Chapter 3
Divide and Conquer Algorithms

Sorting

Sorting is among the most basic problems in algorithm design. We are given a sequence of items, each associated with a given key value. The problem is to permute the items so that they are in increasing (or decreasing) order by key. Sorting is important because it is often the first step in more complex algorithms. Sorting algorithms are usually divided into two classes, internal sorting algorithms, which assume that data is stored in an array in main memory, and external sorting algorithm, which assume that data is stored on disk or some other device that is best accessed sequentially. We will only consider internal sorting. Sorting algorithms often have additional properties that are of interest, depending on the application. Here are two important properties.

In-place: The algorithm uses no additional array storage, and hence (other than perhaps the system's recursion stack) it is possible to sort very large lists without the need to allocate additional working storage.

Stable: A sorting algorithm is stable if two elements that are equal remain in the same relative position after sorting is completed. This is of interest, since in some sorting applications you sort first on one key and then on another. It is nice to know that two items that are equal on the second key, remain sorted on the first key.

Bubble Sort

The bubble sort algorithm Compare adjacent elements. If the first is greater than the second, swap them. This is done for every pair of adjacent elements starting from first two elements to last two elements. At the end of pass1 greatest element takes its proper place. The whole process is repeated except for the last one so that at each pass the comparisons become fewer.

Algorithm

```
BubbleSort(A, n)
{
        for(i = 0; i <n-1; i++)
        {
        for(j = 0; j < n-i-1; j++)
        {
                if(A[j] > A[j+1])
                {
                        temp = A[j];
                        A[j] = A[j+1];
                        A[j+1] = temp;
                }
        }
        }
}
```

Time Complexity:

Inner loop executes for (n-1) times when i=0, (n-2) times when i=1 and so on:

Time complexity = (n-1) + (n-2) + (n-3) + …… +2 +1

$= O(n^2)$

There is no best-case linear time complexity for this algorithm.

Space Complexity:

Since no extra space besides 3 variables is needed for sorting

Space complexity = O(n)

Selection Sort

Idea: Find the least (or gratest) value in the array, swap it into the leftmost(or rightmost) component (where it belongs), and then forget the leftmost component. Do this repeatedly.

Algorithm:

```
SelectionSort(A)
{
    for( i = 0;i < n ;i++)
    {
        least=A[i]; p=i;
        {
        for ( j = i + 1;j < n ;j++)
            if (A[j] < A[i])
            least= A[j]; p=j;
        }
    }
        swap(A[i],A[p]);
}
```

Time Complexity:

Inner loop executes for (n-1) times when i=0, (n-2) times when i=1 and so on:

Time complexity = (n-1) + (n-2) + (n-3) + ……. +2 +1

= $O(n^2)$

There is no best-case linear time complexity for this algorithm, but number of swap operations is reduced greatly.

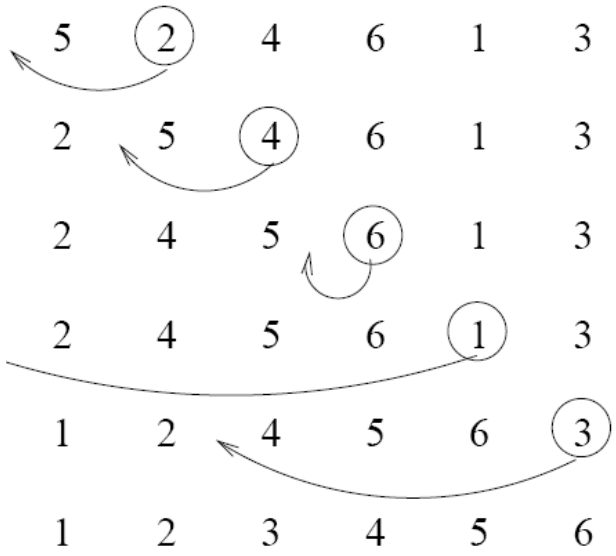
Space Complexity:

Since no extra space besides 5 variables is needed for sorting

Space complexity = O(n)

Insertion Sort

Idea: like sorting a hand of playing cards start with an empty left hand and the cards facing down on the table. Remove one card at a time from the table, and insert it into the correct position in the left hand. Compare it with each of the cards already in the hand, from right to left. The cards held in the left hand are sorted



Algorithm:

```
InsertionSort(A)
    {
        for (i=1;i<n;i++)
        {
            key = A[ i]
             for(j=i; j>0 && A[j] >key; j--)
            {
                A[j + 1] = A[j]
            }
             A[j + 1] = key

        }
    }
```

Time Complexity:

Worst Case Analysis:

Array elements are in reverse sorted order

Inner loop executes for 1 times when i=1, 2 times when i=2... and n-1 times when i=n-1:

Time complexity = 1 + 2 + 3 + ……. +(n-2) +(n-1)

= $O(n^2)$

Best case Analysis:

Array elements are already sorted

Inner loop executes for 1 times when i=1, 1 times when i=2... and 1 times when i=n-1:

Time complexity = 1 + 2 + 3 + ……. +1 +1

= O(n)

Space Complexity:

Since no extra space besides 5 variables is needed for sorting

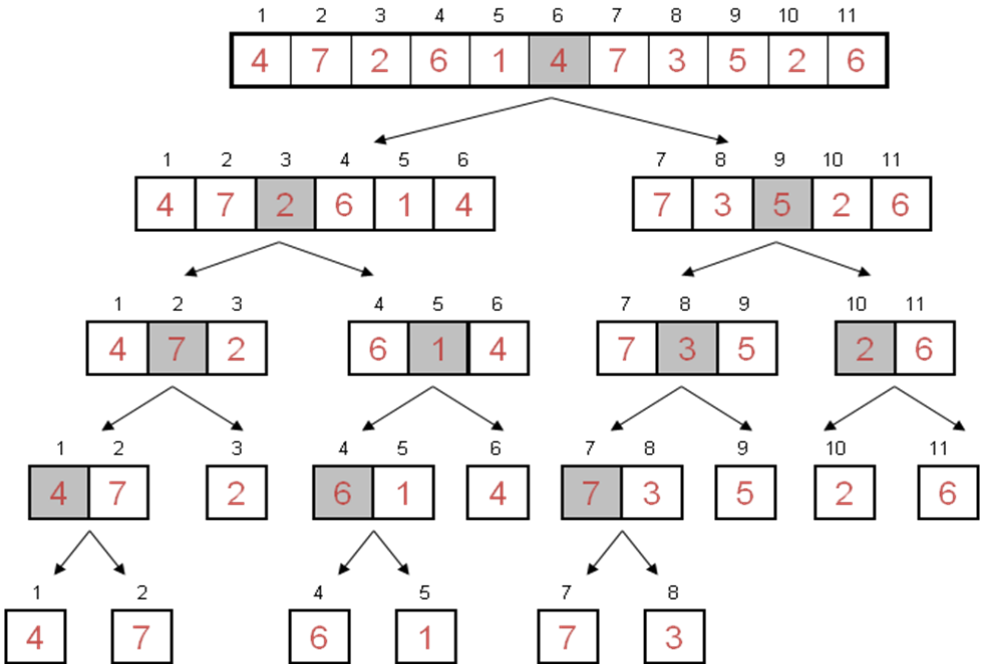
Space complexity = O(n)

Merge Sort

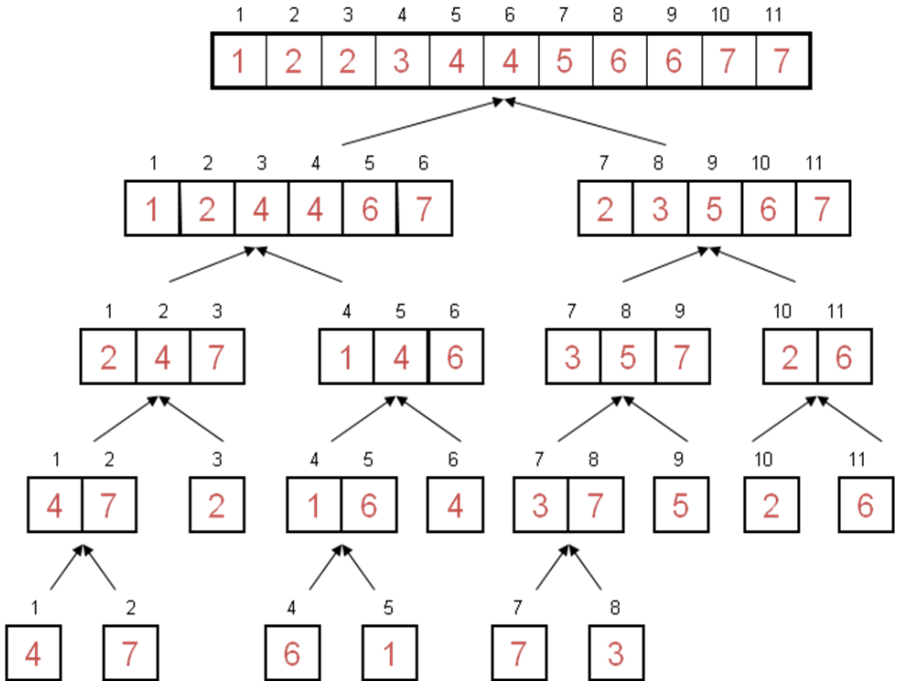
To sort an array A[l . . r]:

- Divide
  - Divide the n-element sequence to be sorted into two subsequences of n/2 elements each
- Conquer
  - Sort the subsequences recursively using merge sort. When the size of the sequences is 1 there is nothing more to do
- Combine
  - Merge the two sorted subsequences

Divide



Merging:




```
MergeSort(A, l, r)
{
        If ( l < r)
        {                                   //Check for base case
           m = ⌊(l + r)/2⌋                  //Divide
                MergeSort(A, l, m)          //Conquer
                MergeSort(A, m + 1, r)      //Conquer
                Merge(A, l, m+1, r)         //Combine
        }
}
```

```
Merge(A,B,l,m,r)
{
        x=l, y=m;
        k=l;
        while(x<m && y<r)
```

```
        {
                if(A[x] < A[y])
                {
                        B[k]= A[x];
                        k++; x++;
                }
                else
                {
                        B[k] = A[y];
                        k++; y++;
                }
        }
        while(x<m)
        {
                A[k] = A[x];
                k++; x++;
        }
        while(y<r)
        {
                A[k] = A[y];
                k++; y++;
        }
        for(i=l;i<= r; i++)
        {
                A[i] = B[i]
        }
}
```

Time Complexity:

Recurrence Relation for Merge sort:

$T(n) = 1$ if $n=1$

$T(n) = 2\ T(n/2) + O(n)$ if $n>1$

Solving this recurrence we get

Time Complexity = O(nlogn)

Space Complexity:

It uses one extra array and some extra variables during sorting, therefore

Space Complexity= $2n + c = O(n)$

Quick Sort

- **Divide**

  Partition the array A[l…r] into 2 subarrays A[l..m] and A[m+1..r], such that each element of A[l..m] is smaller than or equal to each element in A[m+1..r]. Need to find index p to partition the array.

- **Conquer**

  Recursively sort A[p..q] and A[q+1..r] using Quicksort

- **Combine**

  Trivial: the arrays are sorted in place. No additional work is required to combine them.

5	3	2	6	4	1	3	7
x							y
5	3	2	6	4	1	3	7
			x			y	{swap x & y}
5	3	2	3	4	1	6	7
					y	x	{swap y and pivot}
1	3	2	3	4	5	6	7
					p		

Algorithm:

```
QuickSort(A,l,r)
{
    if(l<r)
    {
        p = Partition(A,l,r);
        QuickSort(A,l,p-1);
        QuickSort(A,p+1,r);
    }
}
```

```
Partition(A,l,r)
{
x =l; y =r ; p = A[l];
while(x<y)
{
        do {
                x++;
        }while(A[x] <= p);
        do {
                y--;
        } while(A[y] >=p);
        if(x<y)
                swap(A[x],A[y]);
}
A[l] = A[y]; A[y] = p; return y;   //return position of pivot
}
```

Time Complexity:

We can notice that complexity of partitioning is O(n) because outer while loop executes cn times.

Thus recurrence relation for quick sort is:

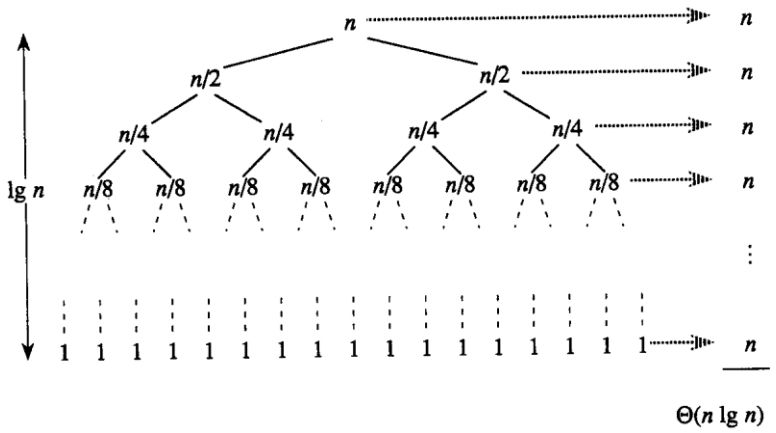
$T(n) = T(k) + T(n-k-1) + O(n)$

Best Case:

Divides the array into two partitions of equal size, therefore

$T(n) = T(n/2) + O(n)$, Solving this recurrence we get,

⇨ Time Complexity = O(nlogn)

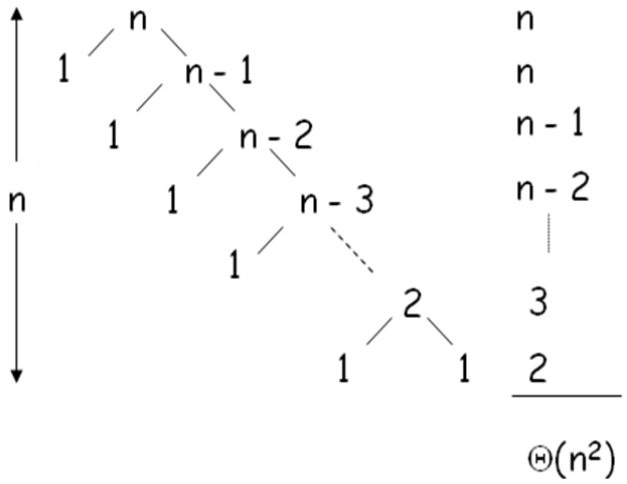


T, TU

When array is already sorted or sorted in reverse order, one partition contains n-1 items and another contains zero items, therefore

T(n) = T(n-1) + O(1), Solving this recurrence we get

⇨ Time Complexity = $O(n^2)$

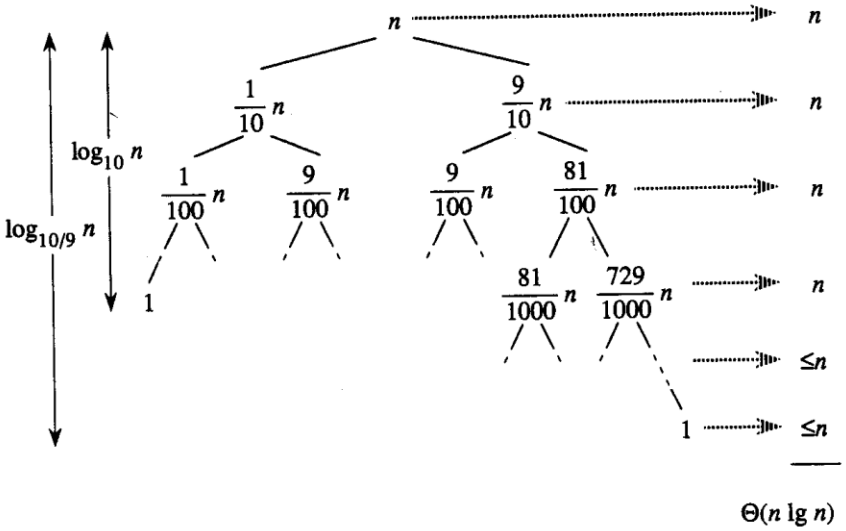


Case between worst and best:

9-to-1 partitions split

T(n) = T(n=9n/10) + T(n/10) + O(n), Solving this recurrence we get

Time Complexity = O(nolgn)



Average case:

All permutations of the input numbers are equally likely. On a random input array, we will have a mix of well balanced and unbalanced splits. Good and bad splits are randomly distributed across throughout the tree

Suppose we are alternate: Balanced, Unbalanced,Balanced, ….

$B(n)= 2UB(n/2) + \Theta(n)$ Balanced

$UB(n)= B(n-1) + \Theta(n)$ Unbalanced

Solving:

$B(n)= 2(B(n/2-1) + \Theta(n/2)) + \Theta(n)$

$= 2B(n/2-1) + \Theta(n)$

$= \Theta(nlogn)$

Randomized Quick Sort:

The algorithm is called randomized if its behavior depends on input as well as random value generated by random number generator. The beauty of the randomized algorithm is that no particular input can produce worst-case behavior of an algorithm. IDEA: Partition around a random element. Running time is independent of the input order. No assumptions need to be made about the input distribution. No specific input elicits the worst-case behavior. The worst case is determined only by the output of a random-number generator. Randomization cannot eliminate the worst-case but it can make it less likely!

Algorithm:

```
RandQuickSort(A,l,r)
{
    if(l<r)
    {
            m = RandPartition(A,l,r);
            RandQuickSort(A,l,m-1);
            RandQuickSort(A,m+1,r);
    }
}
```

```
RandPartition(A,l,r)
{
    k = random(l,r); //generates random number between i and j including both.
    swap(A[l],A[k]);
    return Partition(A,l,r);
}
```

Time Complexity:

Worst Case:

T(n) = worst-case running time

$T(n) = \max_{1 \le q \le n-1} (T(q) + T(n-q)) + \Theta(n)$

Use substitution method to show that the running time of Quicksort is $O(n^2)$

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

- Induction goal: $T(n) \le cn^2$
- Induction hypothesis: $T(k) \le ck^2$ for any $k < n$

Proof of induction goal:

$T(n) \le \max_{1 \le q \le n-1} (cq^2 + c(n-q)^2) + \Theta(n)$

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

The expression $q^2 + (n-q)^2$ achieves a maximum over the range $1 \le q \le n-1$ at one of the endpoints

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

$T(n) \le cn^2 - 2c(n - 1) + \Theta(n)$

$\le cn^2$

Average Case:

To analyze average case, assume that all the input elements are distinct for simplicity. If we are to take care of duplicate elements also the complexity bound is same but it needs more intricate analysis. Consider the probability of choosing pivot from n elements is equally likely i.e. 1/n.

Now we give recurrence relation for the algorithm as

$T(n) = 1/n \sum_{k=1}^{n-1} (T(k) + T(n-k)) + O(n)$

For some k = 1,2, …, n-1, T(k) and T(n-k) is repeated two times

$T(n) = 2/n \sum_{k=1}^{n-1} T(k) + O(n)$

$nT(n) = 2 \sum_{k=1}^{n-1} T(k) + O(n^2)$

Similarly

$(n-1)T(n-1) = 2 \sum_{k=1}^{n-2} T(k) + O(n-1)^2$

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

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

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

Let $An = T(n)/(n+1)$

- ⇨ $A_n = A_{n-1} + (2n+1)/n(n-1)$
- ⇨ $An = \sum_{i=1}^{n} 2i - 1/i(i+1)$
- ⇨ $An \approx \sum_{i=1}^{n} 2i/i(i+1)$
- ⇨ $An \approx 2 \sum_{i=1}^{n} 1/(i+1)$
- ⇨ $An \approx 2\log n$

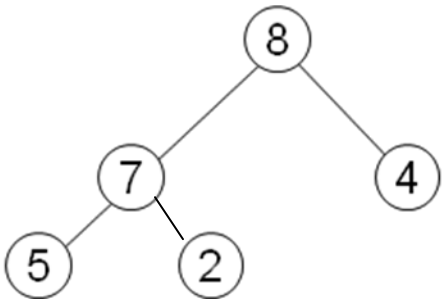
Since $An = T(n)/(n+1)$

$T(n) = n\log n$

Heap Sort

A **heap** is a nearly complete binary tree with the following two properties:

- **Structural property:** all levels are full, except possibly the last one, which is filled from left to right
- **Order (heap) property:** for any node x, Parent(x) ≥ x

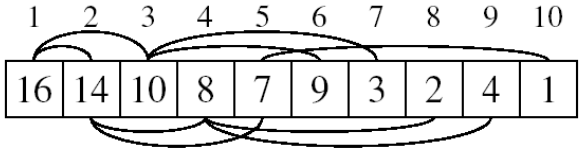


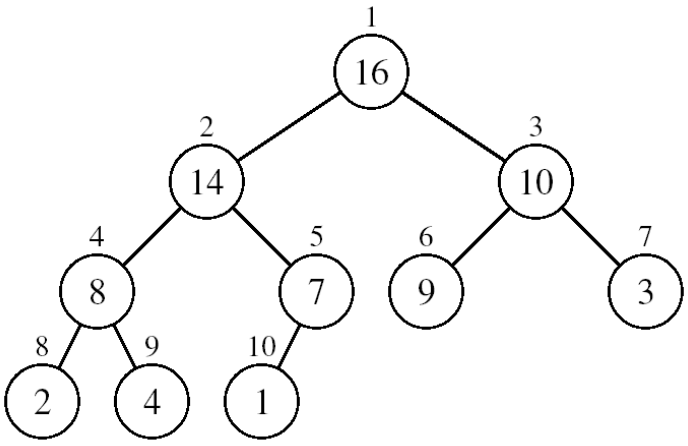
Array Representation of Heaps

A heap can be stored as an array *A*.

- Root of tree is A[1]
- Left child of A[i] = A[2i]
- Right child of A[i] = A[2i + 1]
- Parent of A[i] = A[$\lfloor i/2 \rfloor$]
- Heapsize[A] ≤ length[A]

The elements in the subarray A[($\lfloor n/2 \rfloor$+1) .. n] are leaves





Max-heaps (largest element at root), have the max-heap property:

- for all nodes i, excluding the root:

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

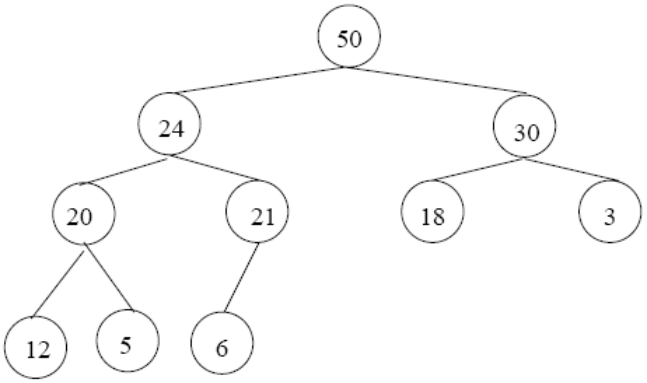
Min-heaps (smallest element at root), have the min-heap property:

- for all nodes i, excluding the root:

$$A[PARENT(i)] \leq A[i]$$

Adding/Deleting Nodes

New nodes are always inserted at the bottom level (left to right) and nodes are removed from the bottom level (right to left).



Operations on Heaps

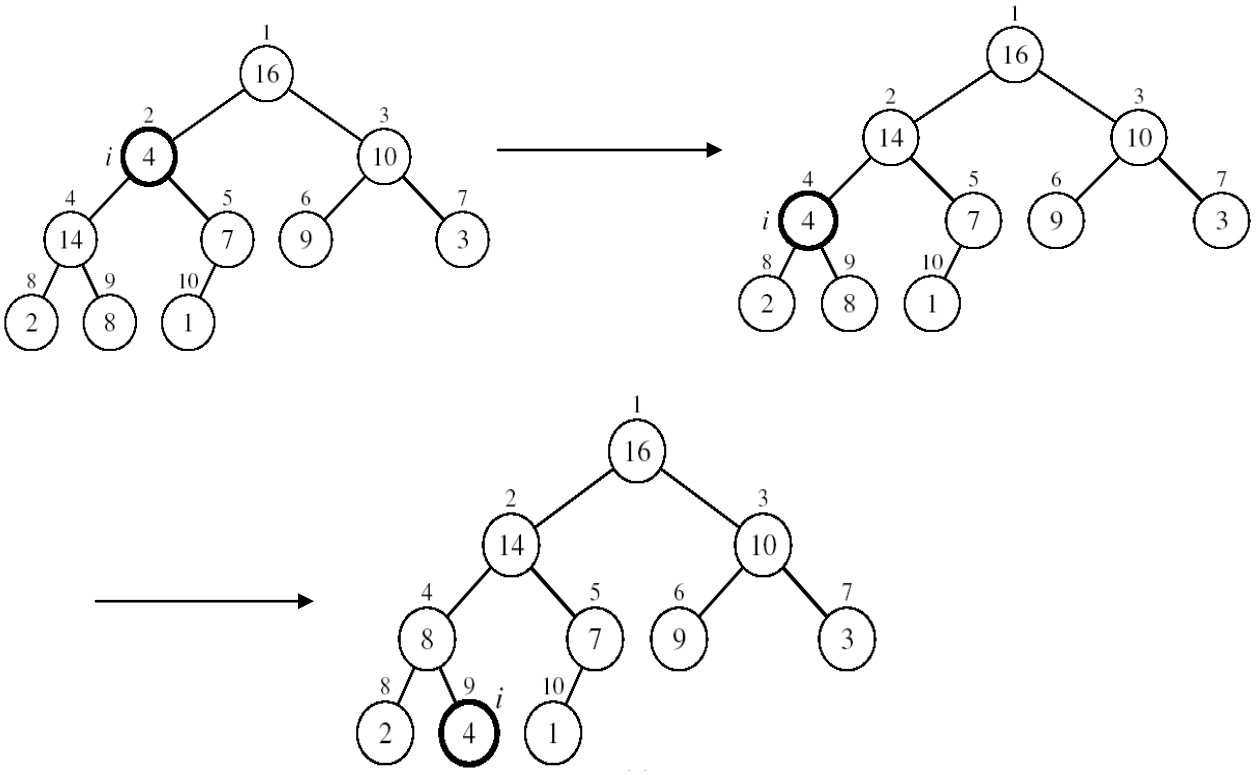
- Maintain/Restore the max-heap property
  - MAX-HEAPIFY
- Create a max-heap from an unordered array
  - BUILD-MAX-HEAP
- Sort an array in place
  - HEAPSORT
- Priority queues

Maintaining the Heap Property

Suppose a node is smaller than a child and Left and Right subtrees of i are max-heaps. To eliminate the violation:

- Exchange with larger child

- Move down the tree
- Continue until node is not smaller than children



Algorithm:

Max-Heapify(A, i, n)

{

l = Left(i)

r = Right(i)

largest=i;

if l ≤ n and A[l] > A[largest]

largest = l

if r ≤ n and A[r] > A[largest]

largest = r

if largest ≠ i

exchange (A[i] , A[largest])

Max-Heapify(A, largest, n)

}

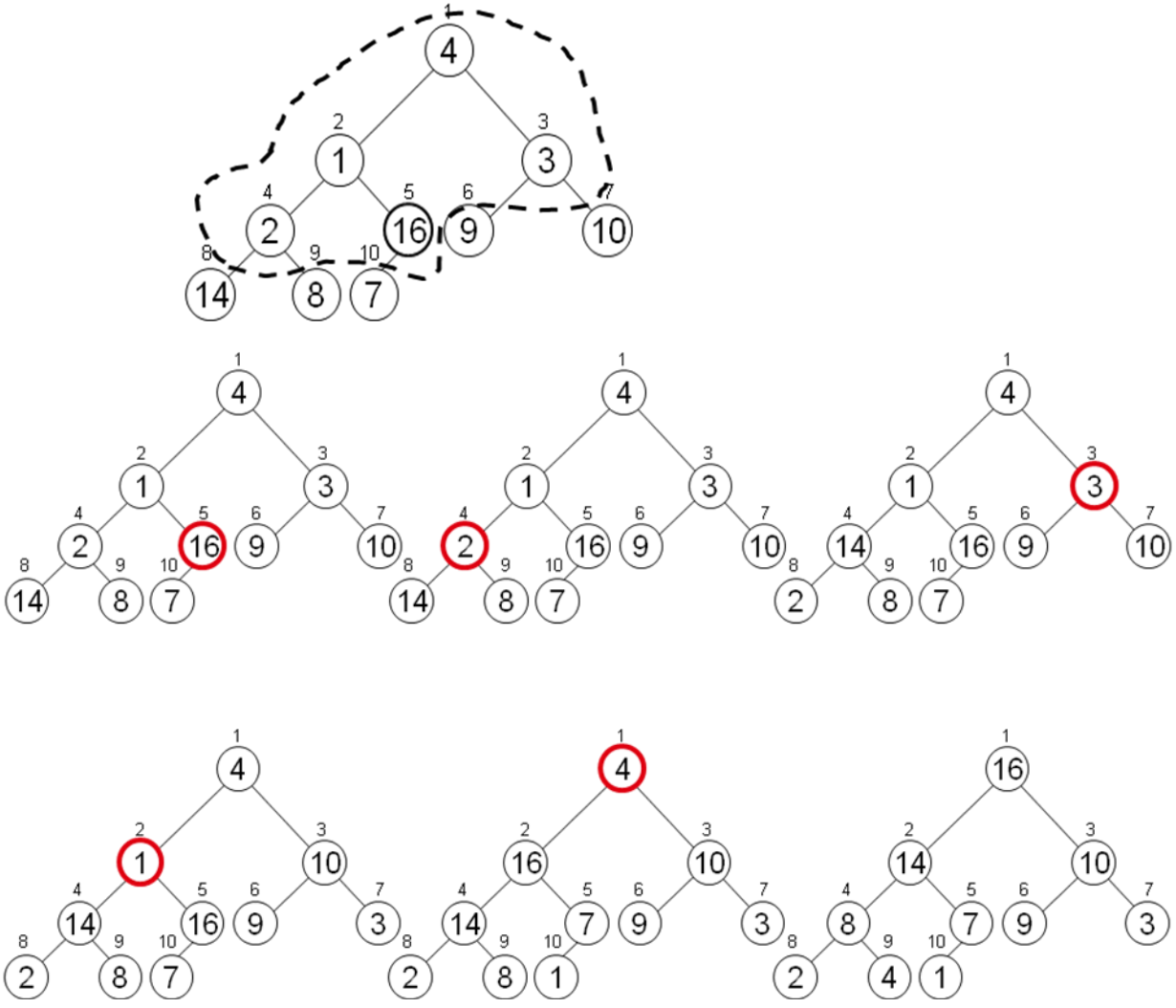
Analysis:

In the worst case Max-Heapify is called recursively h times, where h is height of the heap and since each call to the heapify takes constant time

Time complexity = O(h) = O(logn)

Building a Heap

Convert an array A[1 … n] into a max-heap (n = length[A]). The elements in the subarray A[(⌊n/2⌋+1) .. n] are leaves. Apply MAX-HEAPIFY on elements between 1 and ⌊n/2⌋.



Algorithm:

Build-Max-Heap(A)

n = length[A]

for i ← ⌊n/2⌋ **downto** 1

do MAX-HEAPIFY(A, i, n)

Time Complexity:

Running time: Loop executes O(n) times and complexity of Heapify is O(lgn), therefore complexity of Build-Max-Heap is O(nlogn).

This is not an asymptotically tight upper bound

Heapify takes O(h)

⇒ The cost of Heapify on a node i is proportional to the height of the node i in the tree

$\Rightarrow T(n) = \sum_{i=0}^{h} n_i h_i$

$h_i = h - i$ height of the heap rooted at level i

$n_i = 2^i$ number of nodes at level i

$\Rightarrow T(n) = \sum_{i=0}^{h} 2^i (h-i)$

$\Rightarrow T(n) = \sum_{i=0}^{h} 2^h (h-i) / 2^{h-i}$

Let k= h-i

$\Rightarrow T(n) = 2^h \sum_{i=0}^{h} k / 2^k$

$\Rightarrow T(n) \leq n \sum_{i=0}^{\infty} k / 2^k$

We know that, $\sum_{i=0}^{\infty} x^k = 1/(1-x)$ for $x<1$

Differentiating both sides we get,

$\sum_{i=0}^{\infty} k\, x^{k-1} = 1/(1-x)^2$

$$\sum_{i=0}^{\infty} k\,x^k = x/(1-x)^2$$

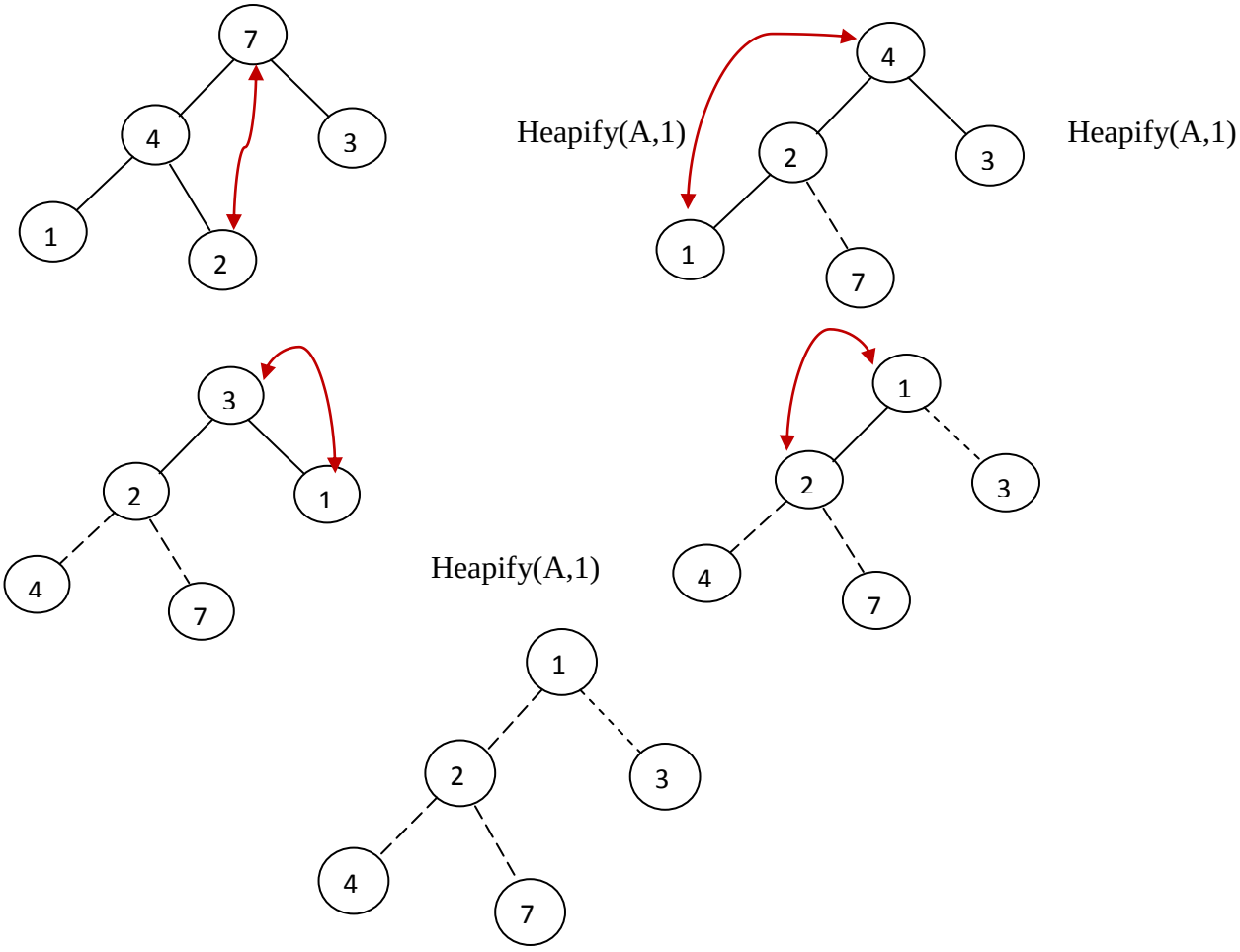
Put x=1/2

$$\sum_{i=0}^{\infty} k/2^k = 1/(1-x)^2 = 2$$

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

Heapsort

- Build a max-heap from the array
- Swap the root (the maximum element) with the last element in the array
- "Discard" this last node by decreasing the heap size
- Call Max-Heapify on the new root
- Repeat this process until only one node remains



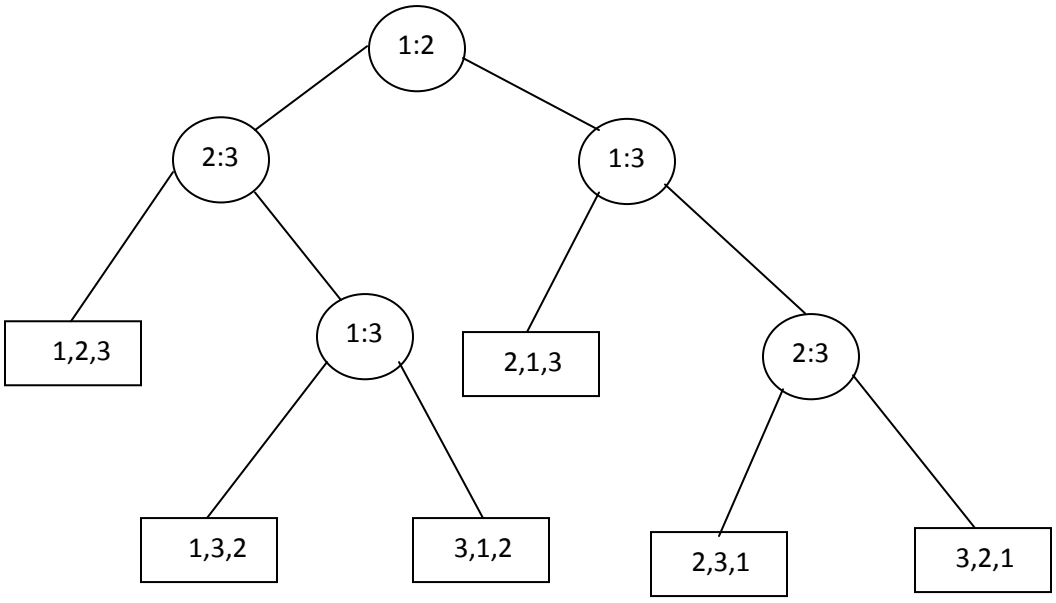
Algorithm:

```
HeapSort(A)
{
        BuildHeap(A); //into max heap
        n = length[A];
        for(i = n ; i >= 2; i--)
        {
                swap(A[1],A[n]);
                n = n-1;
                Heapify(A,1);
        }
}
```

Lower Bound for Sorting

All the sorting algorithms we have seen so far are ***comparison sorts***: only use comparisons to determine the relative order of elements. The best worst-case running time that we've seen for comparison sorting is $O(n\lg n)$. *E.g*., insertion sort, merge sort, quicksort, heapsort.

Sort $\langle a1, a2, \ldots, an\rangle$



Each internal node is labeled i:j for i, j ∈{1, 2,…, n}. The left subtree shows subsequent comparisons if ai≤aj. The right subtree shows subsequent comparisons if ai≥aj. A decision tree

can model the execution of any comparison sort: The tree contains the comparisons along all possible instruction traces.The running time of the algorithm is the length of the path taken.Worst-case running time is height of tree.

The tree must contain ≥n! leaves, since there are n! possible permutations. A height-h binary tree has $\leq 2^h$ leaves.

Thus, n! $\leq 2^h$

$\therefore h \geq \lg(n!)$

Since, lg(n!) = Ω(nlg n)

⇨ h= Ω(nlg n).

Searching

Sequential Search

Simply search for the given element left to right and return the index of the element, if found. Otherwise return "Not Found".

Algorithm:

```
LinearSearch(A, n,key)
{
        for(i=0;i<n;i++)
        {
                if(A[i] == key)
                {
                        return I;
                }
        }

        return -1;//-1 indicates unsuccessful search
}
```

Analysis:
Time complexity = O(n)

Binary Search:

To find a key in a large _le containing keys z[0; 1; : : : ; n-1] in sorted order, we first compare key with z[n/2], and depending on the result we recurse either on the first half of the file, z[0; : : : ; n/2 - 1], or on the second half, z[n/2; : : : ; n-1].

Algorithm

```
BinarySearch(A,l,r, key)
{
        if(l= = r)
        {
                if(key = = A[l])
                return l+1; //index starts from 0
                else
                return 0;
        }

        else
        {
                m = (l + r) /2 ; //integer division
                if(key = = A[m]
                return m+1;
                else if (key < A[m])
                return BinarySearch(l, m-1, key) ;
                else return BinarySearch(m+1, r, key) ;
        }
}
```

Analysis:

From the above algorithm we can say that the running time of the algorithm is:

$$T(n) = T(n/2) + Q(1)$$
$$= O(\log n) .$$

In the best case output is obtained at one run i.e. O(1) time if the key is at middle. In the worst case the output is at the end of the array so running time is O(logn) time. In the average case also running time is O(logn). For unsuccessful search best, worst and average time complexity is O(logn).

Max and Min Finding

Here our problem is to find the minimum and maximum items in a set of n elements. We will see two methods here first one is iterative version and the next one uses divide and conquer strategy to solve the problem.

Iterative Algorithm:

```
MinMax(A,n)
{
max = min = A[0];
for(i = 1; i < n; i++)
{
if(A[i] > max)
max = A[i];
if(A[i] < min)
min = A[i];
}
}
```

The above algorithm requires 2(n-1) comparison in worst, best, and average cases. The comparison A[i] < min is needed only when A[i] > max is not true. If we replace the content inside the for loop by

```
if(A[i] > max)
max = A[i];
else if(A[i] < min)
min = A[i];
```

Then the best case occurs when the elements are in increasing order with (n-1) comparisons and worst case occurs when elements are in decreasing order with 2(n-1) comparisons. For the average case A[i] > max is about half of the time so number of comparisons is 3n/2 – 1.
We can clearly conclude that the time complexity is O(n).

Divide and Conquer Algorithm:

Main idea behind the algorithm is: if the number of elements is 1 or 2 then max and min are obtained trivially. Otherwise split problem into approximately equal part and solved recursively.

```
MinMax(l,r)
{
if(l = = r)
max = min = A[l];
else if(l = r-1)
{
if(A[l] < A[r])
{
max = A[r]; min = A[l];
}
else
{
max = A[l]; min = A[r];
}
}
else
{
//Divide the problems
mid = (l + r)/2; //integer division
//solve the subproblems
{min,max}=MinMax(l,mid);
{min1,max1}= MinMax(mid +1,r);
//Combine the solutions
if(max1 > max) max = max1;
if(min1 < min) min = min1;
}
}
```

Analysis:

We can give recurrence relation as below for MinMax algorithm in terms of number of comparisons.

$T(n) = 2T(n / 2) + 1$, if $n>2$

$T(n) = 1$, if $n \leq 2$

Solving the recurrence by using master method complexity is (case 1) O(n).

Selection

i^{th} order statistic of a set of elements gives i^{th} largest(smallest) element. In general lets think of i^{th} order statistic gives i^{th} smallest. Then minimum is first order statistic and the maximum is last order statistic. Similarly a median is given by i^{th} order statistic where i = (n+1)/2 for odd n and i = n/2 and n/2 + 1 for even n. This kind of problem commonly called selection problem.

This problem can be solved in Θ(nlogn) in a very straightforward way. First sort the elements in Θ(nlogn) time and then pick up the i^{th} item from the array in constant time. What about the linear time algorithm for this problem? The next is answer to this.

Nonlinear general selection algorithm

We can construct a simple, but inefficient general algorithm for finding the *k*th smallest or *k*th largest item in a list, requiring O(*kn*) time, which is effective when *k* is small. To accomplish this, we simply find the most extreme value and move it to the beginning until we reach our desired index.

```
Select(A, k)
{
        for( i=0;i<k;i++)
        {
            minindex = i;
            minvalue = A[i];
           for(j=i+1;j<n;j++)
           {
              if( A[j] < minvalue)
```

```
                {
                        minindex = j;
                        minvalue = A[j];
                }
                swap(A[i],A[minIndex]);
            }
            return A[k];
    }
```

Analysis:

When i=0, inner loop executes n-1 times

When i=1, inner loop executes n-2 times

When i=2, inner loop executes n-3 times

……...

When i=k-1 inner loop executes n-(k+1) times

Thus, Time Complexity = (n-1) + (n-2) + ……..(n-k-1)

$= O(kn) \approx O(n^2)$

Selection in expected linear time

This problem is solved by using the "divide and conquer" method. The main idea for this problem solving is to partition the element set as in Quick Sort where partition is randomized one.

Algorithm:

```
    RandSelect(A,l,r,i)
    {
        if(l = =r )
            return A[p];
        p = RandPartition(A,l,r);
        k = p – l + 1;
        if(i <= k)
            return RandSelect(A,l,p-1,i);
        else
```

```
            return RandSelect(A,p+1,r,i - k);
    }
```

Analysis:

Since our algorithm is randomized algorithm no particular input is responsible for worst case however the worst case running time of this algorithm is $O(n^2)$. This happens if every time unfortunately the pivot chosen is always the largest one (if we are finding minimum element). Assume that the probability of selecting pivot is equal to all the elements i.e 1/n then we have the recurrence relation,

$$T(n) = 1/n\left(\sum_{j=1}^{n-1} T(\max(j,n-j))\right) + O(n)$$

Where, max(j,n-j) = j, if j>= ceil(n / 2)

and max(j,n-j) = n-j, otherwise.

Observe that every T(j) or T(n – j) will repeat twice for both odd and even value of n (one may not be repeated) one time form 1 to ceil(n / 2) and second time for ceil(n / 2) to n-1, so we can write,

$$T(n) = 2/n\left(\sum_{j=n/2}^{n-1} T(j)\right) + O(n)$$

Using substitution method,

Guess T(n) = O(n)

To show T(n) <= cn

Assume T(j) <= cj

Substituting on the relation

$$T(n) = 2/n \sum_{j=n/2}^{n-1} cj + O(n)$$

$$T(n) = 2/n \left\{\sum_{j=1}^{n-1} cj - \sum_{j=1}^{n/2-1} cj\right\} + O(n)$$

$T(n) = 2/n \{ (n(n-1))/2 –((n/2-1)n/2)/2\} + O(n)$

$T(n) \leq c(n-1) – c(n/2-1)/2 + O(n)$

$T(n) \leq cn – c – cn/4 + c/2 + O(n)$

$= cn – cn/4 –c/2 + O(n)$

$\leq cn$ {choose the value of c such that $(-cn/4-c/2 + O(n) \leq 0$ }

⇨ T(n) = O(n)

Selection in worst case linear time

Divide the n elements into groups of 5. Find the median of each 5-element group. Recursively SELECT the median x of the $\lfloor n/5 \rfloor$ group medians to be the pivot.

Algorithm

Divide *n* elements into groups of 5

Find median of each group

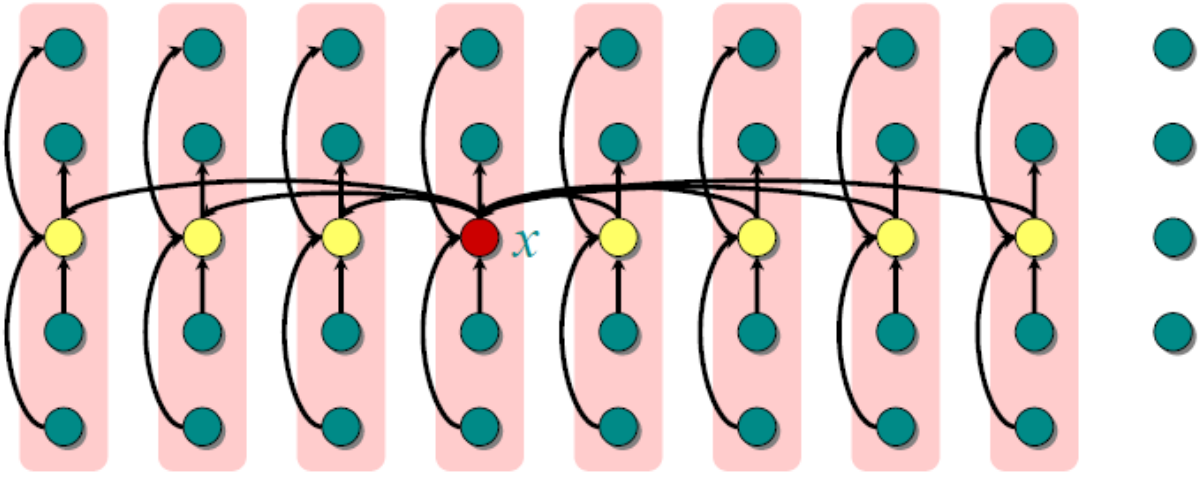
Use Select() recursively to find median *x* of the $\lfloor n/5 \rfloor$ medians

Partition the *n* elements around *x*. Let *k* = rank(*x*) //index of x

if (i == k) **then** return x

if (i < k) **then** use Select() recursively to find *i*th smallest element in first partition

else (i > k) use Select() recursively to find (*i*-*k*)th smallest element in last partition



At least half the group medians are ≤x, which is at least $\lfloor \lfloor n/5 \rfloor /2 \rfloor = \lfloor n/10 \rfloor$ group medians.

Therefore, at least $3\lfloor n/10 \rfloor$ elements are ≤x.

Similarly, at least $3\lfloor n/10 \rfloor$ elements are ≥*x*

For $n \geq 50$, we have $3\lfloor n/10 \rfloor \geq n/4$.

Therefore, for $n \geq 50$the recursive call to SELECT in Step 4 is executed recursively on $\leq 3n/4$elements in worst case.

Thus, the recurrence for running time can assume that Step 4 takes time $T(3n/4)$in the worst case.

Now, We can write recurrence relation for above algorithm as"

$T(n) = T(n/5) + T(3n/4) + \Theta(n)$

Guess $T(n) = O(n)$

To Show $T(n) \leq cn$

Assume that our guess is true for all $k<n$

Now,

$T(n) \leq cn/5 + 3cn/4 + O(n)$

$= 19cn/20 + O(n)$

$= cn - cn/20 + O(n)$

$\leq cn$ { Choose value of c such that $cn/20 - O(n) \leq 0$}

⇨ $T(n) = O(n)$