Chapter 2. Sorting & Order Statistics

We discuss:

- Sorting algor. (HeapSort, QuickSort)
- Median & Order Statistics
- A Lower Bound on Sorting.

Selection Sort.

We've discussed Insertion Sort: it's in-place and has time complexity $O(n^2)$.

Idea of Selection Sort:

- Remove largest element and add it to output sequence until input sequence is exhausted.

```
SELECTION ( A[1..n] )
    j = n
    while   j > 1
        k = j ; max = j ; S = A[j]
        while   k > 1
            k = k - 1
```

if $A[k] > S$ then

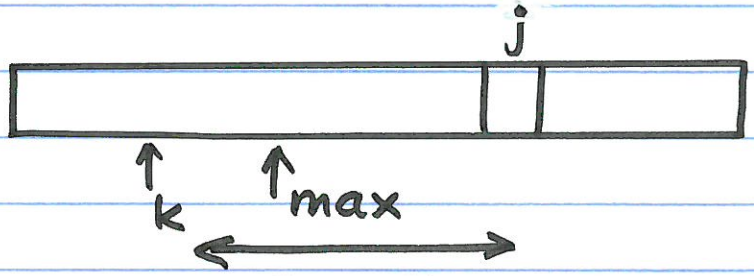
$max = k$

$S = A[k]$

swap $A[max]$ with $A[j]$

$j = j-1$

Illustration



$A[max]$ = max of $A[k+1 .. j]$

Complexity of SELECTION:

- Searching for max in $A[1..j]$ takes $O(j)$ steps
- Outer loop is executed for values $j = n, n-1, \ldots, 2$.

Thus, $T(n) = O\left(\sum_{j=2}^{n} O(j)\right)$

$= O(n^2)$.

HeapSort

Observation: If we can find max faster, we can improve on SELECTION.

$\Rightarrow$ We organize $A[1..n]$ as a **heap**

A **heap** is an array object that can be identified as an almost complete binary tree: $parent(i) = \lfloor i/2 \rfloor$

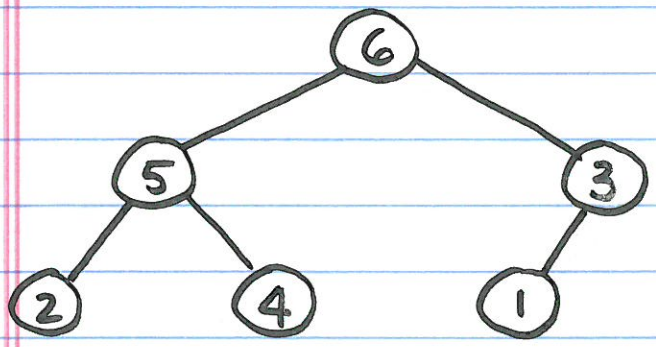
$left(i) = 2i$ and $right(i) = 2i+1$

A heap is a **max heap** if

$$A[parent(i)] \geq A[i]$$

(**min heap** if $A[parent(i)] \leq A[i]$)

Example: (max heap)



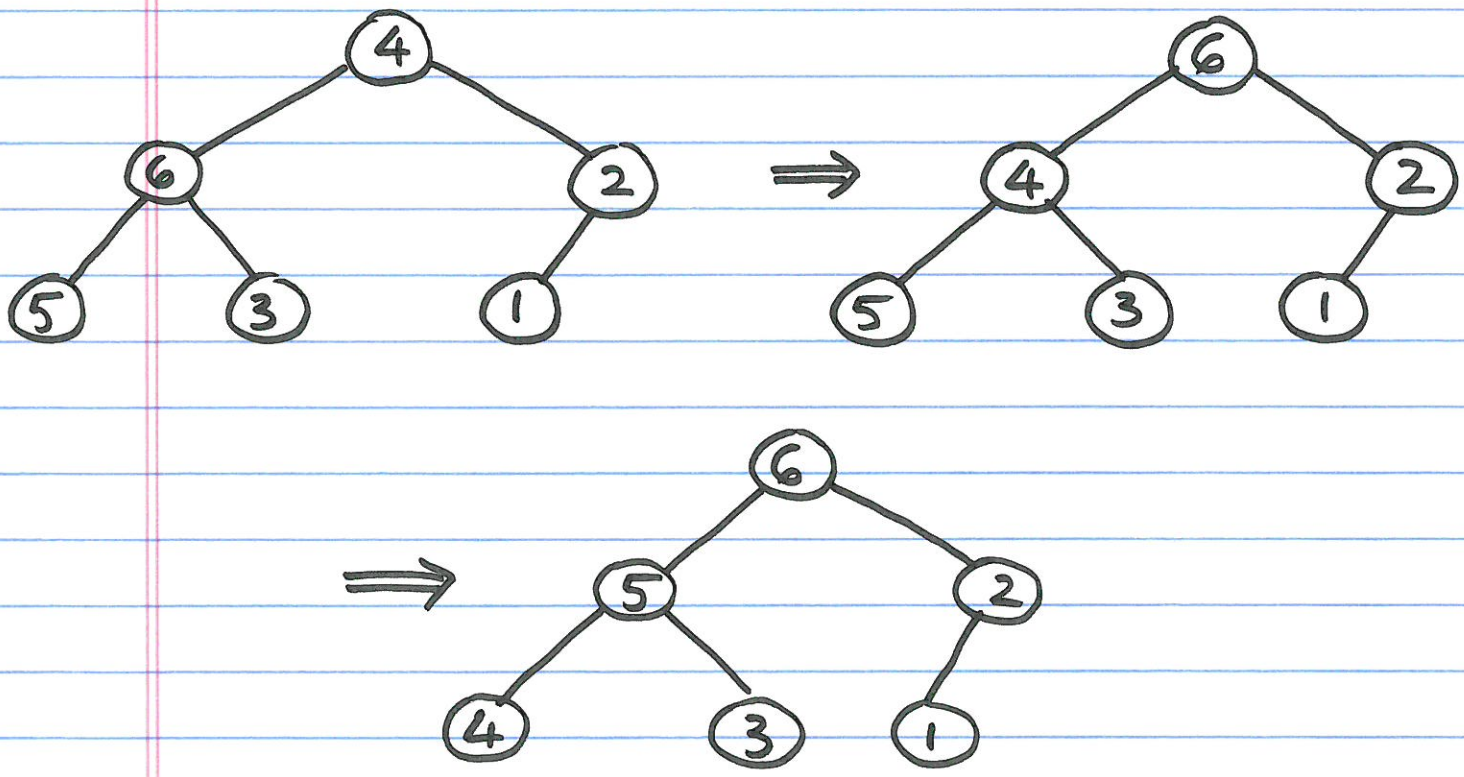
6	5	3	2	4	1

Max-Heapify

Max-Heapify is a procedure on (A, i):

- Assumes subtrees rooted at left(i), right(i) are max-heaps but A[i] may not sat. max-heap property
- Transform subtree rooted at A[i] into a max-heap by "floating A[i] down".

Example.



Max_Heapify (A, i)

```
l = left(i) ; r = right(i)
if l ≤ heap_size[A] and A[l] > A[i]
    then largest = l
    else largest = i ;
if r ≤ heap_size[A] and
        A[r] > A[largest]
    then largest = r ;
if largest ≠ i
    then
        exchange A[i] ↔ A[largest]
        Max. Heapify (A, largest)
```

Complexity of Max_Heapify

- Subtree rooted at A[largest] for recursive call is of size at most $2n/3$, where n = size of subtree rooted at A[i].

(Note left subtree is larger when bottom level is half-full.)

Thus, $T(n) \leq T(\lfloor 2n/3 \rfloor) + \Theta(1)$

$= O(\lg n)$

Remark. If h = height of subtree rooted at $A[i]$, then $T(n) = O(h) = O(\lg n)$.

Building a Heap

Build_Max_Heap (A)

heap_size (A) = length (A)

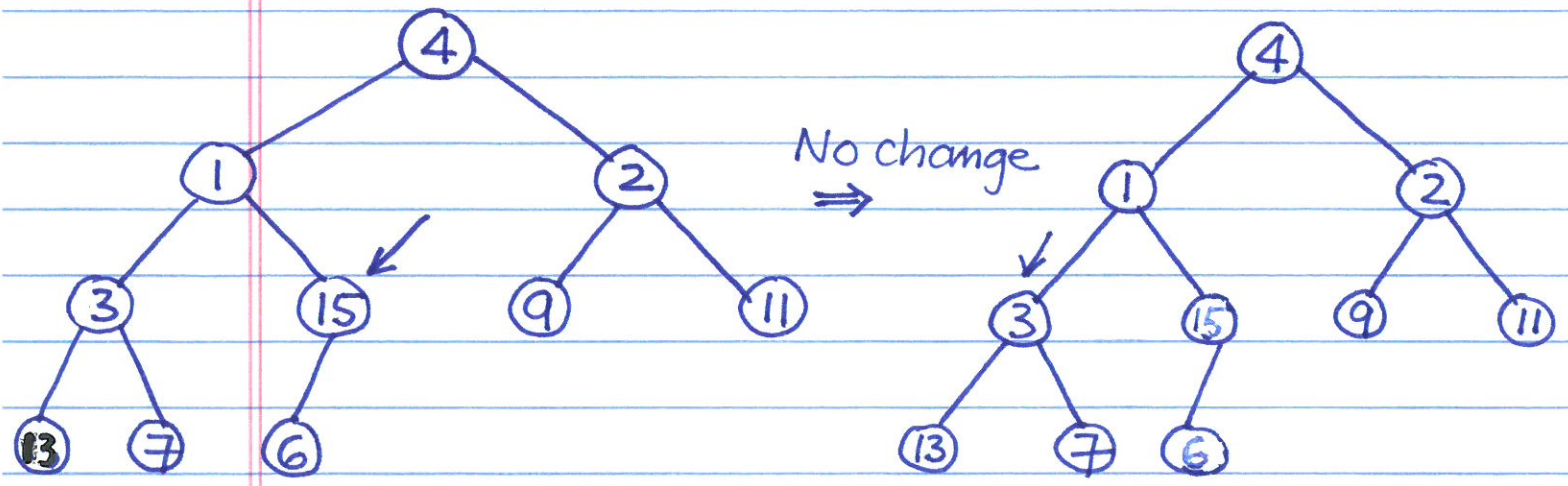
for $i = \lfloor$ length (A)/2 $\rfloor$ downto 1

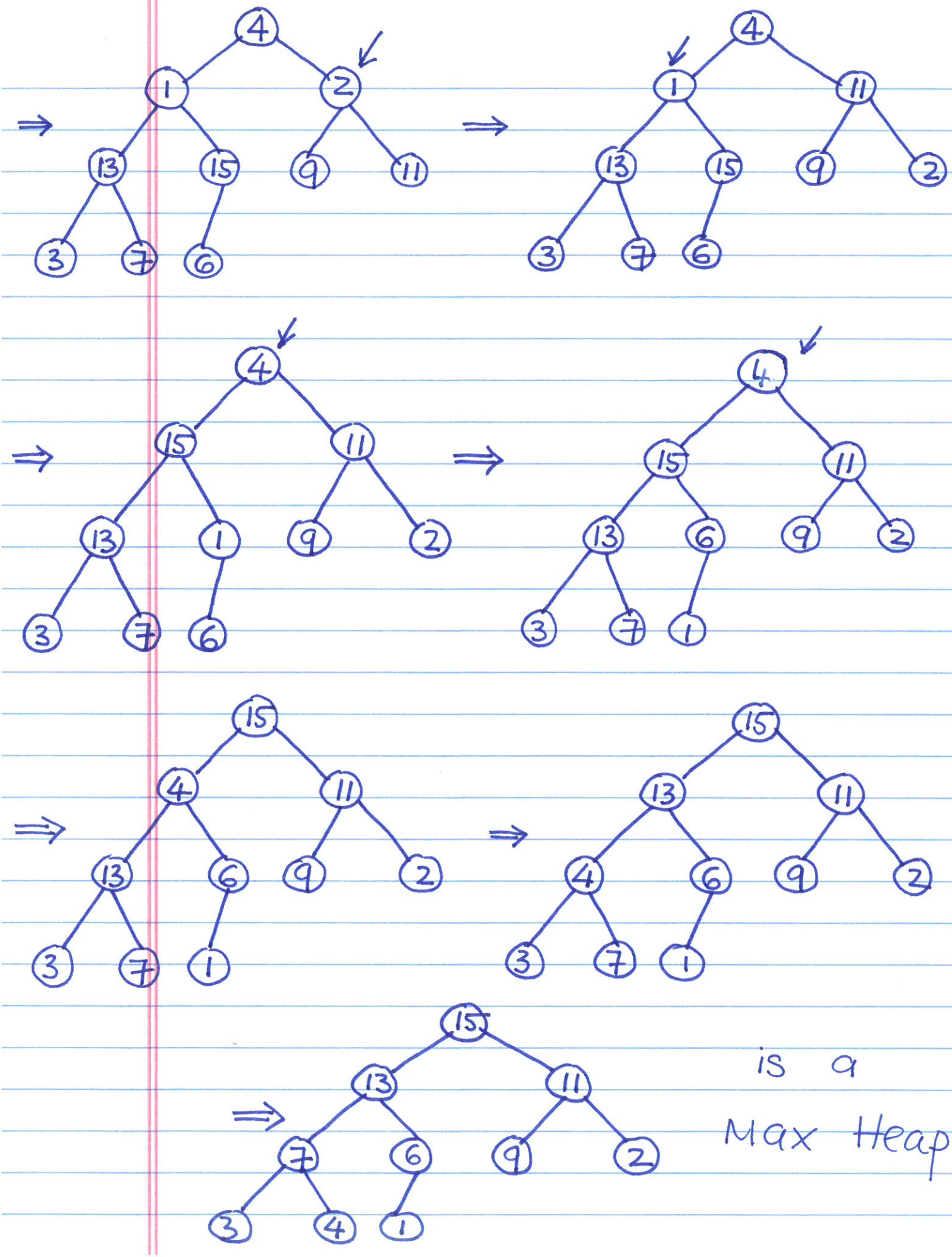
Max_Heapify (A, i)

Note: Building a heap bottom-up

Example:

4	1	2	3	15	9	11	13	7	6




4
1
2
13
15
9
11
3
7
6
4
1
11
13
15
9
2
3
7
6
4
15
11
13
1
9
2
3
7
6
4
15
11
13
6
9
2
3
7
1
15
4
11
13
6
9
2
3
7
1
15
13
11
4
6
9
2
3
7
1
15
13
11
7
6
9
2
3
4
1

is a Max Heap!

Analysis of Build.Max.Heap

Fact. An n-element heap has:

(1) height $= \lfloor \lg n \rfloor$

(2) $\leq \lceil n/2^{h+1} \rceil$ nodes at height h.

Pf (1) ✓

(2) Ind. on h.

Observe:

internal nodes $\leq$ # leaves

$\leq$ # internal nodes + 1

Thus,

nodes at height 0 $\leq \lceil n/2 \rceil$

Now, assume # nodes at height h is $\leq \lceil n/2^{h+1} \rceil$. Then

nodes at height $h+1$ is

$$\leq \left\lceil \lceil n/2^{h+1} \rceil / 2 \right\rceil = \left\lceil \frac{n}{2^{h+2}} \right\rceil$$

□

Thus, running time of Build-Max-Heap:

$$\leq \sum_{h=0}^{\lfloor \lg n \rfloor} \lceil \frac{n}{2^{h+1}} \rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right)$$

$$= O(n)$$

since $\sum_{h=0}^{\infty} \frac{h}{2^h} = 2$

Note: $\sum_{i=0}^{\infty} x^h = \frac{1}{1-x}$

diff $\Rightarrow \sum h x^{h-1} = \frac{1}{(1-x)^2}$

mult. by $x \Rightarrow$

The Heapsort algor.

Heapsort (A)

$O(n)$ Build-Max-Heap (A);

$O(n)$ for i = A.length downto 2

exchange A[1] with A[i]

heap-size (A) = heap-size (A) - 1

$O(\lg n)$ Max-Heapify (A, 1)

Heapsort's running time is:

$$O(n \lg n)$$

Quicksort.

Quicksort:

- has $\Theta(n^2)$ worst-case running time
- has $O(n \lg n)$ expected running time
- is in place
- is method of choice

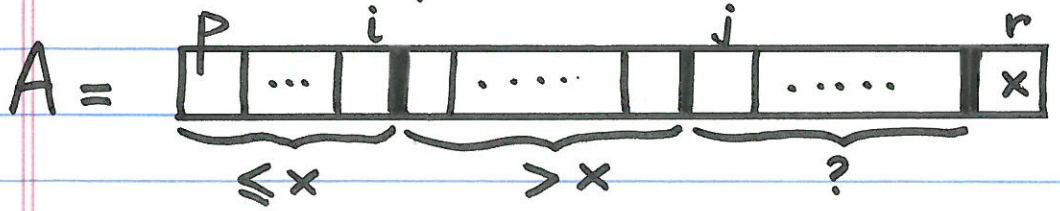
Quicksort is based on divide-and-conquer technique:

- **Divide**: Partition $A[p..r]$ into subarrays $A[p..q-1], A[q..r]$ s.t.
  $A[i] \leq A[q] \quad \forall p \leq i \leq q-1$
  $A[q] \leq A[j] \quad \forall q+1 \leq j \leq r$
- **Conquer**: Recursively sort $A[p..q-1]$, $A[q+1, r]$

(Now A is sorted)

Question: How to do partition?

For $A[p..r]$ let $x = A[r]$ be pivot

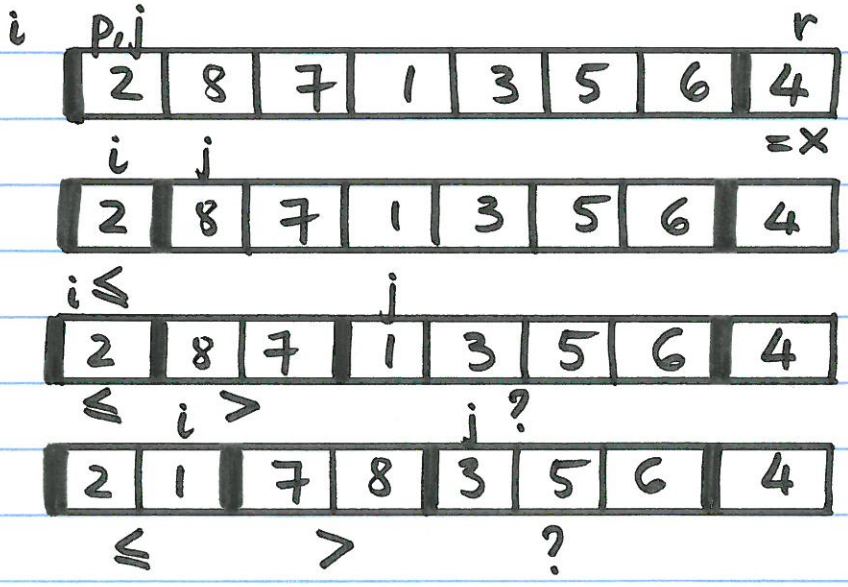


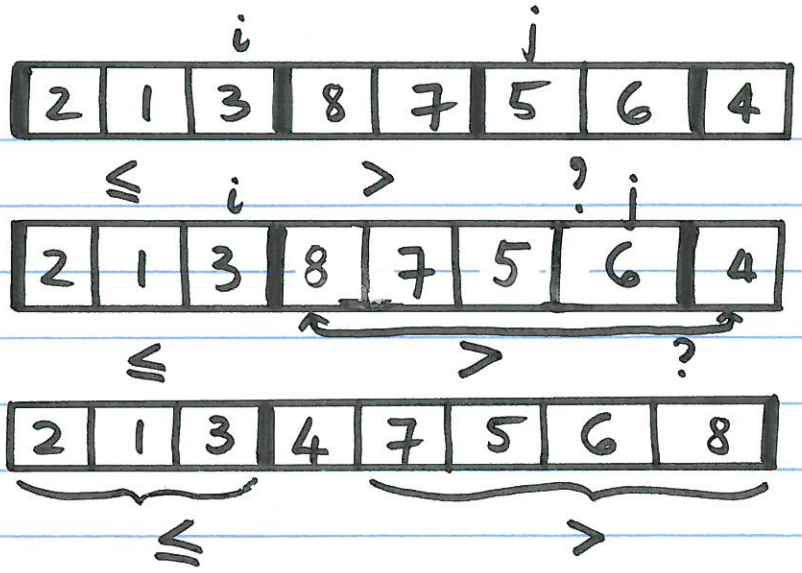
- Search for j with $A[j] \leq x$
- Exchange $A[i+1]$ with $A[j]$

(this increases first region $\leq x$ by one element)

```
Partition(A, p, r)
    x = A[r]
    i = p-1
    for j = p to r-1
        if A[j] ≤ x
            i = i+1
            exchange A[i] with A[j]
    exchange A[i+1] with A[r]
    return i+1
```

Ex.





Complexity of Partition : $\Theta(n)$

Quicksort algorithm:

```
Quicksort (A, p, r)
if p<r
    q = PARTITION (A, p, r)
    QUICKSORT (A, p, q-1)
    QUICKSORT (A, q+1, r)
```

Worst_case complexity of Quicksort

$$T(n) \leq \max_{0 \leq q \leq n-1} \left(T(q) + T(n-q-1)\right) + \Theta(n)$$

Guess: $T(n) \leq cn^2$ for some c.

We have:

$$T(n) \leq \max_{0 \leq q \leq n-1} \left(cq^2 + c(n-q-1)^2\right) + \Theta(n)$$

$$= c \cdot \max_{0 \le q \le n-1} (q^2 + (n-q-1)^2) + \Theta(n)$$

which achieves max. when $q = 0, n-1$.

Thus,

$$\begin{aligned} T(n) &\le c \cdot (n-1)^2 + \Theta(n) \\ &= c \cdot (n^2 - 2n + 1) + \Theta(n) \\ &= c \cdot n^2 - c(2n-1) + \Theta(n) \\ &\le cn^2 \end{aligned}$$

for sufficiently large c. □

On sorted arrays Quicksort requires $\Omega(n^2)$ steps. Thus,

$$T(n) = \Theta(n^2) \qquad \square\square$$

Expected complexity of Quicksort

When partitioning is unbalanced Quicksort has worse performance.

Observation: Expected partition is balanced!

Assume:

- Array elements are distinct
- All permutations are equally likely.

$\Pr(\text{last element has rank } i) = \frac{1}{n}$

$\Rightarrow$ Expected size of subarray is

$\frac{1}{n}(0+1+\cdots+n-1) = \frac{n-1}{2}$ ✓

Guess: $E[T(n)] \leq c \cdot n \cdot \lg n$

We have:

$$E[T(n)] \leq \frac{1}{n}\left(E[T(0)] + E[T(n-1)] + c_1 n\right)$$
$$+ \frac{1}{n}\left(E[T(1)] + E[T(n-2)] + c_1 n\right)$$
$$\vdots$$
$$+ \frac{1}{n}\left(E[T(n-1)] + E[T(0)] + c_1 n\right)$$
$$\leq c_1 n + \frac{2c}{n}\sum_{i=1}^{n-1} i \lg i$$

Fact. $\sum_{i=1}^{n-1} i \lg i \leq \frac{1}{2}n^2 \lg n - \frac{1}{8}n^2$

Thus,

$$E[T(n)] \leq \frac{2c}{n}\left(\frac{1}{2}n^2 \lg n - \frac{1}{8}n^2\right) + c_1 n$$
$$\leq cn\lg n - \frac{c}{4}n + c_1 n$$
$$\leq cn\lg n$$

for sufficiently large c. ⊠

Proof of Fact.

$$\sum_{i=1}^{n-1} i \lg i = \sum_{i=1}^{\lceil n/2 \rceil - 1} \underbrace{i \lg i}_{\leq \lg n - 1} + \sum_{i=\lceil n/2 \rceil}^{n-1} \underbrace{i \lg i}_{\leq \lg n}$$

$$\leq (\lg n - 1) \sum_{i=1}^{\lceil n/2 \rceil - 1} i + \lg n \sum_{i=\lceil n/2 \rceil}^{n-1} i$$

$$= \lg n \sum_{i=1}^{n-1} i - \sum_{i=1}^{\lceil n/2 \rceil - 1} i$$

$$\leq \frac{1}{2} n(n-1) \lg n - \frac{1}{2} \left(\frac{n}{2} - 1 \right) \frac{n}{2}$$

$$\leq \frac{1}{2} n^2 \lg n - \frac{1}{8} n^2$$

for $n \geq 2$. □

A Lower Bound for Sorting

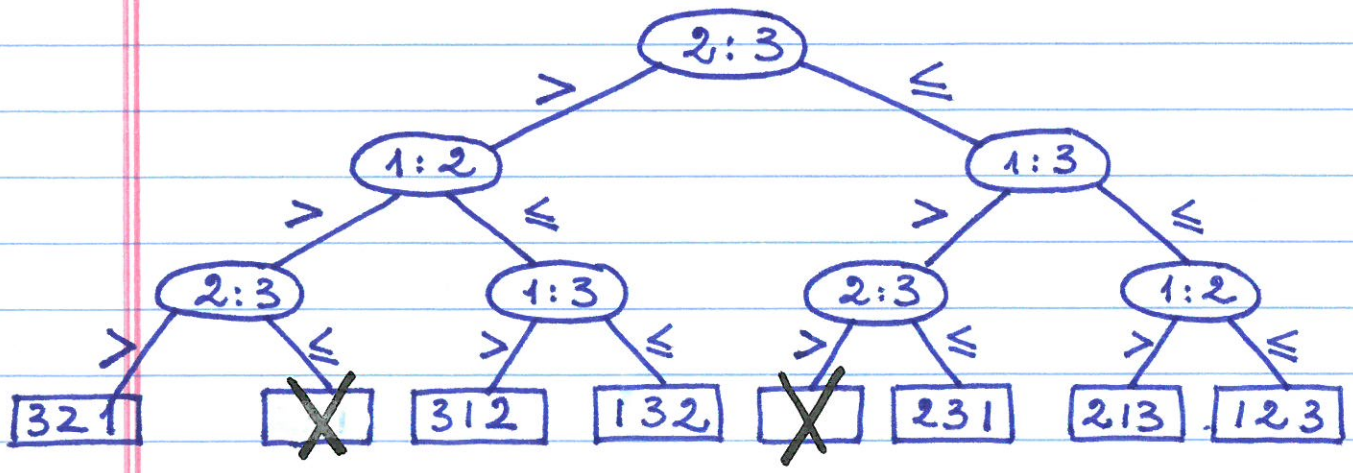
Observ. Sorting algor. are based on comparisons in general!

Question. Are there comparison-based sorting algor. with time complexity $o(n \cdot \log n)$?

Consider for example sorting on $A[1..3]$ by maximum selection!

– Suppose that we first compare $A[2]$ with $A[3]$: (2:3)

if $A[2] > A[3]$ then (1:2)
else (1:3)



This is an example of a decision tree

A **decision tree** is a bin. tree whose nodes have labels of form $i:j$.
The 2 outgoing edges are labeled $>, \leq$.

Observ. On input arrays $A[1..n]$ comparison-based sorting algor. produce decision trees in a natural way.

The leaves correspond to permutations on $A[1..n]$

Def. A decision tree T solves the sorting problem of size n if there is a labeling of the leaves of T by permutations on $\{1,2,..,n\}$ s.t:

For every $A[1..n]$ if a leaf labeled π is reached, then

$$A[\pi(1)] \leq A[\pi(2)] \leq \dots \leq A[\pi(n)]$$

For sorting problem of size n def.

$S(n)$ and $A(n)$ by:

Let T be a decision tree, π a permut.
Let ℓ_π^T = length of path from root to leaf labeled π in T.

$$S(n) := \min_T \underbrace{\underbrace{\max_\pi \ell_\pi^T}_{\text{depth of } T}}_{\text{depth of decision tree of smallest depth}}$$

($S(n) \hat{=}$ # of comparisons required by best comparison-based sort. alg.)

$$A(n) := \min_T \left(\frac{1}{n!} \sum_\pi \ell_\pi^T \right)$$

($A(n) \hat{=}$ depth of decision tree with minimum average depth)

Goal: To obtain lower bounds for $S(n)$ and $A(n)$!

Claim: $S(n) = \Theta(n \lg n)$

Pf. Since there are $n!$ permutations of n elements, any decision tree for sorting n elements has $\geq n!$ leaves.

Thus, $2^{S(n)} \geq n!$

$\Rightarrow$ $S(n) \geq \lceil \log n! \rceil$

Stirling's approximation for $n!$ gives:

$$\log_2 n! = \left(n + \frac{1}{2}\right) \log_2 n - \frac{n}{\ln 2} + O(1)$$

Hence, $S(n) = \Omega(n \lg n)$

On the other hand, known sorting algorithms imply the existence of decision trees of depth $O(n \lg n)$ $\square$

Next: A lower bound for $A(n)$.

Question: What is average depth $D(k)$ of a binary tree with k leaves?

Claim: $D(k) \geq \log_2 k \quad \forall\, k \geq 2$

Pf. By ind. on k.

Basis. $k = 2$ ✓

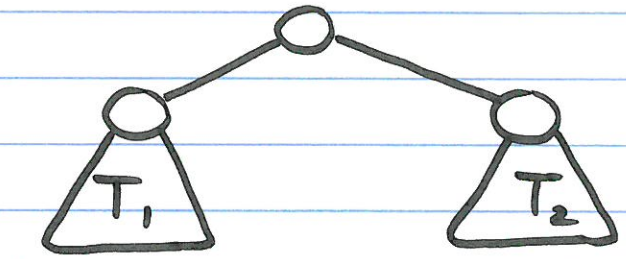
Ind. Step. Suppose Claim holds true for for binary trees with $\leq k-1$ leaves.

We show Claim for k.

Let T be a bin. tree with k leaves.

Without loss of generality (w.l.o.g.)

T is of form:



$$k_1 + k_2 = k$$

Average depth of T =

$$1 + \frac{k_1}{k_1+k_2}(\text{aver. depth of } T_1) + \frac{k_2}{k_1+k_2}(\text{av. depth of } T_2)$$

$$\geq \frac{k_1}{k}\log_2 k_1 + \frac{k_2}{k}\log_2 k_2 + 1 \quad \text{(by I.H.)}$$

$$= \frac{1}{k}\left(k_1 \log_2 k_1 + k_2 \log_2 k_2 + k_1 + k_2\right)$$

$$= \frac{1}{k}\underbrace{\left(k_1 \log_2 2k_1 + k_2 \log_2 2k_2\right)}_{f(k_1, k_2)}$$

$f(k_1, k_2)$ achieves a min. when $k_1 = k_2 = k/2$. Thus,

Average depth of $T \geq \log_2 k$ □

Theorem. Every comparison-based sorting algor. has worst-case and average-case complexities $\Omega(n \lg n)$ □

Remark This lower bound doesn't hold for general models of computation such as RAMs or Turing machines.

Question. Is there some other type of sorting that can be solved faster?

Of course such algorithms cannot be based on comparisons!

Counting Sort.

Array to be sorted: $A[1..n]$

$A[1], \ldots, A[n] \in \{0, 1, \ldots, k\}$, $k = O(n)$

counting_sort (A, n, B, k) $(B[1..n] = \text{output})$

- Let $C[0..k]$ be new array
- **for** $i = 0$ **to** k
  $C[i] = 0$
- **for** $j = 1$ **to** n
  $C[A[j]] = C[A[j]] + 1$

/* $C[i] = \#\{A[j] = i\}$ */

- **for** $i = 1$ **to** k
  $C[i] = C[i] + C[i-1]$

/* $C[i] = \#\{A[j] \leq i\}$ */

/* Now place $A[j]$ into its correct position in B */

- **for** $j = n$ **downto** 1
  $B[C[A[j]]] = A[j]$
  $C[A[j]] = C[A[j]] - 1$

	1	2	3	4	5	6	7	8
A =	2	5	3	0	2	3	0	3

$k = 5$

C →

0	1	2	3	4	5
2	0	2	3	0	1

C →

0	1	2	3	4	5
2	2	4	7	7	8

Where should $A[8]$ be placed in B?

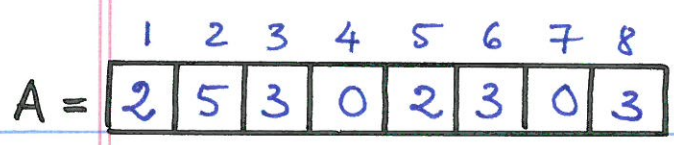
B =

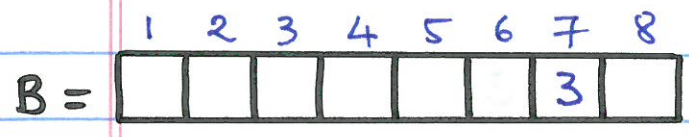
							3	

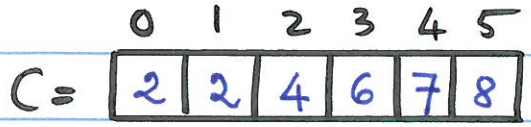
$B[7] = 3$

C →

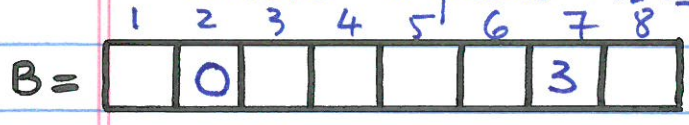
2	2	4	6	7	8

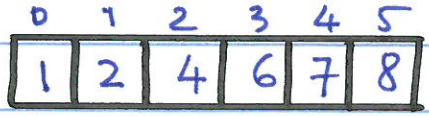




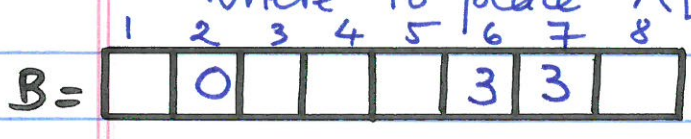


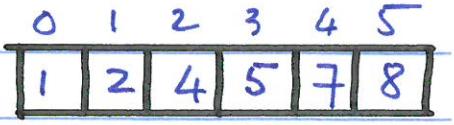
Where to place A[7]? (A[7]=0)



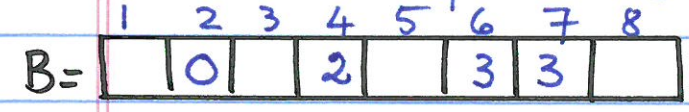


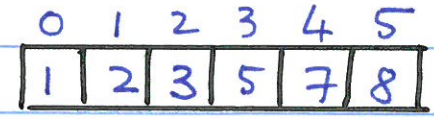
Where to place A[6] = 3?



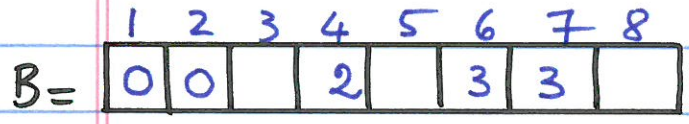


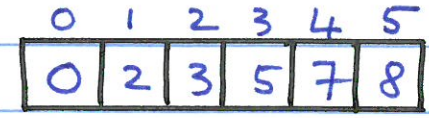
Next we place A[5] = 2 into B





We now place A[4] = 0 into B





Counting Sort

```
for i=0 to k do C[i]=0                          O(k)
for j = 1 to n do C[A[j]]= C[A[j]]+1            O(n)
for i = 1 to k do C[i] = C[i]+C[i-1]            O(k)
for j=n downto 1 do                             O(n)
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] -1
```

Total = $O(n+k)$

Remark: not in-place, stable.

Radix_Sort.

Elements of input array A[1..n] are d-digit numbers

```
Radix_Sort(A, d)
    for i = 1 to d do
        use a stable sort to sort A on digit i
```

Example.

329		436		329		329
457		457		436		436
657	⇒	657	⇒	839	⇒	457
839		329		457		657
436		839		657		839

Proposition. If each digit can take on up to k values, Radix_Sort correctly sorts A[1..n] in time $\Theta(d(n+k))$ if the stable sort used takes $\Theta(n+k)$ time. □

Selection Problem

Input. An array $A[1..n]$ of distinct elements and an integer i, $1 \le i \le n$.

Output. The i-th smallest element of A.

Randomized Selection

```
Randomized_Select (A, p, r, i)
    if p = r then return A[p]
    q = Random_Partition (A, p, r)
    k = q - p + 1 ; if i = k then return A[q];
    if i < k then return Random_Select (A, p, q-1, i)
        else return Random_Select (A, q+1, r, i-k)
```

Claim. Expected running time of Randomized_Select is $O(n)$.

Pf. Let $T(n)$ denote running time of Randomized_Select. We show:

$$E[T(n)] \le c \cdot n \text{ for some } c.$$

Assume by induction that

$$E[T(k)] \le c \cdot k \text{ for } k < n.$$

We have:

$$E[T(n)] \leq \frac{1}{n}\left(\sum_{k=1}^{n-1} E[T(\max(k, n-k))]\right) + c_1 n \quad \text{(Partition)}$$

$$\leq \frac{2}{n}\sum_{k=\lfloor n/2 \rfloor}^{n-1} E[T(k)] + c_1 n$$

$$\leq \frac{2}{n}\sum_{k=\lfloor n/2 \rfloor}^{n-1} c \cdot k + c_1 n \quad \text{(by I.H.)}$$

$$= \frac{2c}{n}\left(\sum_{k=1}^{n-1} k - \sum_{k=1}^{\lfloor n/2 \rfloor - 1} k\right) + c_1 n$$

$$\leq \frac{2c}{n}\left(\frac{(n-1)n}{2} - \frac{(n/2-2)(n/2-1)}{2}\right) + c_1 n$$

$$= \frac{2c}{n}\left(\frac{n^2 - n - n^2/4 + 3n/2 - 2}{2}\right) + c_1 n$$

$$= c\left(\frac{3n}{4} + \frac{1}{2} - \frac{2}{n}\right) + c_1 n$$

$$\leq \frac{3cn}{4} + \frac{c}{2} + c_1 n$$

$$= cn - \underbrace{\left(\frac{cn}{4} - \frac{c}{2} - c_1 n\right)}_{\geq 0 \text{ for suitable } c}$$

$$\leq cn. \qquad \square$$

Linear Time Selection (Worst-Case)

Observation: Worst-case running time of Select is good if partition is "balanced."

$\Rightarrow$ Choose pivot element for partition close to median, and find it fast!

Select: $A[1..n]$, $1 \le i \le n$.

(1) Divide $A[1..n]$ into $\lceil n/5 \rceil$ groups of 5 elements each. (Last group may have < 5 elements)

(2) Find median of each group

(3) Recursively use Select ~~the~~ compute the median x of the $\lceil n/5 \rceil$ medians found in (2)

(4) Using x as pivot, partition A
Let k = size of left subarray

(5) Recursively use Select to find i-th smallest element
- in left subarray if $i < k$
- in right subarray if $i > k$

(if $i = k$ then return x)

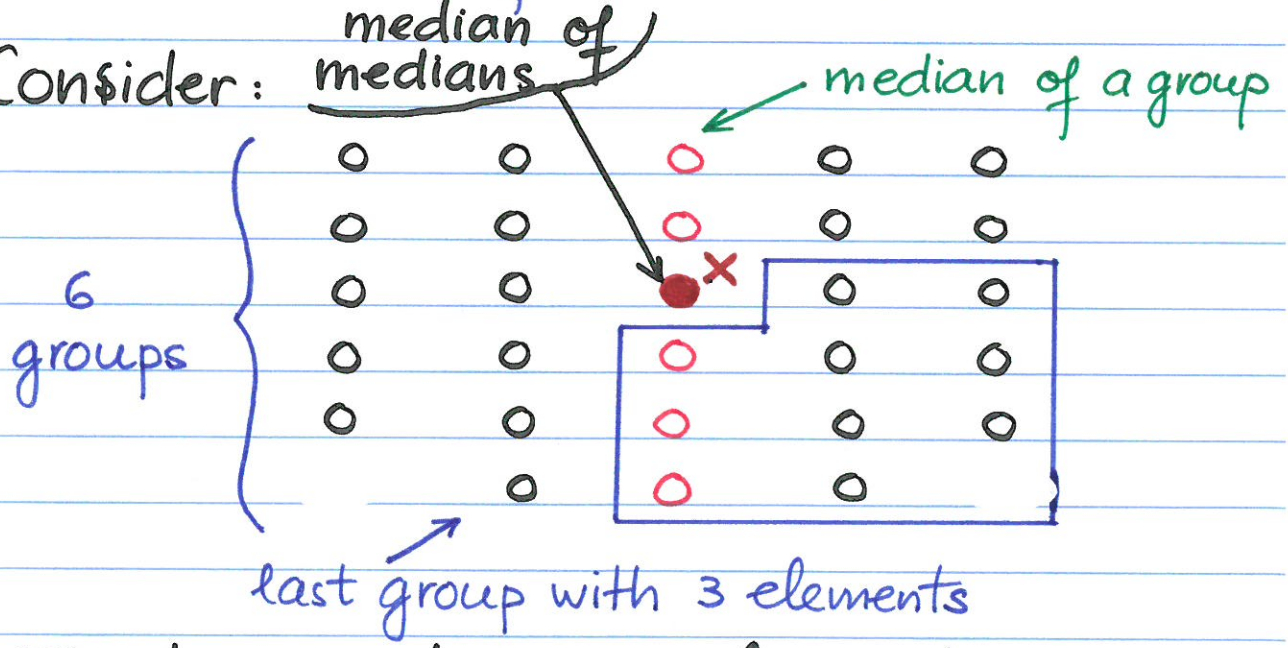
(Remark. If there are 2 possible medians, take the smaller one)

Observation. Recursive call in Step (3) applies to array of size $\lceil n/5 \rceil$.

Question. What is size of subarrays in recursive calls in Step (5)?

What is # of elements $> x$?

Consider: median of medians



The box containing elements $> x$ must have at least

$$3\left(\underbrace{\left\lceil \frac{1}{2}\left\lceil \frac{n}{5} \right\rceil \right\rceil}_{\text{Half of groups}} - 2\right) \geq \frac{3n}{10} - 6$$

-2: -1 group containing x
$-$ last group with < 5 elements.

Similary, the # of elements $< x$ is also $\geq \frac{3n}{10} - 6$.

Thus, subarrays in rec. call in Step(5) must have size $\leq \frac{7n}{10} + 6$.

The recurrence for Select is:

$$T(n) \leq \begin{cases} O(1) & \text{if } n < 140 \\ T(\lceil \frac{n}{5} \rceil) + T(\frac{7n}{10} + 6) + O(n) & \text{otherwise} \end{cases}$$

Claim. $T(n) \leq cn$ for some c.

Pf.
$$\begin{aligned} T(n) &\leq c\lceil \tfrac{n}{5} \rceil + c(\tfrac{7n}{10} + 6) + c_1 n \\ &\leq c\tfrac{n}{5} + c + \tfrac{7cn}{10} + 6c + c_1 n \\ &= \tfrac{9cn}{10} + 7c + c_1 n \\ &= cn + (-\tfrac{cn}{10} + 7c + c_1 n) \\ &\leq cn \end{aligned}$$

if $-\frac{cn}{10} + 7c + c_1 n \leq 0$

or $c \geq 10 c_1 \frac{n}{n-70}$ when $n > 70$.

Now if $n \geq 140 \Rightarrow \frac{n}{n-70} \leq 2$.

Thus, if $c \geq 20 c_1$, then

$$c \geq 10 \cdot c_1 \cdot 2 \geq 10 c_1 \frac{n}{n-70}$$ □