Week 1

Lecturer: Pritam Bhattacharya, BITS Pilani, Goa Campus

PRITAMB@GOA.BITS-PILANI.AC.IN

Date: 24/Jul/2021

Topics Covered

1. What is an algorithm?
2. What is time and space complexity?

What is an algorithm?

An algorithm is a finite sequence of steps to be followed to reach a pre determined goal for a predetermined set of inputs. An algorithm has the following properties:
1. **Finiteness**: the sequence of steps the algorithm has must be finite
2. **Definiteness**: Each step should be atomic and precise and cause no confusion on what it does
3. **Input**: There may or may not be an input passed to the algorithm to work on
4. **Termination**: Due to the finiteness of algorithms the algorithm must terminate and produce an output
5. **Correctness**: The output produces must be the right one for any given input

What is time and space complexity?

It is a way to analyze the performance of algorithms. Experimental studies for analyzing algorithms have some limitations:
1. It is difficult to compare running times of two algorithms in different software/hardware configurations
2. To truly know the performance we need to implement and run the algorithms on a computer since things like type of data provided for an algorithm can greatly alter the running time of the algorithm (For example an algorithm can run really slow for a single but large value and faster for multiple values)
3. One must need to make sure that the set of inputs used to analyze the algorithms are representative, meaning that the inputs must cover some particular use case (say for a sorting

algorithm, we can have one input where all values are already sorted, sorted in reverse, all values being same and all values being jumbled)

Consider the following example:

```
Algorithm arrayMax(A, n):
    Input: An array A storing n >= 1 integers.
    Output: The maximum element in A

    currentMax <- A[0]
    for i <- 1 to n - 1 do
        if currentMax < A[i] then
            currentMax <- A[i]
    return currentMax
```

The above example is what we call a *pseudo code* . It is meant to represent the flow of logic to find the maximum value in a given array.

Consider A = [3, 2, 7, 5]

Step i	What we are comparing it with	currentMax
0	null	= A[0] = 3
1	A[1] = 2	3 > 2 so 3
2	A[2] = 7	3 < 7 so 7
3	A[3] = 5	7 > 5 so 7

So going over the steps we see that at the end we can determine that the max value is A[2] = 7. We see that:

> "Algorithm arrayMax runs in time proportional to n"
>
> > If we were to actually run experiments, then the running time of arrayMax given any input of size n would never exceed c.n where c is the amount of time taken by the given software to run for an input of size 1.

As seen above the constant c depends on the language used to run the algorithm and the

hardware used to run the algorithm. A workstation can compute a more complex algorithm faster than a slow computer would for a less complex algorithm, hence to truly compare two algorithms all the parameters (The language, software and the hardware used) must be the same.

Given two algorithms has two implementations where:
1. **A algorithm**: that runs proportional to N and
2. **B algorithm** that runs proportional to N^2
we need to ideally makes sure that the algorithm chosen is the one that is proportional to N since for a very large input A would perform better.

Addition, Multiplication, Subtraction and Division are examples of primitive operations. Primitive operation are those that cannot be further broken down to more simpler steps.

In the above algorithm we can see the analysis as follows:

```
currentMax <- A[0]              ---> 2 = 1 (for indexing) + 1 (for assignment)
i <- 1                          ---> 1 (for assignment)
while i < n                     ---> n (1 for the every comparison)
    if currentMax < A[i] then     ---> 2 = 1 (for indexing) + 1 (for comparison)
        currentMax <- A[i]        ---> 2 = 1 (for indexing) + 1 (for assignment)
    i = i + 1                     ---> 2 = 1 (for addition) + 1 (for assignment)
return currentMax               ---> 1
```

From the above analysis we can say that the algorithm time complexity at the worst case where every iteration goes into the if block, is:

$$TimeComplexity < (2 + 1 + n + (n - 1) * (2 + 2 + 2) + 1)$$

$$TimeComplexity < 7n - 2$$

$$TimeComplexity < 7n$$

We say $<$ because there are cases where the statements under the if condition is not calculated.

Tags: !DatastructuresAndAlgorithmsIndex

Week 2

Lecturer: Pritam Bhattacharya, BITS Pilani, Goa Campus

PRITAMB@GOA.BITS-PILANI.AC.IN

Date: 31/Jul/2021

Topics Covered

1. Questions Answered on Previous Week Topics
2. Concept of Recursion

Questions Answered on Previous Week Topics

- **Question**: If an algorithm for solving a problem has a loop that runs n times and another algorithm to solve the same problem and has a loop that runs $\frac{n}{2}$, then what is the comparative analysis for these two algorithms?
  Answer: Both are proportional to n since the only thing that changed here is the constant c, which is 1 in the first case and $\frac{1}{2}$ in the other.
- **Question**: Do all primitives have the same constant running time?
  Answer: This is not the case but as a concept it is taken as to be the same since the running time for a given operation is not dependent on the number of inputs.
- **Question**: From the array max example (discussed here: Week1DSA#What is time and space complexity) we see that the algorithm in its worst case is $7n - 2$, what is it in its best case?
  Answer: In the best case, the maximum value is in the starting of the array $A[0]$ itself. In this case the inside of the if block will not even execute so we can reduce 2 primitive steps from the calculation, this will lead to the following analysis:

$$TimeComplexity < (2 + 1 + n + (n - 1) * (2 + 0 + 2) + 1)$$

$$TimeComplexity < 5n$$

Concept of Recursion

Consider the algorithm for finding the factorial of a number:

```
Algorithm factorial(n):
    Input: A number n > 0.
    Output: The factorial of n.

    product <- 1
    for i <- 1 to n do
        product <- i * product
    return product
```

The above algorithm is called an iterative one since it employs loops, the same can be written through recursive function calls such as below:

```
Algorithm factorial(n):
    Input: A number n > 0.
    Output: The factorial of n.

    product <- 1
    if n = 1 then
        return 1
    else
        result = n * factorial(n - 1)
        return result
```

Here you see that the loop is removed and the algorithm is called recursively, but it yields the same result. One must be careful to have at least one base case (Here $product \longleftarrow 1$) must exist, else recursion will go on indefinitely.

Now a recursive implementation for the array max algorithm (Week1DSA#What is time and space complexity) would be as follows:

```
Algorithm recursiveMax(A, n):
    Input: An array A storing n >= 1 integers.
    Output: The maximum element in A

    if n = 1 then
        return A[0]
    return max{recursiveMax(A, n-1), A[n-1]}
```

The running time for the same will be shows as below:

$$T(n) = 3,\ if\ n = 1$$

$$T(n) = T(n-1) + 7,\ otherwise$$

Tags: !DatastructuresAndAlgorithmsIndex

Week 4

Lecturer: Pritam Bhattacharya, BITS Pilani, Goa Campus

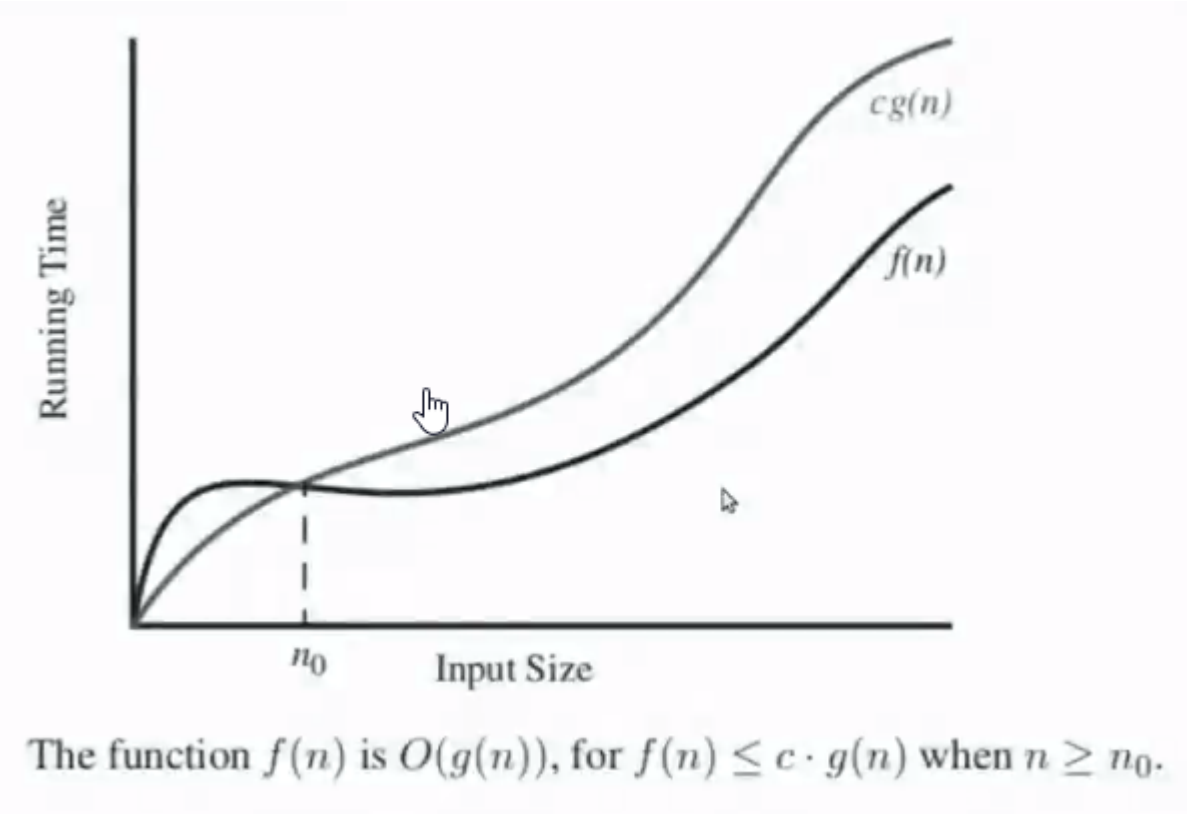
PRITAMB@GOA.BITS-PILANI.AC.IN

Date: 21/Aug/2021

Topics Covered

1. Deriving Big$\mathcal{O}$ From Growth Rate
    a. Significance of Growth Rate
    b. Examples
2. Rules to Find Big$\mathcal{O}$ Notation

Deriving Big$\mathcal{O}$ From Growth Rate



Significance of Growth Rate

Consider f(n) and g(n), Let us consider what a growth rate of a function is.
Growth rate

$f(n) = 1000n$
$g(n) = 2n^2$

n	$f(n)$	$g(n)$	$f(n)/f(n-1)$	$g(n)/g(n-1)$
1	1000	2	-	-
2	2000	8	2	4
3	3000	18	1.5	2.25
4	4000	32	1.33	1.78
5	5000	50	1.2	1.56

Seeing the above pattern in the rate of growth, we see that for any particular value of n the $g(n)$ ratio is bigger than that for $f(n)$, and this is seen as we incrementing n. So we can conclude that $g(n)$ has a faster rate of growth.

We can also find the value of n for which the value of $g(n)$ will shoot up more than $f(n)$:

$g(n) \geq f(n)$
$2.n^2 \geq 1000n$
$n \geq 500$

This shows that since the $g(n)$ grows faster (quadratic growth), whenever n is greater than 500 then $g(n)$ will always be greater than $f(n)$ so we can conclude that the algorithm that has a run time of $f(n)$ will be overall better than one that has a run time of $g(n)$.

Examples

Example 1: $f(n) = 20.n^3 + 10.n.log(n) + 5$ is $\mathcal{O}(n^3)$
Proof: $20.n^3 + 10.n.log(n) + 5 \leq 35.n^3$, for $n \geq 1$

Example 2: $f(n) = 2^{100}$ is $\mathcal{O}(1)$
Proof: $2^{100} \leq 2^{100}.1$, for $n \geq 1$

Example 3: $f(n) = 3^log(n) + log(log(n))$ is $\mathcal{O}(log(n))$
Proof: $3^log(n) + log(log(n)) \leq log(n)$, for $n \geq 2$ and $c = 4$

Rules to Find Big$\mathcal{O}$ Notation

1. If $d(n)$ is $O(f(n))$, then $ad(n)$ is $O(f(n))$, for any constant $a > 0$.
2. If $d(n)$ is $O(f(n))$ and $e(n)$ is $O(g(n))$, then $d(n)+e(n)$ is $O(f(n)+g(n))$.
3. If $d(n)$ is $O(f(n))$ and $e(n)$ is $O(g(n))$, then $d(n)e(n)$ is $O(f(n)g(n))$
4. If $d(n)$ is $O(f(n))$ and $f(n)$ is $O(g(n))$, then $d(n)$ is $O(g(n))$.
5. If $f(n)$ is a polynomial of degree d (that is, $f(n) = a_0 + a_1 n + \cdots + a_d n^d$), then $f(n)$ is $O(n^d)$.
6. n^x is $O(a^n)$ for any fixed $x > 0$ and $a > 1$.
7. $\log n^x$ is $O(\log n)$ for any fixed $x > 0$.
8. $\log^x n$ is $O(n^y)$ for any fixed constants $x > 0$ and $y > 0$.

Week 4 (cont.)

Lecturer: Pritam Bhattacharya, BITS Pilani, Goa Campus

PRITAMB@GOA.BITS-PILANI.AC.IN

Date: 22/Aug/2021

Topics Covered

1. Rules to Find Big$\mathcal{O}$ Notation (cont.)
2. Little$o$ and Littleω
3. Bubble Sort
    a. Sort Function
    b. Optimized Sort Function
    c. Algorithm Analysis
        i. Time Complexity
        ii. Space Complexity

Rules to Find Big$\mathcal{O}$ Notation (cont.)

- It is considered poor taste to include constant factors and lower order terms in the big$\mathcal{O}$ notation
- Consider a function $2n^2$ is $\mathcal{O}(4n^2 + 6nlog(n))$, although this is right it is always easier and simpler to write $\mathcal{O}(n^2)$

Logrithmic	Linear	Quadractic	Polynomial	Exponential
$\mathcal{O}(log(n))$	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^k)$ $(k \geq 1)$	$\mathcal{O}(a^n)$ $(a > 1)$

- Even though in general we ignore constants in the big$\mathcal{O}$ notation, we need to be careful and check if the constants have very large constants, in this case big$\mathcal{O}$ might not be the right

assumption for an algorithm.

Some Functions Ordered By Growth Rate	Commone Name
$\mathcal{O}(log(n))$	Logrithmic
$\mathcal{O}(log^2(n))$	Polytlogrithmic
$\mathcal{O}(\sqrt{n})$	Square Root
$\mathcal{O}(n)$	Linear
$\mathcal{O}(nlog(n))$	Linearithmic
$\mathcal{O}(n^2)$	Quadratic
$\mathcal{O}(n^3)$	Cubic
$\mathcal{O}(2^n)$	Exponential

n	$\log n$	$\log^2 n$	$\sqrt{n}$	$n \log n$	n^2	n^3	2^n
4	2	4	2	8	16	64	16
16	4	16	4	64	256	4,096	65,536
64	6	36	8	384	4,096	262,144	1.84×10^{19}
256	8	64	16	2,048	65,536	16,777,216	1.15×10^{77}
1,024	10	100	32	10,240	1,048,576	1.07×10^{9}	1.79×10^{308}
4,096	12	144	64	49,152	16,777,216	6.87×10^{10}	10^{1233}
16,384	14	196	128	229,376	268,435,456	4.4×10^{12}	10^{4932}
65,536	16	256	256	1,048,576	4.29×10^{9}	2.81×10^{14}	10^{19728}
262,144	18	324	512	4,718,592	6.87×10^{10}	1.8×10^{16}	10^{78913}

- In the above chart you see that $\sqrt{n}$ crosses over much later over $log^2(n)$ in comparison to other function pairs.

Little$o$ and Littleω

No matter what value of $c > 0$ we choose, we need to see if $g(n) \geq f(n)$. For example for a functions $12n^2 + 6n$, the little$o$ is $o(n^3)$.

if $f(n)$ is $o(g(n)$ then $g(n)$ is $\omega(f(n))$

Bubble Sort

Sort Function

```
BubbleSort(int [] A, int n)
    Input: An array A containing n>= 1 integers
    Output: The sorted version of the array A

    for i = 0 to (n - 1)
        for j = 0 to (n - 1 - i)
            if A[i] > A[j + 1]
                # swap A[j] with A[j + 1]
                A[j] <-> A[j + 1]
    return A
```

Optimized Sort Function

```
BubbleSortOptimized(int [] A, int n)
    Input: An array A containing n>= 1 integers
    Output: The sorted version of the array A

    for i = 0 to (n - 1)
        swaps = 0
        for j = 1 to (n - 1 - i)
            if A[i] > A[j + 1]
                # swap A[j] with A[j + 1]
                A[j] <-> A[j + 1]
                swaps <- swaps + 1
            if swaps == 0
                break
    return A
```

Algorithm Analysis

Time Complexity

In the basic sort function, for every value of i, the inner loop indexed by j is done upto $n-1$ number of times and the outer loop indexed by i also runs $n-1$ times as well.
So the worst case time complexity is $\mathcal{O}((n-1)(n-1)) = \mathcal{O}(n^2)$
Best case time complexity (Where array is already sorted) is also $\mathcal{O}(n^2)$

Now taking the optimized sort function, we see that at the worst case the function behaves exactly

like the basic sort function so the worst case time complexity is still $\mathcal{O}(n^2)$, but in the best case the outer loop will run only once, so the best case time complexity is $\mathcal{O}(n)$ since only the inner loop will run completely once.

Space Complexity

Since no extra data structures were used, the space used is constant, so the space complexity in all cases is $\mathcal{O}(1)$

Week 5

Lecturer: Pritam Bhattacharya, BITS Pilani, Goa Campus

PRITAMB@GOA.BITS-PILANI.AC.IN

Date: 28/Aug/2021

Topics Covered

1. Selection Sort
    a. Steps
    b. Algorithm
        i. Sort Function
        ii. Optimized Sort Function
    c. Algorithm Analysis
        i. Time Complexity
        ii. Space Complexity

Selection Sort

Selection sort uses similar concepts of Bubble sort but then instead of bubbling up the max value in each pass, we select the max value in the given array and then swap that to the last position directly and do the same to the second last element and so on.

Steps

Let us consider the following arrays
Initial State: $[\ 4, 1, 7, 5, 2, 3\ ]$
At each pass we select the maximum value of the array and move it to the end:
Pass 1: $[\ 4, 1, 3, 5, 2, 7\ ]$
Pass 2: $[\ 4, 1, 3, 2, 5, 7\ ]$
Pass 3: $[\ 2, 1, 3, 4, 5, 7\ ]$

Pass 4: $[\ 2, 1, 3, 4, 5, 7\ ]$
Pass 5: $[\ 1, 2, 3, 4, 5, 7\ ]$

Algorithm

Sort Function

```
sort(A):
    Input: An array of size n
    Ouput: The sorted version of the array A

    for i = 1; i < n do
        max = A[0]
        maxIndex = 0
        for j = 1; j < (n - i) do
            if A[j] > max then
                max = A[j]
                maxIndex = j
        A[maxIndex] <-> A[n - i]
```

Optimized Sort Function

```
optimizedSort(A):
    Input: An array of size n
    Ouput: The sorted version of the array A

    for i = 1; i < n do

        inversions = 0
        for j = 0; j < (n - 1 - i) do
            if A[j] > A[j + 1] then
                inversion = inversion + 1
        if invversions == 0 then
            break

        max = A[0]
        maxIndex = 0
        for j = 1; j < (n - i) do
            if A[j] > max then
                max = A[j]
                maxIndex = j
        A[maxIndex] <-> A[n - i]
```

Algorithm Analysis

Time Complexity

Just like the bubble sort, here we see that the outer loop runs in the order of n and the inner loop is also running in the order of n
So the worst case time complexity is $\mathcal{O}(n^2)$
Best case time complexity (Where array is already sorted) is also $\mathcal{O}(n^2)$

Space Complexity

Since no extra data structures were used, the space used is contant, so the space complexity in all cases is $\mathcal{O}(1)$

Tags: IDatastructuresAndAlgorithmsIndex

Week 5 (cont.)

Lecturer: Pritam Bhattacharya, BITS Pilani, Goa Campus

PRITAMB@GOA.BITS-PILANI.AC.IN

Date: 29/Aug/2021

Topics Covered

1. Insertion Sort
    a. Steps
    b. Algorithm
        i. Sort Function
        ii. Optimized Sort Function
    c. Algorithm Analysis
        i. Time Complexity
        ii. Space Complexity

Selection Sort

Steps

Let us consider the following arrays
Initial State: $[\ 1, 4, 7, 5, 2, 3\ ]$
At each pass we sort the array that we get by adding one element to it, So we first sort indexes $1, 2$ of the array, after that we sort the indexes $1, 2, 3$ and so on till $1, 2, 3, \ldots, n$ to get the final sorted array
Pass 1: $[\ 1, 4, 7, 5, 2, 3\ ], j = 0$
Pass 2: $[\ 1, 4, 7, 5, 2, 3\ ]$
Pass 3: $[\ 1, 4, 5, 7, 2, 3\ ]$
Pass 4: $[\ 1, 2, 4, 5, 7, 3\ ]$
Pass 5: $[\ 1, 2, 3, 4, 5, 7\ ]$

Algorithm

Sort Function

```
sort(A):
    Input: An array of size n
    Ouput: The sorted version of the array A

    for i = 1 to n - 1
        j = i - 1
        while A[i] < A[j]
            j = j - 1
            if j < 0
                break
        curr = A[i]
        k = i - 1
        while k >= j + 1
            A[k + 1] = A[k]
            k = k - 1
        A[j + 1] = curr
```

Algorithm Analysis

Time Complexity

Just like the selection sort, here we see that the outer loop runs in the order of n and the inner while loop is also running in the order of n
So the worst case time complexity is $\mathcal{O}(n^2)$
Best case time complexity (Where array is already sorted) is $\mathcal{O}(n)$, since none of the loops are executed if the elements are sorted, so the order of n operations that is contributed by them will not happen.

Space Complexity

Since no extra data structures were used, the space used is constant, so the space complexity in all cases is $\mathcal{O}(1)$

Tags: !DatastructuresAndAlgorithmsIndex

Week 6

Lecturer: Pritam Bhattacharya, BITS Pilani, Goa Campus

PRITAMB@GOA.BITS-PILANI.AC.IN

Date: 4/Sep/2021

Topics Covered

1. Merge Sort
    a. Steps
    b. Algorithm
        i. Merge Function
        ii. Sort Function
    c. Algorithm Analysis
        i. Space Complexity
        ii. Time Complexity

Merge Sort

Merge Sort is not an in-place sort like the other three that was explained above

Steps

Let us consider the following arrays
$4, 1, 7, 5, 2, 3$
Now let us split this array into two halves:
$4, 1, 7 \mid 5, 2, 3$
And let us sort these two parts:
$1, 4, 7 \mid 2, 3, 5$
When merging the above two parts, we maintain two pointers that start on either parts, and we compare the elements pointed by these two and place the smaller element into a new array and

advance that pointer and repeat the above procedure till all the elements are compared.

Algorithm

Merge Function

```
merge(S1, S2, S):
    Input: Two arrays, S1, S2, of size n1 and n2 sorted in non decreasing order,
and an empty array S of size n1 + n2
    Ouput: S, containing the elements from S1 and S2 in sorted order

    i <- 0
    j <- 0
    while i < n1 and j < n2 do
        if S1[i] <= S2[j] then
            S[i + j] <- S1[i]
            i <- i + 1
        else
            S[i + j] <- S2[j]
            j <- j + 1
    while i <= n1 do
        S[i + j] <- S1[i]
        i <- i + 1
    while i <= n2 do
        S[i + j] <- S2[j]
        j <- j + 1
```

Sort Function

The actual sorting is done in a divide and conquer fashion denoted by the below steps:
1. **Divide:** If S has zero or one element, return S immediately, it is already sorted, Otherwise put the elements of S into two sequences S_1 and S_2, each containing about half of the elements of S
2. **Recur:** Recursively sort the sequences S_1 and S_2
3. **Conquer:** Put back the elements into S by merging the sorted sequences S_1 and S_2 into a sorted sequence.

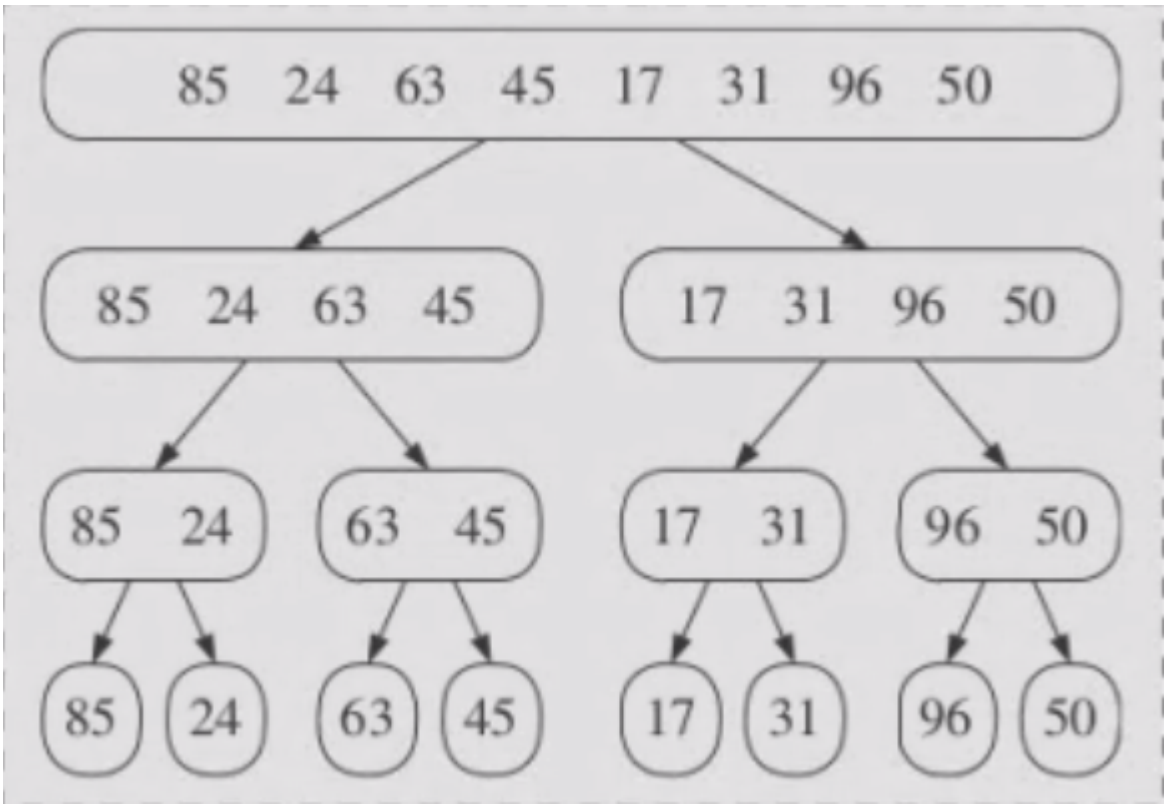
```
sort(S):
    Input: An array of size n
    Ouput: The sorted version of the array S

    if n == 1 then
        return S
    if n == 2 then
        if S[0] > S[1] then
            S[0] <-> S[1]
        return S
    S1 <- {S[0], S[1], S[2], .... , S[n/2]}
    S1 <- {S[(n/2)+1], .... , S[n - 2], S[n - 1]}
```

```
    R1 <- sort(S1)
    R2 <- sort(S2)
    merge(R1, R2, R)         # R is an empty array of size n
    return R
```

Algorithm Analysis

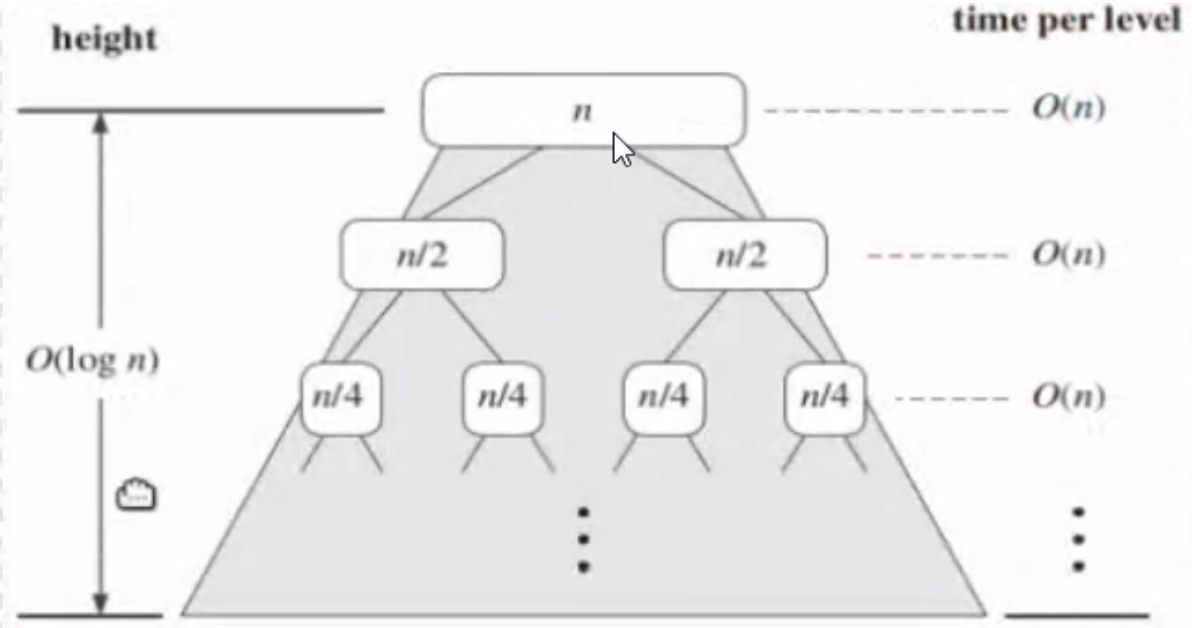
Space Complexity

The recursive tree for a random input array would look something like this:



From the above image you can see that the number of memory blocks from the root to the leaf node of the recursive tree is at max n, so:
Space Complexity: $\mathcal{O}(n)$

Time Complexity


height
time per level
n
O(n)
n/2
n/2
O(n)
O(log n)
n/4
n/4
n/4
n/4
O(n)

Week 6 (cont.)

Lecturer: Pritam Bhattacharya, BITS Pilani, Goa Campus

PRITAMB@GOA.BITS-PILANI.AC.IN

Date: 5/Sep/2021

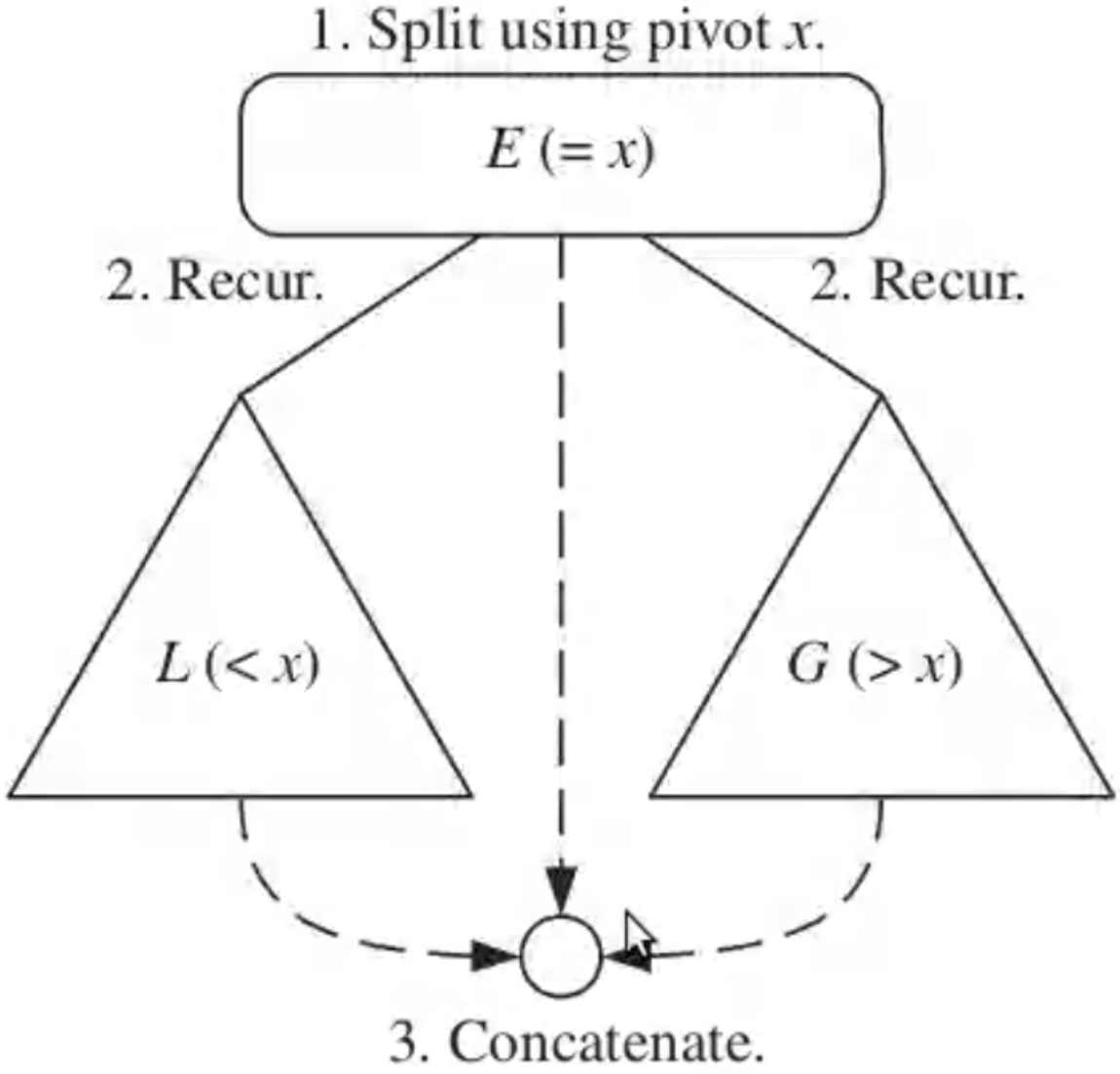
Topics Covered

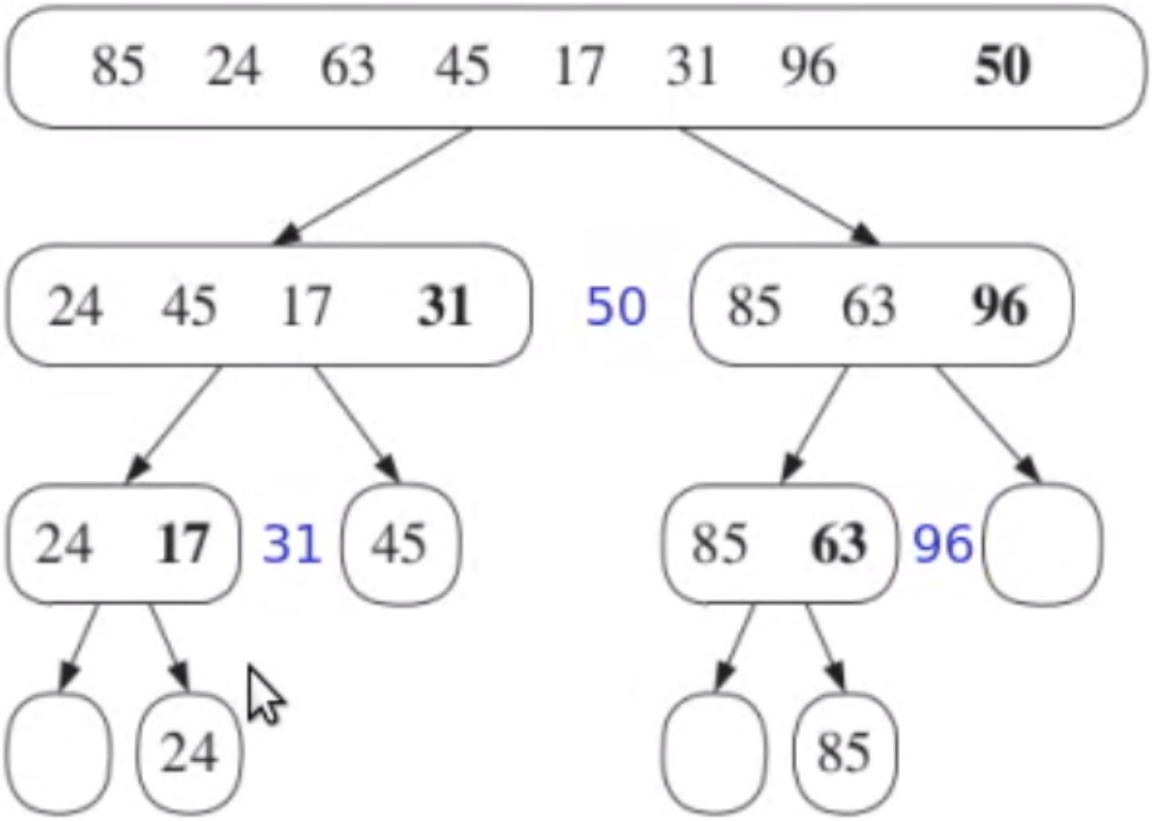
1. Quick Sort
    a. Steps Done
    b. Algorithm
        i. Pivot Function
        ii. Quick Sort Function
    c. Algorithm Analysis
        i. Space Complexity
        ii. Time Complexity

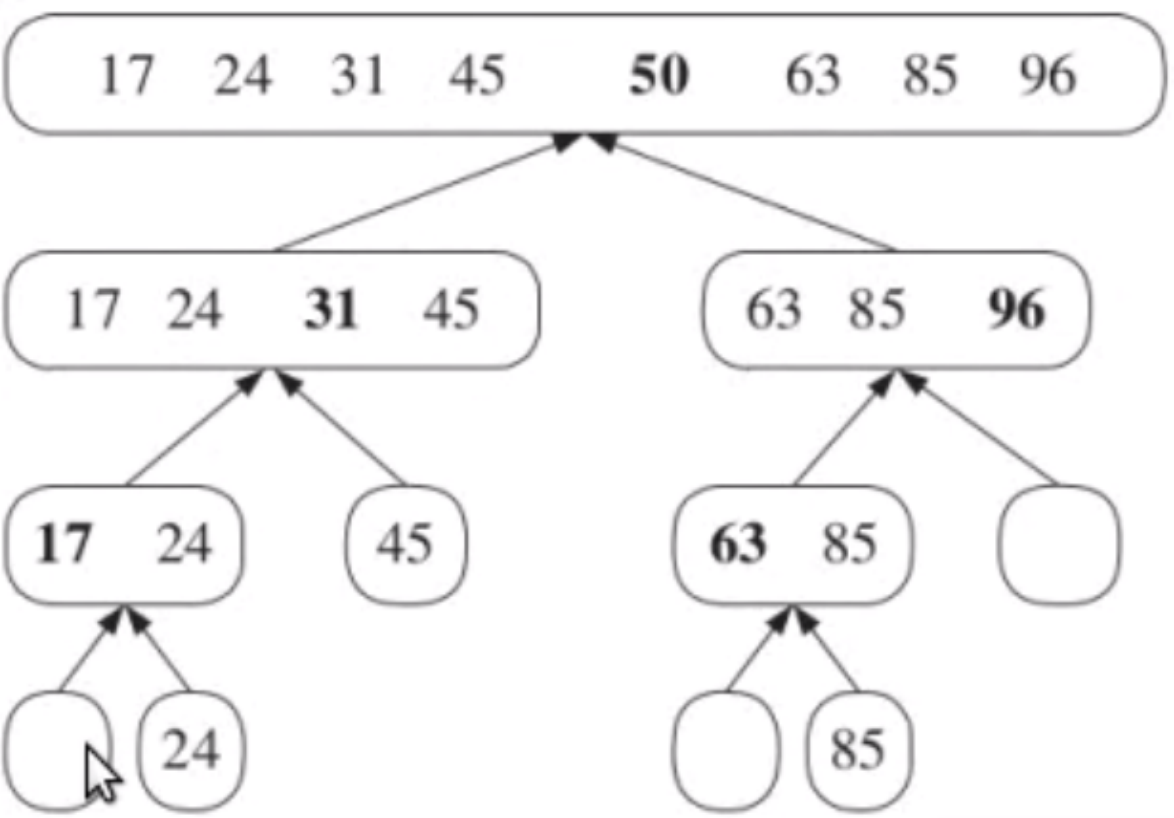
Quick Sort

Steps Done

The actual sorting is done in a divide and conquer fashion denoted by the below steps:
1. **Divide:** If S has zero or one element, return S immediately, it is already sorted, Otherwise pick a random element as a pivot element, and generate 3 parts:
1. L : The elements less than the pivot element
2. E : The element that is used as a pivot
3. G : The elements greater than the pivot element
2. **Recur:** Recursively sort the sequences L and G
3. **Conquer:** Concatenate the three sequences and return that


1. Split using pivot x.
$E\ (= x)$
2. Recur.
2. Recur.
$L\ (< x)$
$G\ (> x)$
3. Concatenate.

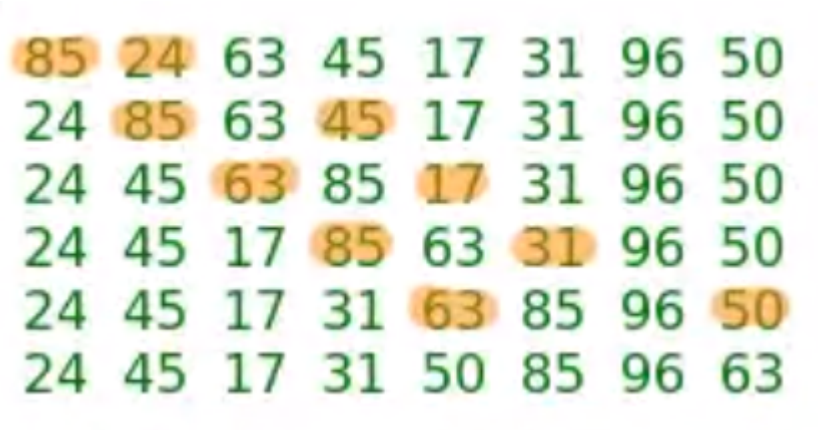

85 24 63 45 17 31 96 50
24 45 17 31
50
85 63 96
24 17
31
45
85 63
96
24
85



Algorithm

Pivot Function

```
partition(arr, low, high)
    Input: An array of integers arr the lower index for the partition in the
array and high the upper index
    Ouput: pi, Partitioning index, the index where the pivot is placed

    pivot = arr[high]
    i = (low - 1)
    for(j = low; j <= high - 1; j++) then
        if arr[i] < pivot then
            i++
            arr[i] <-> arr[j]
    arr[i + 1] <-> arr[high]
    return (i + 1)
```



Quick Sort Function

```
sort(arr, low, high)
    Input: An array of integers arr the lower index for the partition in the
array and high the upper index
    Ouput: pi, Partitioning index, the index where the pivot is placed

    if low < high then
        pi = partition(arr, low, high)
```