quiz 1:
54:00 in video.
1. What is the tight bound on the asymptotic running time, in terms of the number of states n and the number of representatives R, of the ApportionCongress algorithm described on page 9 of your textbook if the priority queue is implemented efficiently with a heap data structure?

**********0(R log(n))*********
None of the above
0(r)
0(r)
0(Rn)
0(n)

2. Which of the following values for c and n_0 can be used to prove that 3n+20 = O(n)? Pick ALL correct answers.
c=23, n0 =1
c=2, n0=100
c=3, n0=21
****c=4, n0=20****

3. Consider this recursive implementation of Horner's method for polynomial evaluation:

```
def recHorner(a, i, n, z):
    if i > n:
        return 0
    return a[i] + z * recHorner(a, i+1, n, z)
```

Which one of the following is the correct way to call this function on a polynomial of degree n whose coefficients are stored in array a and a value z:

*******recHorner (a,0,n,z)*******
recHorner(a, 0,z,n)
recHorner(a,n,0,z)
recHorner(a,n,z,0)
none of the above.

4. Suppose you repeatedly divide (using integer division) an integer n by 2. For example, if n is 120, then this process will generate values 60, 30, 15, 7, etc. How many divisions, in terms of n, will it take to get down to 1?

0(n log(n))
0(n^2)
*************0(log(n))************
0(n)

5. Suppose you append a constant number of values to a sorted integer array of size n. What would be the running time of insertionSort on such limited type of input?

0(n log(n))
none of the above
****0(n)***
0(n^2)
0(log(n))

Quiz 2:

1. Which of the following insights is NOT used in the development of the divide-and-conquer algorithm dicussed in the week 2 lecture?

Every set of n points has the property that when sorted by y-coordinate, points that are close to each other in the ordering are also close to each other on the plane.

Before the recursive divide-and-conquer procedure is even started, we must construct two different arrays containing the input points: one array holds the points in order by x-coordinate and the other in order by y-coordinate.

The closest pair of points are not necessarily on the same (left or right) side of line l (slides 67-72)

When considering pairs of points lying on opposite sides of line l, we only need to consider those points whose distance form line l is less than delta.

Each one of the 16 equal sized dashed squares shown on slides 74-75 may contain one point at most.

2. Not including the memory space necessary to hold the input array A, how much extra memory space is used by MergeSort and by QuickSort? Choose all that apply.

Question 2 options:

****

Θ(1) extra memory space is used by MergeSort
****

Θ(n) extra memory space is used by QuickSort

Θ(n) extra memory space is used by MergeSort

3. Match each recurrence relation on the right to a solution on the left.

1.
$T(n) = 4T(n/2) + O(n)$
$o(n^2)$

2.
$T(n) = 3T(n/2) + O(n)$
$o(n \log n)$

3.
T(n) = 2T(n/2) + O(n)
0(n^LOG2^3)

4.
T(n) = T(n/2) + O(n)
0(n)

4. Which of the following algorithms covered in lecture 2 have a worst case running time of O(log n)? Choose all that apply.

Question 4 options:

QuickSort

FastMultiply

MergeSort

****PingalaPower*****

****Binary Search****

5. The initial call to multiply x=1,234,567 and y=9,876,543 using function FastMultiply on page 41 of your textbook will make three recursive calls to FastMultiply. What three pairs of values will the three recursive calls be multiplying?

Question 5 options:

12 and 98, 34567 and 76543, -34555 and -76445

1234 and 9876, 567 and 543, 667 and 9333

1234 and 6543, 567 and 987, 667 and 5556

123 and 543, 4567 and 9876, 4444 and 9333

****123 and 987, 4567 and 6543, -4444 and -5556****

Quiz 3:

1. Execute by hand the PlaceQueens algorithm with n=5, i.e. on a 5x5 chessboard until you reach a valid solution. What is the column of the queen in the last row?

5
2
3
4
1

2. Figure 2.5 on page 75 of your textbook shows levels 0, 1, and 2 of the fake-sugar packer game tree. No two states in these levels are the same. What is the shallowest level at which some states will appear several times?
****3****
6
4
7
5

3. The Subset Sum algorithm on lecture slide 64 will output all valid solutions when executed on problem instance S = [4, 7, 6, 3, 1] and T = 10. Which valid solution is found first? Second? Third?

first ------> [6,3,1]

second----> [7,3]

Third -------> [4,6]

4. Suppose you run the Subset Sum algorithm on lecture slide 64 on problem instance S = [4, 7, 6, 3, 1] and T = 10. Which of the following partial solutions will be pruned because of the first pruning rule, i.e. the one described on lecture slide 67?

X = [1,0,0,1]

*****X = [0,1,1]******

X = [0,0,1,1,0]

X = [1,0,0]

X = [1,0,0,0,1,1]

5. Which of the following are correct statements about the prof's solution to the Kattis Class Picture problem?

Question 5 options:

Regardless of what pairs of people don't like each other, the program will never output a lineup in which the names are sorted in reverse lexicographic order (i.e., in reverse dictionary order).

The prof's initial solution was accepted by Kattis.

The re-sorting of the set remaining in every recursive call of function solve is critical to ensure that the output is accepted by Kattis.

Excluding the recursive call, the running time of the body of the for loop in function solve is O(1). You may assume that the Python set class methods and operators used in the body of the for loop run in O(1) time.

If input m is 0 then the lineup that is output is lexicographically sorted (i.e., sorted in dictionary order).

The initial solution that was rejected by Kattis used the Python set data type to store student names.

Quiz 4 :

Question 1 (2 points)
Which of the following are correct statements about the text segmentation algorithm on slide 42 and/or the two partial solutions on slide 29 of Lecture 4?

The partial solution starting with BLUE will be generated and extended by the algorithm before the partial solution starting with BLUEST is ever generated.

It is possible that during the execution of the Splittable algorithm the recursive call Splittable(i) for some index i between 1 and n is executed repeatedly.

The running time of Splittable on an input sequence of length n is quadratic in n.

It is posible that during the execution of the Splittable algorithm a recursive call Splittable(i) is made for some i greater than n+1.

Question 2
Which of the following are longest increasing subsequences of sequence

2 6 1 4 2 4 2 9 5 3 5 7 8 3

? If more than one is a longest increasing subsequence, you must choose all of them.

Question 2 options:

1 2 3 5 7 8

1 2 4 5 7 8

6 1 4 9 5 7

1 4 5 3 7 8

1 2 3 4 7 8

Question 3 (2 points)
Suppose that a recursive call to LISbigger (on slide 61 of Lecture 4) is made with prev = 6 and A[1..N] = [2 6 4 2 9 5 3 5 7 8 3] and that you then observe the execution of the algorithm, i.e. execution of successive recursive calls, until the value of prev changes. What value does prev change to?
Question 3 options:

2

6

5

9

4

Question 4 (2 points)
What is the total cost of accessing keys 3, 5, and 7 if they are 1) stored in a binary search tree with root 3 with child 5 and grandchild 7 and 2) key 3 is accessed six times, 5 is accessed four times, and 7 is accessed twice. (You will need to use the Cost function given on slides 80-81 of Lecture 4.)

Question 4 options:

20

23

19

21

22

Question 5 (2 points)
Which of the following is the binary search tree that minimizes the total cost of accessing keys 3, 5, 7, and 9 with access frequencies 8, 2, 6, and 2, respectively? (Note that the below descriptionsdo implicitly specify the position(s) in the tree of the key(s) not explicitly mentioned.)
Question 5 options:

Root 5 with right child 9

Root 3 with right child 5 whose right child is 7

Root 5 with right child 7-

Root 3 with right child 9 whose left child is 7

Root 3 with right child 7

8 questions total.

2. written.

6. multiple choice. 2 from quize directly.

the 2 written question questions are about:

1. Divide & Conquer : solve a problem
A divide-and-conquer algorithm recursively breaks down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.

The divide-and-conquer technique is the basis of efficient algorithms for many problems, such as sorting (e.g., quicksort, merge sort), multiplying large numbers (e.g., the Karatsuba algorithm), finding the closest pair of points, syntactic analysis (e.g., top-down parsers), and computing the discrete Fourier transform (FFT).[1]

Similarly, decrease and conquer only requires reducing the problem to a single smaller problem, such as the classic Tower of Hanoi puzzle, which reduces moving a tower of height n to moving a tower of height n − 1.

Binary Search is a searching algorithm. ...
Quicksort is a sorting algorithm. ...
Merge Sort is also a sorting algorithm. ...
Closest Pair of Points The problem is to find the closest pair of points in a set of points in x-y plane.

A typical Divide and Conquer algorithm solves a problem using the following three steps.

Divide: Break the given problem into subproblems of same type. This step involves breaking the problem into smaller sub-problems. Sub-problems should represent a part of the original problem. This step generally takes a recursive approach to divide the problem until no sub-problem is further divisible. At this stage, sub-problems become atomic in nature but still represent some part of the actual problem.
Conquer: Recursively solve these sub-problems. This step receives a lot of smaller sub-problems to be solved. Generally, at this level, the problems are considered 'solved' on their own.
Combine: Appropriately combine the answers. When the smaller sub-problems are solved, this stage recursively combines them until they formulate a solution of the original problem. This algorithmic approach works recursively and conquer & merge steps works so close that they appear as one.
This method usually allows us to reduce the time complexity by a large extent.

One last time

Divide / Break. Break the problem into smaller sub-problems.
Conquer / Solve. Solves all the sub-problems.
Merge / Combine. Merges all the sub-solutions into one solution.

Towers of Hanoi 

The Towers of Hanoi is a mathematical problem which compromises 3 pegs and 3 discs. This problem is mostly used to teach recursion, but it has some real-world uses. The number of pegs & discs can change.

Each disc is a different size. We want to move all discs to peg C so that the largest is on the bottom, second largest on top of the largest, third largest (smallest) on top of all of them. There are some rules to this game:

We can only move 1 disc at a time.
A disc cannot be placed on top of other discs that are smaller than it.
We want to use the smallest number of moves possible. If we have 1 disc, we only need to move it once. 2 discs, we need to move it 3 times.

The number of moves is a power of 2 minus 1. Say we have 4 discs, we calculate the minimum number of moves as
2
4
=
16
−
1
=
15
2
4
=16−1=15.

To solve the above example we want to store the smallest disc in a buffer peg (1 move). See below for a gif on solving Tower of Hanoi with 3 pegs and 3 discs.
Notice how we need to have a buffer to store the discs.

We can generalise this problem. If we have n discs: move n-1 from A to B recursively, move largest from A to C, move n-1 from B to C recursively.

If there is an even number of pieces the first move is always into the middle. If it is odd the first move is always to the other end.

Let's code the algorithm for ToH, in pseudocode.

```
function MoveTower(disk, source, dest, spare):
    if disk == 0, then:
        move disk from source to dest
```

edit
play_arrow

brightness_4

```
// Java code to demonstrate Divide and
// Conquer Algorithm
class GFG{
```

```
// Function to find the maximum no.
// in a given array.
static int DAC_Max(int a[], int index, int l)
{
    int max;

    if (index >= l - 2)
    {
        if (a[index] > a[index + 1])
            return a[index];
        else
            return a[index + 1];
    }

    // Logic to find the Maximum element
    // in the given array.
    max = DAC_Max(a, index + 1, l);

    if (a[index] > max)
        return a[index];
    else
        return max;
}

// Function to find the minimum no.
// in a given array.
static int DAC_Min(int a[], int index, int l)
{
    int min;
    if (index >= l - 2)
    {
        if (a[index] < a[index + 1])
            return a[index];
        else
            return a[index + 1];
    }

    // Logic to find the Minimum element
    // in the given array.
    min = DAC_Min(a, index + 1, l);

    if (a[index] < min)
        return a[index];
    else
        return min;
}

// Driver Code
public static void main(String[] args)
{

    // Defining the variables
    int min, max;
```

```
    // Initializing the array
    int a[] = { 70, 250, 50, 80, 140, 12, 14 };

    // Recursion - DAC_Max function called
    max = DAC_Max(a, 0, 7);

    // Recursion - DAC_Max function called
    min = DAC_Min(a, 0, 7);

    System.out.printf("The minimum number in " +
                      "a given array is : %d\n", min);
    System.out.printf("The maximum number in " +
                      "a given array is : %d", max);
}
}

// This code is contributed by Princi Singh
```

2. Backtracking --> Write a recursive solution

Backtracking can be defined as a general algorithmic technique that considers searching every possible combination in order to solve a computational problem.

Decision Problem – In this, we search for a feasible solution.
Optimization Problem – In this, we search for the best solution.
Enumeration Problem – In this, we find all feasible solutions.

Pseudo Code for Backtracking :

Recursive backtracking solution.

```
void findSolutions(n, other params) :
    if (found a solution) :
        solutionsFound = solutionsFound + 1;
        displaySolution();
        if (solutionsFound >= solutionTarget) :
            System.exit(0);
        return

    for (val = first to last) :
        if (isValid(val, n)) :
            applyValue(val, n);
            findSolutions(n+1, other params);
            removeValue(val, n);
```

Finding whether a solution exists or not

```
boolean findSolutions(n, other params) :
    if (found a solution) :
        displaySolution();
        return true;

    for (val = first to last) :
        if (isValid(val, n)) :
            applyValue(val, n);
            if (findSolutions(n+1, other params))
                return true;
            removeValue(val, n);
        return false;
```