UNIT – 1 INTRODUCTION TO ALGORITHM

UNIT STRUCTURE

1.1 LEARNING OBJECTIVES

After going through this unit, you will be able to:

- understand the concept of algorithm
- know the process of algorithm analysis
- know the notations for defining the complexity of algorithm
- learnthe method to calculate time complexity of algorithm
- know the different operations on disjoint set
- learn methods for sorting data in linear time

1.2 INTRODUCTION

The concept of an algorithm is the basic need for any programming development in computer science. Algorithm exists for many common problems, but designing an efficient algorithm is a challenge and it plays a crucial role in large scale computer system. In this unit we will discuss about the algorithm analysis. Also we will discuss few algorithms for sorting data in linear time. We will also discuss algorithm for disjoint set operations.

1.3DEFINITION OF ALGORITHM

Definition: An algorithm is a well-defined computational method, which takes some value(s) as input and produces some value(s) as output.In other words, an algorithm is a sequence of computational steps that transforms input(s) into output(s).

Each algorithm must have

- **Specification:** Description of the computational procedure.
- **Pre-conditions:** The condition(s) on input.
- **Body of the Algorithm:** A sequence of clear and unambiguous instructions.
- **Post-conditions**: The condition(s) on output.

Consider a simple algorithm for finding the factorial of ***n***.

Algorithm Factorial (n)

```
Step 1: FACT = 1
Step 2: for i = 1 to n do
Step 3: FACT = FACT * i
Step 4: print FACT
```

In the above algorithm we have:

Specification: Computes *n*!.
Pre-condition: *n*>= 0
Post-condition: FACT = *n*!

1.4 ALGORITHM ANALYSIS

Programming is a very complex task, and there are a number of aspects of programming that make it so complex. The first is that most programming projects are very large, requiring the coordinated efforts of many people. (This is the case of software engineering.) The next is that manyprogramming projects involve storing and accessing large quantities of data efficiently. (This is the case of data structures and databases.) The last is that many programming projects involve solving complexcomputational problems, for which simplistic or naive solutions may not be efficient enough. The complexproblems may involve numerical data which need to computed accurately up to high precision (in caseof numerical analysis). This is where the topic of algorithm design and analysis is important.

Although the algorithms discussed in this course will often represent only a tiny fraction of the code that isgenerated in a large software system, this small fraction may be very important for the success of the overallproject.

If unfortunately someonedesign an inefficient algorithm anddata structure to solve the problem, and then take the poor design and attempt to fine-tune its performance, then often no amount of fine-tuning is going to make a substantialdifference. So at the design phase of the algorithm itself care should be taken to design an efficient algorithm.

The focus of this course is on how to design good algorithms, and how to analyze their efficiency. This is one of the most basic aspects of good programming.

1.5 COMPLEXITY

Once we develop an algorithm, it is always better to check whether the algorithm is efficient or not. The efficiency of an algorithm depends on the following factors:

- Accuracy of the output
- Robustness of the algorithm
- User friendliness of the algorithm
- Time required to run the algorithm
- Space required to run the algorithm
- Reliability of the algorithm
- Extensibility of the algorithm

To be a good program, all the above mentioned factors are very important. When we design some algorithm it should be user friendly and produce correct output(s) for all the possible set of input(s). A well designed algorithm should not take very long amount of time and also it should not uselarge amount of main memory. A well design algorithm is always reliable and it can be extended as per requirement.

In case of complexity analysis, we mainly concentrate on the time and space required by a program to execute.So complexity analysis is broadly categorized into two classes

- Space complexity
- Time complexity

1.5.1 SPACE COMPLEXITY

Now a day's,memory is becoming more and more cheaper, even though it is very much important to analyze the amount of memory used by a program. Because, if the algorithm takes memory beyond the capacity of the machine, thenthe algorithm will not beable to execute. So, it is very much important to analyze the space complexity before execute it on the computer.

Definition [SpaceComplexity]: The Space complexity of an algorithm is the amount of main memory needed to run the program till completion.

To measure the space complexity in absolute memory unit has the following problems

The space required for an algorithm depends on space required by the machine during execution, they are

i) Programspace
ii) Data space.

i) The program space is fixed and it is used to store the temporary data, object code,etc.
ii) The data space is used to store the different variables, data structures defined in the program.

In case of analysis we consider only the data space, since program space is fixed and dependon the machine where it is executed.

Consider the following algorithms for exchange two numbers:

```
Algo1_exchange (a, b)

Step 1: tmp = a;
Step 2: a = b;
Step 3: b = tmp;
```

```
Algo2_exchange (a, b)

Step 1: a = a + b;
Step 2: b = a - b;
Step 3: a = a - b;
```

The first algorithm uses three variables a, b and tmpand the second one take only two variables, so if we look from the space complexity perspective the second algorithm is better than the first one.

1.5.2TIME COMPLEXITY

Definition [Time Complexity]: The Time complexity of an algorithm is the amount of computer time it needs to run the program till completion.

To measure the time complexity in absolute time unit has the following problems

1. The time required for an algorithm depends on number of instructions executed by the algorithm.
2. The execution time of an instruction depends on computer's power. Since, different computers take different amount of time for the same instruction.
3. Different types of instructions take different amount of time on same computer.

For time complexity analysis we design a machine by removing all the machine dependent factors called Random Access Machine (RAM). The random access machine model of computation was devised by John von Neumann to study algorithms. The design of RAM is as follows

1. Each “simple” operation (+, -, =, if, call) takes exactly 1 unit cost.
2. Loops and subroutine calls are not simple operations, they depend upon the size of the data and the contents of a subroutine.
3. Each memory access takes exactly 1 unit cost.

Consider the following algorithm for add two number

```
Algo_add (a,b)

Step 1. C = a + b;
Step 2.return C;
```

Here this algorithm has only two simple statements so the complexity of this algorithm is 2

Consider another algorithm for add n even number

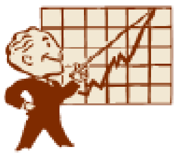
```
Algo_addeven (n)

Step 1. i = 2;
Step 2.sum = 0;
Step 3.while i <= 2*n
Step 4.sum = sum + i
Step 5. i = i + 2;
Step 6.end while;
Step 7.return sum;
```

Here,
Step 1, Step 2 and Step 7 are simple statements and they will execute only once.
Step 3 is a loop statement and it will execute as many times as the loop condition is true and one more time for check the condition is false.
Step 5 and Step 6 are inside the loop so it will run as much as the loop condition is true
Step 6 just indicate the end of while and no cost associated with it.

Statement	**Cost**	**Frequency**	**Total cost**
Step 1. i = 2;	1	1	1
Step 2.sum = 0;	1	1	1
Step 3.while i <= 2*n	1	n+1	n+1
Step 4.sum = sum + i	1	n	n
Step 5. i = i + 2;	1	n	n
Step 6.end while;	0	1	0
Step 7.return sum;	1	1	1

Total cost 3n+4



CHECK YOUR PROGRESS

1. State True or False
   - a) Time complexity is the time taken to design an algorithm.
   - b) Space complexity is the amount of space required by a program during execution
   - c) An algorithm may not produce any output.
   - d) Algorithm are computer programs which can be directly run into the computer.
   - e) If an algorithm is designed for a problem then it will work for all the valid inputs for the problem.

1.6ASYMPTOTIC NOTATION

When we calculate the complexity of an algorithm we often get a complex polynomial. To simplify this complex polynomial we use some notation to represent the complexity of an algorithm called: Asymptotic Notation.

Θ (Theta) Notation

For a given function g(n), Θ(g(n)) is defined as

$$\Theta(g(n)) = \begin{cases} f(n) : \text{there exist constants } c_1 > 0,\ c_2 > 0 \text{ and } n_0 \epsilon\ N \\ \quad \text{such that } 0 \le c_1\, g(n) \le f(n) \le c_2\, g(n) \text{ for all } n \ge n_0 \end{cases}$$

In other words a function f(n) is said to belongs to Θ(g(n)), if there exists positive constants c_1 and c_2 such that $0 \le c_1\, g(n) \le f(n) \le c_2 g(n)$ for sufficiently large value of n. Fig 1.1 gives a intuitive picture of functions f(n) and g(n), where f(n) = Θ (g(n)). For all the values of n at and to right of n_0, the values of f(n) lies at or above $c_1 g(n)$and at or below $c_2 g(n)$.In other words, for all n $\ge n_0$, the function f(n) is equal to g(n) to within a constant factor. So,g(n) is said an **asymptotically tight bound** for f(n).

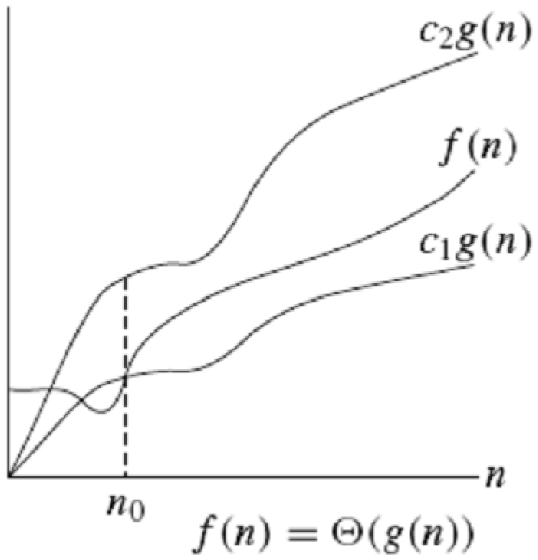


Fig 1.1 : Graphic Example of Θ notation.

For example

$f(n) = ½\, n^2 - 3\, n$

let, $g(n) = n^2$

to proof f(n) = Θ (g(n)) we must determine the positive constants c_1, c_2 and n_0 such that

$c_1\, n^2 \le ½\, n^2 - 3\, n \le c_2\, n^2$ for all $n \ge n_0$

dividing the whole equation by n^2, we get

$$c_1 \le ½ - 3/n \le c_2$$

We can make the right hand inequality hold for any value of $n \ge 1$ by choosing $c_2 \ge ½$. Similarly we can make the left hand inequality hold for any value of $n \ge 7$ by choosing $c_1 \le {}^1/_{14}$. Thus, by choosing $c_1 = {}^1/_{14}$, $c_2 = ½$. And $n_0 = 7$ we can have $f(n) = \Theta(g(n))$. That is $½n^2 - 3n = \Theta(n^2)$.

O (Big O) Notation

For a given function g(n), O(g(n)) is defined as

$$O(g(n)) = \begin{cases} f(n): \text{there exist constants } c > 0, \text{ and } n_0 \in N \\ \quad \text{such that } 0 \le f(n) \le c\, g(n) \text{ for all } n \ge n_0 \end{cases}$$

In other words a function f(n) is said to belongs to O(g(n)), if there exists positive constant c such that $0 \le f(n) \le c\,g(n)$ for sufficiently large value of n.Fig 1.2 gives a intuitive picture of functions f(n) and g(n), where $f(n) = O(g(n))$. For all the values of n at and to the right of n_0, the values of f(n) lies at or below cg(n). Sog(n) is said an **asymptotically upper bound** for f(n).

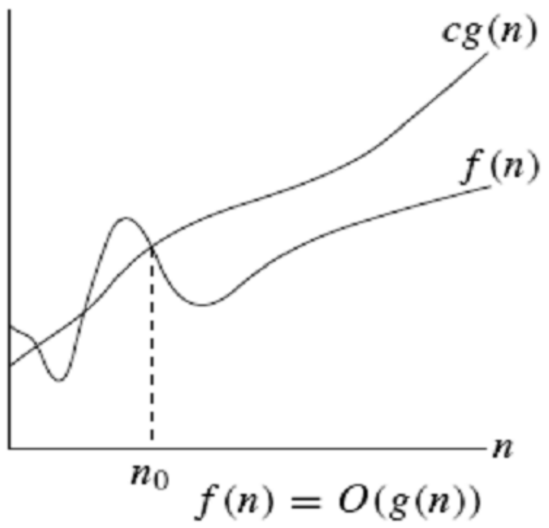


Fig 1.2 : Graphic Example of O notation.

Ω (Big Omega) Notation

For a given function g(n), Ω (g(n)) is defined as

$$\Omega(g(n)) = \begin{cases} f(n): \text{there exist constants } c > 0, \text{ and } n_0 \in N \\ \quad \text{such that } 0 \le c\, g(n) \le f(n) \text{ for all } n \ge n_0 \end{cases}$$

In other words, a function f(n) is said to belongs to Ω (g(n)), if there exists positive constant c such that 0 ≤ c g(n) ≤ f(n) for sufficiently large value of n. Fig 1.3 gives a intuitive picture of functions f(n) and g(n), where f(n) = Ω (g(n)). For all the values of n at and to the right of n_0, the values of f(n) lies at or above cg(n). Sog(n) is said an **asymptotically lower bound** for f(n).

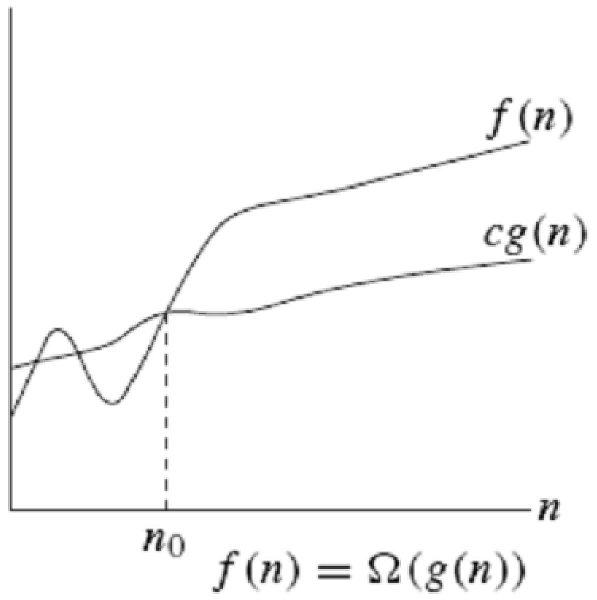


Fig 1.3 : Graphic Example of notation

The growth patterns of order notations have been listed below:

O(1) < O(log(n)) < O(n) < O(n log(n)) < O(n^2) < O(n^3)… <O(2^n).

The common name of few order notations is listed below:

Notation	Name
O(1)	Constant
O(log(n))	Logarithmic
O(n)	Linear
O(n log(n))	Linearithmic
O(n^2)	Quadratic
O(c^n)	Exponential
O(n!)	Factorial

A Comparison of typical running time of different order notations for different input size listed below:

$\log_2 n$	n	$n\log_2 n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65536
5	32	160	1024	32768	4294967296

Now let us take few examples of above asymptotic notations

1. **Prove that $3n^3 + 2n^2 + 4n + 3 = O(n^3)$**

Here,

$f(n) = 3n^3 + 2n^2 + 4n + 3$
$g(n) = O(n^3)$

to proof f(n) = O (g(n)) we must determine the positive constants c and n_0 such that

$$3n^3 + 2n^2 + 4n + 3 \le c\, n^3 \qquad \text{for all } n \ge n_0$$

dividing the whole equation by n^3, we get

$$3 + 2/n + 4/n^2 + 3/n^3 \le c$$

We can make the inequality hold for any value of $n \ge 1$ by choosing $c \ge 12$. Thus, by choosing $c \ge 12$and $n_0 = 1$ we can have
f(n) = O(g(n)).

Thus,$3n^3 + 2n^2 + 4n + 3 = O(n^3)$.

2. **Prove that $3n^3 + 2n^2 + 4n + 3 = \Omega(n^3)$**

Here,

$f(n) = 3n^3 + 2n^2 + 4n + 3$
$g(n) = O(n^3)$

to proof f(n) = Ω (g(n)) we must determine the positive constants c and n_0 such that

$$c\, n^3 \le 3n^3 + 2n^2 + 4n + 3 \qquad \text{for all } n \ge n_0$$

dividing the whole equation by n^3, we get

$$c \le 3 + 2/n + 4/n^2 + 3/n^3$$

We can make the inequality hold for any value of $n \ge 1$ by choosing $c \le 3$. Thus, by choosing $c = 3$ and $n_0 = 1$ we can have
f(n) = Ω (g(n)).

Thus, $3n^3 + 2n^2 + 4n + 3 = \Omega(n^3)$.

3. **Prove that $7n^3 + 7 = \Theta(n^3)$**

Here,

$f(n) = 7n^3 + 7$
$g(n) = O(n^3)$

to proof f(n) = Θ (g(n)) we must determine the positive constants c_1, c_2 and n_0 such that

$$c_1 n^3 \le 7n^3 + 7 \le c_2 n^3 \text{ for all } n \ge n_0$$

dividing the whole equation by n^3, we get

$$c_1 \le 7 + 7/n^3 \le c_2$$

We can make the right hand inequality hold for any value of $n \ge 1$ by choosing $c_2 \ge 14$. Similarly we can make the left hand inequality hold for any value of $n \ge 1$ by choosing $c_1 \le 7$. Thus, by choosing $c_1 = 7$, $c_2 = 14$. And $n_0 = 1$ we have $f(n) = \Theta(g(n))$.

Thus,$7n^3 + 7 = \Theta(n^3)$.

Now let us take few examples of Algorithms and represent their complexity in asymptotic notations

Example 1. Consider the following algorithm to find out the sum of all the elements in an array

Statement	**Cost**	**Frequency**	**Total cost**
Sum_Array(arr[], n)			
Step 1. i = 0;	1	1	1
Step 2. s = 0;	1	1	1
Step 3.while i < n	1	n+1	n+1
Step 4. s = s + arr [i]	1	n	n
Step 5. i = i + 1;	1	n	n
Step 6.end while;	0	1	0
Step 7.return s;	1	1	1

Total Cost 3n + 4

So,
Here $f(n) = 3n + 4$
Let, $g(n) = n$
If we want to represent it in O notation then we have to show that for some positive constant c and n_0

$$0 \le f(n) \le c\, g(n)$$
$$\Rightarrow 0 \le 3n + 4 \le c\, n$$

Now if we take n = 1 and c = 7

$$\Rightarrow 0 \le 3 \times 1 + 4 \le 7 \times 1$$

Which is true, so we can say that for $n_0 = 1$ and $c = 7$

$f(n) = O(g(n))$ that is
$3n+4 = O(n)$

Example 2. Consider the following algorithm to addtwo square matrix.

Statement	Cost	Frequency	Total cost
Mat_add(a[],n,b[])			
Step 1. i = 0	1	1	1
Step 2. j = 0;	1	1	1
Step 3.while i < n	1	n+1	n+1
Step 4.while j < n	1	n(n+1)	n(n+1)
Step 5.c[i][j] = a[i][j] + b[i][j]	1	n*n	n*n
Step 6. j = j + 1	1	n*n	n*n
Step 7.end inner while;	0	n	0
Step 8. i = I + 1	1	n	n
Step 9.end outer while	0	1	0
Step 10.return c;	1	1	1

Total Cost $3n^2 + 3n + 4$

Here $f(n) = 3n^2 + 3n + 4$

Let, $g(n) = n^2$

If we want to represent it in Ω notation then we have to show that for some positive constant c and n_0

$$0 \le c\, g(n) \le f(n)$$

$$\Rightarrow 0 \le c\, n^2 \le 3n^2 + 3n + 4$$

Now if we take n = 1 and c = 3

$$\Rightarrow 0 \le 3 \times 1 \le 3 \times 1^2 + 3 \times 1 + 4$$

Which is true, so we can say that for $n_0 = 1$ and c = 3,

$f(n) = \Omega\,(g(n))$

i.e.$3n^2 + 3n + 4 = O(n^2)$

In analysis of algorithms three different cases may be considered depending on the input to the algorithms.These are,

Worst Case: This is the upper bound for execution time with anyinput(s). It guarantees that irrespective of the type of input(s), the algorithm will not take any longer than the worst case time.

Best Case: This is the lower bound for execution time with any input(s). It guarantees that under any circumstances,the algorithm will beexecuted at leastfor best case time. That is the minimum time required by the algorithm to execute for any input.

Average case: This is the execution time taken by thealgorithm forany random input(s)to the algorithm. In this case, for the inputs, the algorithm takes a time which is in between the upper and lower bound.

Example 2. Consider the following Insertion sort algorithm

Algorithm Insertion_Sort (a[n])

```
Step 1: i = 2
Step 2: while i < n
Step 3: num = a[i]
Step 4: j = i
Step 5: while (( j>1) && (a[j-1] >num))
Step 6: a[j] = a[j-1]
Step 7: j = j-1
Step 8: end while (inner)
Step 9: a[j] = num
Step 10: i = i + 1
Step 11: end while (outer)
```

Worst case Analysis of Insertion Sort

In worst case, inputs to the algorithm will be reversely sorted.Sothe loop statementswill run for maximum time. In worst case, every time we will find a[j-1]>num in statement 5 as true, so statement 5 will run for 2 + 3 + 4 + … + n times total n(n+1) - 1 times. Statement 6 will run for 1 + 2 + 3 + … + n-1 times total n(n-1) times. Same time as statement 6 will be taken by statement 7.

Statement	Cost	Frequency	Total cost
Step 1	1	1	1
Step 2	1	n	n
Step 3	1	n-1	n-1
Step 4	1	n-1	n-1
Step 5	1	n(n+1)-1	n(n+1)-1
Step 6	1	n(n-1)	n(n-1)
Step 7	1	n(n-1)	n(n-1)
Step 8	0	n-1	0
Step 9	1	n-1	n-1
Step 10	1	n-1	n-1
Step 11	0	1	0

Total Cost $3n^2 + 4n - 4$

Here $f(n) = 3n^2 + 4n - 4$
Let, $g(n) = n^2$

If we want to represent it in O notation then we have to show that for some positive constant c and n_0, the following must be true,

$$0 \le f(n) \le c\, g(n)$$

$\Rightarrow 0 \le 3n^2 + 4n - 4 \le c\, n^2$

Now if we take n = 1 and c = 7

$\Rightarrow 0 \le 3 x 1^2 + 4 x 1 - 4 \le 7 \times 1^2$

Which is true.So we can say that for $n_0 = 1$ and c = 7

$f(n) = O(g(n))$

i.e $3n^2 + 4n - 4 = O(n^2)$

The worst case time complexity of insertion sort is $O(n^2)$.

Average case Analysis of Insertion Sort

In Average case, inputs to the algorithm will be random. Here, half of the time we will find a[j-1]>num is true and false in other half. So statement 5 will run for (2 + 3 + 4 + … + n)/2 times total (n(n+1)–1)/2 times. Statement 6 will run for (1 + 2 + 3 + … + n-1)/2 times total (n(n-1))/2 times. Same for statement 7 as statement 6.

Statement	Cost	Frequency	Total cost
Step 1	1	1	1
Step 2	1	n	n
Step 3	1	n-1	n-1
Step 4	1	n-1	n-1
Step 5	1	(n(n+1)-1)/2	(n(n+1)-1)/2
Step 6	1	(n(n-1))/2	(n(n-1))/2
Step 7	1	(n(n-1))/2	(n(n-1))/2
Step 8	0	n-1	0
Step 9	1	n-1	n-1
Step 10	1	n-1	n-1
Step 11	0	1	0

Total Cost $\frac{3}{2}n^2 + \frac{7}{2}n - 4$

Now, $\frac{3}{2}n^2 + \frac{7}{2}n - 4 = O(n^2)$

The average case time complexity of insertion sort is $O(n^2)$.

Best case Analysis of Insertion Sort

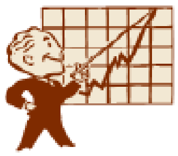
In best case, inputs will be already sorted. So a[j-1]>num will be false always. Statement 5 will run for n times (only to check the condition is false). Statement 6 will run for 0 times since while loop will be false always. Statement 7 will also run for same times as statement 6.

Statement	Cost	Frequency	Total cost
Step 1	1	1	1
Step 2	1	n	n
Step 3	1	n-1	n-1
Step 4	1	n-1	n-1
Step 5	1	n	n
Step 6	1	0	0
Step 7	1	0	0
Step 8	0	n-1	0
Step 9	1	n-1	n-1
Step 10	1	n-1	n-1
Step 11	0	1	0

Total Cost 5n- 3

Now 5n-3 = O(n)

The best case time complexity of insertion sort is O(n)



CHECK YOUR PROGRESS

2. State True or False.

a) $7 n^3 + 4n + 27 = O (n^3)$

b) $2n^2 + 34 = \Omega (n^3)$

c) $2n^2 + 34 = O (n^3)$

d) $2n^2 + 34 = \Theta (n^3)$

e) $2n^2 + 34 = \Omega (n)$

f) $2n^2 + 34 = \Theta (n^2)$

g) $2n^7 + 4n^3 + 2n = \Omega (n^3)$

h) $2n^4 + 3n^3 + 17n^2 = O (n^3)$

1.7 HEAPS AND HEAP SORT