Sorting: complexity & algorithm design strategies

Petr Kurapov

MIPT

Fall 2024

Agenda

- Incremental strategy: Insertion sort
- Recursive strategy: merge-sort, quicksort
- Heap sort
- Quick select

Why sorting?

- Many problems become *much* easier when the input is sorted, ie:
  - Searching, min/max, k-th statistics
- Applications:
  - Uniqueness and duplicate removal
  - Prioritization & scheduling
  - Set operations (intersection/union)
  - Efficient searching
  - …

Insertion sort

- Start with one element in target sorted array
- Pick next element and insert it into its proper sorted order
- Repeat for others left

Insertion sort

```
def insertionSort(A):
    for j in range(1, len(A)):
        k = A[j]
        i = j - 1
        while i >= 0 and A[i] > k:
            A[i+1] = A[i]
            i = i - 1
        A[i+1] = k
```

Insertion sort

```
def insertionSort(A):
    for j in range(1, len(A)):
        k = A[j]
        i = j - 1
        while i >= 0 and A[i] > k:
            A[i+1] = A[i]
            i = i - 1
        A[i+1] = k
```

Insertion sort

```python
def insertionSort(A):
    for j in range(1, len(A)):
        k = A[j]
        i = j - 1
        while i >= 0 and A[i] > k:
            A[i+1] = A[i]
            i = i - 1
        A[i+1] = k
```

Costs:

N

$\sum t_j$

Insertion sort

```python
def insertionSort(A):
    for j in range(1, len(A)):
        k = A[j]
        i = j - 1
        while i >= 0 and A[i] > k:
            A[i+1] = A[i]
            i = i - 1
        A[i+1] = k
```

Total:

$$T(n) = C + C_1 n + C_2 \sum t_j$$

Best case: $t_j = 1$

Worst case: $t_j = j$

Insertion sort

```
def insertionSort(A):
    for j in range(1, len(A)):
        k = A[j]
        i = j - 1
        while i >= 0 and A[i] > k:
            A[i+1] = A[i]
            i = i - 1
        A[i+1] = k
```

Total:

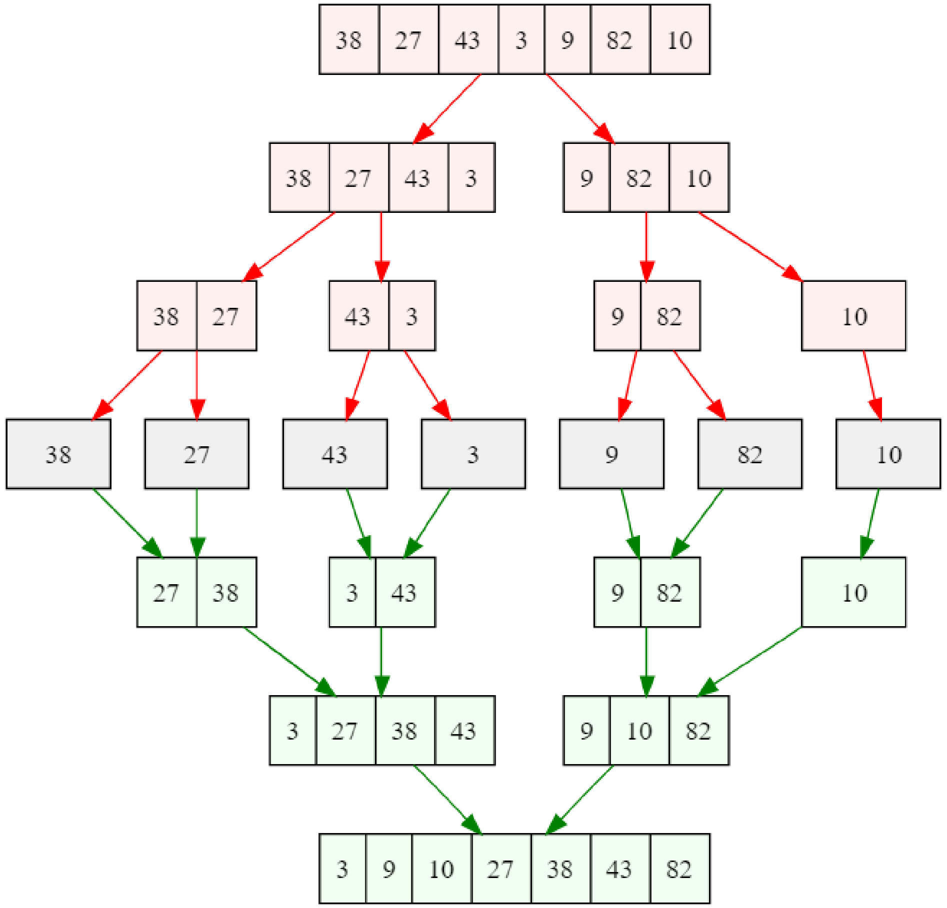
$$T(n) = C + C_1 n + C_2 \sum t_j$$

Best case: $t_j = 1 \rightarrow \Theta(n)$

Worst case: $t_j = j \rightarrow \Theta(n^2)$

Merge sort

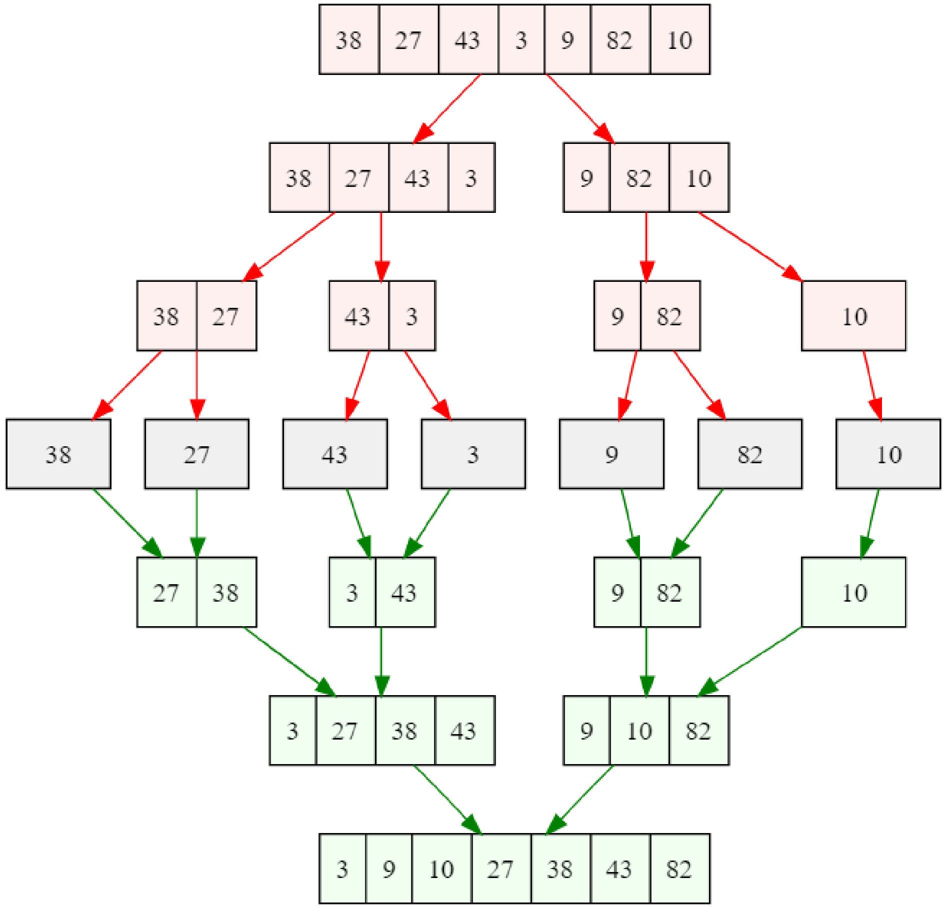
- Divide
  - N-element base is split into two n/2 subsequences
- Conquer
  - Sort these two subsequences recursively
- Combine
  - Merge sorted sequences to get the result



Source: https://en.wikipedia.org/wiki/Merge_sort#/media/File:Merge_sort_algorithm_diagram.svg

Merge sort

- Divide
  - ?
- Conquer
  - ?
- Combine
  - ?

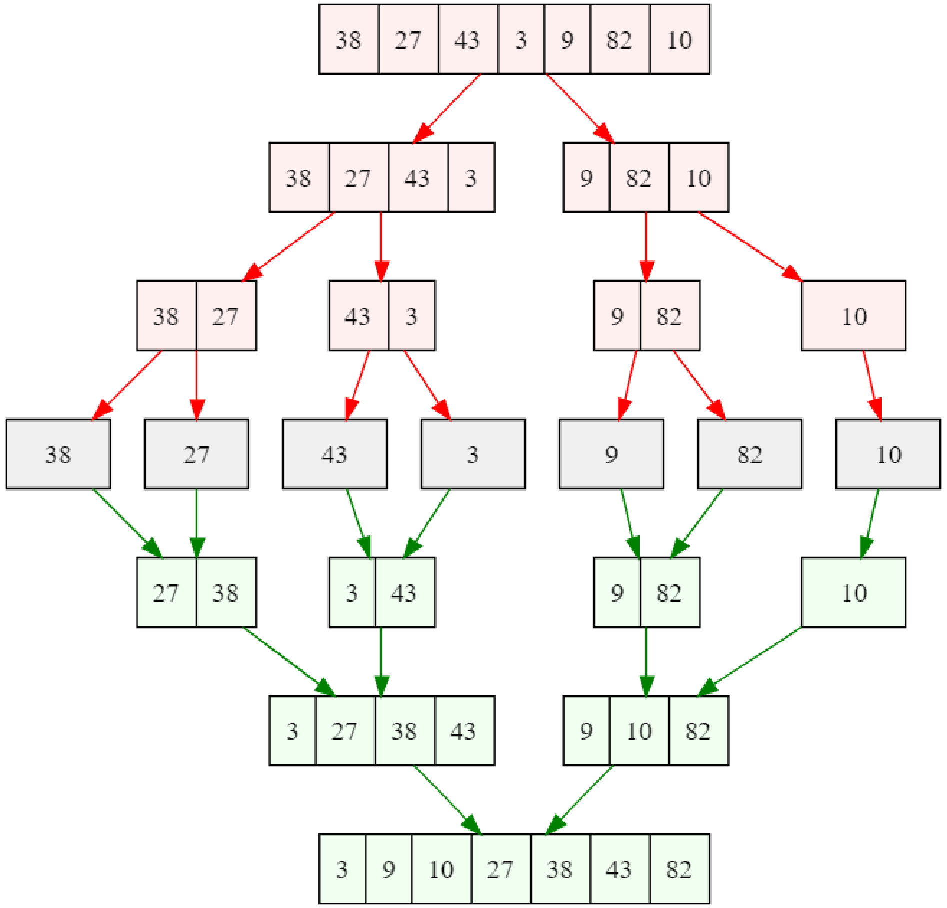


Source: https://en.wikipedia.org/wiki/Merge_sort#/media/File:Merge_sort_algorithm_diagram.svg

Merge sort

- Divide
  - $\Theta(1)$
- Conquer
  - $2T(n/2)$
- Combine
  - $\Theta(n)$

- $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$
- $T(1) = \Theta(1)$



Source: https://en.wikipedia.org/wiki/Merge_sort#/media/File:Merge_sort_algorithm_diagram.svg

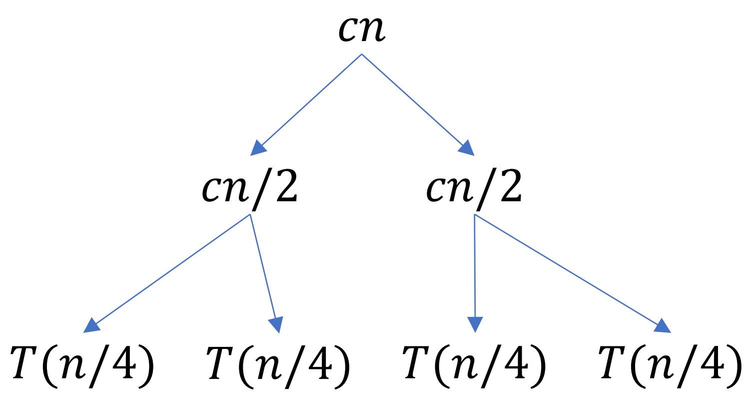
Merge sort

- $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$
- $T(1) = \Theta(1)$

$$
\begin{tikzcd}
 & cn \arrow[dl] \arrow[dr] & \\
T(n/2) & & T(n/2)
\end{tikzcd}
$$

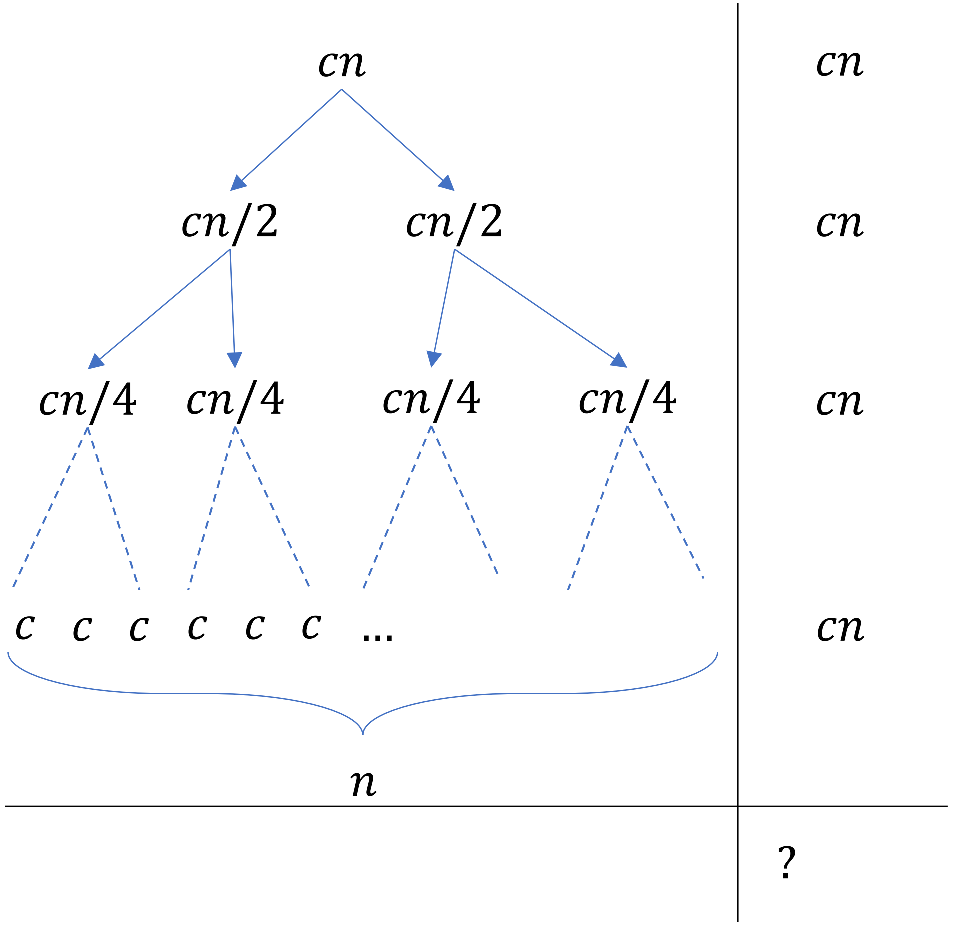
Merge sort

- $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$
- $T(1) = \Theta(1)$



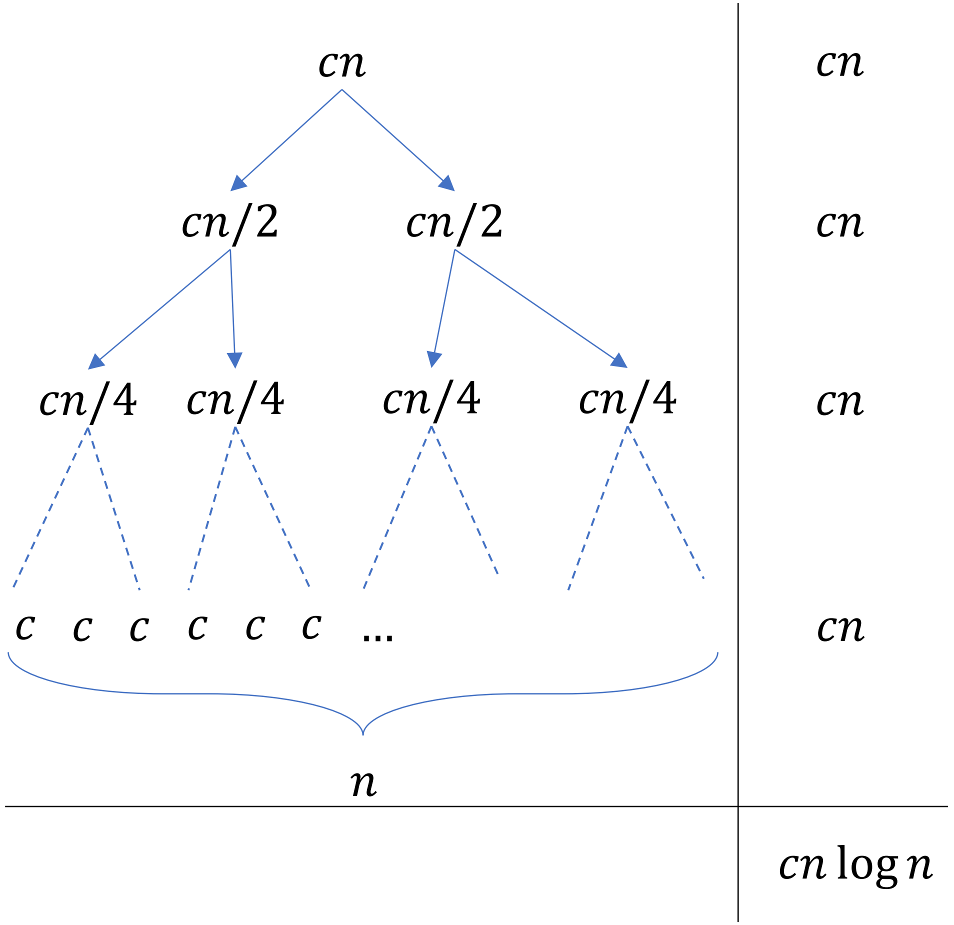
Merge sort

- $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$
- $T(1) = \Theta(1)$



Merge sort

- $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$
- $T(1) = \Theta(1)$

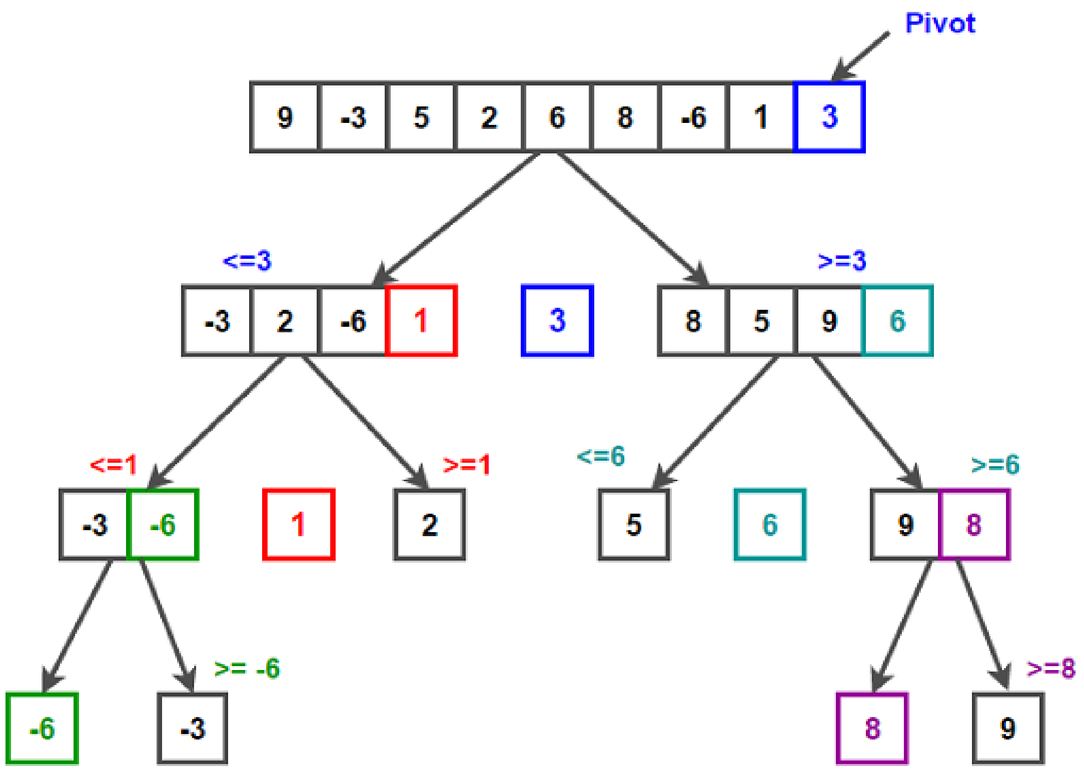


Quicksort

- Choose pivot item and do partition – split into two parts greater than pivot and less than pivot
- Recursively sort the two parts
- Combining not required as sorting is done in-place

Quicksort

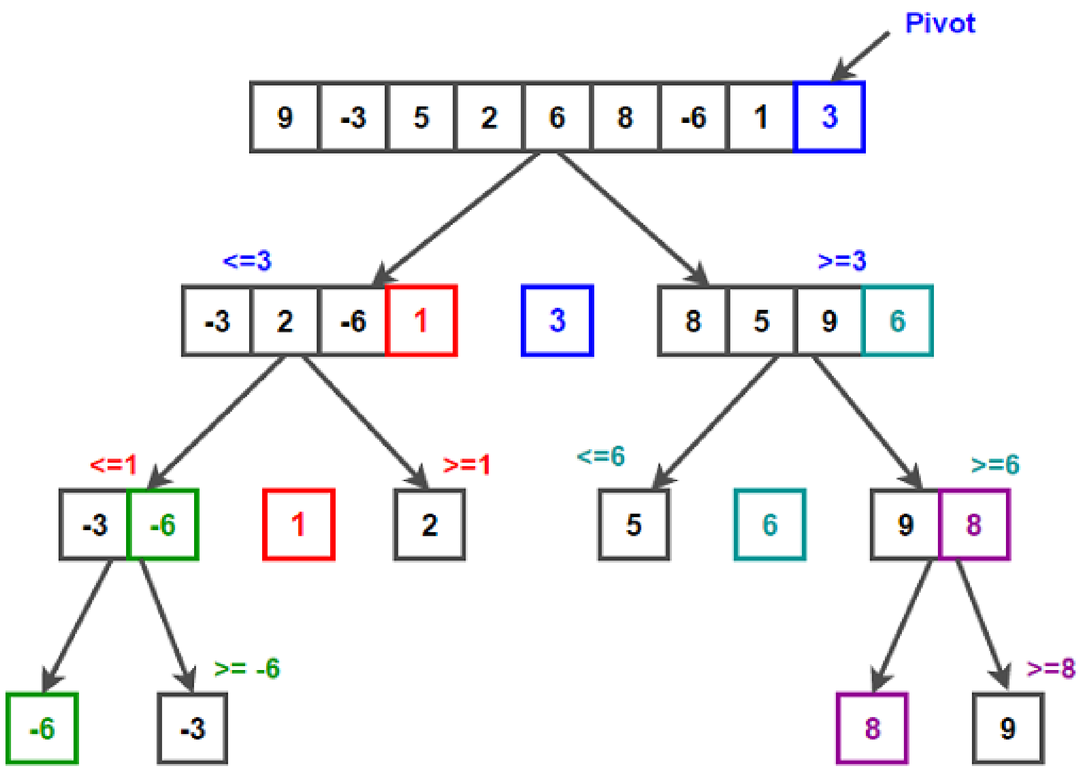
```
def qsort(A, p, r):
    if p < r:
        q = partition(A, p, r)
        qsort(A, p, q-1)
        qsort(A, p, q+1)
```



Source: https://www.techiedelight.com/quicksort/

Quicksort

```
def partition(A, p, r):
    x = A[r]
    i = p - 1
    for j in range(p, r):
        if A[j] <= x:
            i = i + 1
            swap(A[i], A[j])
    swap(A[i+1], A[r])
```



Source: https://www.techiedelight.com/quicksort/

Quicksort

- Worst case
  - Partition splits into n-1 and 0 elements, works in $\Theta(n)$
  - $T(n) = T(n-1) + T(0) + \Theta(n)$
  - Arithmetic progression, so $\Theta(n^2)$ in total
- Best case
  - Partition splits into n/2 parts
  - $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$
  - $\Theta(n \log n)$

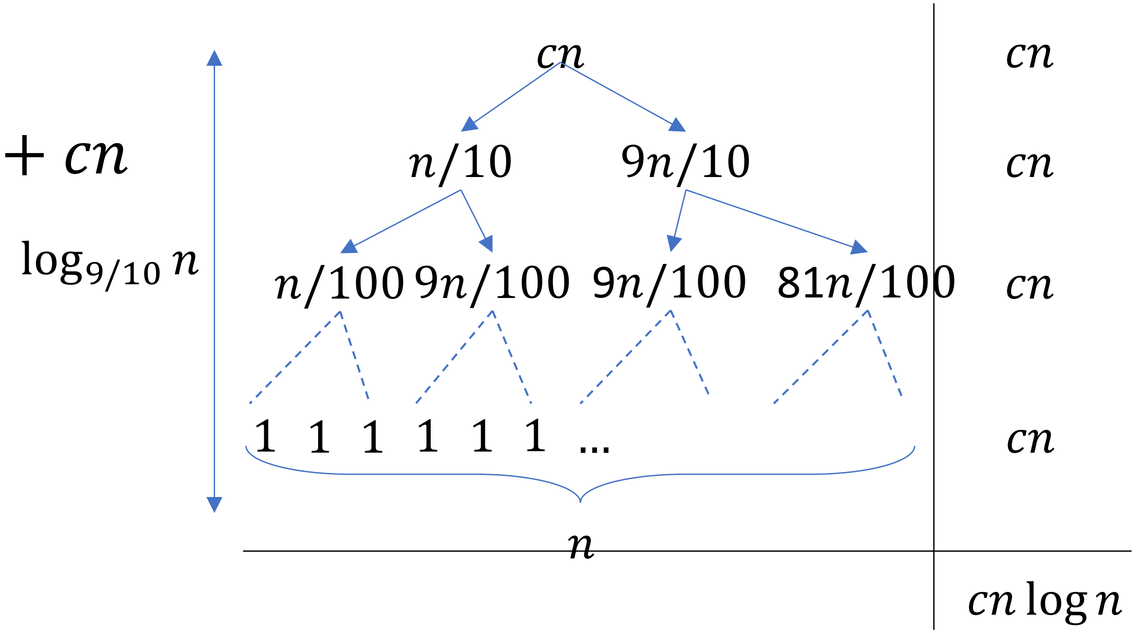
Quicksort

- Worst case
  - Partition splits into n-1 and 0 elements, works in $\Theta(n)$
  - $T(n) = T(n-1) + T(0) + \Theta(n)$
  - Arithmetic progression, so $\Theta(n^2)$ in total

Notice for $T(n) = T\left(\frac{9n}{10}\right) + T\left(\frac{n}{10}\right) + cn$

recursion tree:

- Best case
  - Partition splits into n/2 parts
  - $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$
  - $\Theta(n \log n)$



Quicksort

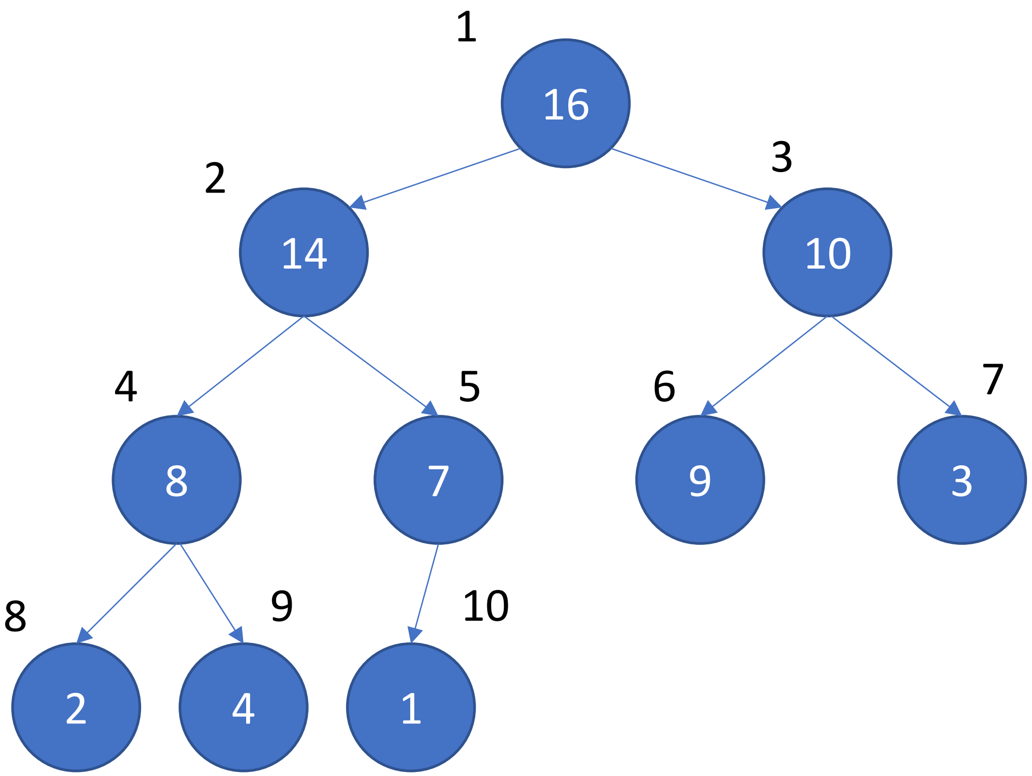
- Complexity is highly susceptible to the input and can be manipulated
- i.e. DoS attack on a service

- Best case: $\Theta(n \log n)$
- Worst case: $\Theta(n^2)$

Solution: change pivot & partition choosing technique:
Hoare’s, rand(), median-of-three

Heap

- Max-Heap:



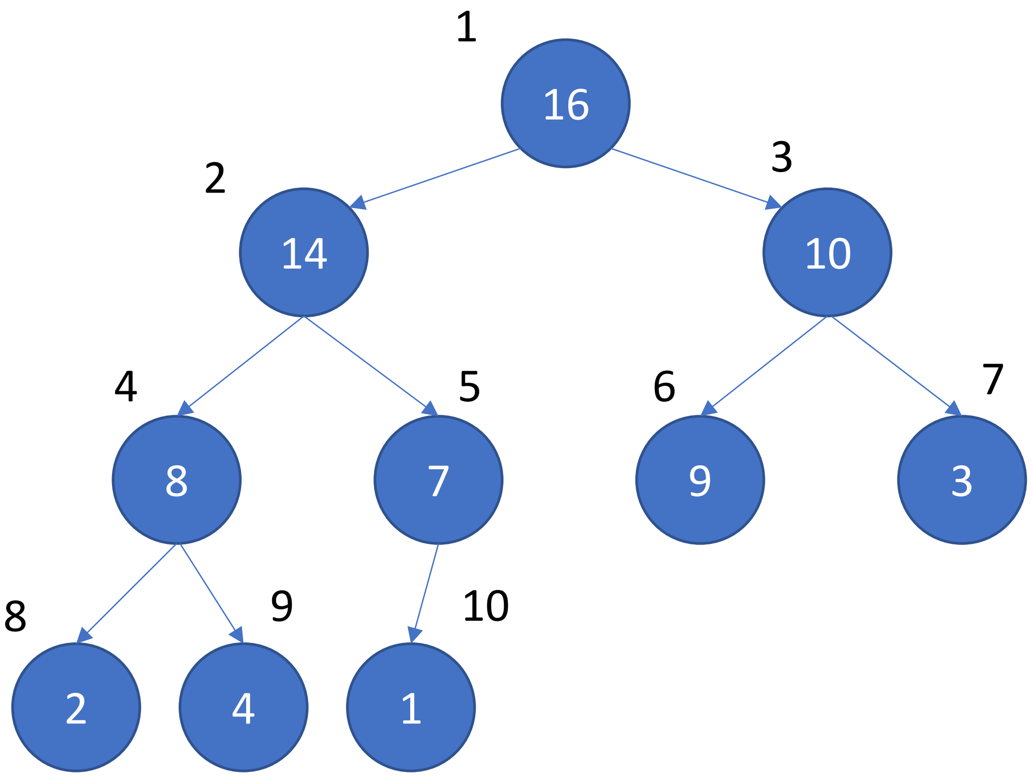
Complete* binary tree

- Completeness
- Heap property

1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

Heap

- Max-Heap:



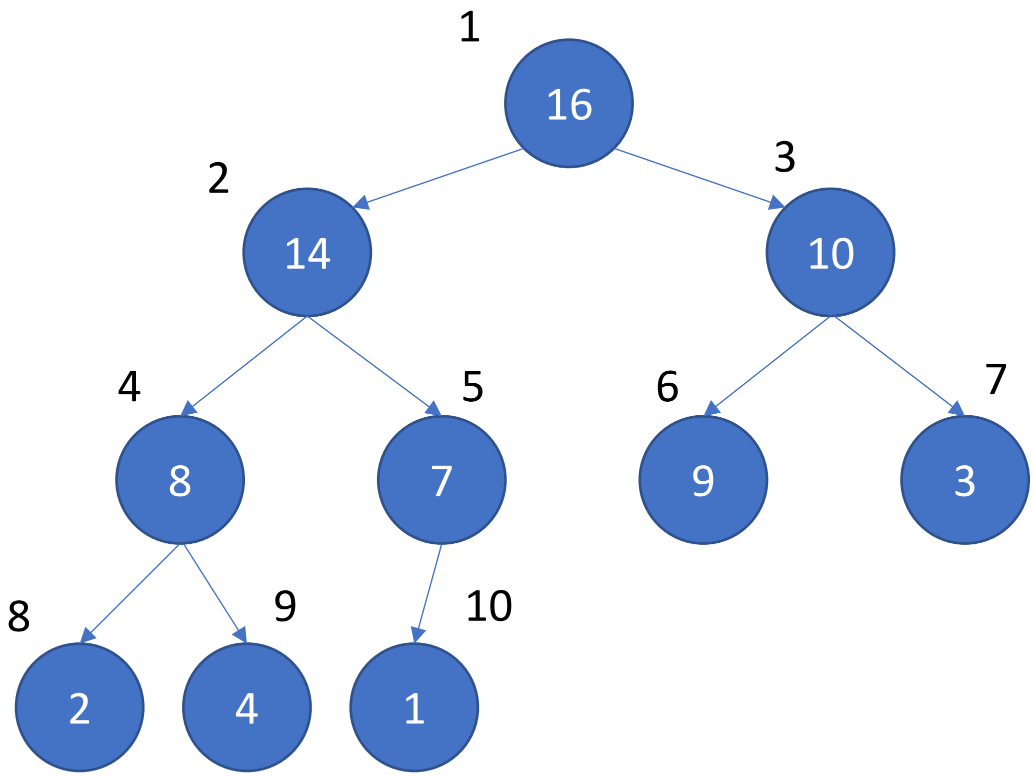
Complete* binary tree

- Get parent:
- Get left:
- Get right:

1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

Heap

- Max-Heap:



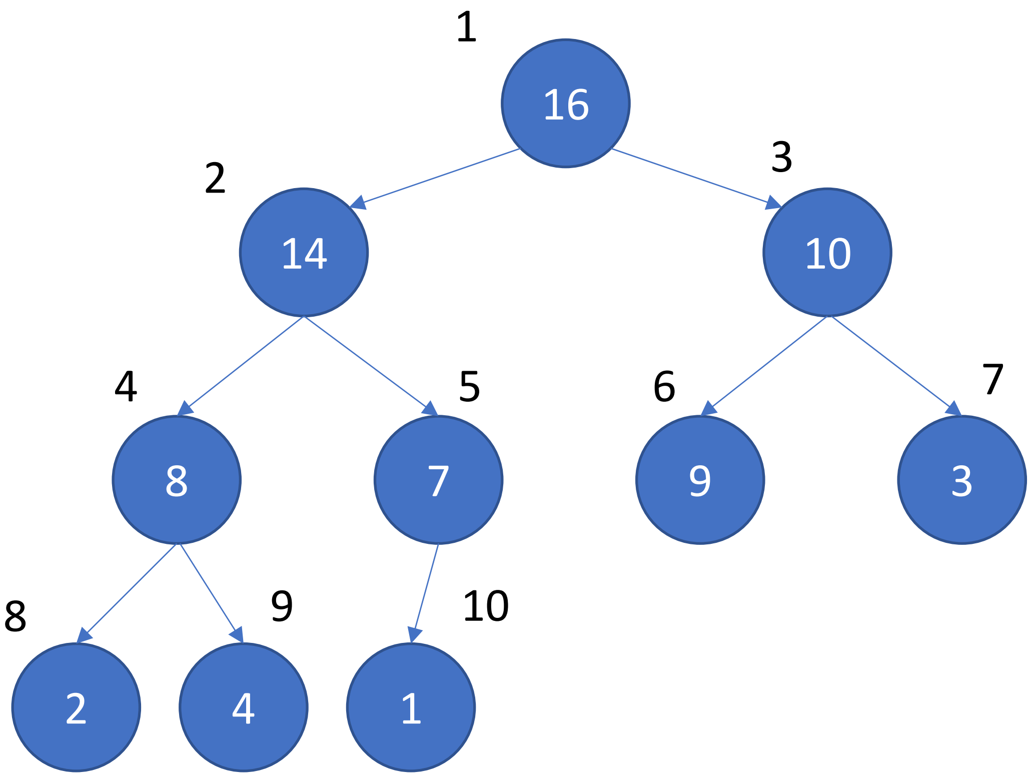
Complete* binary tree

- Get parent: floor(i/2)
- Get left: 2i
- Get right: 2i+1

1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

Heap: insertion

- Max-Heap:



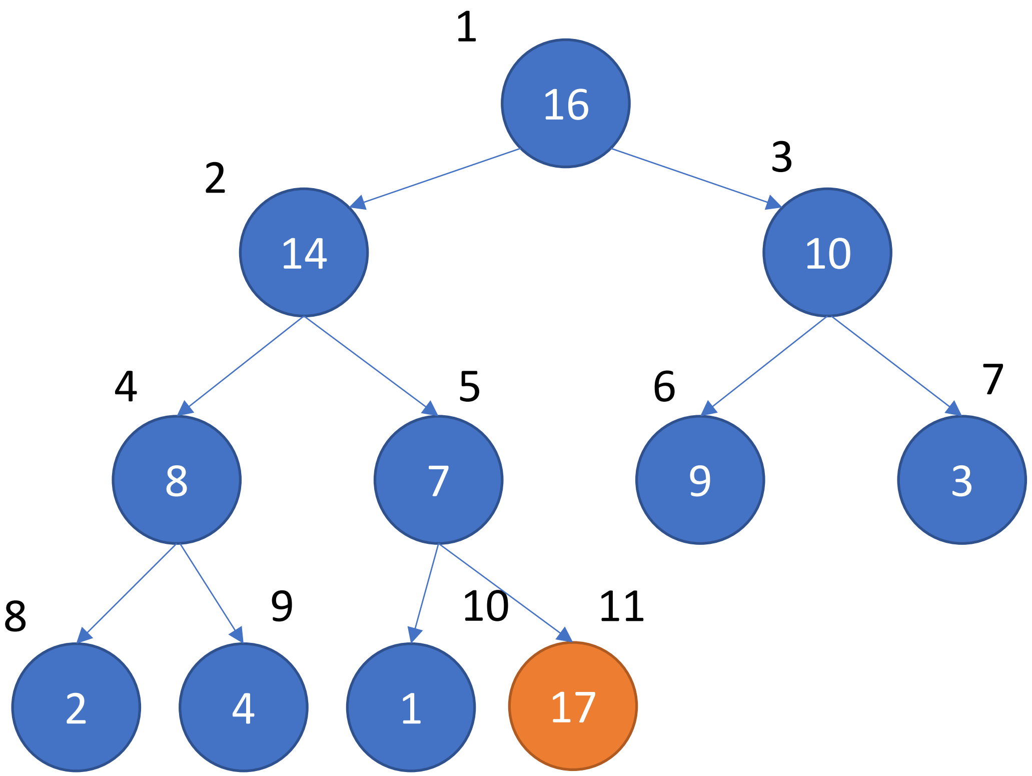
Complete* binary tree

New node

1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

Heap: insertion

- Max-Heap:



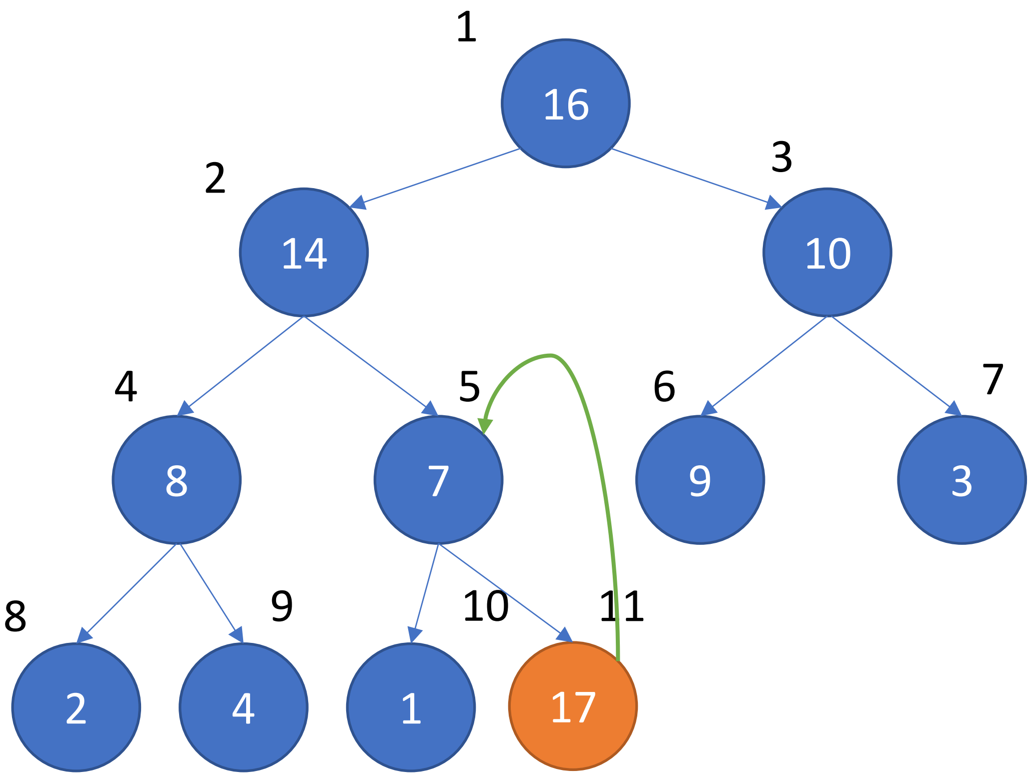
Complete* binary tree

- Push back

1	2	3	4	5	6	7	8	9	10	11
16	14	10	8	7	9	3	2	4	1	17

Heap: insertion

- Max-Heap:



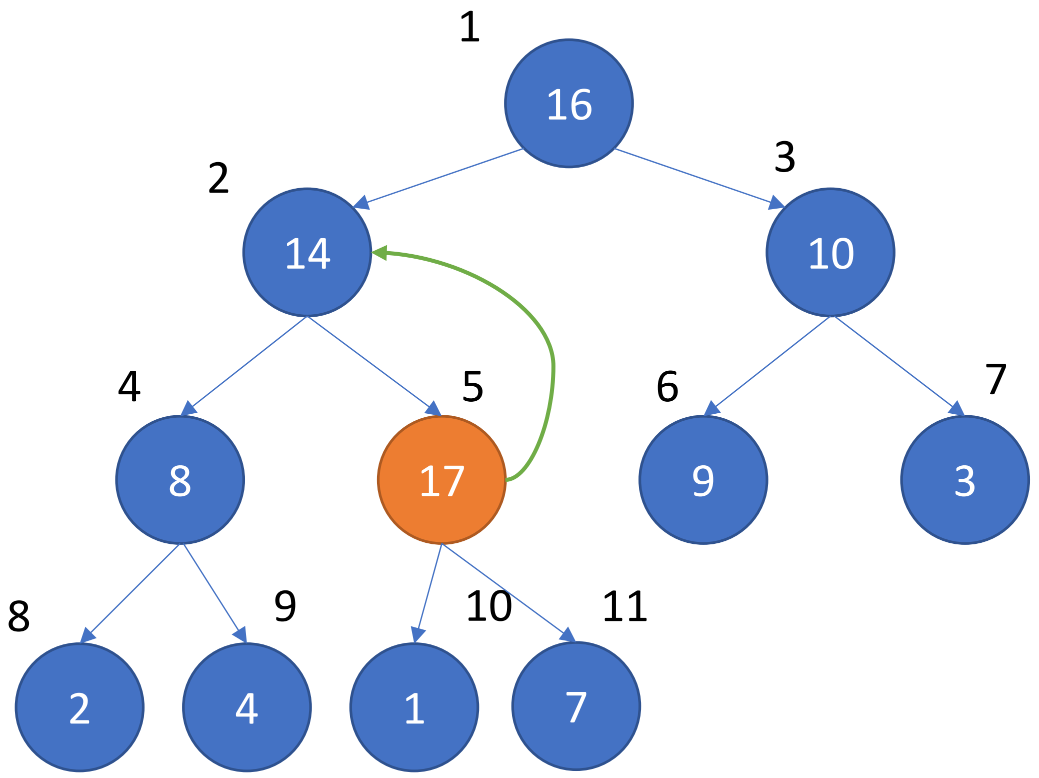
Complete* binary tree

- Push back
- Bubble it up, comparing to parent

1	2	3	4	5	6	7	8	9	10	11
16	14	10	8	7	9	3	2	4	1	17

Heap: insertion

- Max-Heap:



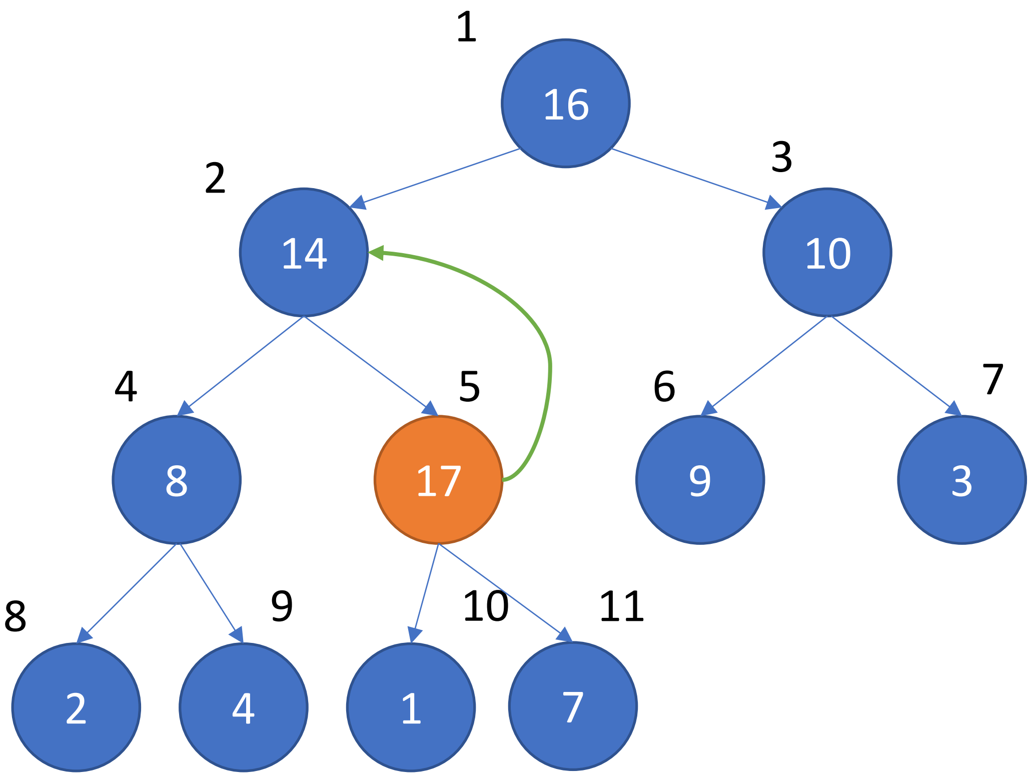
Complete* binary tree

- Push back
- Bubble it up, comparing to parent
- Until in place
- Complexity?

1	2	3	4	5	6	7	8	9	10	11
16	14	10	8	17	9	3	2	4	1	7

Heap: insertion

- Max-Heap:



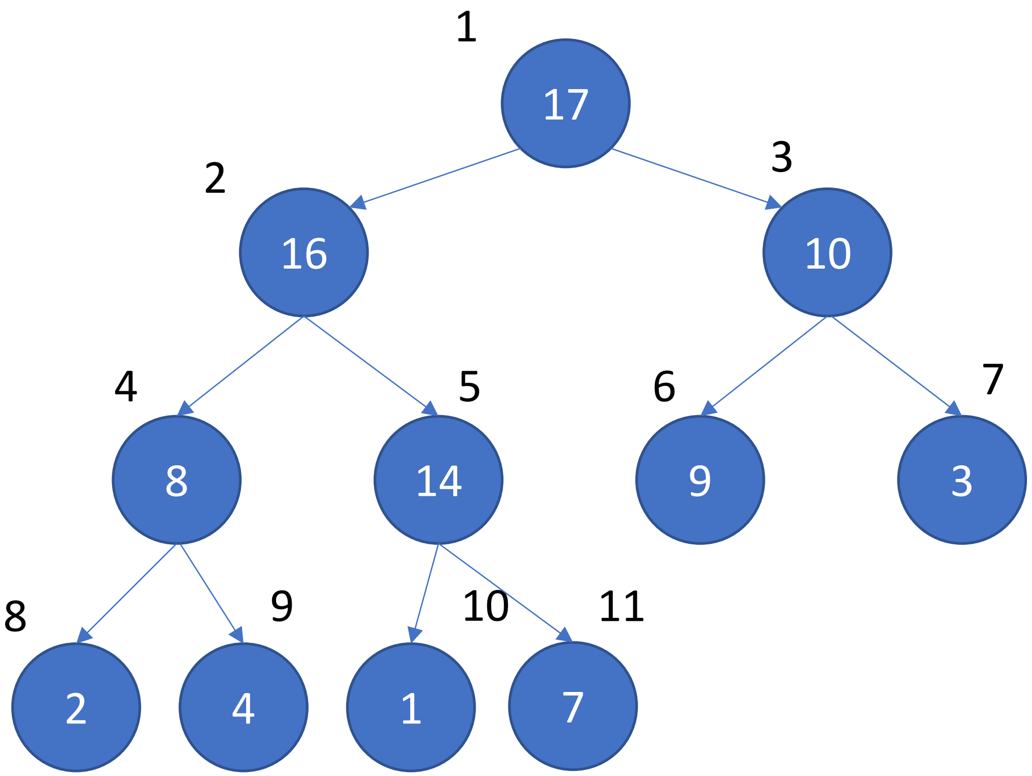
Complete* binary tree

- Push back
- Bubble it up, comparing to parent
- Until in place
- Complexity: $\Theta(\log n)$

1	2	3	4	5	6	7	8	9	10	11
16	14	10	8	17	9	3	2	4	1	7

Heap: pop

- Max-Heap:



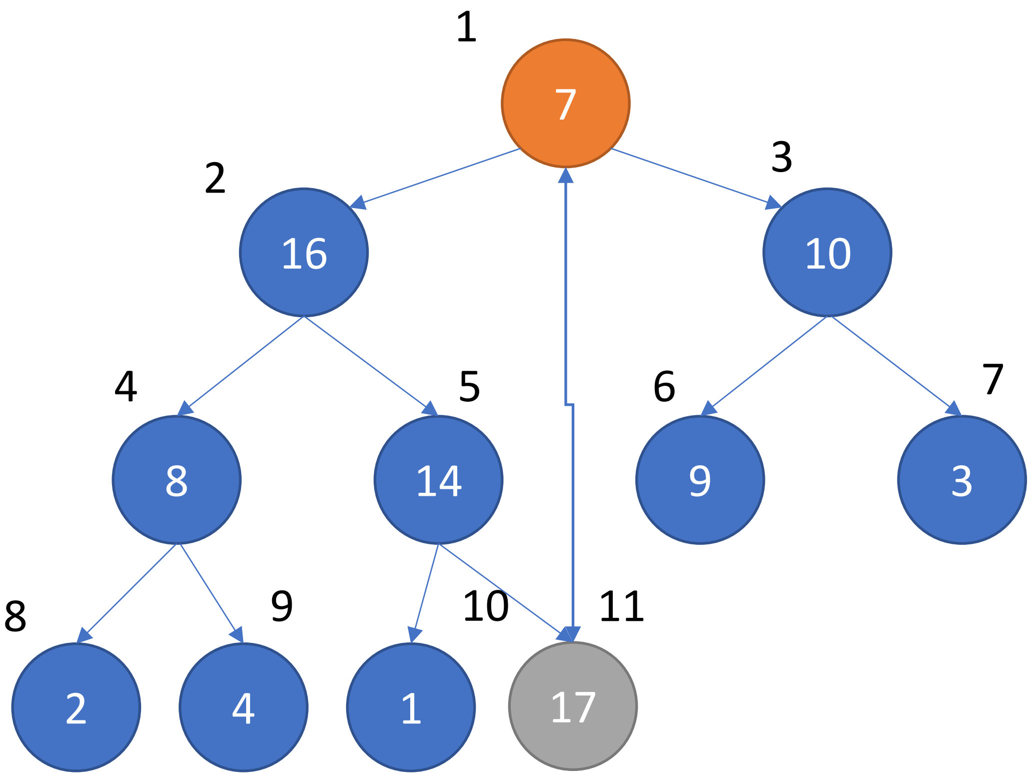
Complete* binary tree

- Let’s pop largest (min) element

1	2	3	4	5	6	7	8	9	10	11
17	16	10	8	14	9	3	2	4	1	7

Heap: pop

- Max-Heap:



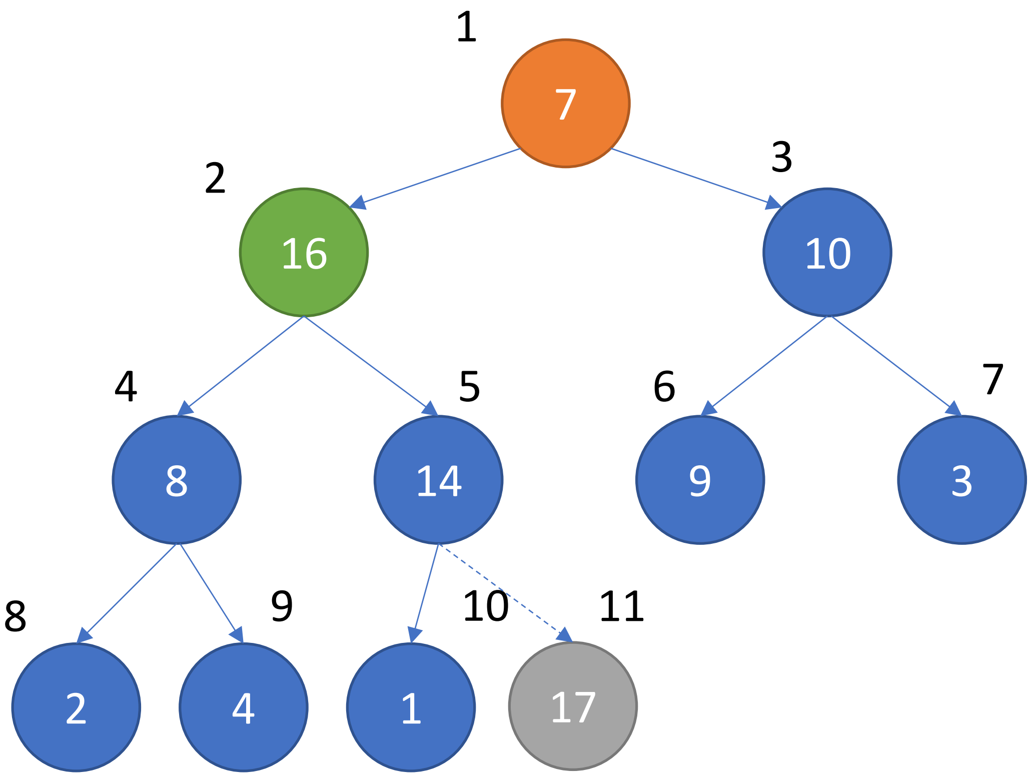
Complete* binary tree

- Let’s pop largest (min) element
- Swap last with root

1	2	3	4	5	6	7	8	9	10	11
7	16	10	8	14	9	3	2	4	1	17

Heap: pop

- Max-Heap:



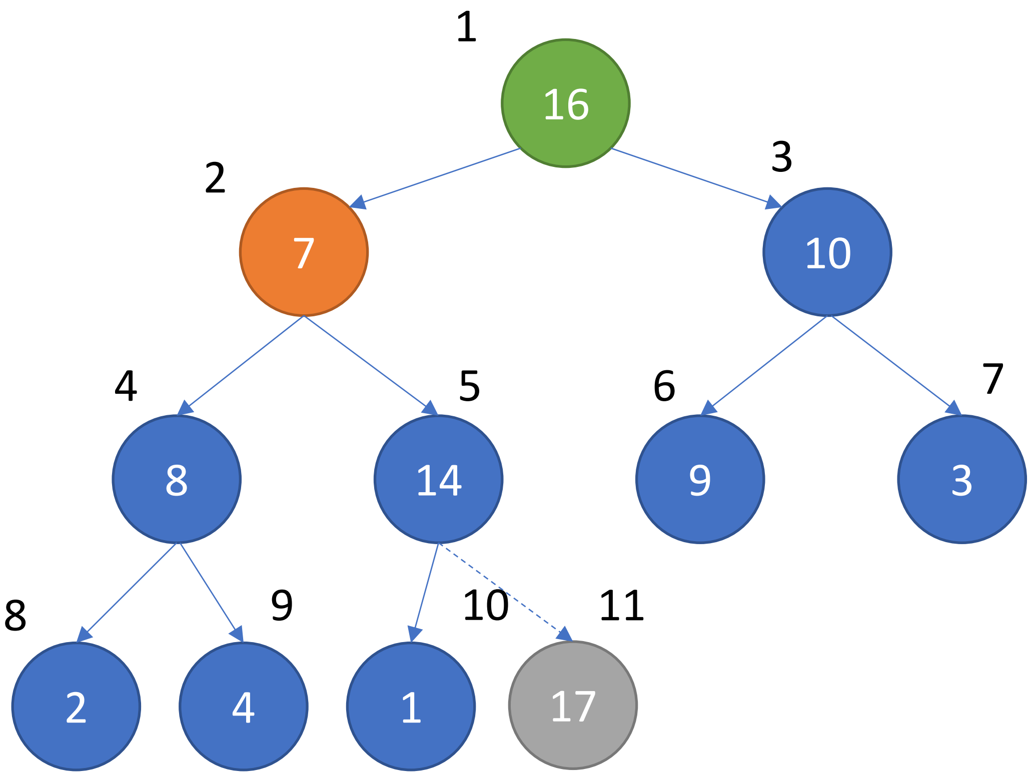
Complete* binary tree

- Let's pop largest (min) element
- Swap last with root
- Push down swapping with largest child

1	2	3	4	5	6	7	8	9	10	11
7	**16**	10	8	14	9	3	2	4	1	17

Heap: pop

- Max-Heap:



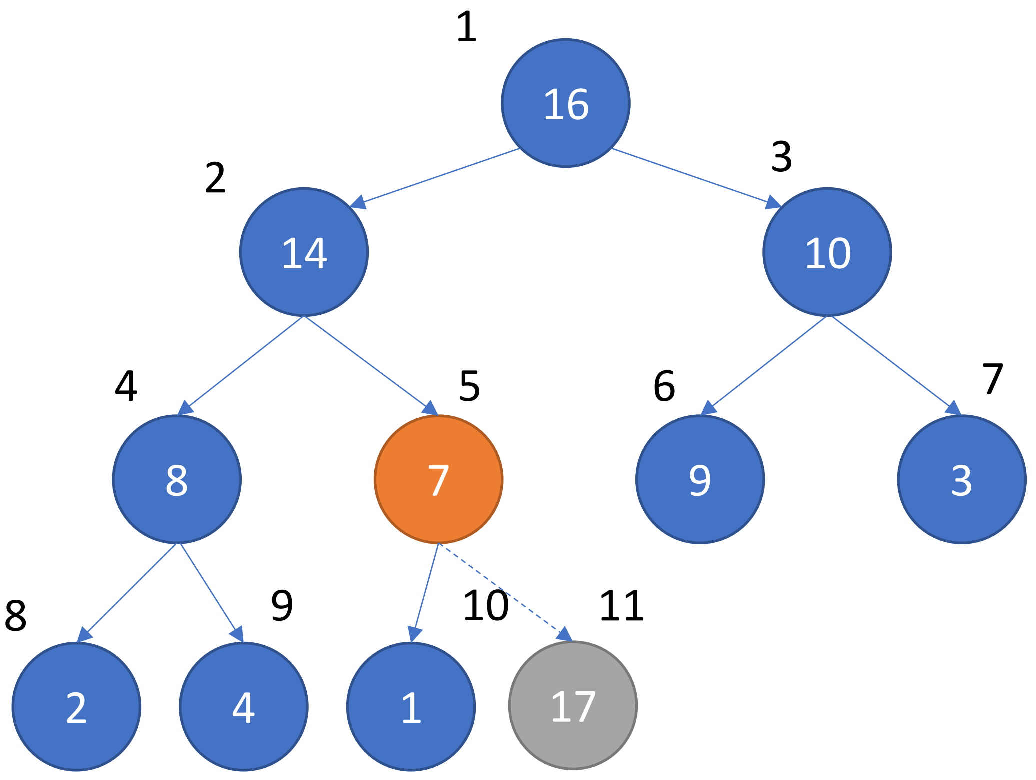
Complete* binary tree

- Let’s pop largest (min) element
- Swap last with root
- Push down swapping with largest child

1	2	3	4	5	6	7	8	9	10	11
16	7	10	8	14	9	3	2	4	1	17

Heap: pop

- Max-Heap:



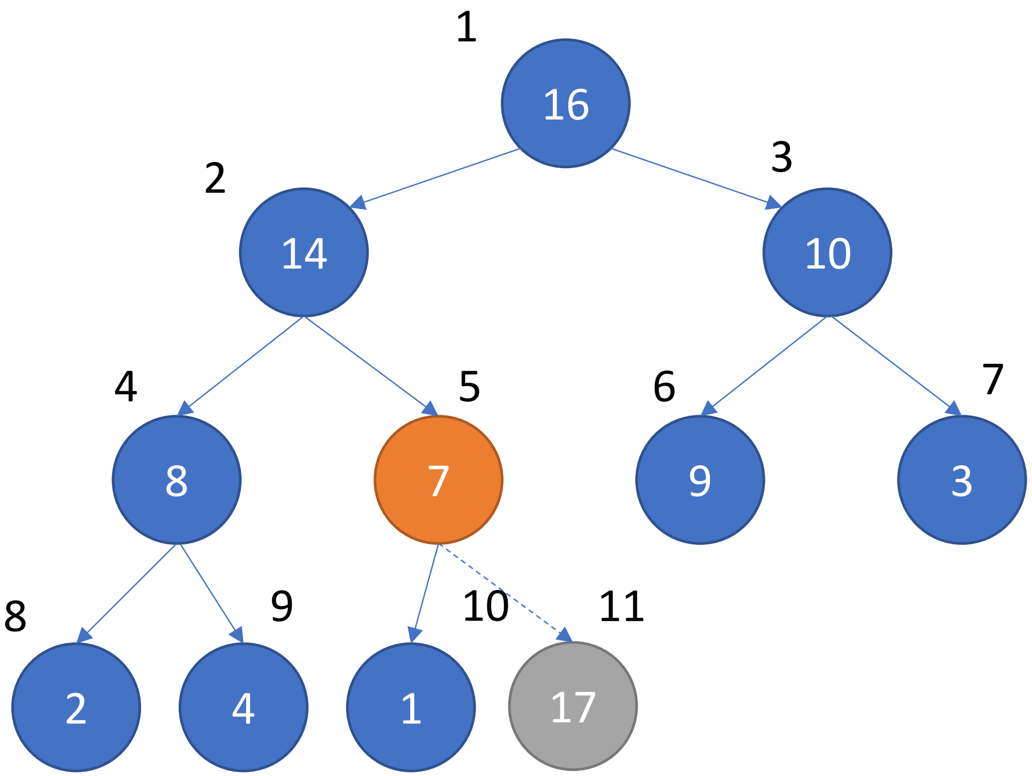
Complete* binary tree

- Let's pop largest (min) element
- Swap last with root
- Push down swapping with largest child
- Complexity?

1	2	3	4	5	6	7	8	9	10	11
16	14	10	8	7	9	3	2	4	1	17

Heap: pop

- Max-Heap:



Complete* binary tree

- Let’s pop largest (min) element
- Swap last with root
- Push down swapping with largest child
- Complexity: $\Theta(\log n)$

1	2	3	4	5	6	7	8	9	10	11
16	14	10	8	7	9	3	2	4	1	17

Heapsort

- Building a heap + popping elements yields sorted sequence
- Complexity?

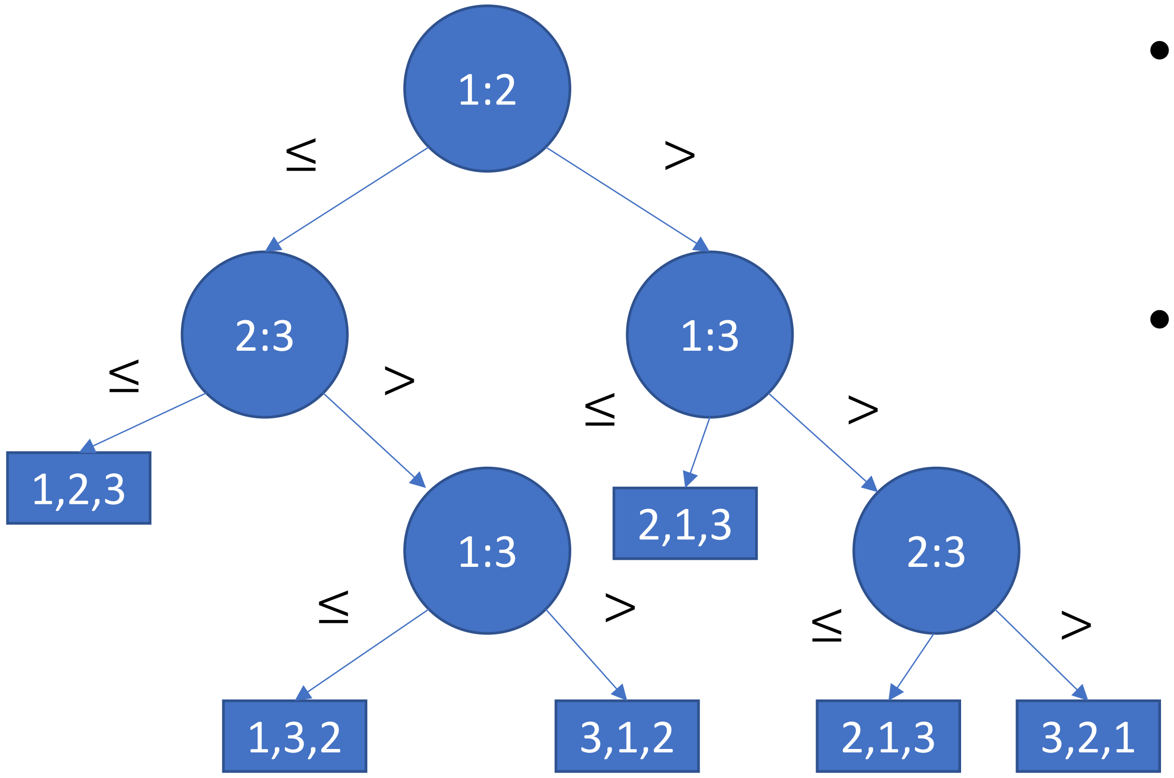
```
def heapsort(A):
    heapsize = len(A)
    buildMaxHeap(A)
    for i in reversed(range(1, len(A))):
        swap(A[0], A[i])
        heapsize = heapsize - 1
        maxHeapify(A, 1)
```

Heapsort

- Building a heap + popping elements yields sorted sequence
- Complexity: $\Theta(n \log n)$

```
def heapsort(A):
    heapsize = len(A)
    buildMaxHeap(A)
    for i in reversed(range(1, len(A))):
        swap(A[0], A[i])
        heapsize = heapsize - 1
        maxHeapify(A, 1)
```

Lower bound for comparison-base sorting



- Decision tree model for comparison-based sorting of 3 elements
- $\Omega(n \log n)$

K-th statistic

- K-th smallest (greatest) element in an unordered collection

Quickselect

- K-th smallest (greatest) element in an unordered collection
- Same idea as in quicksort – divide the collection in two with pivot element

Quickselect

- K-th smallest (greatest) element in an unordered collection
- Same idea as in quicksort – divide the collection in two with pivot element
- Complexity?

```
def qselect(A, l, r, k):
    if l == r:
        return A[l]
    pivot = choosePivot(l, r)
    pivot = partition(A, l, r, pivot)
    if k == pivot:
        return A[k]
    if k < pivot:
        select(A, l, pivot-1, k)
    else:
        select(A, pivot+1, r, k)
```

Quickselect

- K-th smallest (greatest) element in an unordered collection
- Same idea as in quicksort – divide the collection in two with pivot element
- Complexity:
  - Best case: $O(n)$
  - Worst case: $O(n^2)$
  - Average case: ?

```
def qselect(A, l, r, k):
    if l == r:
        return A[l]
    pivot = choosePivot(l, r)
    pivot = partition(A, l, r, pivot)
    if k == pivot:
        return A[k]
    if k < pivot:
        select(A, l, pivot-1, k)
    else:
        select(A, pivot+1, r, k)
```

Quickselect: average case analysis

- Complexity:
  - Best case: $O(n)$
  - Worst case: $O(n^2)$
  - Average case: ?

```
def qselect(A, l, r, k):
    if l == r:
        return A[l]
    pivot = choosePivot(l, r)
    pivot = partition(A, l, r, pivot)
    if k == pivot:
        return A[k]
    if k < pivot:
        select(A, l, pivot-1, k)
    else:
        select(A, pivot+1, r, k)
```

Floyd-Rivest algorithm

- K-th smallest (greatest) element in an unordered collection
- $s(n)$ – sample size
- $g(n)$ – rank gap
- The algorithm picks a random sample S from X and two pivots u, v: $u \leq x_k^* \leq v$ with high prob.
- Partition X into less than u, between u, v and greater than v
- Either detect $x_k^* = v \; or \; u$ or determine a subset of X and do selection recursively

- If $n = 1$ return x_1. Choose sample size s and gap $g > 0$
- Pick random sample $S = \{y_1 \dots y_s\}$
- Pivots: $i_u = \max(\left\lceil \frac{ks}{n} - g \right\rceil, 1)$ & $i_v = \min(\left\lceil \frac{ks}{n} + g \right\rceil, s)$
- Partitions: $L = \{x < u\}, M = \{u < x < v\}, R = \{v < x\}$
- If $k = |L| + 1$ return u, if $k = n - |R|$ return v
- If $k \leq |L|$ set $X = L$ etc.
- Recursion

http://people.csail.mit.edu/rivest/pubs/FR75b.pdf

Floyd-Rivest algorithm

- On average at most $n + \min\{k, n - k\} + o(n)$ comparisons
- Pivot selections takes $c_s + c_{s-i_u}$ steps
- Partition takes at most $2(n - s)$

- If $n = 1$ return x_1. Choose sample size s and gap $g > 0$
- Pick random sample $S = \{y_1 \dots y_s\}$
- Pivots: $i_u = \max(\left\lceil \frac{ks}{n} - g \right\rceil, 1)$ & $i_v = \min(\left\lceil \frac{ks}{n} + g \right\rceil, s)$
- Partitions: $L = \{x < u\}, M = \{u < x < v\}, R = \{v < x\}$
- If $k = |L| + 1$ return u, if $k = n - |R|$ return v
- If $k \leq |L|$ set $X = L$ etc.
- Recursion

On Floyd and Rivest’s SELECT algorithm: https://core.ac.uk/download/pdf/82672439.pdf

Median-of-three killer sequence & Introselect algorithm

- How do you choose a good pivoting element in Quicksort?
- A **good** pivoting element is the one that makes the search set decrease **exponentially**.
- Find an approximate median in linear time → worst-case complexity of Quicksort can be reduced to $\Theta(n \log n)$
- Quick select + median of medians (or heap select – std::nth_element cpp standard*)
- Good average + optimal worst-case performance
- Optimistically start with a quick select and fall back to median of medians if the recursion progress is slow

Introspective sorting and selection algorithms (Musser): https://dl.acm.org/doi/10.5555/261387.261395

Afterword

- Memory usage:
  - In-place, constant space
- Comparison-based vs non-comparison based
  - Lower bound for comparison-based sorting
- Stable vs non-stable

Resources

- Introduction to Algorithms, Thomas H. Cormen, chapters 2, 6.
- https://visualgo.net/en/sorting
- The input/output complexity of sorting and related problems (https://dl.acm.org/doi/10.1145/48529.48535)

Backup