

Unit 3: Divide and Conquer Algorithms

3.0. Introduction to Divide-and-conquer paradigm

3.1. Searching Algorithms:

- Binary Search
- Min-Max Finding

3.2. Sorting Algorithms:

- Merge Sort
- Quick Sort and Analysis (Best Case, Worst Case and Average Case), Randomized quick sort.
- Heap Sort (Heapify, Build Heap and Heap Sort Algorithms and their Analysis)

3.3. Order Statistics (selection problem):

- Selection in Expected Linear Time, Selection in Worst Case Linear Time and their Analysis

What is divide-and-conquer strategy?

- Since the ancient times, countries often used the concept of dividing the enemy unity to conquer them.
- While designing any algorithm, the same strategy can be used.
- Suppose we have a large problem to solve which is difficult to solve. Then, we can divide the problem into smaller problems which are easier to solve. This solution technique is called “Divide and Conquer”.
- The divide-and-conquer paradigm involves three steps:
 - i) **Divide**: *Divide the problem into subproblems of smaller size.*
 - ii) **Conquer**: *Solve the subproblems recursively*
 - iii) **Combine**: *Combine the solutions to the subproblems to obtain the solution to the original problem.*

Analyzing DAC algorithms

- In DAC, we solve a problem recursively, applying the previously mentioned three steps at each level of recursion.
- When the problem is large enough to solve recursively, we call that the *recursive case*. Once the subproblems become small enough to solve directly without recursion, we call it *base case*.
- We already know that recurrence relations are best way to represent recursive algorithms.
- Recurrences (recurrence relations) go hand in hand with the DAC paradigm, because they give us a natural way to characterize the running time of DAC algorithms.

Binary search

- Binary search is a DAC algorithm to solve search problem with a sorted array of elements.

Input: An sorted array A of n elements and a key K .

Output: Position of K in A (if present)

Basic idea

To solve above problem, we will use the DAC strategy.

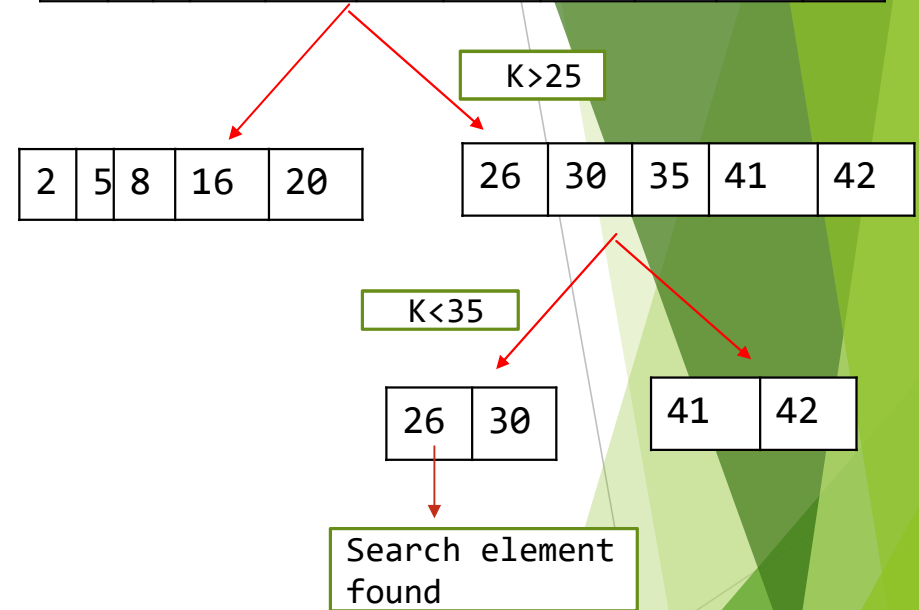
- If there is only one element, then compare with k and return.
- Otherwise divide the array into two halves (subarrays) and compare K with element in the middle (say x).
 - If $x = K$ then return. we are done.
 - Otherwise if $x > K$ then continue search on left subarray. if not search on right subarray.

Example:

$K = 26$

$A =$

2	5	8	16	20	25	26	30	35	41	42
---	---	---	----	----	----	----	----	----	----	----



Algorithm:

```
bin_search(A, f_indx, l_indx, Key)
{
    if(f_indx == l_indx)
    {
        if(Key == A[f_indx]) //one element in
            array
            return f_indx;
        else
            return "error"    //element not found
    }
    else
    {
        mid =  $\left\lfloor \frac{f\_indx + l\_indx}{2} \right\rfloor$  //floor value
        if((Key==A[mid]
            return mid
        else
        {
            if(Key<A[mid]
                return bin_search(A,f_indx,mid-1,Key)
            else
                return bin_search(A,mid+1,l_indx,Key)
        }
    }
}
```

5

Analysis:

- the Algorithm recursively divides the array of size n into half until possibly there is only one element.
- the running time can be represented by the recurrence:

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

cost of dividing (pointing to $T(n/2)$)
cost of combine (pointing to $O(1)$)

Solving by Master method,

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

Thus binary search is very efficient.

Best case complexity: $O(1)$ (how ??)

However, binary search requires the array to be presorted. If we count the cost of sorting then the running time will be higher.

Min-max finding problem

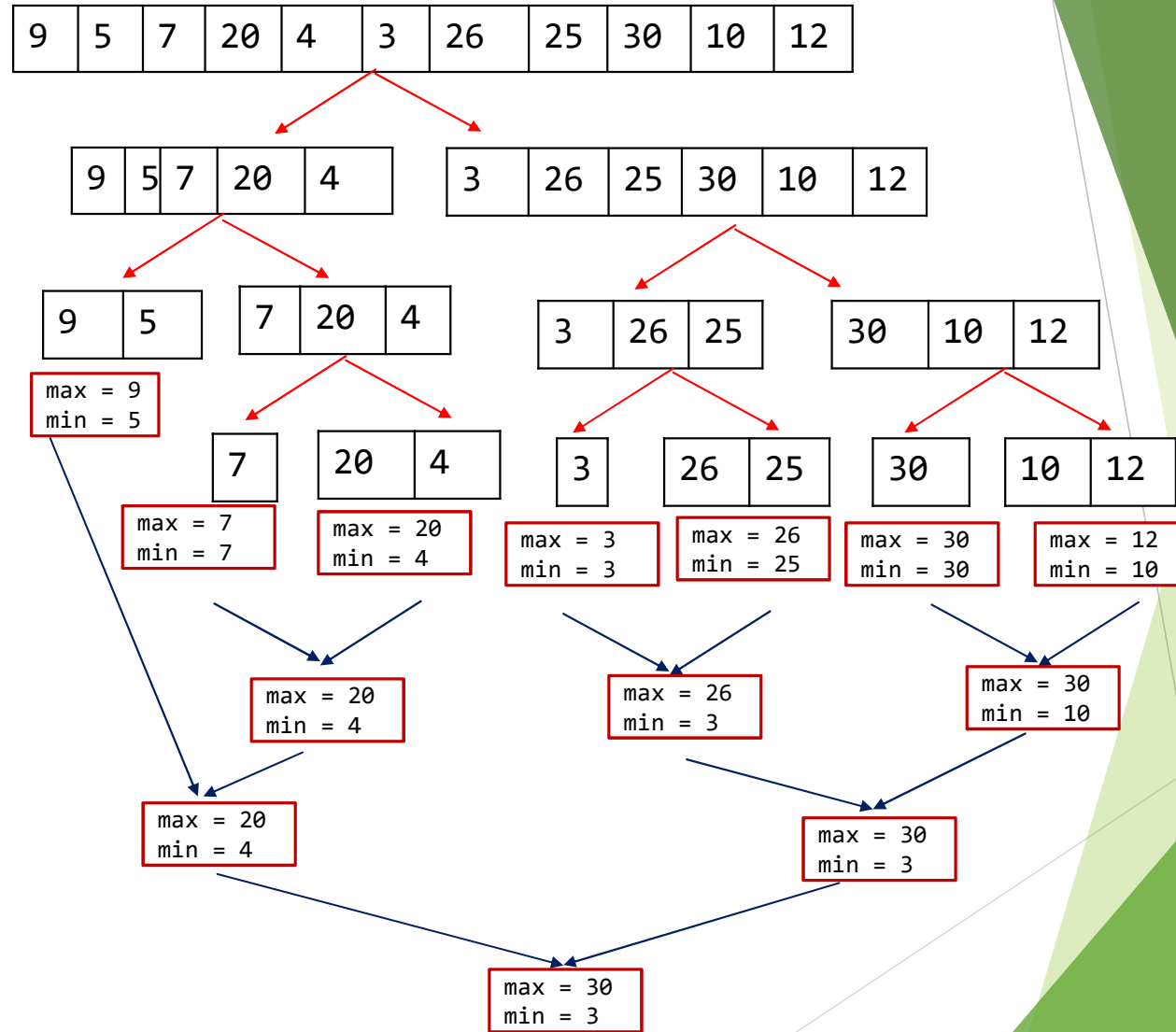
- Here our problem is to find the minimum-value and maximum-value items in an array A of n elements.
- We can solve the problem iteratively as follows:

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

Complexity ??


Now let us try to solve the problem with DAC approach.

- Idea:
 - if the number of elements is 1 the element itself is max and min
 - Else if the number of elements is 2 then max and min are obtained by one comparison.
 - Otherwise split the array into equal parts and solve recursively.



pseudocode

These are called
output arguments.



```
DAC-MinMax(A, f_indx, l_indx, max, min)
{
    if(f_indx == l_indx)        //only one element
        max = min = A[i]
    else if(f_indx == l_indx-1) //two elements
    {
        if(A[f_indx] < A[l_indx])
        {
            max = A[l_indx]
            min = A[f_indx]
        }
    }
    else
    {
        max = A[f_indx]
        min = A[l_indx]
    }
}
else                            //more than two elements
{
```

```
//Divide the problem
mid = (f_indx + l_indx)/2 //integer division
//solve the subproblems
DAC-MinMax(A, f_indx, mid, max, min)
DAC-MinMax(A, mid + 1, l_indx, max1, min1)
//Combine the solutions
if(max1 > max)
    max = max1
if(min1 < min)
    min = min1
}
}
```


Analysis:

- The input array is divided into two halves and each is solved recursively. The running time can be represented by the recurrence:

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

cost of dividing

cost of combine

- Solving by master method we get,

$$T(n) = \Theta(n)$$



- This complexity is same as in case of iterative approach.

Which approach is better to use - Iterative or DAC ??

- If both iterative and DAC solutions have same complexity then we must use iterative solution because DAC approach has recursion overhead (which requires more memory space to maintain stack. It also requires extra stack operations).

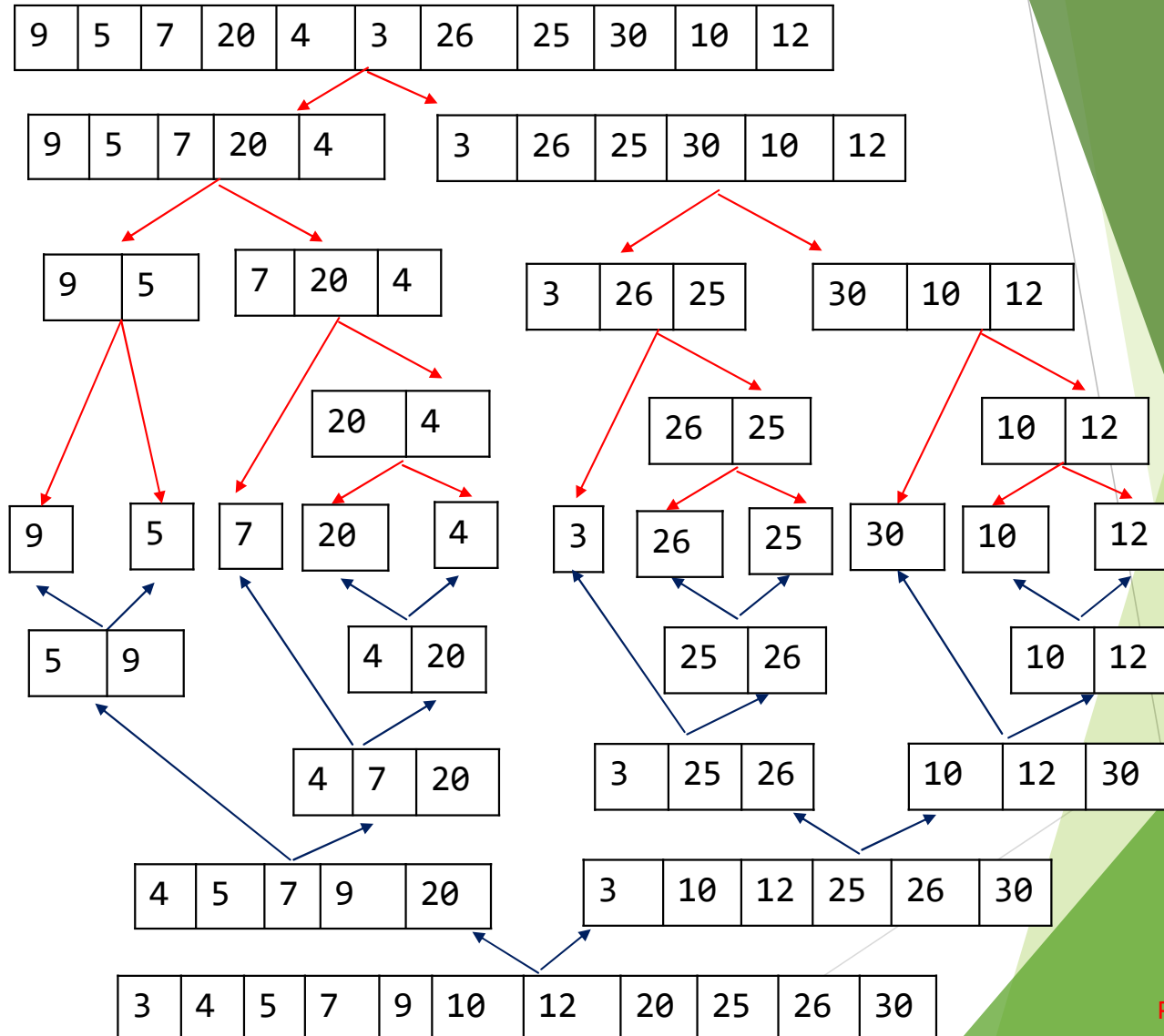
Merge sort

- Merge sort is an efficient sorting algorithm that uses DAC approach to sort an array of elements.
- The basic idea behind merge sort is:
 - i) If the input array contains single element then it is already sorted itself.
 - ii) If the input array contains more than one element then
 - *Divide* the array into equal halves (subarrays) recursively until each subarray contains one element.
 - The one-element subarrays are themselves sorted.
 - *Merge* the sorted subarrays

Divide

Conquer

Merge



Merging two sorted arrays

- Merging is an important step in merge sort.
- Merging is the process of combining two sorted arrays into single sorted array.

Idea:

Let we have two arrays which are already sorted.

$A = \{a_1, a_2, \dots, a_m\}$ and

$B = \{b_1, b_2, \dots, b_n\}$

Then we can merge them as follows:

- Let i be the pointer to elements in A
- Let j be the pointer to elements in B
- i and j initially point to first elements
- Compare a_i with b_j and store the smaller element in a temporary array T .
- Move the pointer of smaller element.
- Repeat step iv and v until one or both of i, j exceed the array limit.

- vii) If any of array has remaining elements, copy the elements to temporary array T .

Example:



Continue the process.

Merge sort algorithm

```
Mergesort(A, first, last)
{
    if(first==last)
        return;
    mid = (first+last)/2
    mergesort(A, first, mid)
    mergesort(A, mid+1, last)
    simplemerge(A, first, mid+1, last)
}
simplemerge(A, first, second, last)
{
    i=first
    j=second
    l=0
```

```
//compare corresponding
elements and store the
smaller.
```

```
while(i<second and
j<=last)
{
    if(A[i]<A[j])
    {
        temp[l] = a[i]
        l=l+1
        i=i+1
    }
    else
    {
        temp[l] = a[j]
        l=l+1
        j=j+1
    }
}
```

```
//copy the remaining
elements (if any)
```

```
while(j<=last)
{
    temp[l]=A[j]
    l=l+1
    j=j+1
}
while(i<second)
{
    l=l+1
    temp[l] = A[i]
    i=i+1
}
//Copy the temp array
into original array A
l=0
for(i=first;i<=last;i++)
{
    A[i]=temp[l]
    l++
}
```

Analysis

- The mergesort algorithm divides input array into two nearly equal halves and solves them recursively. Finally, the sorted subarrays are merged together.
- The running time can be represented by the recurrence:

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) \leftarrow \text{cost of merging: During the merging process, the while loops may run } n \text{ times.}$$
$$= \Theta(n \log n)$$

- Thus merge sort is efficient than bubble, selection and insertion sort.
- However, the space complexity is higher because of the use of temporary array during merging.

Did you know ?

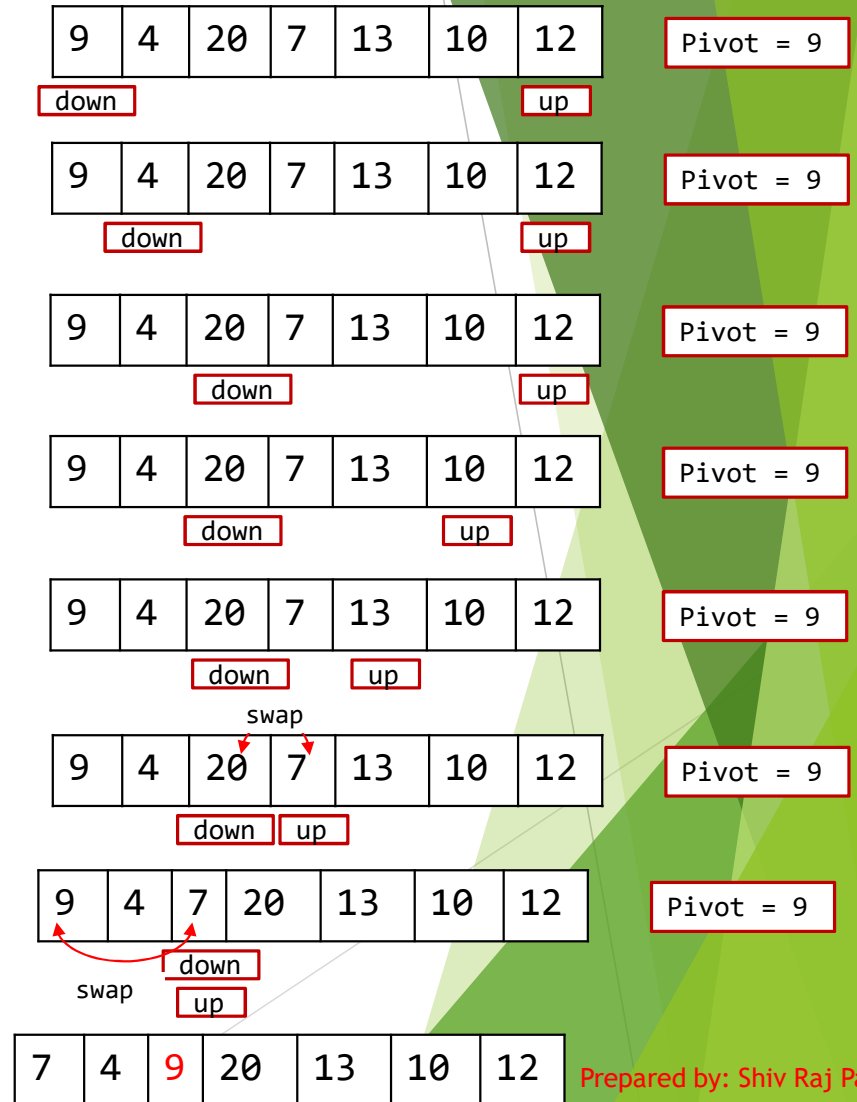
Insertion sort is a special case of Merge sort where, we merge a single element with a sorted subarray in each iteration.

- Useful resource: <https://levelup.gitconnected.com/sorting-algorithms-selection-sort-bubble-sort-merge-sort-and-quicksort-75479f8f80b1>

Quick sort

- Quick sort is an efficient sorting algorithm that uses DAC approach to sort an array A of n elements.
- The basic idea behind Quick sort is:
 - i) If the input array contains single element then it is already sorted itself.
 - ii) If the input array contains more than one element then
 - *Pick an element v in A. v is called pivot.*
 - *Divide* the array $A - \{v\}$ into two subarrays recursively.
$$A_1 = \{x \mid x \leq v\}$$
$$A_2 = \{x \mid x > v\}$$
 - Combine the subarrays
- The divide step is tricky as we need to compare the elements with pivot to put them on correct side (left or right) of pivot.

Example: A single iteration of dividing.



Dividing (partitioning) strategy

1. Initialize pointers “down” and “up” to point to first and last element of array
2. initialize pivot as first element of array (*More on this later*)
3. repeatedly move pointer “down” until element larger than pivot is encountered.
4. repeatedly move pointer “up” until element smaller than pivot is found.
5. if $down < up$ then swap the elements.
6. repeat step 3 to step 5 until $down > up$ (i.e. down and up cross each other)
7. Now the “up” is the position for pivot. swap “first” element with element at “up” position

Pseudocode

```
quicksort(A,first,last)
{
    if(first==last)
        return;
    r = partition(A,first,last)
    if(r>first)
        quicksort(A,first,r-1)
    if(r<last)
        quicksort(A,r+1,last)
}
```

```
partition(A,first,last)
{
    down = first
    up = last
    pivot = A[first]
    while(down<=up)
    {
        while(A[down]<=pivot and down<=up)
            down++
        while(A[up]>pivot and down<=up)
            up++
        if(down<up)
            swap(A[down],A[up])
    }
    swap(A[up],A[first])
    return up
}
```

Analysis:

- Quicksort divides the input array into subarrays and solves them recursively.
- The size of the two subarrays depends on the position of the pivot during partitioning.
- the partitioning step may take $O(n)$ time as the pointers “down” and “up” may move n times in total doing n comparisons.
- Suppose, the pivot lies at r^{th} position. Then, the time complexity can be represented as

$$T(n) = T(r) + T(n-r) + \Theta(n)$$

cost of
partitioning
(finding pivot
position)

1. **Best case:** Best case occurs when the pivot lies exactly in the middle and the array is divided into two nearly equal halves in each recursion.

- so the running time can be expressed as

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + \Theta(n) \\ &= \Theta(n \log n) \end{aligned}$$

2. Worst case

- Worst case occurs when the pivot lies at the end point (first or last) of the array during each recursion.
- In this case the recurrence becomes,

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

3. **Average case:** Consider an average case scenario where the array is divided into $1/10$ and $9/10$ sized parts.

Then,

$$\begin{aligned} T(n) &= T\left(\frac{9n}{10}\right) + T\left(\frac{n}{10}\right) + \Theta(n) \\ &= ?? \end{aligned}$$

Picking the pivot

- The performance of the quicksort depends on where the pivot lies at each recursion.
- The selection of pivot is crucial!
- Poor pivot selection strategy may result in worst case complexity.
- Following are some strategies for picking the pivot for partitioning in quicksort.

1. *Picking first element as pivot*
2. *Median-of-three strategy*
3. *Random pivot*

1. *Picking first element as pivot*

- Poor strategy.
- May result in worst case complexity ($O(n^2)$) if array (or large part of array) is already reverse-sorted.

2. *Median-of-three strategy*

- To select a pivot in each subarray do the following:
 - i. select first, middle and last elements
 - ii. Take median of the three elements.

Example:

9	4	20	7	13	10	12
---	---	----	---	----	----	----

Pivot = median(9, 7, 12) = 9

- Good strategy. Never gives worst case.

3. *Random pivot*

- Take random element as pivot.
 - i. Generate a random index between range of current array indices.
 - ii. Use the randomly selected element as pivot.
- Good strategy to avoid worst case but overhead in generating random number.

Difference between quick and merge sort

Merge sort

1. Employs DAC strategy.
2. Input always divided into equal halves.
3. Dividing step is straightforward.
Combine(Merge) step is tricky.
4. All case time complexity: $O(n \log n)$
5. Space complexity is higher than quicksort (use of temporary array during merging)
6. Insertion sort is special case of merge sort where we merge one element with another sorted array.

Quick sort

1. Employs DAC strategy.
2. Input may not always divided into equal halves. The size of subproblem depends on position of pivot.
3. Dividing step is tricky. Combine step is straightforward.
4. Best case complexity: $O(n \log n)$
Average case complexity: $O(n \log n)$
Worst case complexity : $O(n^2)$
5. Space complexity is less than merge sort.
6. Selection sort is special case of quick sort where the selected minimum element at each iteration is the pivot. Here we have no array on one side of pivot.

Randomized Quicksort

- when we select random pivot, it avoids worst case and almost always results in average case.
- This strategy gives a new version of quicksort called “randomized quicksort”.

Algorithm:

```
randquicksort(A,first,last)
{
    if(first==last)
        return;
    r = rand_partition(A,first,last)
    if(r>first)
        randquicksort(A,first,r-1)
    if(r<last)
        randquicksort(A,r+1,last)
}
```

```
rand_partition(A,p,q)
{
    i=random(p,q)
    swap(A[p],A[i])
    partition(A,p,q)
}

partition(A,f,l)
{
    //Same as ordinary quicksort
    .. ..
}
```

Analysis of randquicksort algorithm:

- The only difference between ordinary version and randomized version of quicksort is the choice of pivot.
- The random pivot strategy make choosing any element equally likely, regardless of the position of elements in the input array.
- So it can be said that, the randomized quicksort gives average case complexity.
- Let the array is divided into subarrays of size q and $n-q$.
- Consider the probability of choosing pivot among n elements is equally likely.

Then the RR is

$$T(n) = 1/n \sum_{q=1}^{n-1} (T(q) + T(n-q)) + \Theta(n).$$

Now for some $j = 1, 2, \dots, n-1$

$T(q) + T(n-q)$ is repeated two times so we can write the relation as,

LL

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

$$\text{Or, } nT(n) = 2 \sum_{j=1}^{n-1} T(j) + n^2. \text{ ---- (I) } \quad [\text{taking } \Theta(n) = n \text{ for simplicity}]$$

For $n = n-1$ we have,

$$(n-1)T(n-1) = 2 \sum_{j=1}^{n-2} T(j) + (n-1)^2. \text{ ---- (II)}$$

(I) – (II) gives,

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

$$\text{Or, } T(n)/(n+1) = T(n-1)/n + 2n - 1 / n(n+1).$$

Put $T(n)/(n+1)$ as A_n then we have above relation as

$$A_n = A_{n-1} + 2n - 1 / n(n+1).$$

The above relation is written as,

$$A_n = \sum_{i=1}^n 2i - 1 / i(i + 1). \quad [2i - 1 \approx 2i]$$

$$\approx 2 \sum_{i=1}^n 1 / (i + 1). \quad [\text{Harmonic series } H_n = \sum_{i=1}^n 1/i \approx \log n]$$

$$\approx 2 \log n$$

But we have $T(n) = (n+1) A_n \approx 2(n+1) \log n$.

So,

$$T(n) = \Theta(n \log n).$$

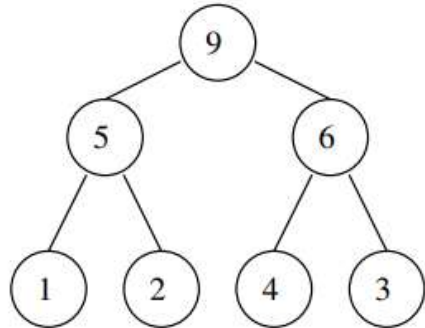
Prepared by: Shiv Raj Pant

Heap sort

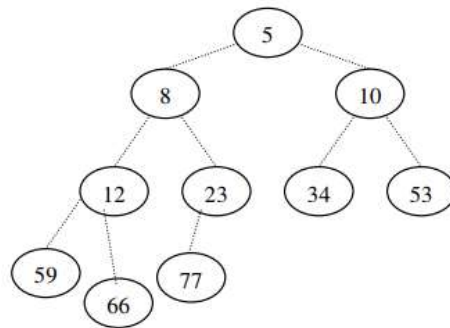
- Heap sort is an efficient sorting algorithm that makes use of a special data structure called *heap*.

What is heap ?

- Heap is a tree data structure where the value of every node is either greater (max heap) or less (min heap) than it's children.



A max heap



A min heap

Representation of heaps:

- We can represent heap in array.
- Example:

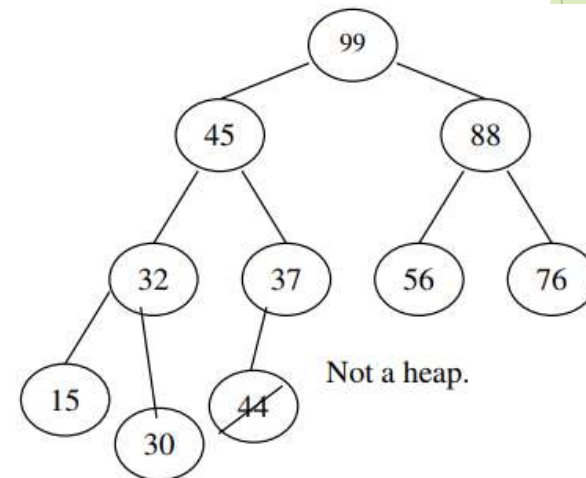
We represent the above max heap as an array like:

1	2	3	4	5	6	7
9	5	6	1	2	4	3

if parent p is at $A[i]$ place then its left child is at $A[2i]$ and the right child is at $A[2i + 1]$

- Example:

Is the sequence 99, 45, 88, 32, 37, 56, 76, 15, 30, 44 a max heap?



Not a heap.

Basic idea of heapsort

- Suppose we want to sort in ascending order
- Heapsort includes two phases

1. *Heap construction phase:*

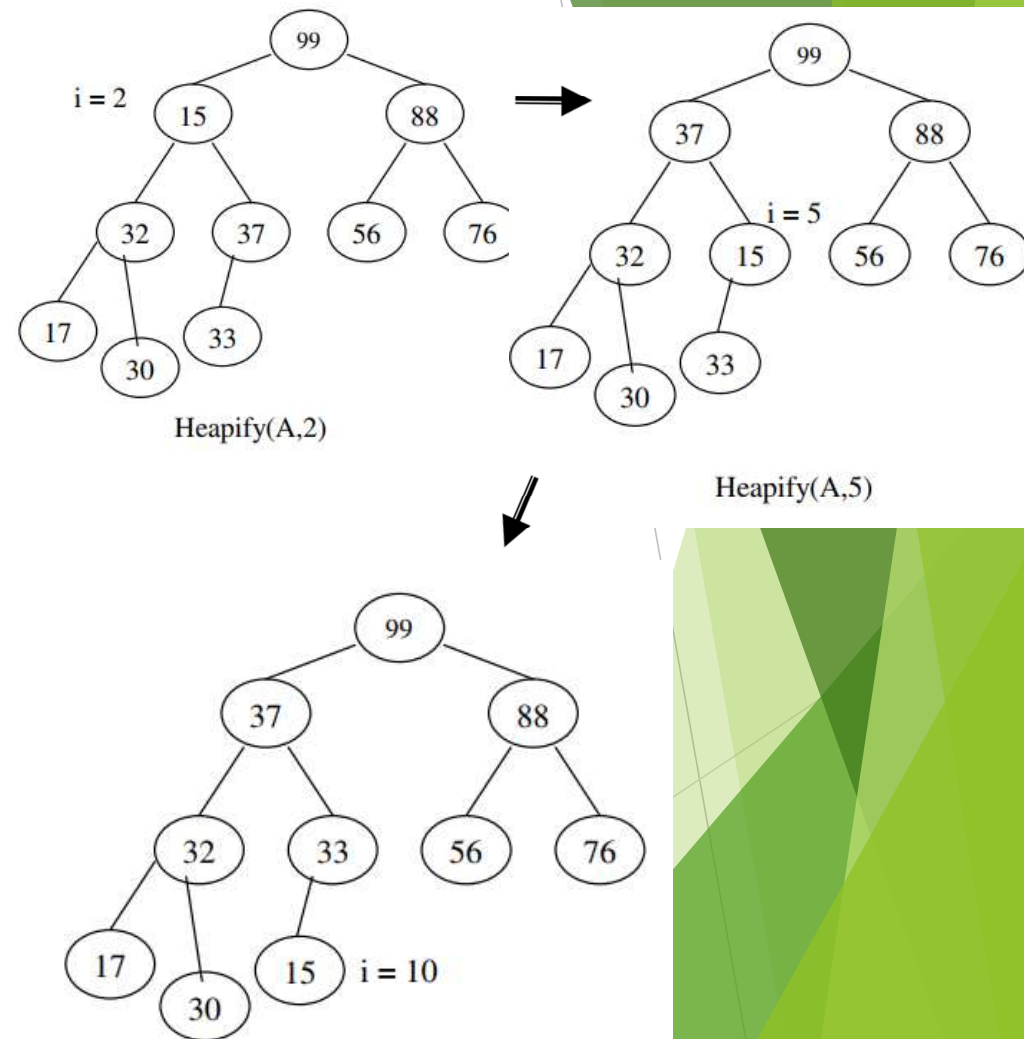
- Build a heap (max heap) by repeatedly heapifying given array

2. *Sort phase:*

- swap first and last elements
- ignore the last element and heapify the remaining heap
- repeat above two steps until no element remains to sort

- We can see that heapify is the major process in heapsort.
- So let us first look at an example of heapify.

The *heapify* process example:



Build-Heap algorithm

BuildHeap(A)

{

heapsize[A] = length[A]

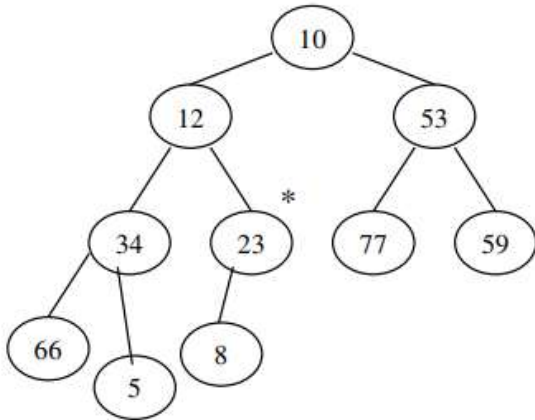
for i = $\lfloor \text{length}[A]/2 \rfloor$ to 1

do Heapify(A,i)

}

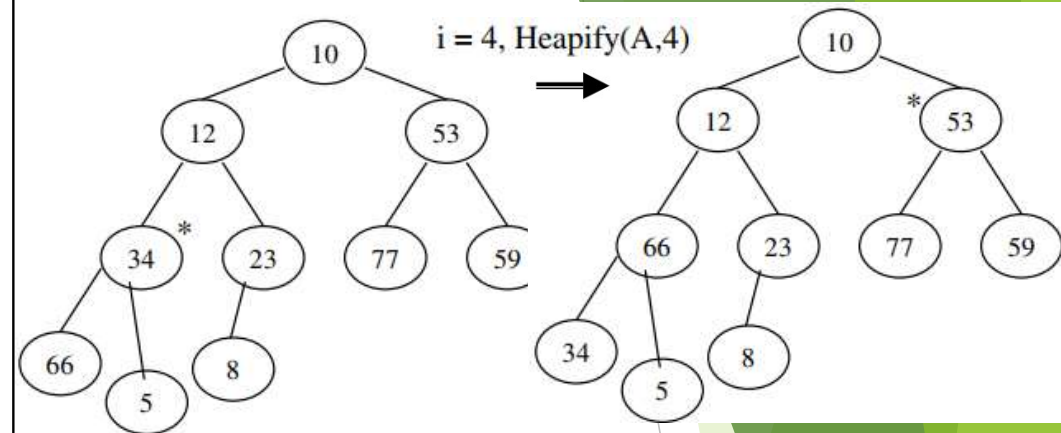
Example:

$A[] = \{10, 12, 53, 34, 23, 77, 59, 66, 5, 8\}$

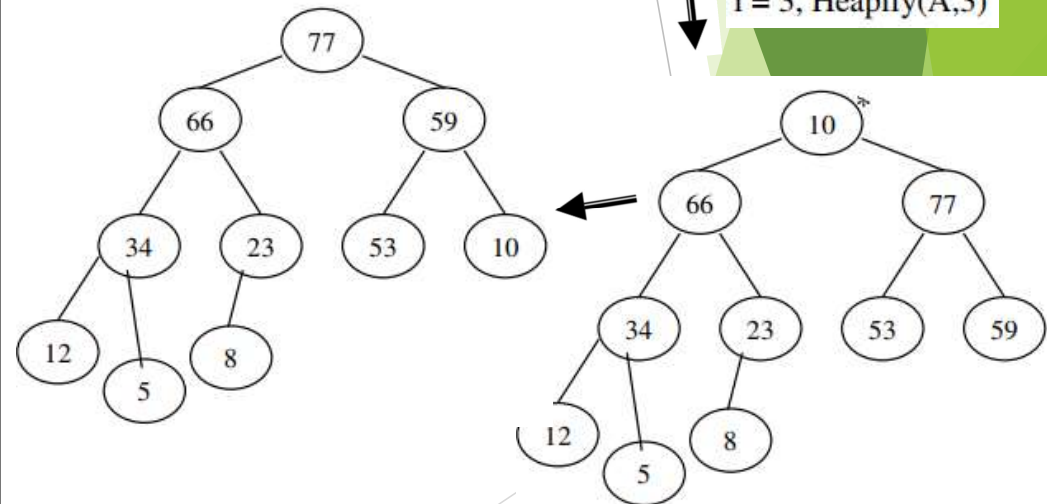


$i = 5, \text{Heapify}(A, 5)$ (no change)

25



$i = 3, \text{Heapify}(A, 3)$



Prepared by: Shiv Raj Pant

Heapify algorithm

```
Heapify(A,i)
{
    l = left(i)
    r = Right(i)
    if l <= heapsize[A] and A[l] > A[i]
        max = l
    else
        max = i
    if r <= heapsize[A] and A[r] > A[max]
        max = r
    if max != i
    {
        swap(A[i],A[max])
        Heapify(A,max)
    }
}
```

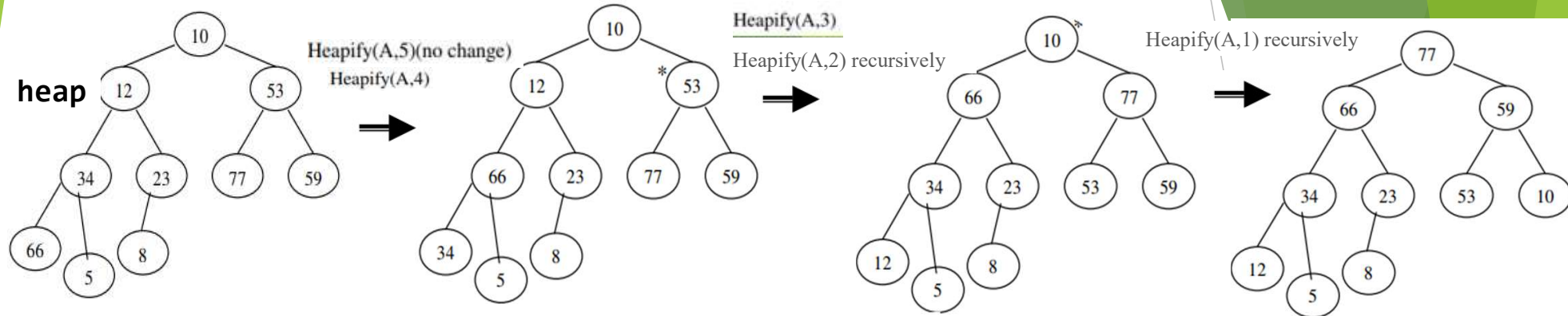
Heapsort algorithm

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

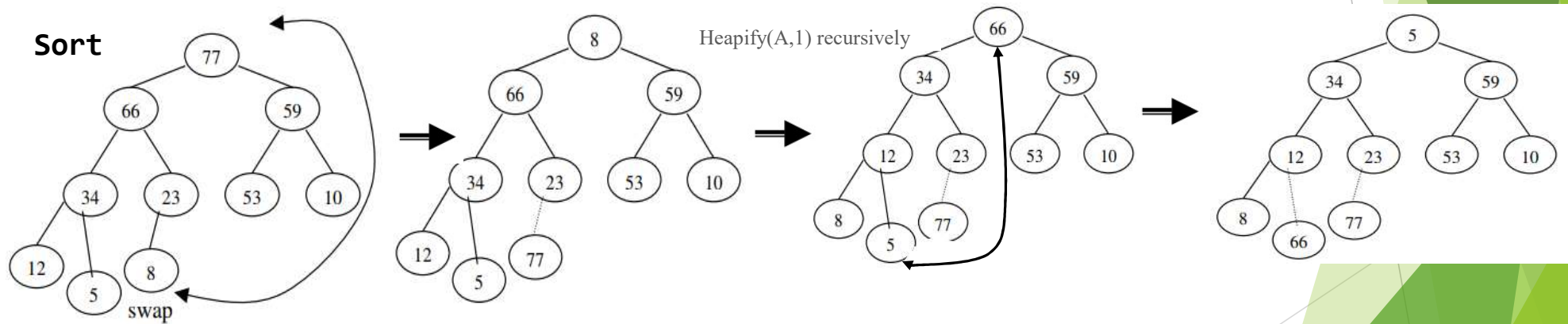
Example of heapsort

$A[] = \{10, 12, 53, 34, 23, 77, 59, 66, 5, 8\}$

Build heap



Sort



Continue the process...

Analysis:

1. Build heap process:
 - the “for” loop runs $n/2$ times
 - The “heapify” process may take at most $\log n$ time (depth of binary tree)
 - Total time during build heap = $O(n \log n)$
2. Sort process:
 - the “for” loop runs $n-1$ times
 - The “heapify” process inside “for” loop may take at most $\log n$ time
 - Total time during sort process = $O(n \log n)$

Hence time complexity of heapsort = $O(n \log n)$

Note: From in-depth (exact) analysis, it has been shown that “build heap” actually takes $O(n)$ time. However, this level of analysis is beyond our scope. But asymptotically, our analysis is also correct in terms of big-oh notation.

Order statistics (Selection problem)

- The i^{th} order statistic of a set of n elements is the i^{th} smallest element.
- e.g., the minimum of a set of elements is the first order statistic ($i=1$), and the maximum is the n^{th} order statistic ($i=n$).

Selection problem:

“Given a set A of n distinct numbers and an integer i , ($1 \leq i \leq n$), select an element $x \in A$ that is larger than exactly $i-1$ other elements.”

How to solve the problem?

1. Naïve approach

- Sort the numbers (in $\Theta(n \log n)$ time)
- Pick the i^{th} element (in constant time)

Total time = $\Theta(n \log n)$

2. DAC approach

- Utilize the partitioning (pivoting) strategy of quicksort.
- Note that “if the pivot lies at r^{th} position then the pivot itself is r^{th} order statistic”.

Steps:

- Partition the array. Let pivot lies at r^{th} position
- if $r=i$ then we are done.
- else if ($r < i$)
partition the right subarray and go to step ii.
- else if ($r > i$)
Partition the left subarray and go to step ii.

Example: find 6th order statistic

Pivot = 9

9	5	7	20	4	3	26	25	30	10	12
---	---	---	----	---	---	----	----	----	----	----

down

up

4	5	7	3	9	20	26	25	30	10	12
---	---	---	---	---	----	----	----	----	----	----

6 > pivot position.
Leave left subarray and partition right subarray

20	26	25	30	10	12
----	----	----	----	----	----

down

up

10	12	20	30	25	26
----	----	----	----	----	----

6 < pivot position.
Leave right subarray and partition left subarray

10	12
----	----

down

up

10	12
----	----

Pivot = 10

6 = pivot position.
6th order statistic = 10

Algorithm

```
DACselection(A, first, last, i)
{
    if (first == last)
        return A[first]
    r = partition(A, first, last, i)
    if (r == i)
        return A[r]
    else if (r > first and r > i)
        DACselection(A, first, r-1, i)
    else if (r < last and r < i)
        DACselection(A, r+1, last, i)
}
```

We have assumed
array index
starts from 1

Analysis

- The algorithm divides given array into two parts and solves one part.
- the partitioning cost is at most $O(n)$.
- Since the size of the subarray to be solved depends on the position of pivot, we have three cases:

Best case:

- Best case scenario is when the first pivot chosen is luckily the required i^{th} order statistic.
- In this case, the complexity is $\Theta(1)$

Worst case:

- worst case occurs when pivot always lies at one end during each partitioning and we are finding the element at the other end.
- In this case the complexity is

$$\begin{aligned} T(n) &= T(n-1) + \Theta(n) \\ &= ? \end{aligned}$$

Average case:

- consider different scenarios where the array is divided into different sized fractions during each partitioning.

$$T(n) = T\left(\frac{2n}{3}\right) + \Theta(n)$$

$$T(n) = T\left(\frac{n}{3}\right) + \Theta(n)$$

$$T(n) = T\left(\frac{9n}{10}\right) + \Theta(n)$$

$$T(n) = T\left(\frac{n}{10}\right) + \Theta(n)$$

In all above cases, we see that the complexity is ??

Selection in expected linear time(Randomized algorithm for selection problem)

- The DAC algorithm for selection problem gives worst case complexity of $O(n^2)$.
- To avoid the worst case, we can use randomized strategy for partitioning.

```
RandDACselection(A, first, last, i)
{
    if(first==last)
        return A[first]
    r = randPartition(A, first, last, i)
    if(r==i)
        return A[r]
    else if(r>first and r>i)
        RandDACselection(A, first, r-1, i)
    else if(r<last and r<i)
        RandDACselection(A, r+1, last, i)
}
```

33

```
randPartition(A, p, q)
```

```
{
    i=random(p, q)
    swap(A[p], A[i])
    partition(A, p, q)
}
```

```
partition(A, f, l)
{
    //Same as ordinary partition of quicksort
    .. ..
}
```

Analysis:

- The expected running time of above algorithm is $O(n)$.
- However, in worst case, it might go $O(n^2)$.

Prepared by: Shiv Raj Pant

Worst-case linear time selection

- The performance of DACSelection algorithm depends on the choice of pivot.
- Randomized version of the algorithm may still give worst case complexity.
- To guarantee worst case linear time $O(n)$ complexity, we have to use a good pivot selection strategy.
- The pivot selection strategy we use here is called *median-of-medians* strategy.

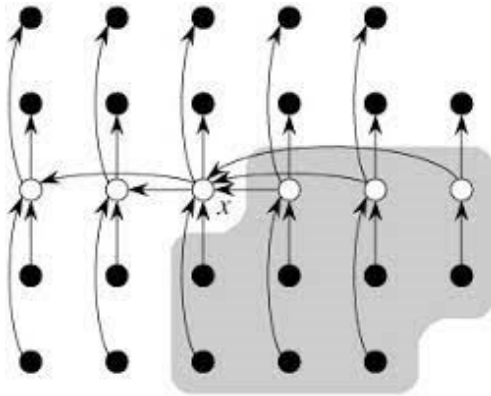
Algorithm

1. Divide n input elements into $\left\lfloor \frac{n}{5} \right\rfloor$ groups of 5 elements each.
2. Find the median of each of the $\left\lfloor \frac{n}{5} \right\rfloor$. (we can use insertion sort for this)
3. Use SELECT algorithm recursively to find the median x of the $\left\lfloor \frac{n}{5} \right\rfloor$ medians found in step 2. (if there are two medians, then we take minimum one).
4. Partition the array around the median-of-medians (x). Let k be the position of x after partition.
5. If $i == k$, then return x . Otherwise, use SELECT recursively the left subarray (if $i < k$) or right subarray (if $i > k$).

Analysis:

At least half of the medians found in **step 2** are greater than or equal to the median-of-medians x .

Thus at least half of the $\lceil \frac{n}{5} \rceil$ groups contribute at least 3 elements that are greater than or equal to x , except for the one group that has less than 5 elements (if 5 does not divide n) and the group containing x itself.



We ignore the group containing x and the group with less than 5 elements.

So the numbers of elements greater than x is at least $3 \left(\left\lceil \frac{1}{2} \left\lceil \frac{n}{5} \right\rceil \right\rceil - 2 \right) \geq \frac{3n}{10} - 6$

Similarly, the number of elements less than x is at least $\frac{3n}{10} - 6$

Thus, in worst case, **step 5** calls **SELECT** recursively on at most $n - (\frac{3n}{10} - 6) = \frac{7n}{10} + 6$ elements.

Now, let $T(n)$ be the running time of the algorithm.

- Step 1 takes $O(n)$ time
- Step 2 takes $O(n)$ time. ($O(n)$ insertion sort calls on constant-sized $O(1)$ set.)
- Step 4 takes $O(n)$ time.
- Step 3 takes $T(\lceil \frac{n}{5} \rceil)$ time
- Step 5 takes at most $T(\frac{7n}{10} + 6)$

So, we obtain the recurrence,

$$T(n) = T\left(\left\lceil \frac{n}{5} \right\rceil\right) + T\left(\frac{7n}{10} + 6\right) + O(n)$$

$$\text{or } T(n) \leq T\left(\left\lceil \frac{n}{5} \right\rceil\right) + T\left(\frac{7n}{10} + 6\right) + an, \text{ for some } a > 0 \dots \dots 1$$

We can show that the running time is linear ($O(n)$) by substitution.

$$\text{Let } T(n) = O(n)$$

$$\text{Then } T(n) \leq cn \quad \text{for some } c > 0$$

So,

$$T\left(\frac{n}{5}\right) \leq \frac{cn}{5} \quad \text{and} \quad T\left(\frac{7n}{10} + 6\right) \leq c\left(\frac{7n}{10} + 6\right)$$

From the recurrence 1,

$$\begin{aligned} T(n) &\leq \frac{cn}{5} + c\left(\frac{7n}{10} + 6\right) + an = \frac{9c}{10} + 6c + an \\ &= cn - \frac{cn}{10} + 6c + an \\ &= cn - \left(\frac{cn}{10} - 6c - an\right) \end{aligned}$$

Our guess will be correct if the quantity on RHS is less than or equal to cn

i.e. if,

$$\frac{cn}{10} - 6c - an \geq 0$$

$$\Rightarrow c \geq 10a\left(\frac{n}{n-60}\right) \dots \dots 2$$

We can take any $n > 60$ to satisfy the inequality 2.

Therefore, the worst case running time of the SELECT algorithm is linear.

From above analysis, the correct recurrence becomes,

$$\begin{aligned} T(n) &= T\left(\left\lceil \frac{n}{5} \right\rceil\right) + T\left(\frac{7n}{10} + 6\right) + O(n), \quad n > 61 \\ T(n) &= 1, \quad n \leq 60 \end{aligned}$$

End of unit 3

Thank You!