Aim for excellence in Engineering
Algorithms

Introduction 4
Analysis of Algorithms 9
Asymptotic notation 10
Recurrences 13
Design Approaches 24
Divide and Conquer 25
Dynamic Programming 28
Greedy Algorithms 32
Graphs 41
Minimum Spanning Tree (MST) 46
Shortest Paths 50
Trees 65
Binary Search Trees (BST) 70
AVL Trees 73
Heaps 75
Hashing 82
Searching 86
Sorting 90
Complexity Classes 102
Introduction to Algorithms

An algorithm is a set of well-defined steps required to accomplish some task. If you’ve ever baked a cake, or followed a recipe of any kind, then you’ve used an algorithm. Algorithms also usually involve taking a system from one state to another, possibly transitioning through a series of intermediate states along the way.

An algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output. An algorithm is thus a sequence of computational steps that transform the input into the output.

An algorithm is a tool for solving a well-specified computational problem. For example, one might need to sort a sequence of numbers into non-decreasing order. This problem arises frequently in practice and provides fertile ground for introducing many standard design techniques and analysis tools. Here is how we formally define the sorting problem:

Input: A sequence of n numbers a1, a2, ..., an.

Output: A permutation (reordering) (a’1,a’2……a’n) of the input sequence such that a’1<=a’2<=….a’n.

Properties Of Algorithm

An algorithm has the following five basic properties

- A number of quantities are provided to an algorithm initially before the algorithm begins. These are the inputs that are processed by the algorithm.

- The processing rules specified in the algorithm must be precise, unambiguous and lead to a specific action. An instruction which can be carried out is called an effective instruction.

- Each instruction must be sufficiently basic such that it can be carried out in a finite time by a person with paper and pencil.

- The total time to carry out all the steps in the algorithm must be finite.

- An algorithm must have one or more output.

Fundamentals Of Algorithmic Problem Solving

- Algorithms are considered to be procedural solutions to problems.

- The solutions are not answers to the problems but specific instructions for getting the answers.
Algorithm Design & Analysis Process

1. Understand the problem

2. Decide on: computational means, exact vs. approximate solving, data structure(s), algorithm design technique

3. Design an algorithm

4. Prove correctness

5. Analyze the algorithm

6. Code the algorithm

Algorithm Design Technique

- It is a general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing.

- Reasons to study different techniques
  - They provide guidance for designing algorithms for new problems.
  - Problems that do not have satisfactory solutions.
  - Algorithm design technique makes it possible to classify algorithms according to an underlying idea.
  - They serve as a natural way to categorize and study the algorithms

Methods Of Specifying An Algorithm

- Using Natural Language
  - This method leads to ambiguity.
  - Clear description of algorithm is difficult.
• Using Pseudo codes
  • It is a mixture of natural language and programming language like constructs.
  • It is more precise than a natural language.
  • Earlier days we used another method.
• Using Flow Charts
  • It is a method of expressing an algorithm by a collection of connected geometric shapes containing the description of algorithm.

**Analyzing An Algorithm**

• Two kinds of algorithm efficiency
  • Time efficiency
    • How fast the algorithm runs.
  • Space efficiency
    • How much extra space the algorithm needs.
• Other desirable characteristics
  • Simplicity
    • Simpler algorithms are easier to understand.
    • It depends on the user.
  • Generality
    • Two issues
    • Generality of the problem the algorithm solves.
    • Range of inputs.

**IMPORTANT PROBLEM TYPES**

**Sorting**

It refers to rearranging the items of a given list in ascending order. For example,

• Sort numbers, characters, strings, records.

We need to choose a piece of information to be ordered.

• This piece of information is called a key.
• The important use of sorting is searching.
• There are many algorithms for sorting.
• Although some algorithms are indeed better than others but there is no algorithm that would be the best solution in all situations.
• The two properties of sorting algorithms are
  • Stable
  • In place

• A sorting algorithm is called stable if it preserves the relative order of any two equal elements in its input.

• An algorithm is said to be in place if it does not require extra memory, except possibly for a few memory units.

**Searching**

• It deals with finding a given value called search key, in a given set.
  • There are several algorithms ranging from sequential search to binary search.
  • Some algorithms are based on representing the underlying set in a different form more conductive to searching.
  • They are used in large databases.
  • Some algorithms work faster than others but require more memory.
  • Some are very fast only in sorted arrays.

**String Processing**

A string is a sequence of characters. We are interested in three kinds of strings

• Text strings
  • Comprises of letters, numbers and special characters

• Bit strings
  • Comprises of zeroes and ones.

• Gene sequences
  • Modeled by strings of characters from the four character alphabet A, C, G, T

• String processing algorithms have been important for computer science for a long time in conjunction with computer languages and compiling issues.

• String matching is one kind of such problem.

**Graph Problem**

• A graph can be thought of as a collection of points called vertices, some of which are connected by line segments called edges.

• They can be used for modeling wide variety of real life applications.
- Basic graph algorithm includes
- Graph traversal algorithms
- Shortest path algorithms
- Topological sorting for graphs with directed edges.

**Combinatorial Problems**

These problems ask to find a combinatorial object such as a permutation, a combination, or a subset – that satisfies certain constraints and has some desired property. These are the most difficult problems.

**Reasons**

- The number of combinatorial objects grows extremely fast with a problem’s size reaching unimaginable magnitude even for moderate sized instances.
- There are no algorithms for solving such problems exactly in an acceptable amount of time.

**Geometric Problems**

They deal with geometric objects such as points, lines, and polygons.

These algorithms are used in developing applications for computer graphics, robotics.

The method is used in radiology, archaeology, biology, geophysics, oceanography, materials science, astrophysics and other sciences.

**Numerical Problems**

These are the problems that involve mathematical objects of continuous nature:

- Solving equations, system of equations, computing definite integrals, evaluating functions.
- The majority of such problems can be solved only approximately.
- Such problems require manipulating real numbers, which can be represented in computer only approximately.
- Large number of arithmetic operation leads to round off error which can drastically distort the output.
**Analysis Of Algorithms**

There may be many different ways to solve a given problem, i.e. there may be many possible algorithms. Given a choice, which of the algorithms should be chosen? What are the possible reasons for choosing one algorithm over another?

We would like to choose the “better” algorithm. But what does it mean for an algorithm to be better? To answer this question we need to ‘analyse’ the algorithms at hand. Analysis of an algorithm is meant to predict the amount of resources it requires. In computing terms, one of the important resources is ‘time’. If an algorithm takes lesser time to complete a task, it would generally be considered the ‘better’ algorithm. These notions will be formalized mathematically.

As an example, consider the first problem that was mentioned: sorting. Given n numbers, a1, a2, …. an, we need to sort them in ascending order. A simple algorithm and its analysis is given below. The notation used will be the array notation a[1], a[2] and so on.

```plaintext
For i = 1 to n do
  pos ← i
  For j = i+1 to n do
    If (a[j] < a[pos])
      pos = j
    tmp ← a[i]
    a[i] ← a[pos]
    a[pos] ← tmp
```

In this algorithm (selection sort) we loop through the list and find the smallest element and move it to the start of the array. Then we look for the smallest element from the second position onwards, and so on. Assuming that every step takes the same amount of time, some constant ‘c’, the total running time could be expressed as:

\[
T(n) = c \left[ n + 2 \sum_{j=2}^{n} (n - j + 1) + 3 n \right] = c[4n + 2 n(n - 1)/2] = 3cn + cn^2.
\]

Here we have considered the worst case, when the condition in the inner loop is satisfied every time. So in every case the number of steps required will be at max 3n + n^2. Also note, that we have not counted the operation of incrementing i and j. Adding these will result in a change in the coefficients of n and n^2.

What happens as we run the algorithm on large arrays, i.e. the value of n is large? Of the two terms in the polynomial, the term with a lower degree becomes more or less insignificant as n increases. That is to say, the square term, n^2 (in this case) becomes more important, and for getting a practical idea of the running time within reasonable error limits, we may ignore the 3cn term altogether. This idea is formalized in the next section on asymptotic notations.
Asymptotic Notation

As noted in the previous section, all terms other than the leading term may be ignored when judging the efficiency of an algorithm. We can also ignore the leading term's constant coefficient, since constant factors are less significant than the rate of growth in determining computational efficiency for large inputs. Thus, from $cn^2 + 3cn$, we can get an abstraction of $n^2$.

We say that the running time of the selection sort algorithm is $O(n^2)$. When we look at input sizes large enough to make only the order of growth of the running time relevant, we are studying the asymptotic efficiency of algorithms. That is, we are concerned with how the running time of an algorithm increases with the size of the input in the limit, as the size of the input increases without bound.

Theta notation (in bound) $\Theta$

- $\Theta (g(n))$ is the set of functions $f(n)$ such that there exist positive constants $c_1$, $c_2$ and $n_0$ such that
  - $0 \leq c_1g(n) \leq f(n) \leq c_2g(n)$ for all $n \geq n_0$
  - $\Theta$ notation bounds a function within constant factors

Example: $3n^2 + 2n + 10 = \Theta(n^2)$

Big ‘O’ notation (upper bound) $O$

- $O (g(n))$ is the set of functions $f(n)$ such that there exist positive constants $c$ and $n_0$ such that
  - $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$
  - $O$ gives an upper bound for a function within a constant factor. In the figure $c=c_2$

NOTE: Different choice of $c$ gives different values of $n_0$. Example: $2n = O(n^2)$ as $2n \leq 1.n^2$ for all $n \geq 2$ (so here $c=1$ & $n_0 =2$)

Omega notation (lower bound) $\Omega$

- $\Omega (g(n))$ is the set of functions $f(n)$ such that there exist positive constants $c$ and $n_0$ such that
  - $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0$

NOTE: Different choice of $c$ gives different values of $n_0$

$\Omega$ gives a lower bound for a function within a constant factor. In the figure $c = c_1$

Example: $3n^2 = \Omega (n)$ as $3.n^2 \geq 3.n$ for all $n \geq 1$ (so here $c = 3$ and $n_0 =1$)
**Time Complexity**

The total number of steps involved in a solution to solve a problem is the function of the size of the problem, which is the measure of that problem’s time complexity. Some general order that we can consider

\[ O(1) < O(\log n) < O(n) < O(n \log n) < O(n^c) < O(c^n) < O(n!), \]

where \( c \geq 2 \) is some constant.

**Space Complexity**

Space complexity is measured by using polynomial amounts of memory, with an infinite amount of time.

The difference between space complexity and time complexity is that space can be reused. Space complexity is not affected by determinism or non-determinism. Amount of computer memory required during the program execution, as a function of the input size.

A small amount of space, deterministic machines can simulate nondeterministic machines, where as in time complexity, time increase exponentially in this case. A nondeterministic TM using \( O(n) \) space can be changed to a deterministic TM using only \( O^{2(n)} \) space. Generally, the efficiency of an algorithm is judged based on the time complexity, rather than space complexity for two reasons:

- Firstly time is a more valuable resource as far as computing is concerned. Space (or storage) is actually cheap.
- Secondly, most of the algorithms that are generally used do not have large space requirements. Most of them actually require space of the order \( O(n) \). So there is not much difference between the various algorithms, as far as space complexity is concerned.

Due to these reasons, when the term “efficient algorithm” is used, we generally mean a lower time complexity.

**Worst Case And Average Case**

The running time of an algorithm may vary depending on the type of input provided to it. For instance, there are sorting algorithms whose running time depends on the type of array provided. Their time complexity may vary depending on the initial arrangement of the numbers in the array. For instance the simple quick sort algorithm is not very efficient for sorting if the array happens to be initially sorted in the reverse order. But in general practical cases, quick sort is found to be efficient.

Therefore, for many algorithms, two different types of analysis need to be made. The first is the worst case, which is a measure of the worst performance possible over the set of all possible inputs. The second is the average case, which is based on a probabilistic analysis of the various types of input possible. If the algorithm has a high worst case complexity, but a good average
case complexity, it may still be a good option, because the particular worst case inputs that
deteriorate the algorithm’s performance are not expected to be encountered very often.

When the worst case and average complexities are quite different, they will be mentioned
separately as and when new algorithms are encountered. One may also consider the ‘best-case’,
i.e. time complexity for inputs for which the algorithm runs the fastest. However, this is not of
much practical significance since the best case is not expected to be encountered on a regular
basis.

**Questions**

1. Is \(2^{n+1} = O(2^n)\)
2. Is \(2^{2n} = O(2^n)\)
3. Show that for constants \(a\) and \(b > 0\), \((n + a)^b = \Theta(n^b)\)

**Solutions**

1. Here \(f(n) = 2^{n+1} = 2 \cdot 2^n\), \(g(n) = 2^n\). Taking \(c = 2\) and \(n_0 = 1\), we have \(f(n) = 2^{n+1} = 2 \cdot 2^n \leq 2 \cdot 2^n = c \cdot g(n)\) for \(n \geq n_0\). Therefore \(2^{n+1} = O(2^n)\).
2. Suppose \(2^{2n} = O(2^n)\). Then there exist \(c\) and \(n_0\) such that for all \(n \geq n_0\) \(2^{2n} \leq c \cdot 2^n\). Taking \(\lg\) on both sides, we get \(2n \leq \lg c + n\), i.e. \(n \leq \lg c\). Since the LHS will go on increasing continuously, there is no value of \(c\) that will satisfy the relation for all \(n \geq n_0\). Therefore \(2^{2n} \neq O(2^n)\)
3. We need to find \(c_1\), \(c_2\), and \(n_0\) such that for all \(n \geq n_0\),
   \[c_1 \cdot n^b \leq (n + a)^b \leq c_2 \cdot n^b.\]
   If \(a \geq 0\), choose \(c_1 = 1\) to satisfy the first inequality. Now for all \(n \geq a\), \((n+a)^b \leq (2n)^b = 2^b n^b\). Therefore choosing \(n_0 = a\), and \(c_2 = 2^b\) satisfies the relation.
   If \(a \leq 0\), choose \(c_2 = 1\) to satisfy the second inequality. Now for all \(n \geq -2a\), \((n/2)^b \geq a\). Adding \((n/2)\) on both sides we get, \(n \geq (n/2) - a\). Rearrange this to get \(n + a \geq (n/2)\). So \((n+a)^b \geq (n/2)^b = 2^{-b} n^b\). Therefore choosing \(n_0 = -2a\), \(c_1 = 2^{-b}\) satisfies the first relation as well.
   Therefore \((n + a)^b = \Theta(n^b)\) for all \(b > 0\).
Recurrences

Many algorithms are recursive in nature (as opposed to the example of selection sort which is iterative), i.e. they contain a call to themselves. The solutions are found by repeatedly breaking the problem into smaller subproblems of the same type and then combining the subsolutions to get the complete solution. Such algorithms always have a terminating condition or the base case, when the recursive call is not made. A recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs. For e.g. $T(n) = T(n - 1) + f(n)$ for some function $f(n)$ that has a closed-form sum, where $T(0) = \text{some constant}$.

Iteration Method

Recurrences of the above form can be solved using the iteration method. This means that we simply unwind the recursion: since $T(n) = T(n-1) + f(n)$, it must be the case that

- $T(n - 1) = T(n - 2) + f(n - 1)$. Then we can substitute for $T(n-1)$ in the first equation:
- $T(n) = (T(n - 2) + f(n - 1)) + f(n)$. We can continue this until we just have $T(0)$ on the right side.
- So $T(n) = T(0) + f(1) + f(2) + \ldots + f(n)$. Since $f(n)$ has a closed-form sum, the right hand side can be simplified.

Recursion Tree Method

Consider the following recurrence

$$T(n) = 2T(n/2) + cn \text{ for } n > 1, \quad T(1) = c$$

This recurrence tells that an instance of the problem of size $n = 1$ can be solved in constant time. For any value greater than 1, the time required is dependent on the time required to solve two instances of half the size. This could also be represented in asymptotic notation as:

$$T(n) = 2T(n/2) + \Theta(n) \text{ for } n > 1, \quad T(1) = \Theta(1)$$

We can solve this by drawing a recursion tree as shown below:
As the self-explanatory recursion tree shows, the time required for each level will be $cn$. The number of levels will be $\lg n$ (representation used for log with base 2). Algorithms with base 2 are generally used in algorithmic analysis. Thus we get the result $T(n) = cn \lg n = \Theta(n \lg n)$.

Note that the recursion tree is the most “natural” method available for solving most recurrence relations, as it is the most intuitive one and does not need the memorization of any formulae.

**Example:** Find asymptotic bounds for the following recurrence: $T(n) = 2T(\sqrt{n}) + \Theta(1)$ for $n > 2$; $T(2) = \Theta(1)$.

In terms of powers the recurrence is $T(n) = 2T(n^{1/2}) + \Theta(1) = 2[2T((n^{1/2})^{1/2}) + \Theta(1)] + \Theta(1)$ and so on. At level $k$ of the recursion tree (starting from level 0) the power of $n$ will be $(1/2)^k$. The recursion tree is shown below:
The terms to be added are in the enclosed curve. The summation is as follows—remember that $2^k \Theta(1) = \Theta(2^k)$. For now, we will ignore the $\Theta$ and do a summation of the terms for simplification.

$$T(n) = 1 + 2 + 2^2 + \ldots + 2^{k+1} = (2^{k+2} + 1)/(2 - 1) = 2^{k+2} + 1 \approx 2^{k+2}.$$  

Now $2 = n^{1/2^k} \rightarrow 1 = \log 2 = (1/2^k) \log n \rightarrow 2^k = \log n$.

So $T(n) = 2^{k+2} \Theta(1) = 4, 2^k \Theta(1) = 4 \log n \Theta(1) = \Theta(\log n)$

**NOTE:** While drawing the recursion tree, be careful as to what elements are to be added and what elements have actually been represented in their recursive form and do not need to be added.

**Master Method**

Provides a way for solving recurrences of the form $T(n) = aT(n/b) + f(n), a \geq 1, b > 1, f(n)$ is an asymptotically positive function.

This is the running time of an algorithm that divides a problem of size $n$ into a subproblems, each of size $n/b$. each of the sub-problems are solved recursively each in time $T(n/b)$.

According to the master theorem, such recurrences can be solved as follows:
1. If \( f(n) = O\left(n^{\log_b a - \epsilon}\right) \) for some constant \( \epsilon > 0 \), then \( T(n) = \Theta\left(n^{\log_b a}\right) \)

2. If \( f(n) = \Theta\left(n^{\log_b a}\right) \), then \( T(n) = \Theta\left(n^{\log_b a \lg n}\right) \).

3. If \( f(n) = \Omega\left(n^{\log_b (a+\epsilon)}\right) \) for some constant \( \epsilon > 0 \), and if \( af(n/b) \leq cf(n) \) for some constant \( c < 1 \) and all sufficiently large \( n \), then \( T(n) = \Theta(f(n)) \).

We can solve the recurrence \( T(n) = 2T(n/2) + cn \) using the master theorem. Here, \( a = 2 \), \( b = 2 \), \( f(n) = cn \). So \( f(n) = \Theta(n^{\log_b a}) = \Theta(n) \). Therefore, it falls in case 2. So \( T(n) = \Theta(f(n)\lg n) = \Theta(n \lg n) \).

Note that the master theorem does not cover all recurrences; there are possibilities when the problem does not fall in any of the three cases. In such cases, the master theorem cannot be used, and other methods such as recursion tree must be explored.

Some common recurrence relations and their asymptotic complexity is given below:

- \( T(n) = T(n-1) + c \) \( \Theta(n) \)
- \( T(n) = 2T(n/2) + cn \) \( \Theta(n \lg n) \)
- \( T(n) = 2T(n/2) + c \) \( \Theta(n) \)
- \( T(n) = T(n-1) + cn \) \( \Theta(n^2) \)
- \( T(n) = T(n/2) + c \) \( \Theta(\lg n) \)

**Questions**

Solve the following (1 to 4) using the master theorem:

1. \( T(n) = 4T(n/2) + n \)
2. \( T(n) = 4T(n/2) + n^2 \)
3. \( T(n) = 4T(n/2) + n^3 \)
4. \( T(n) = 2T(n/2) + n \lg n \)
5. Describe a \( \Theta(n \lg n) \)-time algorithm that, given a set \( S \) of \( n \) integers and another integer \( x \), determines whether or not there exist two elements in \( S \) whose sum is exactly \( x \).

**Solutions**

For 1, 2, 3, comparing with the general recurrence relation for the master theorem, \( T(n) = aT(n/b) + f(n) \), we have \( a = 4 \), \( b = 2 \). So \( \log_b a = \log_2 4 = 2 \), i.e. \( n^{\log_b a} = n^2 \).

1. \( f(n) = n = O(n^{2-\epsilon}) \) for \( 0 < \epsilon < 1 \). So it falls in case 1. Therefore \( T(n) = \Theta(n^2) \)

2. \( f(n) = n^2 = \Theta(n^2) \). So it falls in case 2. Therefore \( T(n) = \Theta(n^2 \lg n) \)

3. \( f(n) = n^3 = O(n^{2+\epsilon}) \) for \( 0 < \epsilon < 1 \). So it falls in case 3. Therefore \( T(n) = \Theta(n^3) \)

4. Consider question 4. We have \( a = b = 2 \). So \( n^{\log_b a} = n^1 = n \). Since \( f(n) = n \lg n \), clearly it does not belong to either case 1 or 2 of the master theorem. But does it belong to case 3? The question is: is \( (n \lg n) = \Omega(n^{1+\epsilon}) \) for some \( \epsilon > 0 \)? Since \( n^{1+\epsilon} = n.n^\epsilon \), we have \( n \) as a
common factor on both sides. So the equivalent question is whether \( \lg n = \Omega(n^\varepsilon) \) for some \( \varepsilon > 0 \)? The answer is no since the rate of growth of \( n^\varepsilon \) for any \( \varepsilon \), no matter how small, is greater than the rate of growth of \( \lg n \), as long as it is greater than zero. So this case falls in one the small gap between case 2 and 3 of the master theorem and cannot be solved by it.

5. The required complexity is \( \Theta(n \lg n) \) which means that comparing all pairs is not an option since comparing all pairs would require \( \Theta(n^2) \) time. The solution lies in sorting the elements in ascending order, which takes time \( \Theta(n \lg n) \). Now, we can start by keeping a pointer at the first (smallest) and last (largest) element. If the sum is greater than \( x \), we need to decrease it. So we move the second pointer to the second largest number. If the sum is lesser than \( x \), we need to increase it. So we move the first pointer to the second smallest number. Since this operation constitutes only one traversal of the array, it takes \( O(n) \) time. The dominating term is \( \Theta(n \lg n) \), hence the complexity of the overall algorithm is \( \Theta(n \lg n) \). Formally the algorithm is given below:

- Sort \( S \) in ascending order.
- Set \( i \leftarrow 1, j \leftarrow n \)
- While \( i < j \) do
  - If \( (S[i] + S[j] = x) \) Return TRUE
  - Else if \( (S[i] + S[j] > x) \)
    - Set \( j \leftarrow j - 1 \)
  - Else
    - Set \( i \leftarrow i + 1 \)
- Return FALSE

**Previous years GATE questions (2003)**

1. Consider the following three claims
   1. \( (n + k)^m = \Theta(n^m) \), where \( k \) and \( m \) are constants
   2. \( 2^n + 1 = O(2^n) \)
   3. \( 2^n + 1 = O(2^n) \)
   Which of these claims are correct?
   (a) 1 and 2    (b) 1 and 3    (c) 2 and 3    (d) 1, 2 and 3

2. The following are the starting and ending times of activities A, B, C, D, E, F, G; and H respectively in chronological order: “a, b, c, d, e, f, g, e, f, h, g, e, f, h, e”. Here, \( x_s \) denotes the starting time and \( x_e \) denotes the ending time of activity \( x \). We need to schedule the activities in a set of rooms available to us. An activity can be scheduled in a room only if the room is reserved for the activity for its entire duration. What is the minimum number of rooms required?
   (a) 3    (b) 4    (c) 5    (d) 6
3. Two matrices $M_1$ and $M_2$ are to be stored in arrays $A$ and $B$ respectively. Each array can be stored either in row-major or column-major order in contiguous memory locations. The time complexity of an algorithm to compute $M_1 \times M_2$ will be

(a) best if $A$ is in row-major, and $B$ is in column-major order
(b) best if both are in row-major order
(c) best if both are in column-major order
(d) independent of the storage scheme

4. Suppose each set is represented as a linked list with elements in arbitrary order. Which of the operations among union, intersection, membership, cardinality will be the slowest?

(a) union only  
(b) intersection, membership
(c) membership, cardinality  
(d) union, intersection

5. The time complexity of the following C function is (assume $n > 0$)

```c
int recursive (int n) {
    if (n == i)
        return (1);
    else
        return (recursive (n-1)+recursive (n-1));
}
```

(a) $O(n)$  
(b) $O(n \log n)$  
(c) $O(n^2)$  
(d) $O(2^n)$

6. The recurrence equation

$$T(1) = 1; T(n) = 2T(n-1) + n, n \geq 2$$

 evaluates to

(a) $2^{n+1} - n - 2$  
(b) $2^n - n$  
(c) $2^{n+1} - 2n - 2$  
(d) $2^n + n$

2005

7. Suppose $T(n) = 2T(n/2) + n$, $T(0) = T(1) = 1$. Which one of the following is FALSE?

(a) $T(n) = O(n^2)$  
(b) $T(n) = \Theta(n \log n)$  
(c) $T(n) = \Omega(n^2)$  
(d) $T(n) = O(n \log n)$

**Data for Q. 8 & Q. 9 are given below.**

Solve the problems and choose the correct answers. We are given 9 tasks $T_1$, $T_2$ ..., $T_9$. The execution of each task requires one unit of time. We can execute one task at a time. $T_i$ has a profit $P_i$ and a deadline $d_i$. Profit $p_i$ is earned if the task is completed before the end of the $d_i$th unit of time.

<table>
<thead>
<tr>
<th>Task</th>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_3$</th>
<th>$T_4$</th>
<th>$T_5$</th>
<th>$T_6$</th>
<th>$T_7$</th>
<th>$T_8$</th>
<th>$T_9$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Profit</td>
<td>15</td>
<td>20</td>
<td>30</td>
<td>18</td>
<td>18</td>
<td>10</td>
<td>23</td>
<td>16</td>
<td>25</td>
</tr>
<tr>
<td>Deadline</td>
<td>7</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>2</td>
<td>7</td>
<td>3</td>
</tr>
</tbody>
</table>

8. Are all tasks completed in the schedule that gives maximum profit?

(a) All tasks are completed  
(b) $T_1$ and $T_6$ are left out
(c) \(T_1\) and \(T_8\) are left out  
(d) \(T_4\) and \(T_6\) are left out

9. What is the maximum profit earned?
(a) 147  (b) 165  (c) 167  (d) 175

2006

10. Consider the polynomial \(p(x) = a_0 + a_1x + a_2x^2 + a_3x^3\), where \(a_i \neq 0\), for all \(i\). The minimum number of multiplications needed to evaluate \(p\) on an input \(x\) is
(a) 3  (b) 4  (c) 6  (d) 9

11. Consider the following C-program fragment in which \(i\), \(j\), and \(n\) are integer variables.

```c
for (i=n,j = 0; i > 0; if =2,j +=i);
```

Let \(\text{Val}(j)\) denote the value stored in the variable \(j\) after termination of the for loop. Which one of the following is true?
(a) \(\text{val}(j) = \Theta(\log n)\)  (b) \(\text{val}(j) = \Theta(\sqrt{n})\)  (c) \(\text{val}(j) = \Theta(n)\)  (d) \(\text{val}(j) = \Theta(n \log n)\)

12. A set \(X\) can be represented by an array \(x[n]\) as follows:

\[
x[i] = \begin{cases} 
1 & \text{If } i \in X \\
0 & \text{otherwise} 
\end{cases}
\]

Consider the following algorithm in which \(x\), \(y\), and \(z\) are boolean arrays of size \(n\):

```c
algorithm zzz (x[], y[], z[]){
    int i;
    for (i = 0; i < n; ++ i)
        z[i] = (x[i] ^ ~y[i]) \lor (~x[i] ^ y[i])
}
```

The set \(Z\) computed by the algorithm is
(a) \((X \cup Y)\)  (b) \((X \cap Y)\)  (c) \((X-Y) \cap (Y-X)\)  (d) \((X-Y) \cup (Y-X)\)

13. Consider the following recurrence:

\(T(n) = 2T(\lceil \sqrt{n} \rceil) + 1, T(1) = 1\)

Which one of the following is true?
(a) \(T(n) = \Theta(\log \log n)\)  (b) \(T(n) = \Theta(\log n)\)  (c) \(T(n) = \Theta(\sqrt{n})\)  (d) \(T(n) = \Theta(n)\)
2007

14. What is the time complexity of the following recursive function:

```c
int DoSomething (int n) {
    if (n <= 2)
        return 1;
    else
        return (DoSomething (floor (sqrt (n))) + n);
}
```

(a) Θ(n^2)  (b) Θ(n log n)  (c) Θ(log n)  (d) Θ(log log n)

15. An array of n numbers is given, where n is an even number. The maximum as well as the minimum of these n numbers needs to be determined. Which of the following is TRUE about the number of comparisons needed?

(a) At least 2n - c comparisons, for some constant c, are needed.
(b) At most 1.5n - 2 comparisons are needed.
(c) At least n log n comparisons are needed.
(d) None of the above

2008

16. Consider the following functions: f(n) = 2^n; g(n) = n!; h(n) = n^log n. Which of the following statements about the asymptotic behavior of f(n), g(n), and h(n) is true?

(a) f(n) = O(g(n)); g(n) = O(h(n))  (b) f(n) = Ω(g(n)); g(n) = O(h(n))
(c) g(n) = O(f(n)); h(n) = O(f(n))  (d) h(n) = O(f(n)); g(n) = Ω(f(n))

2009

17. The running time of an algorithm is represented by the following recurrence relation:

```
T(n) = \begin{cases} 
n & \text{n \leq 3} 
\frac{n}{3} + cn & \text{otherwise} 
\end{cases}
```

Which one of the following represents the time complexity of the algorithm?
(a) Θ(n)  (b) Θ(n log n)  (c) Θ(n^2)  (d) Θ(n^2 log n)

2010

18. Two alternative packages A and B are available for processing a database having 10^k records. Package A requires 0.0001 n^2 time units and package B requires 10n log_{10} n time units to process n records. What is the smallest value of k for which package B will be preferred over A?

(a) 12  (b) 10  (c) 6  (d) 5

19. The weight of a sequence a_0 a_1 ... a_{n-1} of real numbers is defined as a_0 + a_1/2 + ... + a_{n-1}/2^{n-1}. A subsequence of a sequence is obtained by deleting some elements from the sequence,
keeping the order of the remaining elements the same. Let X denote the maximum possible weight of a subsequence of \(a_0 \ a_1 \ldots \ \ a_{n-1}\) and Y the maximum possible weight of a subsequence of \(a_1 \ a_2 \ldots \ \ a_{n-1}\). Then X is equal to:

(a) \(\max (Y, a_0 + Y)\)  
(b) \(\max (Y, a_0 + Y/2)\)  
(c) \(\max (Y, a_0 + 2Y)\)  
(d) \(a_0 + Y/2\)

**Solutions**

1. (a)  2. (b)  3. (d)  4. (d)  5. (d)  6. (a)  7. (c)  8. (d)  9. (a)  10. (a)  11. (c)  12. (d)  13. (a)  14. (d)  15. (b)  16. (d)  17. (a)  18. (c)  19. (b)

**Explanations**

1. (a)

1 is true. Statement 2: \(2^{n+1} = 2.2^n = O(2^n)\). So 2 is true.
Statement 3: \(2^{2n+1} = 2.2^{2n} = O(2^n)\). So 3 is false.

2. (b)

The gap between starting and ending time of B is greater than all other activity. Consider the order b, c, a, d, e, f, b; the gap between b_a to b_e is 6 in which two activity a_e and c_e ended so minimum number of required rooms is 4.

3. (d)

Access time is equal for all element if it is stored in an array.

4. (d)

If each set is represented as a linked list with elements in arbitrary order. Membership and cardinality takes O(n) time for an element in the set. For union and intersection, duplicate elements must be checked for. To do this effectively, we could traverse one linked list and store it in a balanced binary tree, and then match with the elements of the second for duplicates. The best we can do is O(n log n).

5. (d)

The function is recursive with the base case being \(T(1) = 1\). The recurrence is \(T(n) = 2T(n-1)\). Expanding this, we get \(T(n) = 2T(n-1) = 4T(n-2) = \ldots = 2^{n-1} T(1) = 2^{n-1} = O(2^n)\)

6. (a)

\[ T(1) = 1 \]
\[ T(n) = 2T(n-1) + n \text{ for } n > 1 \]
\[ T(2) = 2T(1) + 2 = 2.1 + 2 = 4 \]
\[ T(3) = 2T(2) + 3 = 2.4 + 3 = 11 \]
\[ T(4) = 2T(3) + 4 = 2.11 + 4 = 26 \]
So \(T(n) = 2^{n+1} - n - 2\) since this equation satisfies the above values.
7. (c)
This is a standard recurrence with the result $T(n) = \Theta(n \log n)$. Therefore $O(n^2)$ is just another upper bound that is loose. $T(n) = O(n \log n)$ is also true. But $T(n) = \Omega(n^2)$ is false since $\Omega(n^2)$ represents a higher lower bound than $\Theta(n \log n)$ since $n^2$ dominates $n \log n$.

8. (d)
J is initially empty then according to deadlines it includes $\{T_1, T_2, T_3, T_5, T_7, T_8, T_9\}$. So $T_4$ and $T_6$ can't be included in J.

9. (a)
Total profit earned = 15+20+30+18+23+16+25 = 147

10. (a)
\[ p(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 = p(x) = a_0 + x(a_1 + x(a_2 + a_3 x)) \]
We evaluate the in the following order: t = $a_3 \times x$; t = t+a_2; t = t*x; t = t+a_1; t = t*x; t = t+a_0

11. (c)
The value of ‘i’ across loops is n, n/2, n/4, ... So after termination of loop, $j = n + (n/2) + (n/4) + ... (n/2^{\log n}) = \Theta(n)$

12. (d)
In given algorithm the ‘for’ loop contains a logical expression. The equivalent set representation is: $z = (x \cap y') \cup (y \cap x')$

13. (a)
We know that $\log_2 2 = 1$. So, all the level sums are equal to $\log_2 2$. The problem size at level k of the recursion tree is $(n^2)^k$ and when we stop recursing this value is a constant. Setting $(n^2)^k = 2$ and solving for k gives $2^k \log n = 1$ and $k = \log \log n$. Hence $T(n) = \Theta(\log \log n)$

14. (d)
The recursion is as follows: $T(1) = T(2) = 1$. $T(n) = \sqrt{n} + 1$ for $n > 2$. With the same analysis as the previous question, we get $T(n) = \Theta(\log \log n)$

15. (b)
This can be solved by dividing the array into pairs $A[1,2]$, $A[2,3]$ ... $A[n-1, n]$. We compare $A[1]$ and $A[2]$ and assign the larger one to L and smaller one to S. For each pair, we compare the larger one with L and the smaller one with S to update L and S. Thus for every pair, we perform three comparisons: one between the elements of the pair to find the larger one, one with L and one with S. So the number of comparisons is $(3/2)n - 2$ since the first pair needs only one comparison.

16. (d)
The asymptotic order of functions is $n^{\log n} < c^n < n!$ Hence, the result.
17. (a)
Using Master theorem, \(a = 1, b = 3, \log_b a = \log_3 1 = 0, f(n) = cn = \Theta(n)\). It belongs to the 3rd case where solution is \(T(n) = \Theta(f(n)) = \Theta(n)\).

18. (c)
We need the minimum \(k\) such that, \(0.0001 \, n^2 > 10 \, n \log_{10} n\)
\(n > 10^5 \log_{10} n\). Putting \(n = 10^k\), we get \(10^k > 10^5 \log_{10} 10^k\).
\(10^k > 10^5 \, k\). The minimum \(k\) satisfying this equation is 6.

19. (b)
Let the sub-sequence corresponding to \(Y\) be \(a_i, a_j, \ldots, a_z\). So \(Y = a_i + a_j/2 + \ldots + a_z/2^c\).
Suppose the sub-sequence corresponding to \(X\) contains \(a_0\). Then \(X = a_0 + k\). The maximum \(k\) will correspond to the sub-sequence of \(Y\), with \(k = a_i/2 + a_j/4 + \ldots + a_z/2^{c+1} = Y/2\). So \(X = a_0 + Y/2\).
If the sub-sequence corresponding to \(X\) does not contain \(a_0\), then \(X = Y\). So \(X\) is the maximum of the two values: \(Y\) and \(a_0 + Y/2\).