we can do in this case?
- Represent computation by decision tree:
  - Execution - walk from root to a leaf.

More lower bound

- 1 leaf per each possible answer.
- at least \( n! \) leaves.
- Binary tree with \( n! \) leaves has to be \( \Omega(n \log n) \) deep !
  - worst-case execution time is \( \Omega(n \log n) \)
- In the comparison model, quicksort, mergesort, etc are optimum.
- HW: Why doesn't this work for selection ??
  - What if instead of sorting, we need to divide into groups of, say 10, and sort the groups
    - all element of 1st group + all elements of 2nd group, etc

Counting Sort

- Is \( \Omega(n \log n) \) indeed the limit ?? NO !
- Example: Counting Sort
  - Assume inputs are in \([1,...,k]\), integers.
  - let \( k = \sum_{i=1}^{m} c_i \) for all \( \sum_{i=1}^{m} c_i = n \)
  - compute prefix sum \( c_1(1), c_2(2), \ldots, c_i(i) \) for all \( c_i \in k \)

  ```
  Input: 1, 1, 5, 5, 7
  Output: 1, 1, 5, 5, 7
  ```

  Counting sort: note the stability of intermediate sort requirement !