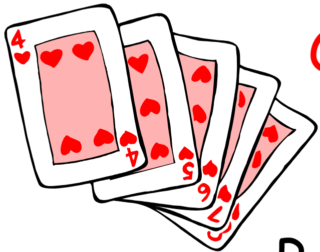


Outcome (2): Sorting algorithms & heap data structure

Sorting (by comparisons)

Problem

Given a sequence of n numbers, output a permutation (reordering) of this input sequence such that the numbers are in increasing (or decreasing) order.

e.g. Input sequence: <5, 34, 7, 86, 3, 16>
Output: 3, 5, 7, 16, 34, 86

How many sorting algorithms do you know?

- Bubble sort
- Selection sort
- Insertion sort
- Merge sort
- Quick sort
- Heap sort
- Counting sort
- Radix sort
- Bucket sort

Note: In our discussion, we assume that we want to rearrange the numbers in increasing order

Bubble sort

Idea: At each pass, scan the numbers from left to right, swap two adjacent numbers if the left is larger than the right.

Q: After the first pass, what can be guaranteed? After the ith pass, what can be guaranteed?

Q: How many passes are needed in the worst case?

Example

56	7	45	34	23	
7	56	45	34	23	
7	45	56	34	23	1^{st} pass
7	45	34	56	23	
7	45	34	23	$\boxed{56}$	
				sorted	
7	34	23	45	56	after 2^{nd} pass
7	23	34	45	56	after 3^{rd} pass
7	23	34	45	56	after 4^{th} pass

56 seems to be "bubbled" up the list

Algorithm

```
/* assuming numbers are store in the array A[1..n] */

BubbleSort(A) {
    for (i = 1 to n-1) {              // n-1 passes
        for (j = 1 to n-i) {          // bubble up the next largest number
            if (A[j] > A[j+1])
                swap(A[j], A[j+1])         // temp = A[j]
        }                                  // A[j] = A[j+1]
    }                                      // A[j+1] = temp
}
```

Overall Time complexity? $O(n^2)$ (for best, average, worst case)

How many swaps the bubble sort will perform in the
(a) worst case; (b) best case?

Insertion sort

Idea: Assume that the input numbers are stored in A[1..n]. At the ith pass, we try to "insert" A[i] into A[1]...A[i-1] which have been considered in previous passes and are in sorted order.

Example

56	7	45	34	23	1st pass
56	7	45	34	23	
56	7	45	34	23	2nd pass
7	56	45	34	23	
7	56	45	34	23	3rd pass
7	45	56	34	23	

in sorted order

7	45	56	34	23	4th pass
7	45	34	56	23	
7	34	45	56	23	
7	45	56	34	23	5th pass
.....					
7	23	34	45	56	

The first pass does not do anything to the numbers, so we can skip it and use n-1 passes

```
Algorithm
InsertionSort(A) {
    for (i = 2 to n) {        // n-1 passes
        j = i-1               // Insert A[i] into A[1], A[2],..., A[i-1]
        while ((j ≥ 1) and (A[j] > A[j+1])) {
            swap(A[j], A[j+1])
            j--;
        }
    }
}
```

Overall Time complexity?

Worst: $O(n^2)$; Best: $O(n)$

Have you noticed that:
Both sorting algorithms only swap adjacent elements.

So, can we do better by swapping elements which are further away?

Try to swap elements which are further away
- Shell sort
- Selection sort

Motivation: Consider the following example.

10, 4, 6, 8, 1 ← 7 inversions

One swap (between 1 and 10) can sort the whole list!

1, 4, 6, 8, 10 ← 0 inversion

<u>Definition</u>: A pair of numbers a_i, a_j in the sequence is called an inversion if $i < j$ and $a_i > a_j$.

0 inversion => sorted

Selection sort

Idea: At the i^{th} pass, look for the i^{th} smallest number and swap it with A[i].

Example

56	7	45	34	23	
7	56	45	34	23	1^{st} pass
7	56	45	34	23	2^{nd} pass
7	23	45	34	56	
7	23	34	45	56	After 3^{rd} pass
7	23	34	45	56	After 4^{th} pass

in final position

Right before the i^{th} pass, A[1]..A[i-1] are in final positions, so look for the smallest number in A[i]..A[n], then swap it with A[i].

Algorithm

```
SelectionSort(A) {
    for (i = 1 to n-1) {        // n-1 passes
        min = i                 // locate the next smallest number
        for (j = i+1 to n) {
            if A[j] < A[min]
                min = j
        }
        swap(A[i], A[min]);  // put the next smallest number in the final pos
    }
}
```

Overall Time complexity?

$O(n^2)$ (for worst and best case)

Only (n – 1) swaps are needed, but still takes $O(n^2)$ time in the worst case.

Merge sort

Idea: If you are given two sorted lists, do you know how to combine ("merge") them into one sorted list?

Example:

14, 18, 40, 57, 60

11, 13, 38, 59, 78, 90

(1) I assume that you know how to write the algorithm for Merge(). [MIT book p.29]

(2) What is the time complexity for Merge()?

O(n+m) where n,m are lengths of lists

(3) Note that O(n+m) extra storage is required to merge the lists

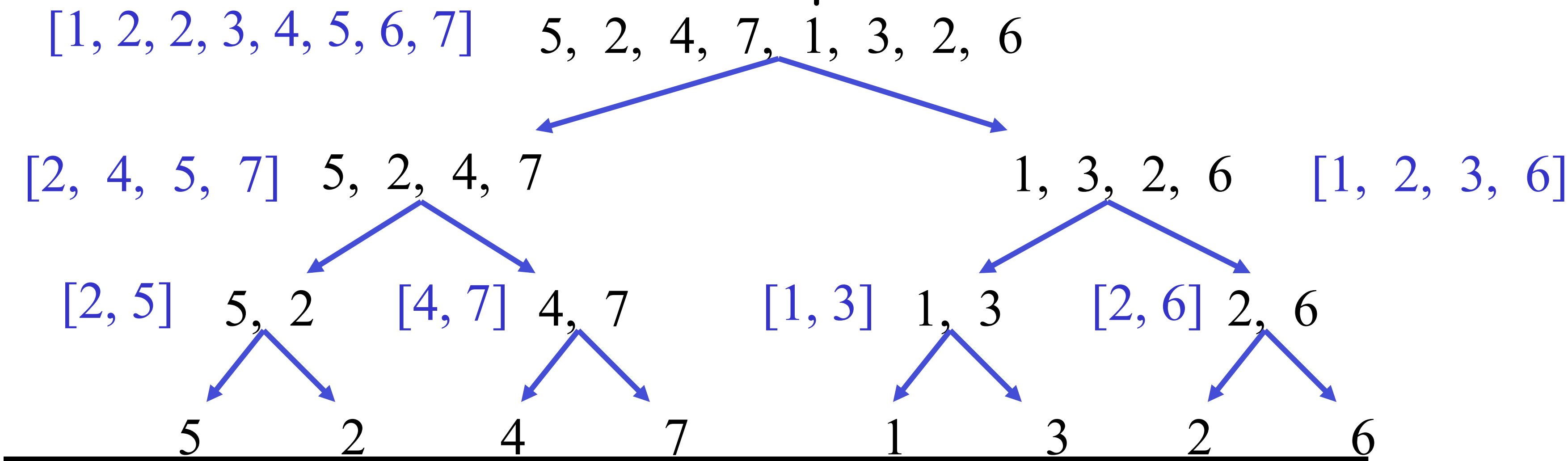
Merge sort:

Divide the numbers into lists of roughly equal lengths

Sort them recursively using merge sort

Then, merge two sorted lists

Example



<u>Algorithm</u>

```
MergeSort(A, p, r) {  // Merge sort A[p] to A[r]
    if (p < r) {   // terminating condition for recursion
        q = ⌊(p+r)/2⌋      // divide into roughly equal length lists
        MergeSort(A, p, q)
        MergeSort(A, q+1, r)
        Merge(A, p, q, r)   // Merge A[p]..A[q] and A[q+1]..A[r]
    }
}
```

Time complexity? $T(n) = \begin{cases} c_1 & \text{if } n = 1 \\ 2T(n/2) + c_2 n & \text{if } n > 1 \end{cases}$ $T(n) = \Theta(n \log n)$

A sorting algorithm is called in-place if only $O(1)$ additional work space is needed besides the initial array that holds the numbers

Bubble sort, insertion sort, and selection sort are in-place sorting algorithms while merge sort is not.

Two more in-place sorting algorithms:

Heap sort

Worst case: $O(n \log n)$

Quick sort

Worst case: $O(n^2)$; Average case: $O(n \log n)$

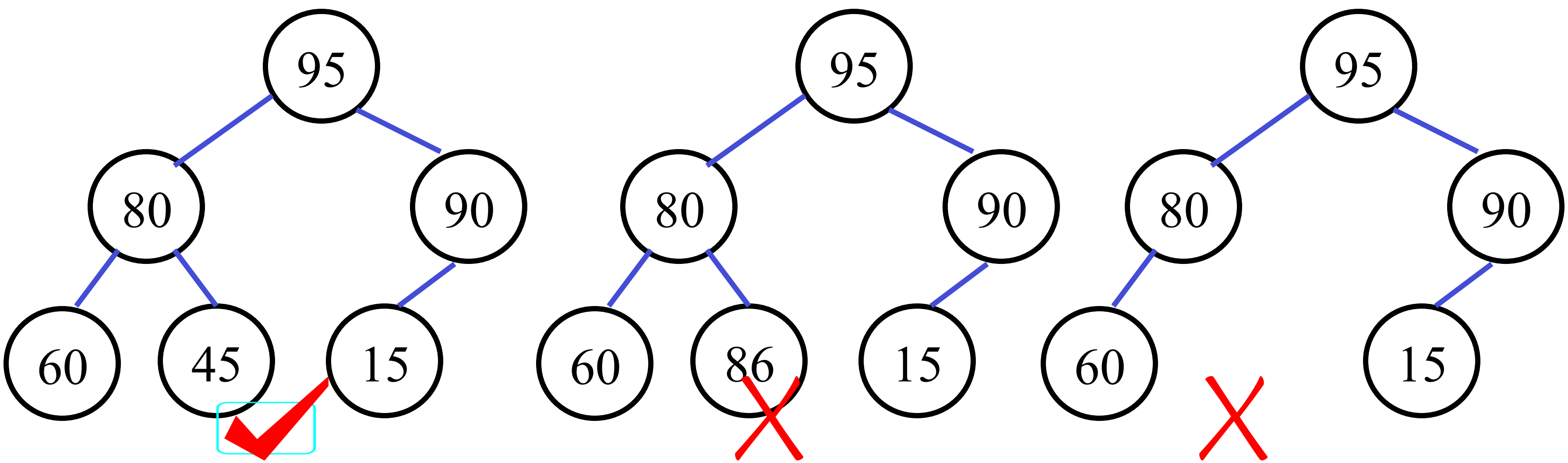
Heap sort

Before we talk about heap sort, let us discuss what is a heap and how to build a heap.

A (max-)heap is a binary tree (note: *NOT a binary search tree*) that satisfies the following.

(1) [(max-)heap property]
The value of a node ≥ the value of any of its children

(2) If the height of the tree is h, then there are 2^i nodes with depth i ($i < h$) and the nodes at depth h are "packed" from the left



Similarly, we can have a min-heap

Which node stores the maximum value? The root

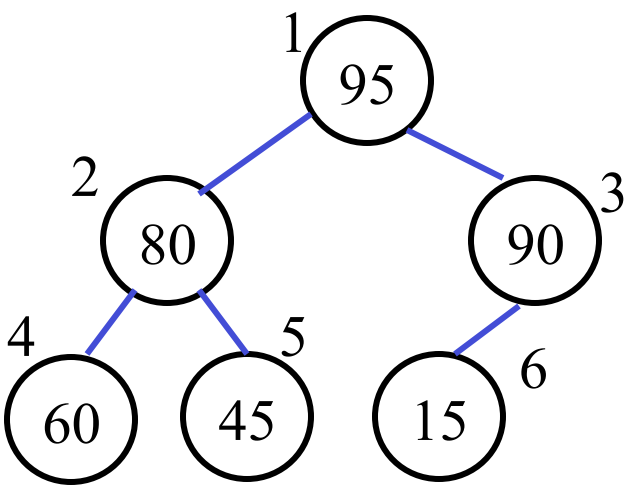
If the height of a heap is h, what is the maximum and minimum number of nodes in the heap?

max: $2^{h+1} - 1$
min: 2^h

If we look for the minimum value in a max-heap with distinct values, which nodes should we examiner? All leaves

We can store the values of the nodes in an array easily:

Starting from the root, label the node from left to right, and top to bottom with 1, 2, ... respectively. The node with label i is stored in the array entry A[i].



1	2	3	4	5	6
95	80	90	60	45	15

Remark:
Left child of A[i] is A[2i]
Right child of A[i] is A[2i+1]
Parent of A[i] is A[$\lfloor i/2 \rfloor$]

Besides sorting, heap is a useful data structure.

Consider the following application:

In a shared computer, we need to maintain the set of available jobs to be executed together with their priorities. When the CPU is free, we need to choose the job with the highest priority. Of course, we need to add new jobs or delete finished jobs (with highest priority) from the set as well.

Q: What kind of ADT we require?

The ADT must support the following operations.

Insert(S, x)
Insert the element x into the set S
Maximum(S)
Returns the element of S with the largest key
Extract-Max(S)
Removes and returns the element of S with the largest key

Priority Queue

Can you suggest an implementation of this ADT?

Approach 1: Use an array

Insert(S, x): always insert at the end of the array. O(1)

Maximum(S): need to search the whole array. O(n)

Extract-Max(S): need to search the whole array and pack elements. O(n)

Maximum and Extract-Max are too slow!

Approach 2: Use an AVL tree

Insert(S, x) O(1og n)

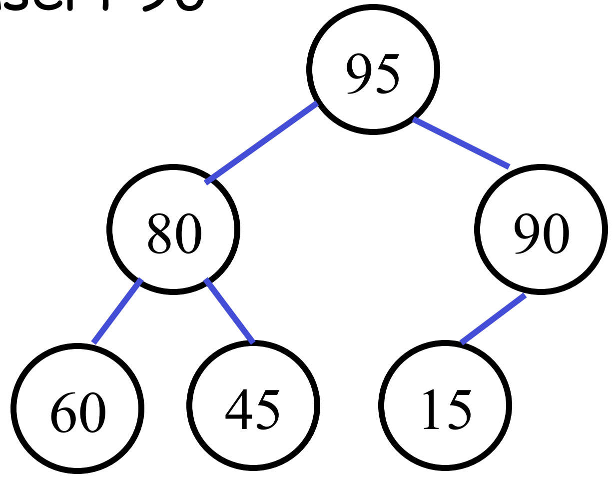
Maximum(S): locate the rightmost node O(log n)

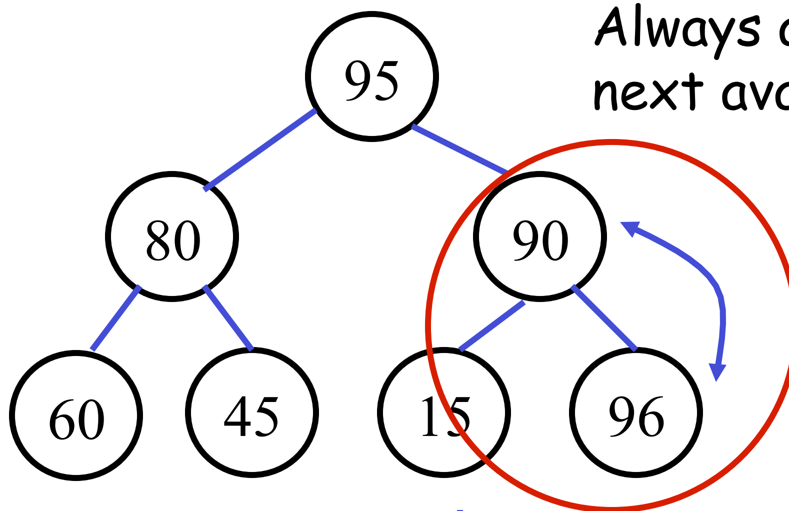
Extract-Max(S): locate the rightmost node and delete it O(log n)

Can we do better?

Yes, we will show that using heap, we can implement a priority queue such that (1) Insert takes O(log n) time; (2) Maximum takes O(1) time; and (3) Extract-Max takes O(log n) time.

Insert 96





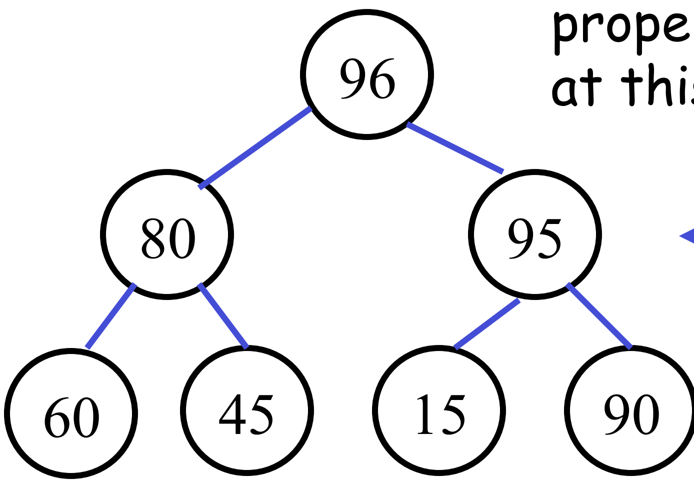
Always add to the next available slot

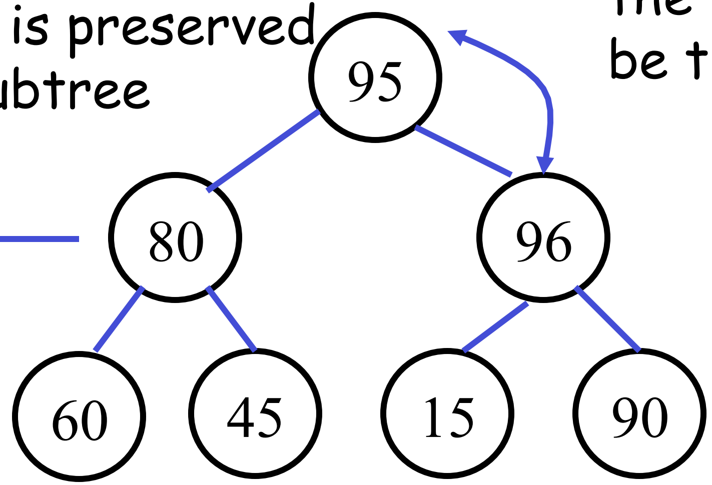
But violates the max-heap property

Repeat the same procedure up to the root.

Note: 90 is already > 15, so after the swap, the max-heap-property is preserved at this subtree

Check with parent, and let the larger value be the parent





If at some point, the original parent is larger, can we stop without going to the root?

```
Insert(A, x) {  // A is the array storing the heap
    A[size+1] = x;   // put x at the end; size stores the no. of elements in heap
    size ++;          // update heap size
    i = size;
    while (i > 1) and (A[i] > A[ ⌊i/2⌋ ]) {   // check heap property with parent
        swap(A[i], A[ ⌊i/2⌋ ]);
        i = ⌊i/2⌋
    }
}
```

Note: It is not necessary to use an array to store the heap. We can use pointers and make the above algorithm more general

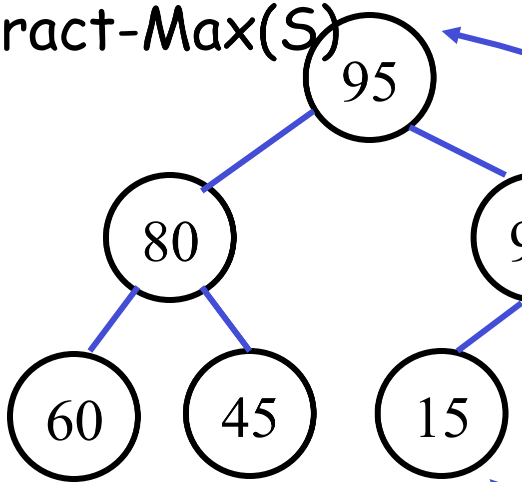
What's the time complexity?

$O(\log n)$

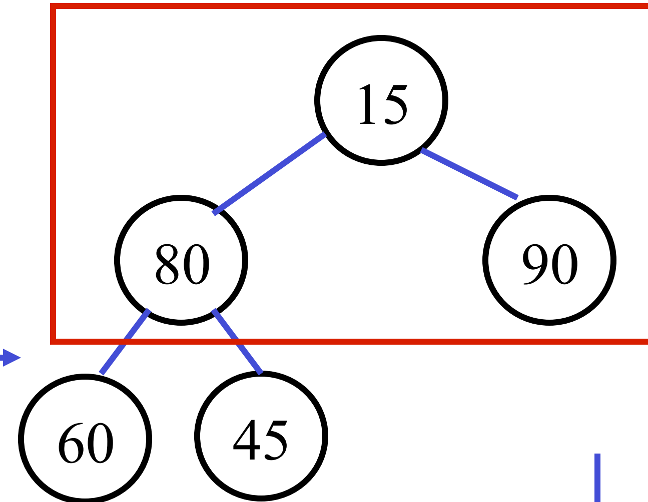
```
Maximum(A) {
    return A[1];
}
```

What's the time complexity? $O(1)$

Extract-Max(S)



1 Copy content

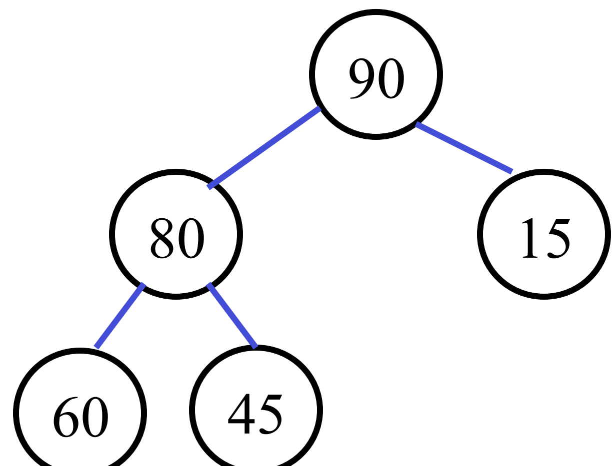


2 Delete this node

Violate max-heap property! What should we do?

3

Again, if the parent is smaller than any of the children, swap the larger child with the parent. Repeat this procedure until no more swapping is needed or reach a leaf.



Algorithm: left as an exercise

Time complexity: O(log n)

So, given n numbers, do you know how to build a heap?

<u>Method 1:</u>

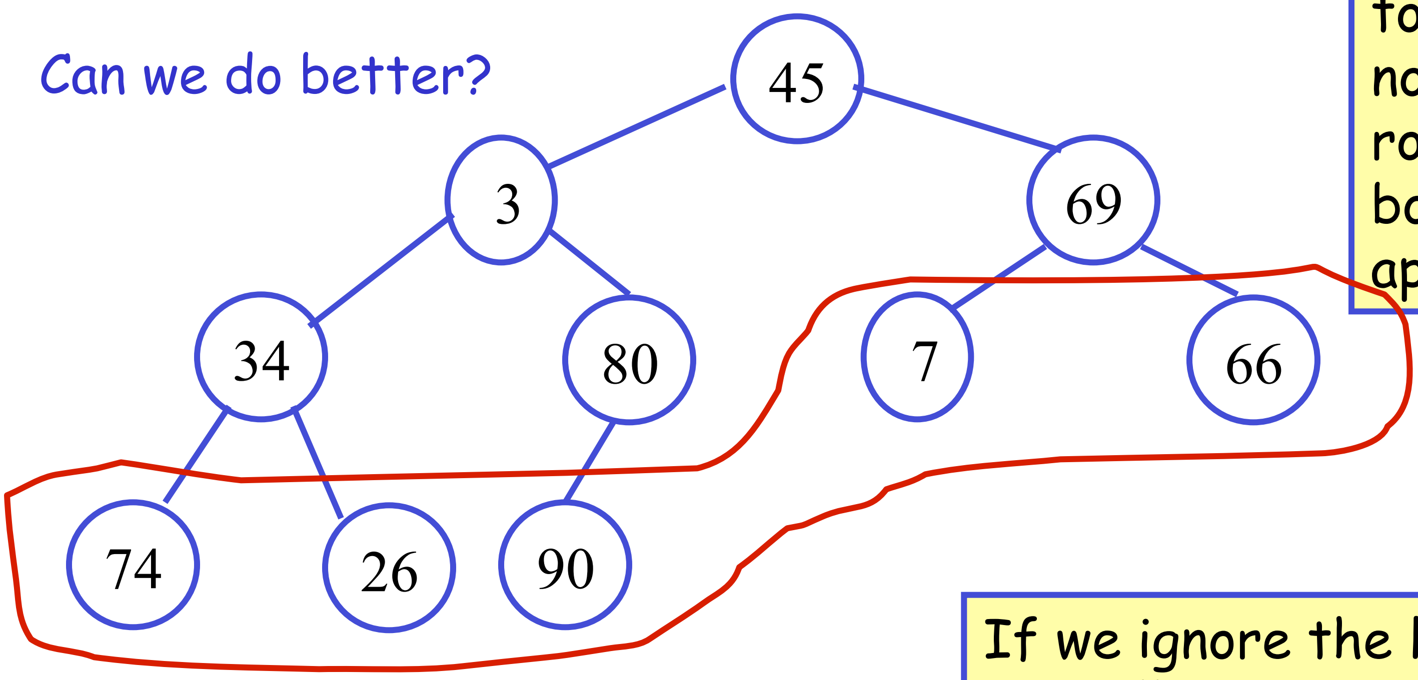
Insert the numbers one by one   Time complexity: O(n log n)

Example: 45, 3, 69, 34, 80, 7, 66, 74, 26, 90

Remarks
Method 1 can be regarded as a top-down approach

Hint: For top-down approach, we have to work on every node except the root. How about a bottom-up approach?

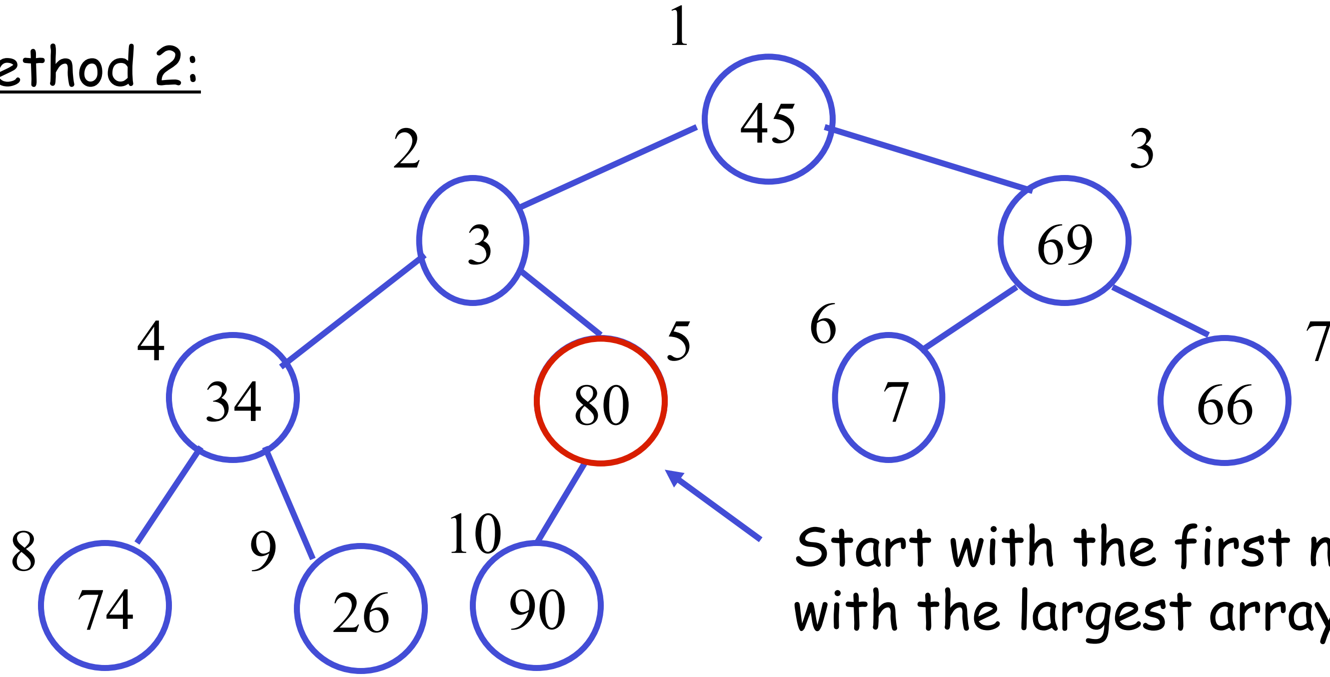
Can we do better?



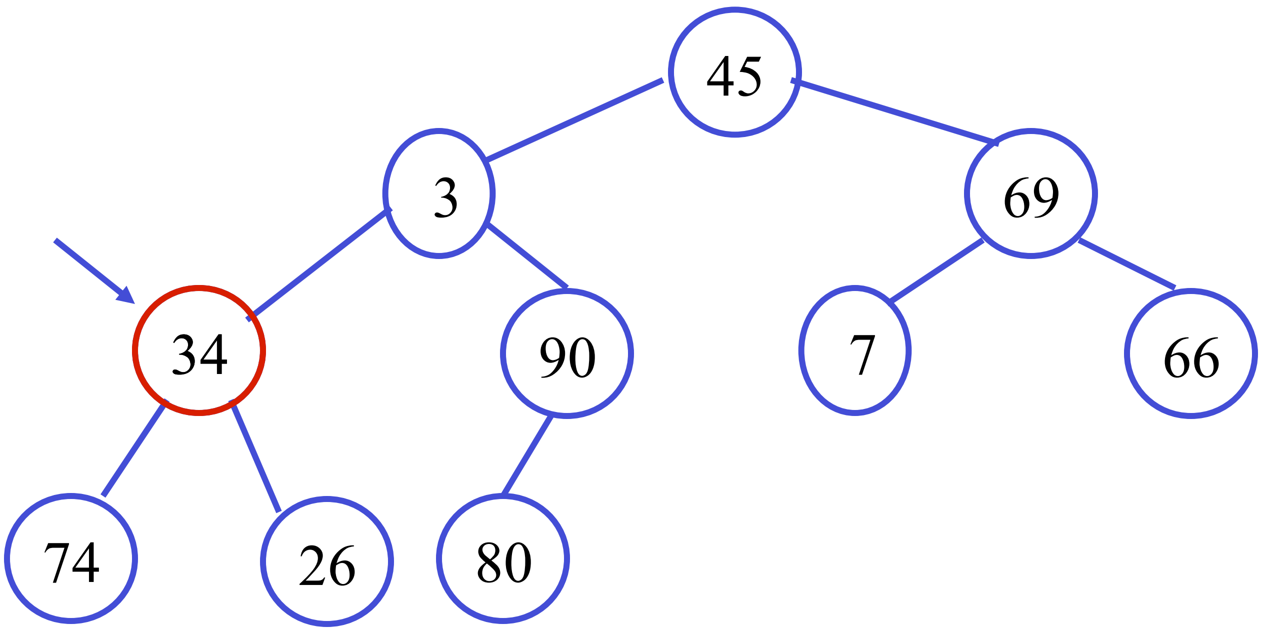
If we ignore the leaves and make sure all other nodes satisfy the max-heap property, then the tree will be a heap.

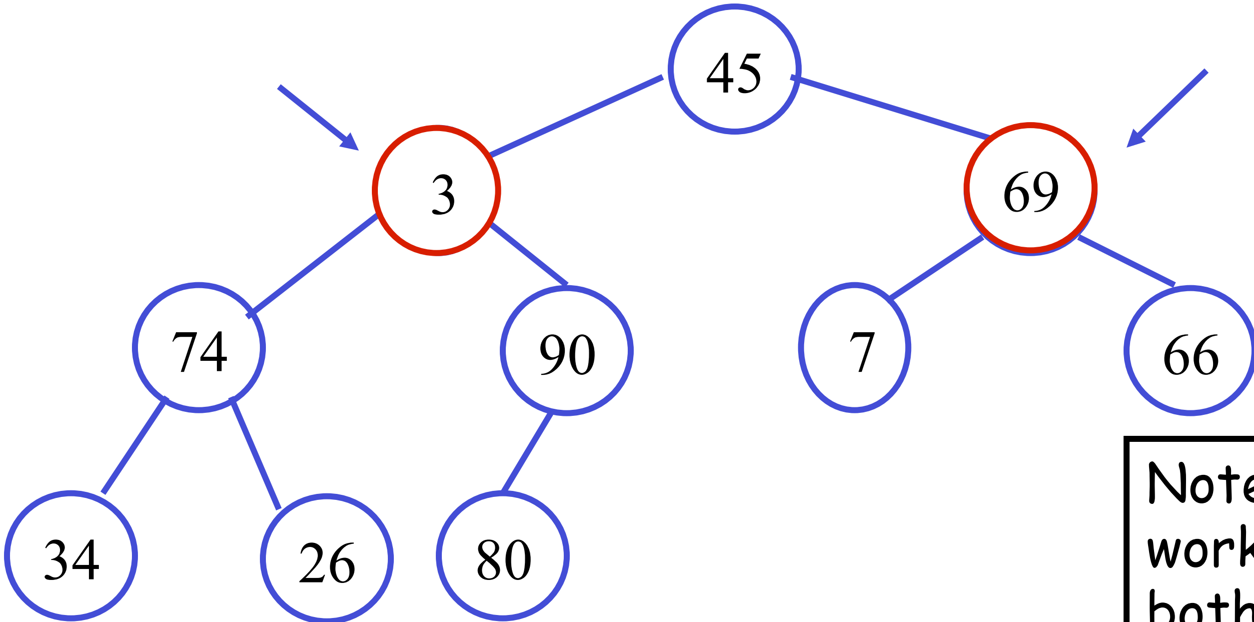
Observations:
a) about half of nodes are leaves;
b) leaves have greater depth

Method 2:

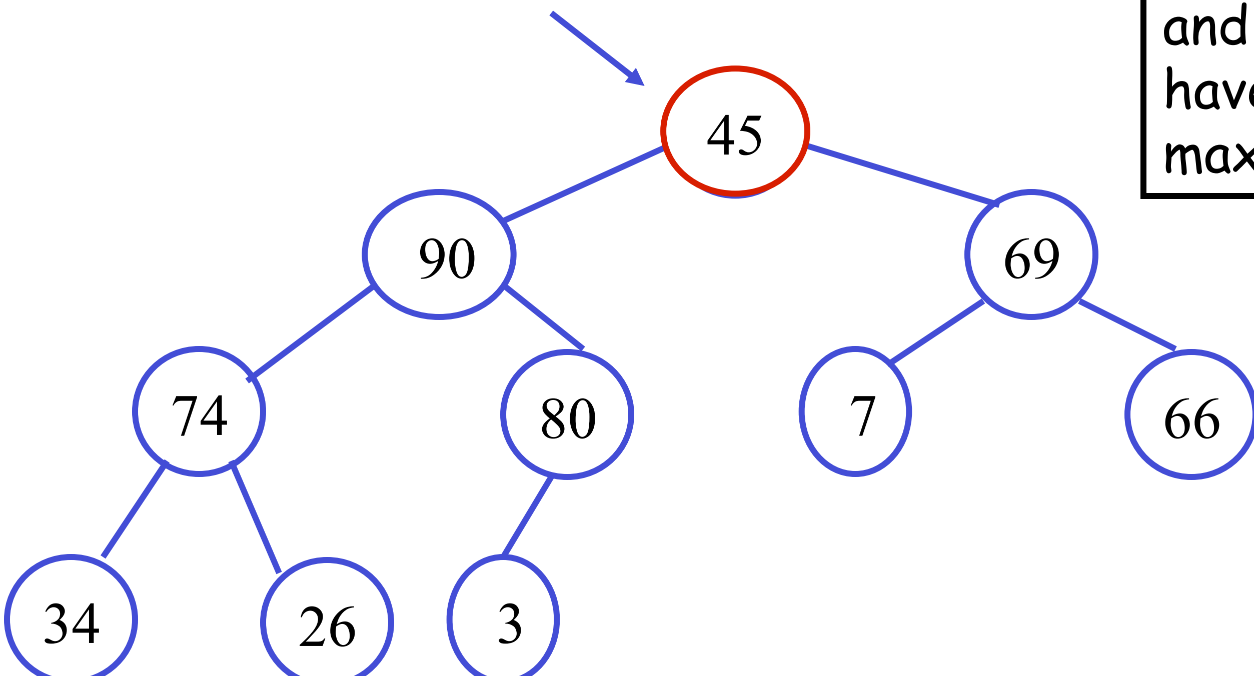


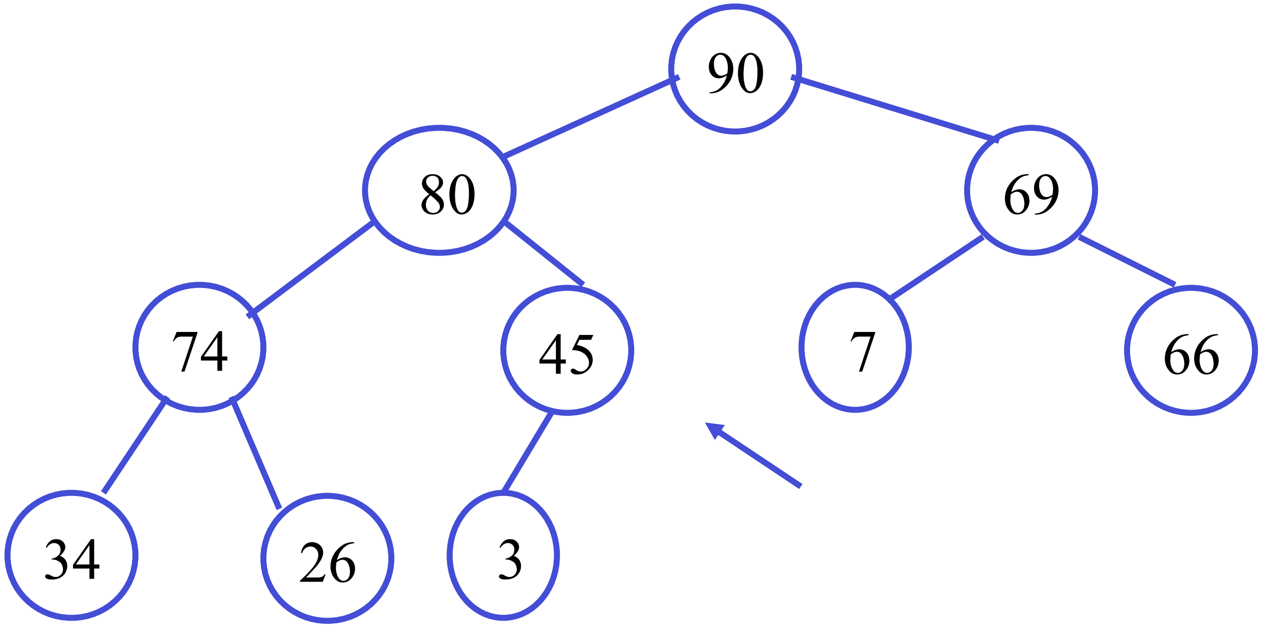
Start with the first non-leave node with the largest array index





Note that when we are working with node x, both the left subtree and right subtree of x have already satisfy the max-heap property!





Q: What is the index of the node that we should begin the process?

Q: Will the time complexity be improved?

<u>Answer to the first question</u>

Note that this node is with the largest index that has a child or in other words, it is the parent of the last node $A[n]$.

So, the index of the node is $A[\lfloor n/2 \rfloor]$

```
// push node A[i] down to make the subtree rooted at A[i] satisfy max-
// heap property. Assuming that both the subtrees of A[i] have satisfied
// max-heap property already.
Max-Heapify(A, i) {
    largest = i;
    if (i > ⌊n/2⌋)      // done if A[i] has no child
        return;
    left = 2i;              // check left child
    if (A[left] > A[i])   // left child value is bigger
        largest = left;
    if (2i + 1 ≤ n) {       // right child exists
        right = 2i+1;
        if (A[right] > A[largest])
            largest = right;
    }
    if (largest ≠ i) {       // one of the child is the largest of them
        swap(A[i], A[largest]);
        Max-Heapify(A, largest);
    }
}
```

```
Build-Max-Heap(A) {
    // size = n: no. of elements
    for (i = ⌊n/2⌋ downto 1) {
        Max-Heapify(A, i)
    }
}
```

Do you know how to write a non-recursive one?

What is the time complexity?

For Max-Heapify, let the size (# of nodes) of the subtree rooted at A[i] be m. Then, $T(m) \le T(p) + c$ where p is the max size of A[i]'s subtree, c is a constant

What is p? $p \le 2m/3$ (need to prove it, left as an exercise!)

$T(m) \le T(2m/3) + c$
$\Rightarrow T(m) = O(\log m)$
or $O(h)$ where h is the height of A[i]

Let the height of heap be h.
Then,
# of nodes with height h = 1= 2^0
# of nodes with height h-1 = 2 = 2^1
# of nodes with height h-2 = 2^2
...
# of nodes with height 1 $\le 2^{h-1}$
# of nodes with height 0 $\le 2^h$ (not included in the algorithm)

```
Build-Max-Heap(A) {
    // size = n: no. of elements
    for (i = ⌊n/2⌋ downto 1) {
        Max-Heapify(A, i)
    }
}
```

Time complexity for Build-Max-Heap =

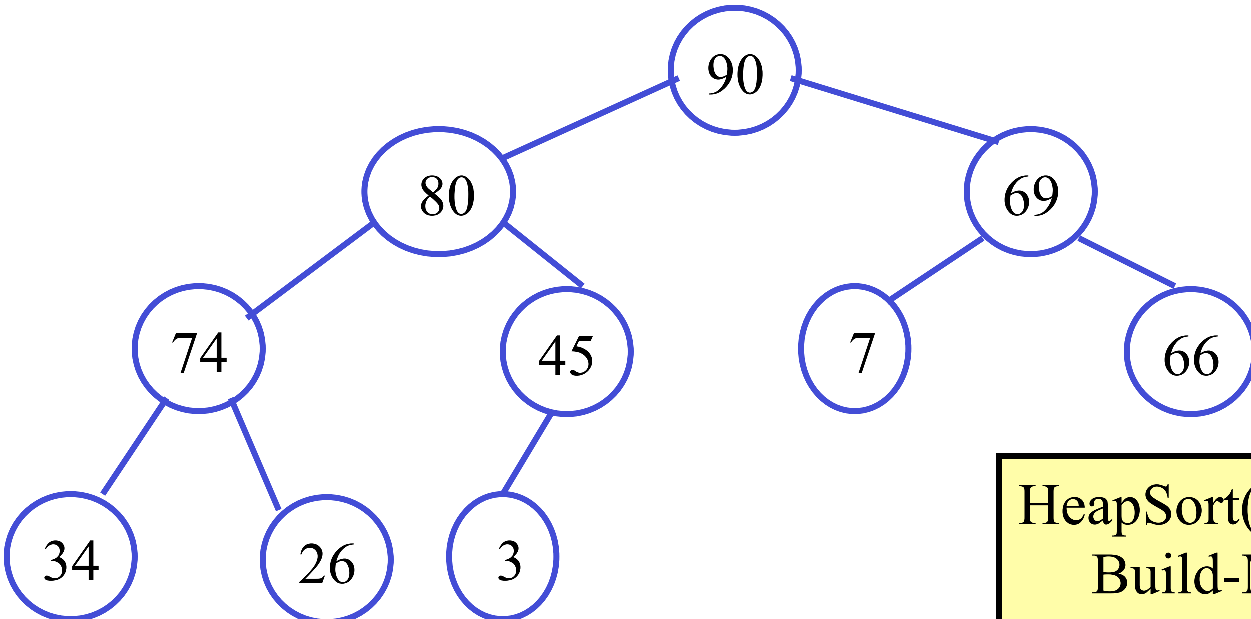
$$O\left(\sum_{i=1}^{h} i(2^{h-i})\right) = O\left(2^h\left(\sum_{i=1}^{h} \frac{i}{2^i}\right)\right)$$

$$= O(2^h) = O(n)$$

Now, you have learnt enough about heap, do you know how to do a heapsort?

Example:

45, 3, 69, 34, 80, 7, 66, 74, 26, 90



Hint: we can extract the next largest number easily!

```
HeapSort(A) {
    Build-Max-Heap(A);
    for (i = 1 to n-1) {
        temp = Extract-Max(A);
        A[size] = temp;
        size --;
    }
}
```

Time complexity: O(n log n)

A final remark on two heap construction methods

Method 1	Method 2
$O(n \log n)$	$O(n)$
Top-down	Bottom-up
On-line	Off-line

In an on-line version, data will only be given to you one by one when it arrives. For example, in the shared computer example, the jobs submitted by the users will be known by the system at the time they are submitted.

In an off-line version, the whole set of data will be given to you before the start of the construction algorithm.

Do you notice that heap sort is also an in-place sorting algorithm?

From the practical point of view, heap sort and merge sort are not the best, we have quick sort

Worst case: $O(n^2)$; Average case: $O(n \log n)$ Why?

Quick sort

Idea: Use the idea of merge sort by dividing the numbers into 2 lists, but in merge sort, it requires a lot of work in combining the two sorted lists? In Quick sort, we want to make the combining part easy.

Example

1	2	3	4	5	6	7	8	9
91	85	90	60	45	15	23	47	35

Note: In merge sort, we just divide the input into 2 lists by cutting roughly in the middle, then proceed to sort the two lists. In quick sort, we do something more:

1) pick an element A (called pivot v)
2) move elements ≤ v to the left of v
3) move elements > v to the right of v

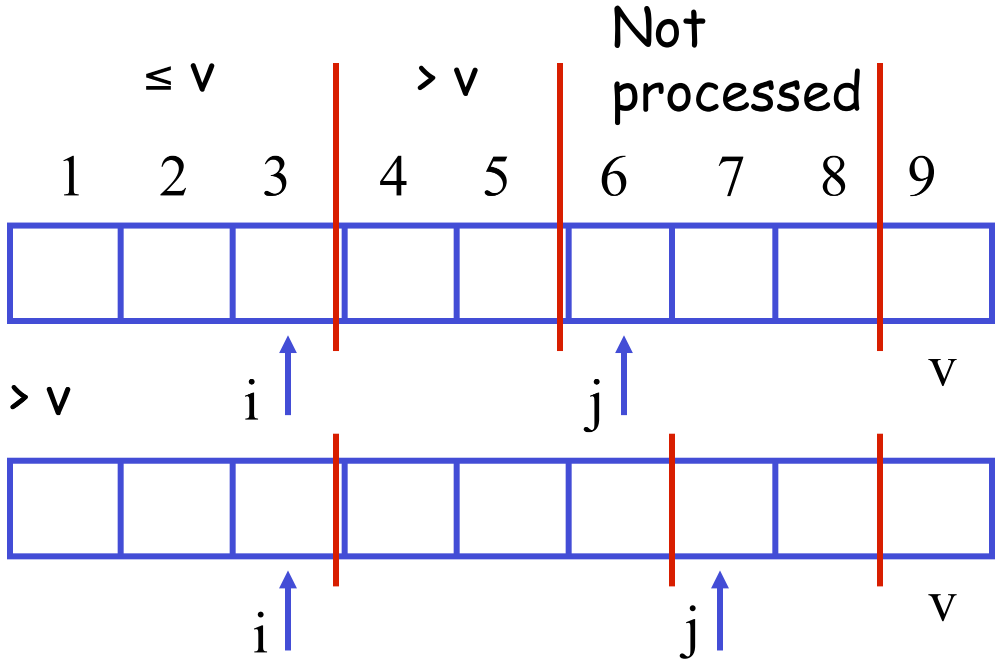
Can you see that no work needs to be done in the combining part in quick sort?

Example: pick v = A[n], after the rearrangement, we divide the numbers into:

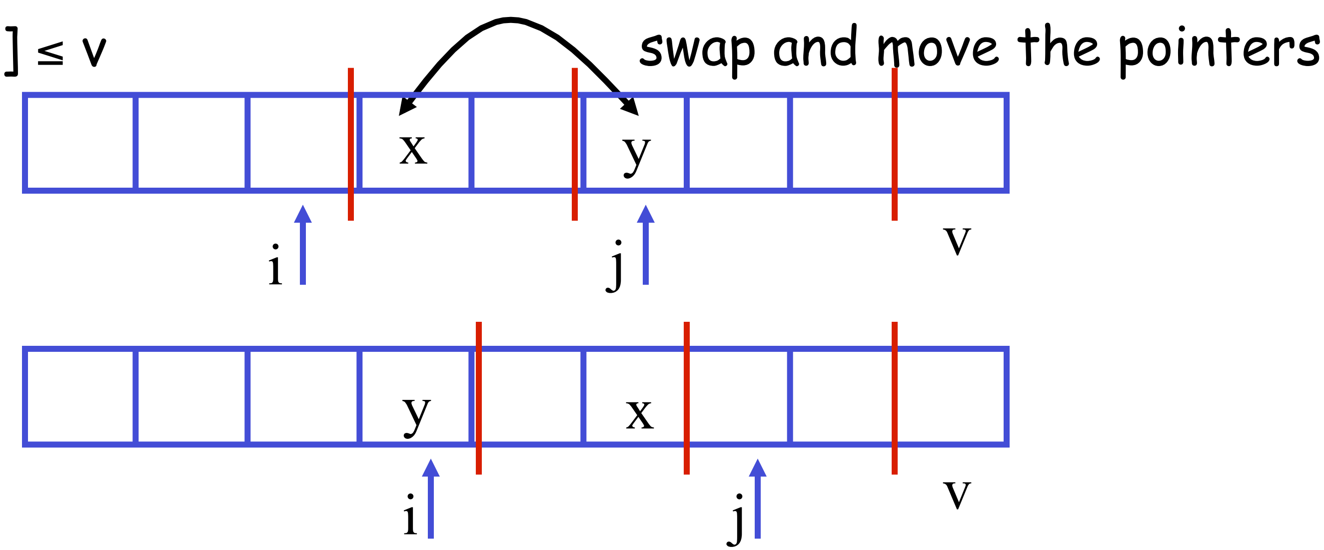
15, 23   35   85, 90, 60, 45, 47, 91

How to do this "in-place"?

Case 1: if A[j] > v



Case 2: if A[j] ≤ v



How about when we get to v? swap(A[n], A[i+1])

Example

1	2	3	4	5	6	7	8	9
91	85	90	60	45	15	23	47	35

i = 0 j

91	85	90	60	45	15	23	47	35

i = 0 j

91	85	90	60	45	15	23	47	35

i = 0 j

91	85	90	60	45	15	23	47	35

i = 0 j

91	85	90	60	45	15	23	47	35

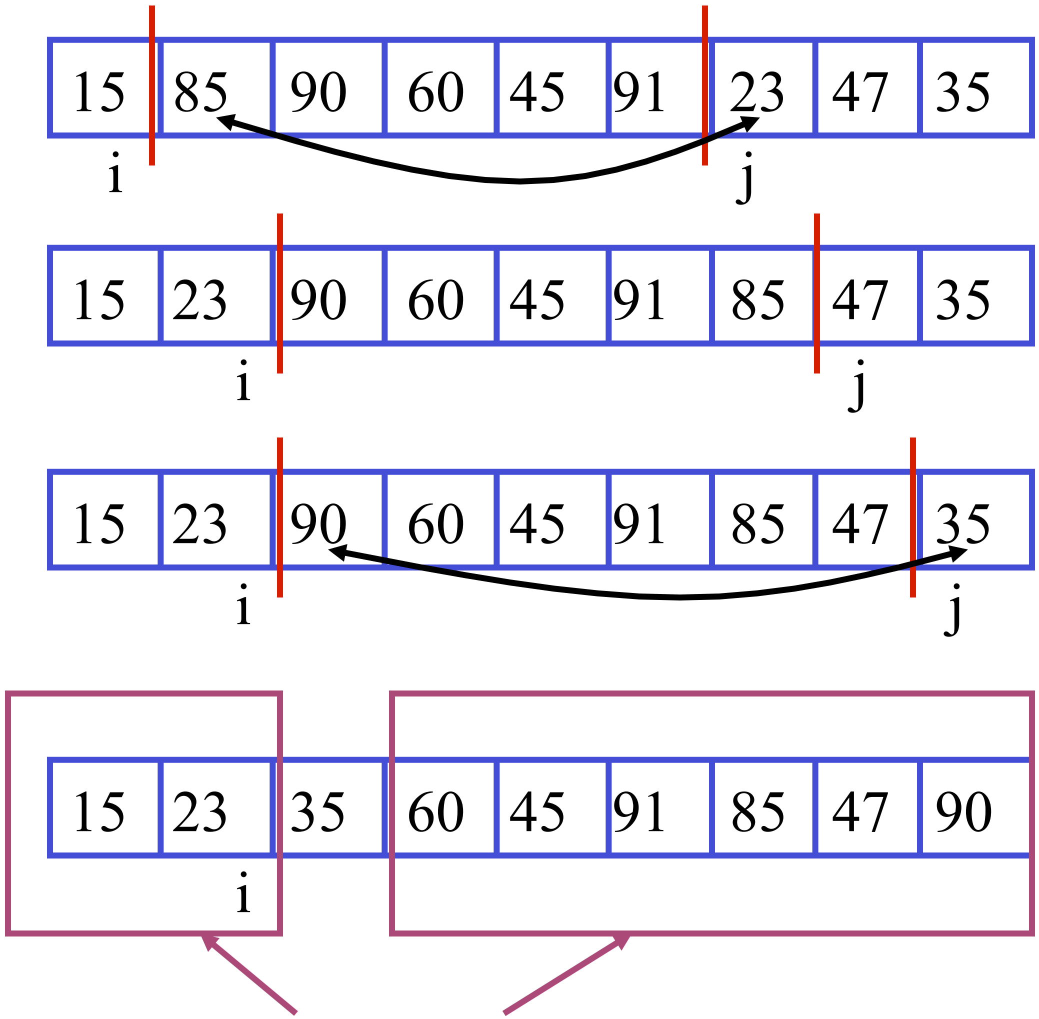
i = 0 j

91	85	90	60	45	15	23	47	35

i = 0 j

15	85	90	60	45	91	23	47	35

i j



Call Quick sort recursively

Note:
Other textbooks may have a different method to do the partitioning
Can we pick other elements to be pivot?

```
Partition(A, p, r) {      // partition A[p..r] using A[r] as pivot
    v = A[r];
    i = p-1;
    for (j = p to r-1) {
        if (A[j] ≤ v) {      // swap!
            swap(A[j], A[i+1]);
            i = i + 1;
        }
    swap(A[r], A[i+1]);       // for the pivot
    return i+1;               // return position of pivot
}
```

Time complexity: O(n)

```
QuickSort(A, p, r) {  // sort A[p..r]
    if (p < r) {  // otherwise, done!
        q = Partition(A, p, r);
        QuickSort(A, p, q-1);
        QuickSort(A, q+1, r);
    }
}
```

Time complexity?

Can you sort this using Quick sort?

91	85	90	60	45	15	23	47	35

From the algorithm, the total size of two subproblems is n-1
$T(n) = T(m) + T(n-m-1) + cn$ // $0 \leq m \leq n-1$

So, what is the worst case, best case, and average case?

Worst case

One subproblem has 0 numbers while the other has n-1 numbers

$T(n) = T(n-1) + cn$ // Note: $T(0) = O(1)$

....

$T(n) = O(n^2)$

e.g. A = ‹ 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1›

Best case

You always locate the median as pivot, so two subproblems are of roughly equal size

$T(n) = 2T(n/2) + cn$

....

$T(n) = O(n \log n)$

Can you come up with an example for this case?

Average case

Assumption: Numbers are all distinct and the input sequence of these n numbers can be any of the possible permutations with equal probability. So, the pivot can be any of these numbers with equal probability.

Possible cases:

$T(n) = T(0) + T(n-1) + cn$

$T(n) = T(1) + T(n-2) + cn$

……

$T(n) = T(n-1) + T(0) + cn$

> Remark: it can be shown that not "too many" cases will make Quick sort run in $O(n^2)$ time, so in practice, it usually runs in $O(n \log n)$ time

$$T(n) = \frac{1}{n}[\sum_{m=0}^{n-1}(T(m) + T(n-m-1))] + cn$$

$$= \frac{2}{n}[\sum_{m=0}^{n-1} T(m)] + cn$$

Using the substitution method and assume that $T(n) \le a\, n \log n + b$, we obtain $T(n) = O(n \log n)$