UNIT – 1
INTRODUCTION

1.1 Notion of Algorithm

Need for studying algorithms:

The study of algorithms is the cornerstone of computer science. It can be recognized as the core of computer science. Computer programs would not exist without algorithms. With computers becoming an essential part of our professional & personal life’s, studying algorithms becomes a necessity, more so for computer science engineers. Another reason for studying algorithms is that if we know a standard set of important algorithms, They further our analytical skills & help us in developing new algorithms for required applications

Algorithm

An algorithm is finite set of instructions that is followed, accomplishes a particular task. In addition, all algorithms must satisfy the following criteria:

1. Input. Zero or more quantities are externally supplied.
2. Output. At least one quantity is produced.
3. Definiteness. Each instruction is clear and produced.
4. Finiteness. If we trace out the instruction of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
5. Effectiveness. Every instruction must be very basic so that it can be carried out, in principal, by a person using only pencil and paper. It is not enough that each operation be definite as in criterion 3; it also must be feasible.
An algorithm is composed of a finite set of steps, each of which may require one or more operations. The possibility of a computer carrying out these operations necessitates that certain constraints be placed on the type of operations an algorithm can include. The fourth criterion for algorithms we assume in this book is that they terminate after a finite number of operations.

Criterion 5 requires that each operation be effective; each step must be such that it can, at least in principal, be done by a person using pencil and paper in a finite amount of time. Performing arithmetic on integers is an example of effective operation, but arithmetic with real numbers is not, since some values may be expressible only by infinitely long decimal expansion. Adding two such numbers would violet the effectiveness property.

- Algorithms that are definite and effective are also called computational procedures.
- The same algorithm can be represented in several ways.
- Several algorithms to solve the same problem
- Different ideas different speed

Example:

Problem: GCD of Two numbers m,n
Input specification: Two inputs, nonnegative, not both zero
Euclids algorithm
-gcd(m,n)=gcd(n,m mod n)
Untill m mod n =0,since gcd(m,0) =m

Another way of representation of the same algorithm

**Euclids algorithm**

Step1: if n=0 return value of m & stop else proceed step 2
Step 2: Divide m by n & assign the value of remainder to r
Step 3: Assign the value of n to m, r to n, Go to step 1.

Another algorithm to solve the same problem

**Euclids algorithm**

Step1: Assign the value of min(m,n) to t
Step 2: Divide m by t, if remainder is 0, go to step 3 else goto step 4
Step 3: Divide n by t, if the remainder is 0, return the value of t as the answer and stop, otherwise proceed to step 4
Step 4: Decrease the value of t by 1, go to step 2
1.2 Review of Asymptotic Notation

Fundamentals of the analysis of algorithm efficiency

- Analysis of algorithms means to investigate an algorithm’s efficiency with respect to resources:
- running time (time efficiency)
- memory space (space efficiency)

Time being more critical than space, we concentrate on Time efficiency of algorithms. The theory developed, holds good for space complexity also.

Experimental Studies: requires writing a program implementing the algorithm and running the program with inputs of varying size and composition. It uses a function, like the built-in clock() function, to get an accurate measure of the actual running time, then analysis is done by plotting the results.

Limitations of Experiments

- It is necessary to implement the algorithm, which may be difficult
- Results may not be indicative of the running time on other inputs not included in the experiment.
- In order to compare two algorithms, the same hardware and software environments must be used

Theoretical Analysis: It uses a high-level description of the algorithm instead of an implementation. Analysis characterizes running time as a function of the input size, \( n \), and takes into account all possible inputs. This allows us to evaluate the speed of an algorithm independent of the hardware/software environment. Therefore theoretical analysis can be used for analyzing any algorithm

Framework for Analysis

We use a hypothetical model with following assumptions

- Total time taken by the algorithm is given as a function on its input size
- Logical units are identified as one step
- Every step require ONE unit of time
- Total time taken = Total Num. of steps executed

Input’s size: Time required by an algorithm is proportional to size of the problem instance. For e.g., more time is required to sort 20 elements than what is required to sort 10
Units for Measuring Running Time: Count the number of times an algorithm’s basic operation is executed. (Basic operation: The most important operation of the algorithm, the operation contributing the most to the total running time.) For e.g., The basic operation is usually the most time-consuming operation in the algorithm’s innermost loop.

Consider the following example:

ALGORITHM sum_of_numbers ( A[0… n-1] )

// Functionality : Finds the Sum
// Input : Array of n numbers
// Output : Sum of ’n’ numbers

i=0
sum=0
while i < n
    sum=sum + A[i]
    i=i + 1
return sum

Total number of steps for basic operation execution, C (n) = n

NOTE: Constant of fastest growing term is insignificant: Complexity theory is an Approximation theory. We are not interested in exact time required by an algorithm to solve the problem. Rather we are interested in order of growth. i.e How much faster will algorithm run on computer that is twice as fast? How much longer does it take to solve problem of double input size?

We can crudely estimate running time by

T (n) \approx \text{Cop} \times C (n) , Where, T (n): running time as a function of n, Cophon : running time of a single operation., C (n): number of basic operations as a function of n.

Order of Growth: For order of growth, consider only the leading term of a formula and ignore the constant coefficient. The following is the table of values of several functions important for analysis of algorithms.

<table>
<thead>
<tr>
<th>(n)</th>
<th>(\log_2 n)</th>
<th>(n)</th>
<th>(n \log_2 n)</th>
<th>(n^2)</th>
<th>(n^3)</th>
<th>(2^n)</th>
<th>(n!)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>3.3</td>
<td>10</td>
<td>3.310</td>
<td>102</td>
<td>103</td>
<td>103</td>
<td>3.6106</td>
</tr>
<tr>
<td>10²</td>
<td>6.6</td>
<td>10²</td>
<td>6.610²</td>
<td>104</td>
<td>105</td>
<td>1.31030</td>
<td>9.310157</td>
</tr>
<tr>
<td>10³</td>
<td>10</td>
<td>10³</td>
<td>1.010³</td>
<td>106</td>
<td>109</td>
<td>1012</td>
<td></td>
</tr>
<tr>
<td>10⁴</td>
<td>13</td>
<td>10⁴</td>
<td>1.310⁴</td>
<td>108</td>
<td>1012</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10⁵</td>
<td>17</td>
<td>10⁵</td>
<td>1.710⁵</td>
<td>10¹⁰</td>
<td>10¹⁵</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10⁶</td>
<td>20</td>
<td>10⁶</td>
<td>2.010⁶</td>
<td>10¹³</td>
<td>10¹⁸</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Worst-case, Best-case, Average case efficiencies

Algorithm efficiency depends on the input size n. And for some algorithms efficiency depends on type of input. We have best, worst & average case efficiencies.

- **Worst-case efficiency**: Efficiency (number of times the basic operation will be executed) for the worst case input of size n. i.e. The algorithm runs the longest among all possible inputs of size n.
- **Best-case efficiency**: Efficiency (number of times the basic operation will be executed) for the best case input of size n. i.e. The algorithm runs the fastest among all possible inputs of size n.
- **Average-case efficiency**: Average time taken (number of times the basic operation will be executed) to solve all the possible instances (random) of the input. NOTE: NOT the average of worst and best case.

Asymptotic Notations

Asymptotic notation is a way of comparing functions that ignores constant factors and small input sizes. Three notations used to compare orders of growth of an algorithm’s basic operation count are:

**Big Oh- O notation**

**Definition:**

A function t(n) is said to be in O(g(n)), denoted t(n) ∈ O(g(n)), if t(n) is bounded above by some constant multiple of g(n) for all large n, i.e., if there exist some positive constant c and some nonnegative integer n0 such that t(n) ≤ cg(n) for all n ≥ n0.
**Big Omega- \( \Omega \) notation**

A function \( t(n) \) is said to be in \( \Omega(g(n)) \), denoted \( t(n) \in \Omega(g(n)) \), if \( t(n) \) is bounded below by some constant multiple of \( g(n) \) for all large \( n \), i.e., if there exist some positive constant \( c \) and some nonnegative integer \( n_0 \) such that \( t(n) \geq cg(n) \) for all \( n \geq n_0 \)

Big-omega notation: \( t(n) \in \Omega(g(n)) \)

**Big Theta- \( \Theta \) notation**

A function \( t(n) \) is said to be in \( \Theta(g(n)) \), denoted \( t(n) \in \Theta(g(n)) \), if \( t(n) \) is bounded both above and below by some constant multiple of \( g(n) \) for all large \( n \), i.e., if there exist some positive constant \( c_1 \) and \( c_2 \) and some nonnegative integer \( n_0 \) such that

\[ c_2 g(n) \leq t(n) \leq c_1 g(n) \text{ for all } n \geq n_0 \]
Basic Efficiency classes

The time efficiencies of a large number of algorithms fall into only a few classes.

<table>
<thead>
<tr>
<th>fast</th>
<th>High time efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>constant</td>
</tr>
<tr>
<td>log n</td>
<td>logarithmic</td>
</tr>
<tr>
<td>n</td>
<td>linear</td>
</tr>
<tr>
<td>n log n</td>
<td>n log n</td>
</tr>
<tr>
<td>n^2</td>
<td>quadratic</td>
</tr>
<tr>
<td>n^3</td>
<td>cubic</td>
</tr>
<tr>
<td>2^n</td>
<td>exponential</td>
</tr>
<tr>
<td>n!</td>
<td>factorial</td>
</tr>
</tbody>
</table>

1.3 Mathematical Analysis of Non-Recursive and Recursive Algorithms

Mathematical analysis (Time Efficiency) of Non-recursive Algorithms

General plan for analyzing efficiency of non-recursive algorithms:

1. Decide on parameter n indicating input size
2. Identify algorithm’s basic operation
3. Check whether the number of times the basic operation is executed depends only on the input size n. If it also depends on the type of input, investigate worst, average, and best case efficiency separately.
4. Set up summation for C(n) reflecting the number of times the algorithm’s basic operation is executed.
5. Simplify summation using standard formulas

Example: Finding the largest element in a given array

```
ALGORITHM MaxElement(A[0..n-1])
//Determines the value of largest element in a given array
//input: An array A[0..n-1] of real numbers
//Output: The value of the largest element in A
currMax ← A[0]
for i ← 1 to n – 1 do
    if A[i] > currMax
        currMax ← A[i]
return currMax
```
Analysis:

1. Input size: number of elements = n (size of the array)
2. Basic operation:
   a) Comparison
   b) Assignment
3. NO best, worst, average cases.
4. Let $C(n)$ denotes number of comparisons: Algorithm makes one comparison on each execution of the loop, which is repeated for each value of the loop’s variable $i$ within the bound between 1 and $n – 1$.

$$C(n) = \sum_{i=1}^{n-1} 1$$

5. Simplify summation using standard formulas

$$C(n) = \sum_{i=1}^{n-1} 1 + 1 + 1 + \ldots + 1 \quad [(n-1) \text{ number of times}]$$

$$C(n) = n - 1$$

Example: Element uniqueness problem

Algorithm UniqueElements ($A[0..n-1]$)

//Checks whether all the elements in a given array are distinct
//Input: An array $A[0..n-1]$  
//Output: Returns true if all the elements in $A$ are distinct and false otherwise
for $i = 0$ to $n - 2$ do
for $j = i + 1$ to $n - 1$ do
    if $A[i] = A[j]$
        return false
return true
Analysis
1. Input size: number of elements = n (size of the array)
2. Basic operation: Comparison
3. Best, worst, average cases EXISTS.
Worst case input is an array giving largest comparisons.
• Array with no equal elements
• Array with last two elements are the only pair of equal elements
4. Let C (n) denotes number of comparisons in worst case: Algorithm makes one comparison for each repetition of the innermost loop i.e., for each value of the loop’s variable j between its limits i + 1 and n – 1; and this is repeated for each value of the outer loop i.e, for each value of the loop’s variable i between its limits 0 and n – 2

\[ C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 \]

5. Simplify summation using standard formulas

\[ C(n) = \sum_{i=0}^{n-2} (n - 1) - (i + 1) + 1 \]

\[ C(n) = \sum_{i=0}^{n-2} (n - 1 - i) \]

\[ C(n) = \sum_{i=0}^{n-2} (n - 1) - \sum_{i=0}^{n-2} i \]

\[ C(n) = (n - 1) \sum_{i=0}^{n-2} 1 - \sum_{i=0}^{n-2} i \]
Mathematical analysis (Time Efficiency) of recursive Algorithms

General plan for analyzing efficiency of recursive algorithms:

1. Decide on parameter n indicating input size
2. Identify algorithm’s basic operation
3. Check whether the number of times the basic operation is executed depends only on the input size n. If it also depends on the type of input, investigate worst, average, and best case efficiency separately.
4. Set up recurrence relation, with an appropriate initial condition, for the number of times the algorithm’s basic operation is executed.
5. Solve the recurrence.

\[
C(n) = (n - 1) \sum_{i=0}^{n-2} 1 - \frac{(n-2) \cdot (n-1)}{2}
\]

\[
C(n) = (n - 1)(n - 1) - \frac{(n-2)(n-1)}{2}
\]

\[
C(n) = (n - 1)((n - 1) - \frac{n - 2}{2})
\]

\[
C(n) = (n - 1)(2n - 2 - n + 2)
\]

\[
C(n) = \frac{1}{2} \cdot \frac{n(n)\slash2}{2}
\]

\[
= \frac{n^2 - n}{2}
\]

\[
= \frac{n^2}{2} - n/2
\]

\[
C(n) \in \Theta(n^2)
\]
Example: Factorial function

**ALGORITHM Factorial (n)**

//Computes n! recursively
//Input: A nonnegative integer n
//Output: The value of n!

if n = = 0
return 1
else
return Factorial (n – 1) * n

Analysis:

1. Input size: given number = n
2. Basic operation: multiplication
3. NO best, worst, average cases.
4. Let M (n) denotes number of multiplications.

\[
M(n) = M(n-1) + 1 \quad \text{for } n > 0
\]

M (0) = 0 \quad \text{initial condition}

Where: M (n – 1) : to compute Factorial (n – 1)
1 :to multiply Factorial (n – 1) by n

5. Solve the recurrence: Solving using “Backward substitution method”:

\[
M(n) = M(n-1) + 1 \\
= [ M(n-2) + 1 ] + 1 \\
= M(n-2) + 2 \\
= [ M(n-3) + 1 ] + 3 \\
= M(n-3) + 3 \\
\vdots
\]

In the ith recursion, we have

\[
= M(n-i) + i
\]

When i = n, we have

\[
= M(n-n) + n = M(0) + n
\]

Since M (0) = 0

\[
= n
\]
**M** (\(n\)) \(\in\) \(\Theta\) (\(n\))

**Example:** Find the number of binary digits in the binary representation of a positive decimal integer

**ALGORITHM** BinRec (\(n\))

//Input: A positive decimal integer \(n\)

//Output: The number of binary digits in \(n\)’s binary representation

if \(n = 1\)
return 1
else
return BinRec (\(\lfloor n/2 \rfloor\)) + 1

**Analysis:**

1. Input size: given number = \(n\)
2. Basic operation: addition
3. NO best, worst, average cases.
4. Let \(A(n)\) denotes number of additions.
   \(A(n) = A(\lfloor n/2 \rfloor) + 1\) for \(n > 1\)
   \(A(1) = 0\) initial condition

Where: \(A(\lfloor n/2 \rfloor)\) : to compute BinRec (\(\lfloor n/2 \rfloor\))
1 : to increase the returned value by 1

5. Solve the recurrence:
   \(A(n) = A(\lfloor n/2 \rfloor) + 1\) for \(n > 1\)

**Assume \(n = 2^k\) (smoothness rule)**

\(A(2^k) = A(2^{k-1}) + 1\) for \(k > 0\); \(A(20) = 0\)

**Solving using “Backward substitution method”:**

\(A(2^k) = A(2^{k-1}) + 1\)

= \([A(2^{k-2}) + 1] + 1\)

= \(A(2^{k-2}) + 2\)

= \([A(2^{k-3}) + 1] + 2\)

= \(A(2^{k-3}) + 3\)

...  

*In the \(i\)th recursion, we have*
= A (2^k) + i

When i = k, we have

= A (2^k) + k = A (2^0) + k

Since A (2^0) = 0

A (2^k) = k

Since n = 2^k, HENCE k = log_2 n

A (n) = log_2 n

A (n) ∈ Θ (log n)

1.4 Brute Force Approaches:

Introduction

*Brute force* is a straightforward approach to problem solving, usually directly based on the problem’s statement and definitions of the concepts involved. Though rarely a source of clever or efficient algorithms, the brute-force approach should not be overlooked as an important algorithm design strategy. Unlike some of the other strategies, brute force is applicable to a very wide variety of problems. For some important problems (e.g., sorting, searching, string matching), the brute-force approach yields reasonable algorithms of at least some practical value with no limitation on instance size. Even if too inefficient in general, a brute-force algorithm can still be useful for solving small-size instances of a problem. A brute-force algorithm can serve an important theoretical or educational purpose.

1.5 Selection Sort and Bubble Sort

Problem: Given a list of *n* orderable items (e.g., numbers, characters from some alphabet, character strings), rearrange them in nondecreasing order.

Selection Sort

**Algorithm** SelectionSort(A[0..n - 1])

// The algorithm sorts a given array by selection sort

// Input: An array A[0..n - 1] of orderable elements

// Output: Array A[0..n - 1] sorted in ascending order

for i=0 to n - 2 do
min = i
for j = i + 1 to n - 1 do
swap A[i] and A[min]

Example:

<table>
<thead>
<tr>
<th>89 45 68 90 29 34 17</th>
</tr>
</thead>
<tbody>
<tr>
<td>17 45 68 90 29 34 89</td>
</tr>
<tr>
<td>17 29 68 90 45 34 89</td>
</tr>
<tr>
<td>17 29 34 90 45 68 89</td>
</tr>
<tr>
<td>17 29 34 45 90 68 89</td>
</tr>
<tr>
<td>17 29 34 45 68 90 89</td>
</tr>
<tr>
<td>17 29 34 45 68 90 90</td>
</tr>
</tbody>
</table>

Selection sort’s operation on the list 89, 45, 68, 90, 29, 34, 17. Each line corresponds to one iteration of the algorithm, i.e., a pass through the list’s tail to the right of the vertical bar; an element in bold indicates the smallest element found. Elements to the left of the vertical bar are in their final positions and are not considered in this and subsequent iterations.

**Performance Analysis of the selection sort algorithm:**
The input’s size is given by the number of elements n.

The algorithm’s basic operation is the key comparison A[j] < A[min]. The number of times it is executed depends only on the array’s size and is given by

\[ C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n - 1) - (i + 1) + 1] = \sum_{i=0}^{n-2} (n - 1 - i). \]

Thus, selection sort is a \(O(n^2)\) algorithm on all inputs. The number of key swaps is only \(O(n)\) or, more precisely, \(n-1\) (one for each repetition of the \(i\) loop). This property distinguishes selection sort positively from many other sorting algorithms.

**Bubble Sort**

Compare adjacent elements of the list and exchange them if they are out of order. Then we repeat the process. By doing it repeatedly, we end up ‘bubbling up’ the largest element to the last position on the list.

**ALGORITHM**
BubbleSort(A[0..n - 1])
//The algorithm sorts array A[0..n - 1] by bubble sort
//Input: An array A[0..n - 1] of orderable elements
//Output: Array A[0..n - 1] sorted in ascending order
for i=0 to n - 2 do
  for j=0 to n - 2 - i do
      swap A[j] and A[j + 1]

Example

The first 2 passes of bubble sort on the list 89, 45, 68, 90, 29, 34, 17. A new line is shown after a swap of two elements is done. The elements to the right of the vertical bar are in their final positions and are not considered in subsequent iterations of the algorithm.

**Bubble Sort the analysis**

Clearly, the outer loop runs \( n \) times. The only complexity in this analysis in the inner loop. If we think about a single time the inner loop runs, we can get a simple bound by noting that it can never loop more than \( n \) times. Since the outer loop will make the inner loop complete \( n \) times, the comparison can't happen more than \( O(n^2) \) times.

The number of key comparisons for the bubble sort version given above is the same for all arrays of size \( n \).

\[
C(n) = \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1 = \sum_{i=0}^{n-2} [(n - 2 - i) - 0 + 1] \\
= \sum_{i=0}^{n-2} (n - 1 - i) = \frac{(n-1)n}{2} \in \Theta(n^2).
\]
The number of key swaps depends on the input. For the worst case of decreasing arrays, it is the same as the number of key comparisons.

\[ S_{\text{worst}}(n) = C(n) = \frac{(n-1)n}{2} \in \Theta(n^2). \]

Observation: if a pass through the list makes no exchanges, the list has been sorted and we can stop the algorithm. Though the new version runs faster on some inputs, it is still in \( O(n^2) \) in the worst and average cases. Bubble sort is not very good for big set of input. However, bubble sort is very simple to code.

**General Lesson From Brute Force Approach**

A first application of the brute-force approach often results in an algorithm that can be improved with a modest amount of effort. Compares successive elements of a given list with a given search key until either a match is encountered (successful search) or the list is exhausted without finding a match (unsuccessful search).

### 1.6 Sequential Search and Brute Force String Matching.

**Sequential Search**

**ALGORITHM** SequentialSearch2\((A[0..n], K)\)

//The algorithm implements sequential search with a search key as a sentinel

//Input: An array A of n elements and a search key K

//Output: The position of the first element in A[0..n - 1] whose value is equal to K or -1 if no such element is found

\( A[n]=K \)

\( i=0 \)

\( \textbf{while } A[i] = K \textbf{ do} \)

\( i=i + 1 \)

\( \textbf{if } i < n \textbf{ return } i \)

\( \textbf{else return} \)
**Brute-Force String Matching**

Given a string of $n$ characters called the *text* and a string of $m$ characters ($m = n$) called the *pattern*, find a substring of the text that matches the pattern. To put it more precisely, we want to find $i$—the index of the leftmost character of the first matching substring in the text—such that

\[
t_i = p_0, \ldots, t_{i+j} = p_j, \ldots, t_{i+m-1} = p_{m-1};
\]

\[
t_0 \ldots t_i \ldots t_{i+j} \ldots t_{i+m-1} \ldots t_{n-1} \quad \text{text } T
\]
\[
p_0 \ldots p_j \ldots p_{m-1} \quad \text{pattern } P
\]

1. Pattern: 001011
   Text: 10010101101001100101111010
2. Pattern: happy
   Text: It is never too late to have a happy childho

The algorithm shifts the pattern almost always after a single character comparison. In the worst case, the algorithm may have to make all $m$ comparisons before shifting the pattern, and this can happen for each of the $n - m + 1$ tries. Thus, in the worst case, the algorithm is in $\theta(nm)$. 
UNIT - 2
DIVIDE & CONQUER

1.1 Divide and Conquer

Definition:
Divide & conquer is a general algorithm design strategy with a general plan as follows:
1. DIVIDE:
   A problem’s instance is divided into several smaller instances of the same problem, ideally of about the same size.
2. RECUR:
   Solve the sub-problem recursively.
3. CONQUER:
   If necessary, the solutions obtained for the smaller instances are combined to get a solution to the original instance.

NOTE:
The base case for the recursion is sub-problem of constant size.

Advantages of Divide & Conquer technique:
• For solving conceptually difficult problems like Tower Of Hanoi, divide & conquer is a powerful tool
• Results in efficient algorithms
• Divide & Conquer algorithms are adapted foe execution in multi-processor machines
• Results in algorithms that use memory cache efficiently.

Limitations of divide & conquer technique:
• Recursion is slow
• Very simple problem may be more complicated than an iterative approach. Example: adding n numbers etc

1.2 General Method

General divide & conquer recurrence:
An instance of size n can be divided into b instances of size n/b, with “a” of them needing to be solved. [ a ≥ 1, b > 1].
Assume size n is a power of b. The recurrence for the running time T(n) is as follows:

\[ T(n) = aT(n/b) + f(n) \]

where:
\[ f(n) \] – a function that accounts for the time spent on dividing the problem into smaller ones and on combining their solutions

Therefore, the order of growth of T(n) depends on the values of the constants a & b and the order of growth of the function f(n)
**Master theorem**

**Theorem:** If \( f(n) \in \Theta(n^d) \) with \( d \geq 0 \) in recurrence equation

\[ T(n) = aT(n/b) + f(n) , \]

then

\[ T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases} \]

**Example:**

Let \( T(n) = 2T(n/2) + 1 \), solve using master theorem.

**Solution:**

Here:  
\[ a = 2 \]
\[ b = 2 \]
\[ f(n) = \Theta(1) \]
\[ d = 0 \]

Therefore:  
\[ a > b^d \text{ i.e., } 2 > 2^0 \]

Case 3 of master theorem holds good. Therefore:

\[ T(n) \in \Theta(n^{\log_2 a}) \]
\[ \in \Theta(n^{\log_2 2}) \]
\[ \in \Theta(n) \]

**1.3 Binary Search**

**Description:**

Binary tree is a dichotomic divide and conquer search algorithm. Ti inspects the middle element of the sorted list. If equal to the sought value, then the position has been found. Otherwise, if the key is less than the middle element, do a binary search on the first half, else on the second half.

**Algorithm:**

Algorithm can be implemented as recursive or non-recursive algorithm.

**ALGORITHM BinSrch ( A[0 … n-1], key)**

//implements non-recursive binary search

//i/p: Array A in ascending order, key k
//o/p: Returns position of the key matched else -1

\[ l \rightarrow 0 \]
\[ r \rightarrow n-1 \]

while \( l \leq r \) do

\[ m \rightarrow (l + r) / 2 \]

if key = = A[m]
return m
else
  if key < A[m]
    r = m-1
  else
    l = m+1
return -1

Analysis:
- **Input size:** Array size, n
- **Basic operation:** key comparison
- **Depend on**
  - Best – key matched with mid element
  - Worst – key not found or key sometimes in the list
- Let C(n) denotes the number of times basic operation is executed. Then
  - Worst case efficiency. Since after each comparison the algorithm divides the problem into half the size, we have
    \[ C_{\text{worst}}(n) = C_{\text{worst}}(n/2) + 1 \quad \text{for } n > 1 \]
    \[ C(1) = 1 \]
- Solving the recurrence equation using master theorem, to give the number of times the search key is compared with an element in the array, we have:
  \[ C(n) = C(n/2) + 1 \]
  \[ a = 1 \]
  \[ b = 2 \]
  \[ f(n) = n^0 ; d = 0 \]
  - case 2 holds:
    \[ C(n) = \Theta (n^{d \log n}) \]
    \[ = \Theta (n^0 \log n) \]
    \[ = \Theta (\log n) \]

Applications of binary search:
- Number guessing game
- Word lists/search dictionary etc

Advantages:
- Efficient on very big list
- Can be implemented iteratively/recursively

Limitations:
- Interacts poorly with the memory hierarchy
- Requires given list to be sorted
- Due to random access of list element, needs arrays instead of linked list.

1.4 Merge Sort
**Definition:**
Merge sort is a sort algorithm that splits the items to be sorted into two groups, recursively sorts each group, and merges them into a final sorted sequence.

**Features:**
- Is a comparison based algorithm
- Is a stable algorithm
- Is a perfect example of divide & conquer algorithm design strategy
- It was invented by John Von Neumann

**Algorithm:**

ALGORITHM Mergesort ( A[0… n-1] )
//sorts array A by recursive mergesort
//i/p: array A
//o/p: sorted array A in ascending order

if n > 1
    copy A[0… (n/2 -1)] to B[0… (n/2 -1)]
    copy A[n/2… n -1)] to C[0… (n/2 -1)]
    Mergesort ( B[0… (n/2 -1)] )
    Mergesort ( C[0… (n/2 -1)] )
    Merge ( B, C, A )

ALGORITHM Merge ( B[0… p-1], C[0… q-1], A[0… p+q-1] )
//merges two sorted arrays into one sorted array
//i/p: arrays B, C, both sorted
//o/p: Sorted array A of elements from B & C

I →0
j→0
k→0
while i < p and j < q do
    if B[i] ≤ C[j]
        A[k] →B[i]
        i→i + 1
    else
        A[k] →C[j]
        j→j + 1
        k→k + 1
if i == p
    copy C [ j… q-1 ] to A [ k… (p+q-1) ]
else
    copy B [ i… p-1 ] to A [ k… (p+q-1) ]

**Example:**
Apply merge sort for the following list of elements: 6, 3, 7, 8, 2, 4, 5, 1
Analysis:

- **Input size**: Array size, n
- **Basic operation**: key comparison
- **Best, worst, average case exists**: 
  Worst case: During key comparison, neither of the two arrays becomes empty before the other one contains just one element.
- Let C(n) denotes the number of times basic operation is executed. Then
  \[ C(n) = 2C(n/2) + C_{merge}(n) \quad \text{for } n > 1 \]
  \[ C(1) = 0 \]
  where, \( C_{merge}(n) \) is the number of key comparison made during the merging stage.
  In the worst case:
  \[ C_{merge}(n) = 2C_{merge}(n/2) + n-1 \quad \text{for } n > 1 \]
  \[ C_{merge}(1) = 0 \]
- **Solving the recurrence equation using master theorem**:
  \[ C(n) = 2C(n/2) + n-1 \quad \text{for } n > 1 \]
  \[ C(1) = 0 \]
  Here \( a = 2 \)
  \[ b = 2 \]
  \[ f(n) = n; \quad d = 1 \]
  Therefore \( 2 = 2^1 \), case 2 holds
  \[ C(n) = \Theta(n^d \log n) \]
  \[ = \Theta(n^1 \log n) \]
\[ = \Theta (n \log n) \]

**Advantages:**
- Number of comparisons performed is nearly optimal.
- Mergesort will never degrade to \(O(n^2)\)
- It can be applied to files of any size

**Limitations:**
- Uses \(O(n)\) additional memory.

### 1.5 Quick Sort and its performance

**Definition:**
Quick sort is a well-known sorting algorithm, based on divide & conquer approach. The steps are:
1. Pick an element called pivot from the list
2. Reorder the list so that all elements which are less than the pivot come before the pivot and all elements greater than pivot come after it. After this partitioning, the pivot is in its final position. This is called the partition operation
3. Recursively sort the sub-list of lesser elements and sub-list of greater elements.

**Features:**
- Developed by C.A.R. Hoare
- Efficient algorithm
- NOT stable sort
- Significantly faster in practice, than other algorithms

**Algorithm**

ALGORITHM Quicksort \((A[1 \ldots r])\)
//sorts by quick sort
//i/p: A sub-array \(A[1..r]\) of \(A[0..n-1]\), defined by its left and right indices \(l\) and \(r\)
//o/p: The sub-array \(A[l..r]\), sorted in ascending order
if \(l < r\)
    Partition \((A[l..r])\) // \(s\) is a split position
    Quicksort\((A[l..s-1])\)
    Quicksort\((A[s+1..r])\)

ALGORITHM Partition \((A[1..r])\)
//Partitions a sub-array by using its first element as a pivot
//i/p: A sub-array \(A[1..r]\) of \(A[0..n-1]\), defined by its left and right indices \(l\) and \(r\) \((l < r)\)
//o/p: A partition of \(A[1..r]\), with the split position returned as this function’s value
\(p\rightarrow A[l]\)
\(i\rightarrow l\)
\(j\rightarrow r + 1;\)
Repeat
    repeat \(i\rightarrow i + 1\) until \(A[i] \geq p\) //left-right scan
    repeat \(j\rightarrow j - 1\) until \(A[j] < p\) //right-left scan
    if \((i < j)\) //need to continue with the scan
        swap\((A[i], a[j])\)
    until \(i \geq j\) //no need to scan
    swap\((A[l], A[j])\)
return j

Analysis:

- **Input size**: Array size, $n$
- **Basic operation**: key comparison
- Best, worst, average case exists:
  - Best case: when partition happens in the middle of the array each time.
  - Worst case: When input is already sorted. During key comparison, one half is empty, while remaining $n-1$ elements are on the other partition.
- Let $C(n)$ denotes the number of times basic operation is executed in worst case:
  Then
  $$C(n) = C(n-1) + (n+1) \text{ for } n > 1 \text{ (2 sub-problems of size 0 and } n-1 \text{ respectively)}$$
C(1) = 1

Best case:
C(n) = 2C(n/2) + Θ(n) \quad (2 \text{ sub-problems of size } n/2 \text{ each})

- Solving the recurrence equation using backward substitution/ master theorem, we have:
  
  \[ C(n) = C(n-1) + (n+1) \text{ for } n > 1; \quad C(1) = 1 \]

  \[ C(n) = \Theta(n^2) \]

  \[ C(n) = 2C(n/2) + \Theta(n). \]
  \[ = \Theta(n^\log n) \]
  \[ = \Theta(n \log n) \]

NOTE:
The quick sort efficiency in average case is \( \Theta(n \log n) \) on random input.
UNIT - 3
THE GREEDY METHOD

3.1 The General Method

Definition:
Greedy technique is a general algorithm design strategy, built on following elements:
- **configurations**: different choices, values to find
- **objective function**: some configurations to be either maximized or minimized

The method:
- Applicable to **optimization problems** ONLY
- Constructs a solution through a sequence of steps
- Each step expands a partially constructed solution so far, until a complete solution to the problem is reached.
  - **On each step, the choice made must be**
    - **Feasible**: it has to satisfy the problem’s constraints
    - **Locally optimal**: it has to be the best local choice among all feasible choices available on that step
    - **Irrevocable**: Once made, it cannot be changed on subsequent steps of the algorithm

NOTE:
- Greedy method works best when applied to problems with the **greedy-choice** property
- A globally-optimal solution can always be found by a series of local improvements from a starting configuration.

Greedy method vs. Dynamic programming method:
- LIKE dynamic programming, greedy method **solves optimization problems**.
- LIKE dynamic programming, greedy method **problems exhibit optimal substructure**
- UNLIKE dynamic programming, greedy method problems exhibit the **greedy choice** property -avoids back-tracing.

Applications of the Greedy Strategy:
- **Optimal solutions**:
  - Change making
  - Minimum Spanning Tree (MST)
  - Single-source shortest paths
  - Huffman codes
- **Approximations**:
  - Traveling Salesman Problem (TSP)
  - Fractional Knapsack problem
3.2 Knapsack problem

- One wants to pack \( n \) items in a luggage
  - The \( i \)th item is worth \( v_i \) dollars and weighs \( w_i \) pounds
  - Maximize the value but cannot exceed \( W \) pounds
  - \( v_i, w_i, W \) are integers
- 0-1 knapsack \( \rightarrow \) each item is taken or not taken
- Fractional knapsack \( \rightarrow \) fractions of items can be taken
- Both exhibit the optimal-substructure property
  - 0-1: If item \( j \) is removed from an optimal packing, the remaining packing is an optimal packing with weight at most \( W - w_j \)
  - Fractional: If \( w \) pounds of item \( j \) is removed from an optimal packing, the remaining packing is an optimal packing with weight at most \( W - w \) that can be taken from other \( n-1 \) items plus \( w_j - w \) of item \( j \)

Greedy Algorithm for Fractional Knapsack problem

- Fractional knapsack can be solvable by the greedy strategy
  - Compute the value per pound \( v_i/w_i \) for each item
  - Obeying a greedy strategy, take as much as possible of the item with the greatest value per pound.
  - If the supply of that item is exhausted and there is still more room, take as much as possible of the item with the next value per pound, and so forth until there is no more room
  - \( O(n \log n) \) (we need to sort the items by value per pound)

O-1 knapsack is harder

- Knapsack cannot be solved by the greedy strategy
  - Unable to fill the knapsack to capacity, and the empty space lowers the effective value per pound of the packing
  - We must compare the solution to the sub-problem in which the item is included with the solution to the sub-problem in which the item is excluded before we can make the choice
3.3 Job sequencing with deadlines

The problem is stated as below.

- There are $n$ jobs to be processed on a machine.
- Each job $i$ has a deadline $d_i \geq 0$ and profit $p_i \geq 0$.
- $P_i$ is earned iff the job is completed by its deadline.
- The job is completed if it is processed on a machine for unit time.
- Only one machine is available for processing jobs.
- Only one job is processed at a time on the machine.
- A feasible solution is a subset of jobs $J$ such that each job is completed by its deadline.
- An optimal solution is a feasible solution with maximum profit value.

**Example**: Let $n = 4$, $(p_1, p_2, p_3, p_4) = (100, 10, 15, 27)$, $(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$
• Consider the jobs in the non-increasing order of profits subject to the constraint that the resulting job sequence J is a feasible solution.

• In the example considered before, the non-increasing profit vector is

\[(100 \ 27 \ 15 \ 10) \quad (2 \ 1 \ 2 \ 1)\]

\[p_1 \ p_4 \ p_3 \ p_2 \quad d_1 \ d \ d_3 \ d_2\]

\(J = \{1\}\) is a feasible one

\(J = \{1, 4\}\) is a feasible one with processing sequence (4,1)

\(J = \{1, 3, 4\}\) is not feasible

\(J = \{1, 2, 4\}\) is not feasible

\(J = \{1, 4\}\) is optimal

**Theorem:** Let J be a set of K jobs and

\[\sigma = (i_1, i_2, \ldots, i_k)\] be a permutation of jobs in J such that \(d_{i_1} \leq d_{i_2} \leq \ldots \leq d_{i_k}\).

• J is a feasible solution iff the jobs in J can be processed in the order \(\sigma\) without violating any deadly.

**Proof:**

• By definition of the feasible solution if the jobs in J can be processed in the order without violating any deadline then J is a feasible solution.
• So, we have only to prove that if J is a feasible one, then \( \sigma \) represents a possible order in which the jobs may be processed.

• Suppose J is a feasible solution. Then there exists \( \sigma^1 = (r_1, r_2, \ldots, r_k) \) such that

\[
d_{r_j} \geq j, \quad 1 \leq j < k
\]

i.e. \( d_{r_1} \geq 1, d_{r_2} \geq 2, \ldots, d_{r_k} \geq k. \)
each job requiring an unit time.

\[
\sigma = (i_1, i_2, \ldots, i_k) \quad \text{and} \quad \sigma^1 = (r_1, r_2, \ldots, r_k)
\]

• Assume \( \sigma^1 \neq \sigma \). Then let a be the least index in which \( \sigma^1 \) and \( \sigma \) differ. i.e. a is such that \( r_a \neq i_a \).

• Let \( r_b = i_a \), so \( b > a \) (because for all indices \( j \) less than \( a \) \( r_j = i_j \)).

• In \( \sigma^1 \) interchange \( r_a \) and \( r_b \).

• \( \sigma = (i_1, i_2, \ldots, i_a, i_b, i_k) \) \[ r_b \text{ occurs before } r_a \]

• \( \sigma^1 = (r_1, r_2, \ldots, r_a, r_b, \ldots, r_k) \)

• \( i_1 = r_1, i_2 = r_2, \ldots, i_{a-1} = r_{a-1}, i_a \neq r_b \) but \( i_a = r_b \)

• We know \( d_{i_1} \leq d_{i_2} \leq \ldots d_{i_a} \leq d_{i_b} \leq \ldots \leq d_{i_k} \).

• Since \( i_a = r_b \), \( d_{r_b} \leq d_{r_a} \) or \( d_{r_a} \geq d_{r_b} \).

• In the feasible solution \( d_{r_a} \geq a \) \( d_{r_b} \geq b \)

• So if we interchange \( r_a \) and \( r_b \), the resulting permutation \( \sigma'^1 = (s_1, \ldots, s_k) \) represents an order with the least index in which \( \sigma'^1 \) and \( \sigma \) differ is incremented by one.

• Also the jobs in \( \sigma'^1 \) may be processed without violating a deadline.

• Continuing in this way, \( \sigma^1 \) can be transformed into \( \sigma \) without violating any deadline.

• Hence the theorem is proved

**GREEDY ALGORITHM FOR JOB SEQUENCING WITH DEADLINE**

Procedure greedy job (D, J, n) J may be represented by

// J is the set of n jobs to be completed // one dimensional array J (1: K)
// by their deadlines // The deadlines are

\[
J \leftarrow \{1\} \quad D(J(1)) \leq D(J(2)) \leq \ldots \leq D(J(K))
\]

for I \( \leftarrow 2 \) to n do To test if \( J \cup \{i\} \) is feasible,
If all jobs in JU{i} can be completed we insert i into J and verify by their deadlines 

\[ D(J®) \leq r \quad 1 \leq r \leq k+1 \]

then J \leftarrow JU{i}

end if

repeat

end greedy-job

Procedure JS(D, J, n, k)

// D(i) \geq 1, 1 \leq i \leq n are the deadlines //
// the jobs are ordered such that //
// p_1 \geq p_2 \geq \ldots \geq p_n //
// in the optimal solution, D(J(i)) \geq D(J(i+1)) //

// 1 \leq i \leq k //

integer D(0:n), J(0:n), i, k, n, r

D(0) \leftarrow J(0) \leftarrow 0

// J(0) is a fictitious job with D(0) = 0 //
K \leftarrow 1; J(1) \leftarrow 1 // job one is inserted into J //

for i \leftarrow 2 to do // consider jobs in non increasing order of pi //

// find the position of i and check feasibility of insertion //

r \leftarrow k // r and k are indices for existing job in J //

// find r such that i can be inserted after r //
while D(J(r)) > D(i) and D(i) \neq r do

// job r can be processed after i and //
// deadline of job r is not exactly r //

r \leftarrow r-1 // consider whether job r-1 can be processed after i //

repeat

if D(J(r)) \geq d(i) and D(i) > r then

// the new job i can come after existing job r; insert i into J at position r+1 //

for l \leftarrow k to r+1 by -1 do

J(l+1) \leftarrow J(l) // shift jobs( r+1) to k right by//

// one position //

repeat
if D(J(r)) ≥ d(i) and D(i) > r then

// the new job i can come after existing job r; insert i into J at position r+1 //
for I ← k to r+1 by −1 do
J(I+1) ← J(I) // shift jobs( r+1) to k right by//
//one position //
Repeat

COMPLEXITY ANALYSIS OF JS ALGORITHM

- Let n be the number of jobs and s be the number of jobs included in the solution.
- The loop between lines 4-15 (the for-loop) is iterated (n-1)times.
- Each iteration takes O(k) where k is the number of existing jobs.

∴ The time needed by the algorithm is O(sn) s ≤ n so the worst case time is O(n²).

If di = n - i+1 1 ≤ i ≤ n, JS takes θ(n²) time

D and J need θ(s) amount of space.

3.4 Minimum-Cost Spanning Trees

Spanning Tree

Definition:

Spanning tree is a connected acyclic sub-graph (tree) of the given graph (G) that includes all of G’s vertices

Example: Consider the following graph
The spanning trees for the above graph are as follows:

Minimum Spanning Tree (MST)

**Definition:**
MST of a weighted, connected graph $G$ is defined as: A spanning tree of $G$ with **minimum total weight**.

**Example:** Consider the example of spanning tree:
For the given graph there are three possible spanning trees. Among them the spanning tree with the minimum weight 6 is the MST for the given graph.

**Question:** Why can’t we use **BRUTE FORCE** method in constructing MST?
**Answer:** If we use Brute force method-
- Exhaustive search approach has to be applied.
- Two serious obstacles faced:
  1. The number of spanning trees grows exponentially with graph size.
  2. Generating all spanning trees for the given graph is not easy.

**MST Applications:**
- **Network design.**
  Telephone, electrical, hydraulic, TV cable, computer, road
- **Approximation algorithms for NP-hard problems.**
  Traveling salesperson problem, Steiner tree
- Cluster analysis.
- Reducing data storage in sequencing amino acids in a protein
- Learning salient features for real-time face verification
- Auto config protocol for Ethernet bridging to avoid cycles in a network, etc
3.5 Prim’s Algorithm

Some useful definitions:

- **Fringe edge**: An edge which has one vertex is in partially constructed tree Ti and the other is not.
- **Unseen edge**: An edge with both vertices not in Ti

Algorithm:

**ALGORITHM Prim (G)**
//Prim’s algorithm for constructing a MST
//Input: A weighted connected graph G = { V, E }
//Output: ET the set of edges composing a MST of G

// the set of tree vertices can be initialized with any vertex

\[V_T \rightarrow \{ v_0 \}\]
\[E_T \rightarrow \emptyset\]

for \(i \rightarrow 1\) to \(|V| - 1\) do

- Find a minimum-weight edge \( e^* = (v^*, u^*) \) among all the edges \((v, u)\) such that \(v\) is in \(V_T\) and \(u\) is in \(V - V_T\)

\[V_T \rightarrow V_T U \{ u^* \}\]
\[E_T \rightarrow E_T U \{ e^* \}\]

return \(E_T\)

The method:

**STEP 1**: Start with a tree, \(T_0\), consisting of one vertex

**STEP 2**: “Grow” tree one vertex/edge at a time

- Construct a series of expanding sub-trees \(T_1, T_2, \ldots T_{n-1}\).
- At each stage construct \(T_{i+1}\) from \(T_i\) by adding the minimum weight edge connecting a vertex in tree (Ti) to one vertex not yet in tree, choose from “fringe” edges (this is the “greedy” step!)

Algorithm stops when all vertices are included
Example:
Apply Prim’s algorithm for the following graph to find MST.

Solution:

<table>
<thead>
<tr>
<th>Tree vertices</th>
<th>Remaining vertices</th>
<th>Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>a (-, -)</td>
<td>b (a, 3) c (-, ∞) d (-, ∞) e (a, 6) f (a, 5)</td>
<td><img src="image1.png" alt="Graph 1" /></td>
</tr>
<tr>
<td>b (a, 3)</td>
<td>c (b, 1) d (-, ∞) e (a, 6) f (b, 4)</td>
<td><img src="image2.png" alt="Graph 2" /></td>
</tr>
<tr>
<td>c (b, 1)</td>
<td>d (c, 6) e (a, 6) f (b, 4)</td>
<td><img src="image3.png" alt="Graph 3" /></td>
</tr>
</tbody>
</table>
Efficiency:

Efficiency of Prim’s algorithm is based on data structure used to store priority queue.
- Unordered array: Efficiency: $\Theta(n^2)$
- Binary heap: Efficiency: $\Theta(m \log n)$
- Min-heap: For graph with $n$ nodes and $m$ edges: Efficiency: $(n + m) \log n$

Conclusion:

- Prim’s algorithm is a “vertex based algorithm”
- Prim’s algorithm “Needs priority queue for locating the nearest vertex.” The choice of priority queue matters in Prim implementation.
  - Array - optimal for dense graphs
  - Binary heap - better for sparse graphs
  - Fibonacci heap - best in theory, but not in practice
3.6 Kruskal’s Algorithm

Algorithm:

ALGORITHM Kruskal (G)
//Kruskal’s algorithm for constructing a MST
//Input: A weighted connected graph G = { V, E }
//Output: ET the set of edges composing a MST of G

Sort E in ascending order of the edge weights

// initialize the set of tree edges and its size
E_T→Ø
edge_counter →0

//initialize the number of processed edges
K →0

while edge_counter < |V| - 1
    k→k + 1
    if E_T U { e_i_k} is acyclic
        E_T→E_T U { e_i_k }
        edge_counter→edge_counter + 1
return E_T

The method:

STEP 1: Sort the edges by increasing weight
STEP 2: Start with a forest having |V| number of trees.
STEP 3: Number of trees are reduced by ONE at every inclusion of an edge
    At each stage:
        • Among the edges which are not yet included, select the one with minimum weight AND which does not form a cycle.
        • the edge will reduce the number of trees by one by combining two trees of the forest

Algorithm stops when |V| - 1 edges are included in the MST i.e : when the number of trees in the forest is reduced to ONE.
Example:
Apply Kruskal’s algorithm for the following graph to find MST.

```
Example:
Apply Kruskal’s algorithm for the following graph to find MST.
```

```
Solution:
The list of edges is:

<table>
<thead>
<tr>
<th>Edge</th>
<th>ab</th>
<th>af</th>
<th>ae</th>
<th>bc</th>
<th>bf</th>
<th>cf</th>
<th>cd</th>
<th>df</th>
<th>de</th>
<th>ef</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weight</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>5</td>
<td>8</td>
<td>2</td>
</tr>
</tbody>
</table>

Sort the edges in ascending order:

<table>
<thead>
<tr>
<th>Edge</th>
<th>bc</th>
<th>ef</th>
<th>ab</th>
<th>bf</th>
<th>cf</th>
<th>af</th>
<th>df</th>
<th>ae</th>
<th>cd</th>
<th>de</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weight</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>8</td>
</tr>
</tbody>
</table>
```

```
Solution:
The list of edges is:

<table>
<thead>
<tr>
<th>Edge</th>
<th>bc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weight</td>
<td>1</td>
</tr>
<tr>
<td>Insertion status</td>
<td>YES</td>
</tr>
<tr>
<td>Insertion order</td>
<td>1</td>
</tr>
</tbody>
</table>

```

```
Solution:
The list of edges is:

<table>
<thead>
<tr>
<th>Edge</th>
<th>ef</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weight</td>
<td>2</td>
</tr>
<tr>
<td>Insertion status</td>
<td>YES</td>
</tr>
<tr>
<td>Insertion order</td>
<td>2</td>
</tr>
</tbody>
</table>
```

```
```
<table>
<thead>
<tr>
<th>Edge</th>
<th>ab</th>
<th><img src="image" alt="Graph" /></th>
</tr>
</thead>
<tbody>
<tr>
<td>Weight</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Insertion status</td>
<td>YES</td>
<td></td>
</tr>
<tr>
<td>Insertion order</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Edge</th>
<th>bf</th>
<th><img src="image" alt="Graph" /></th>
</tr>
</thead>
<tbody>
<tr>
<td>Weight</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Insertion status</td>
<td>YES</td>
<td></td>
</tr>
<tr>
<td>Insertion order</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Edge</th>
<th>cf</th>
<th><img src="image" alt="Graph" /></th>
</tr>
</thead>
<tbody>
<tr>
<td>Weight</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Insertion status</td>
<td>NO</td>
<td></td>
</tr>
<tr>
<td>Insertion order</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Edge</th>
<th>af</th>
<th><img src="image" alt="Graph" /></th>
</tr>
</thead>
<tbody>
<tr>
<td>Weight</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>Insertion status</td>
<td>NO</td>
<td></td>
</tr>
<tr>
<td>Insertion order</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Edge</th>
<th>df</th>
<th><img src="image" alt="Graph" /></th>
</tr>
</thead>
<tbody>
<tr>
<td>Weight</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>Insertion status</td>
<td>YES</td>
<td></td>
</tr>
<tr>
<td>Insertion order</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

Algorithm stops as $|V| - 1$ edges are included in the MST
Efficiency:

Efficiency of Kruskal’s algorithm is based on the time needed for sorting the edge weights of a given graph.

- With an efficient sorting algorithm: Efficiency: \( \Theta(|E| \log |E|) \)

Conclusion:

- Kruskal’s algorithm is an “edge based algorithm”
- Prim’s algorithm with a heap is faster than Kruskal’s algorithm.

3.7 Single Source Shortest Paths.

Some useful definitions:

- Shortest Path Problem: Given a connected directed graph \( G \) with non-negative weights on the edges and a root vertex \( r \), find for each vertex \( x \), a directed path \( P(x) \) from \( r \) to \( x \) so that the sum of the weights on the edges in the path \( P(x) \) is as small as possible.

Algorithm

- By Dutch computer scientist Edsger Dijkstra in 1959.
- Solves the single-source shortest path problem for a graph with nonnegative edge weights.
- This algorithm is often used in routing.

E.g.: Dijkstra's algorithm is usually the working principle behind link-state routing protocols

ALGORITHM Dijkstra\((G, s)\)

//Input: Weighted connected graph \( G \) and source vertex \( s \)
//Output: The length \( D_v \) of a shortest path from \( s \) to \( v \) and its penultimate vertex \( P_v \) for every vertex \( v \) in \( V \)

//initialize vertex priority in the priority queue
Initialize (Q)

for every vertex \( v \) in \( V \) do
  \( D_v \rightarrow \infty \); \( P_v \rightarrow \text{null} \) // \( P_v \), the parent of \( v \)
  insert(Q, v, \( D_v \)) //initialize vertex priority in priority queue
\( d_s \rightarrow 0 \)

//update priority of \( s \) with \( d_s \), making \( d_s \), the minimum
Decrease(Q, s, \( d_s \))

\( V \rightarrow \emptyset \)
for i→0 to |V| - 1 do
  u*→DeleteMin(Q)
  //expanding the tree, choosing the locally best vertex
  V_T→V_T U {u*}
  for every vertex u in V – V_T that is adjacent to u* do
    if Du* + w (u*, u) < Du
      Du→Du + w (u*, u); Pu u*
      Decrease(Q, u, Du)

The method
Dijkstra’s algorithm solves the single source shortest path problem in 2 stages.
Stage 1: A greedy algorithm computes the shortest distance from source to all other nodes in the graph and saves in a data structure.
Stage 2: Uses the data structure for finding a shortest path from source to any vertex v.
  • At each step, and for each vertex x, keep track of a “distance” D(x) and a directed path P(x) from root to vertex x of length D(x).
  • Scan first from the root and take initial paths P(r, x) = (r, x) with
    D(x) = w(rx) when rx is an edge,
    D(x) = ∞ when rx is not an edge.
    For each temporary vertex y distinct from x, set
    D(y) = min{ D(y), D(x) + w(xy) }

Example:
Apply Dijkstra’s algorithm to find Single source shortest paths with vertex a as the source.

Solution:
Length Dv of shortest path from source (s) to other vertices v and Penultimate vertex Pv for every vertex v in V:

Da = 0, Pa = null
Db = ∞, Pb = null
Dc = ∞, Pc = null
Dd = ∞, Pd = null
De = ∞, Pe = null
Df = ∞, Pf = null
### Conclusion:

- Doesn’t work with negative weights
- Applicable to both undirected and directed graphs
- Use unordered array to store the priority queue: Efficiency = $\Theta(n^2)$
- Use min-heap to store the priority queue: Efficiency = $O(m \log n)$
UNIT - 4
Dynamic Programming

4.1 The General Method
Definition
Dynamic programming (DP) is a general algorithm design technique for solving problems with overlapping sub-problems. This technique was invented by American mathematician “Richard Bellman” in 1950s.

Key Idea
The key idea is to save answers of overlapping smaller sub-problems to avoid re-computation.

Dynamic Programming Properties
- An instance is solved using the solutions for smaller instances.
- The solutions for a smaller instance might be needed multiple times, so store their results in a table.
- Thus each smaller instance is solved only once.
- Additional space is used to save time.

Dynamic Programming vs. Divide & Conquer
LIKE divide & conquer, dynamic programming solves problems by combining solutions to sub-problems. UNLIKE divide & conquer, sub-problems are NOT independent in dynamic programming.

<table>
<thead>
<tr>
<th>Divide &amp; Conquer</th>
<th>Dynamic Programming</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Partitions a problem into independent smaller sub-problems</td>
<td>1. Partitions a problem into overlapping sub-problems</td>
</tr>
<tr>
<td>2. Doesn’t store solutions of sub-problems. (Identical sub-problems may arise - results in the same computations are performed repeatedly.)</td>
<td>2. Stores solutions of sub-problems; thus avoids calculations of same quantity twice</td>
</tr>
<tr>
<td>3. Top down algorithms: which logically progresses from the initial instance down to the smallest sub-instances via intermediate sub-instances.</td>
<td>3. Bottom up algorithms: in which the smallest sub-problems are explicitly solved first and the results of these used to construct solutions to progressively larger sub-instances</td>
</tr>
</tbody>
</table>

Dynamic Programming vs. Divide & Conquer: EXAMPLE
Computing Fibonacci Numbers
1. Using standard recursive formula:

\[
F(n) = \begin{cases} 
0 & \text{if } n=0 \\
1 & \text{if } n=1 \\
F(n-1) + F(n-2) & \text{if } n > 1 
\end{cases}
\]

**Algorithm F(n)**

// Computes the nth Fibonacci number recursively by using its definitions
// Input: A non-negative integer n
// Output: The nth Fibonacci number
if n==0 || n==1 then
    return n
else
    return F(n-1) + F(n-2)

**Algorithm F(n): Analysis**

- Is too expensive as it has repeated calculation of smaller Fibonacci numbers.
- Exponential order of growth.

2. Using Dynamic Programming:

**Algorithm F(n)**

// Computes the nth Fibonacci number by using dynamic programming method
// Input: A non-negative integer n
// Output: The nth Fibonacci number
A[0] 0
A[1] 1
for i 2 to n do
return A[n]

**Algorithm F(n): Analysis**

- Since it caches previously computed values, saves time from repeated computations of same sub-instance
- Linear order of growth

**Rules of Dynamic Programming**

1. **OPTIMAL SUB-STRUCTURE:** An optimal solution to a problem contains optimal solutions to sub-problems
2. **OVERLAPPING SUB-PROBLEMS**: A recursive solution contains a “small” number of distinct sub-problems repeated many times
3. **BOTTOM UP FASHION**: Computes the solution in a bottom-up fashion in the final step

**Three basic components of Dynamic Programming solution**
The development of a dynamic programming algorithm must have the following three basic components
1. A recurrence relation
2. A tabular computation
3. A backtracking procedure

**Example Problems that can be solved using Dynamic Programming method**
1. Computing binomial co-efficient
2. Compute the longest common subsequence
3. Warshall’s algorithm for transitive closure
4. Floyd’s algorithm for all-pairs shortest paths
5. Some instances of difficult discrete optimization problems like knapsack problem traveling salesperson problem

### 4.2 Warshall’s Algorithm

**Some useful definitions:**
- **Directed Graph**: A graph whose every edge is directed is called directed graph OR digraph
- **Adjacency matrix**: The adjacency matrix $A = \{a_{ij}\}$ of a directed graph is the boolean matrix that has
  - $1$ - if there is a directed edge from $i$th vertex to the $j$th vertex
  - $0$ - Otherwise
- **Transitive Closure**: Transitive closure of a directed graph with $n$ vertices can be defined as the $n$-by-$n$ matrix $T=\{t_{ij}\}$, in which the elements in the $i$th row ($1 \leq i \leq n$) and the $j$th column ($1 \leq j \leq n$) is $1$ if there exists a nontrivial directed path (i.e., a directed path of a positive length) from the $i$th vertex to the $j$th vertex, otherwise $t_{ij}$ is $0$.
  The transitive closure provides reach ability information about a digraph.

**Computing Transitive Closure:**
- We can perform DFS/BFS starting at each vertex
  - Performs traversal starting at the $i$th vertex.
  - Gives information about the vertices reachable from the $i$th vertex
  - **Drawback**: This method traverses the same graph several times.
  - **Efficiency**: $O(n(n+m))$
- Alternatively, we can use dynamic programming: the Warshall’s Algorithm

**Underlying idea of Warshall’s algorithm:**
- Let $A$ denote the initial boolean matrix.
• The element \( r(k) [i, j] \) in \( i \)th row and \( j \)th column of matrix \( R_k \) \((k = 0, 1, \ldots, n)\) is equal to 1 if and only if there exists a directed path from \( i \)th vertex to \( j \)th vertex with intermediate vertex if any, numbered not higher than \( k \).

• **Recursive Definition:**
  - **Case 1:**
    A path from \( vi \) to \( vj \) restricted to using only vertices from \( \{v1, v2, \ldots, vk\} \) as intermediate vertices does not use \( vk \). Then
    \[ R(k) [i, j] = R(k-1) [i, j]. \]
  - **Case 2:**
    A path from \( vi \) to \( vj \) restricted to using only vertices from \( \{v1, v2, \ldots, vk\} \) as intermediate vertices do use \( vk \). Then
    \[ R(k) [i, j] = R(k-1) [i, k] \AND R(k-1) [k, j]. \]

We conclude:
\[ R(k)[i, j] = R(k-1)[i, j] \OR (R(k-1)[i, k] AND R(k-1)[k, j]) \]

**Algorithm:**

**Algorithm Warshall(A[1..n, 1..n])**

// Computes transitive closure matrix
// Input: Adjacency matrix A
// Output: Transitive closure matrix R

\[ R(0) = A \]

for \( k \rightarrow 1 \) to \( n \) do
  for \( i \rightarrow 1 \) to \( n \) do
    for \( j \rightarrow 1 \) to \( n \) do
      \[ R(k)[i, j] \rightarrow R(k-1)[i, j] \OR (R(k-1)[i, k] \AND R(k-1)[k, j]) \]

return \( R(n) \)

Find Transitive closure for the given digraph using Warshall’s algorithm.

\[ \begin{array}{cccc}
A & B & C & D \\
A & 0 & 0 & 1 & 0 \\
B & 1 & 0 & 0 & 1 \\
C & 0 & 0 & 0 & 0 \\
D & 0 & 1 & 0 & 0 \\
\end{array} \]

**Solution:**
### Design and Analysis of Algorithms

#### 10CS43

**Dept of CSE, SJBIT**

**Page 47**

<table>
<thead>
<tr>
<th>R(0)</th>
<th>k = 1</th>
<th>Vertex 1 can be intermediate node</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

R1[2,3] = R0[2,3] OR R0[2,1] AND R0[1,3] = 0 OR (1 AND 1) = 1

<table>
<thead>
<tr>
<th>R(1)</th>
<th>k = 2</th>
<th>Vertex {1,2} can be intermediate nodes</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

R2[4,1] = R1[4,1] OR R1[4,2] AND R1[2,1] = 0 OR (1 AND 1) = 1

<table>
<thead>
<tr>
<th>R(2)</th>
<th>k = 3</th>
<th>Vertex {1,2,3} can be intermediate nodes</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

R2[4,3] = R1[4,3] OR R1[4,2] AND R1[2,3] = 0 OR (1 AND 1) = 1


<table>
<thead>
<tr>
<th>R(3)</th>
<th>k = 4</th>
<th>Vertex {1,2,3,4} can be intermediate nodes</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>


**NO CHANGE**
Efficiency:
- Time efficiency is $\Theta(n^3)$
- Space efficiency: Requires extra space for separate matrices for recording intermediate results of the algorithm.

4.3 Floyd’s Algorithm to find - ALL PAIRS SHORTEST PATHS

Some useful definitions:
- **Weighted Graph**: Each edge has a weight (associated numerical value). Edge weights may represent costs, distance/lengths, capacities, etc. depending on the problem.
- **Weight matrix**: $W(i,j)$ is
  - 0 if $i=j$
  - $\infty$ if no edge b/n i and j.
  - “weight of edge” if edge b/n i and j.

**Problem statement:**

Given a weighted graph $G(V, E_w)$, the all-pairs shortest paths problem is to find the shortest path between every pair of vertices $(v_i, v_j) \in V$.

**Solution:**
A number of algorithms are known for solving All pairs shortest path problem
- **Matrix multiplication based algorithm**
- **Dijkstra's algorithm**
- **Bellman-Ford algorithm**
- **Floyd's algorithm**

**Underlying idea of Floyd’s algorithm:**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

$R_4[2,2] = R_3[2,2] \lor R_3[2,4] \land R_3[4,2] = 0 \lor (1 \land 1) = 1$

TRANSITIVE CLOSURE for the given graph
• Let $W$ denote the initial weight matrix.
• Let $D(k)[i, j]$ denote cost of shortest path from $i$ to $j$ whose intermediate vertices are a subset of \{1,2,\ldots,k\}.

**Recursive Definition**

**Case 1:**
A shortest path from $v_i$ to $v_j$ restricted to using only vertices from $\{v_1,v_2,\ldots,v_k\}$ as intermediate vertices does not use $v_k$. Then

$$D(k)[i, j] = D(k-1)[i, j].$$

**Case 2:**
A shortest path from $v_i$ to $v_j$ restricted to using only vertices from $\{v_1,v_2,\ldots,v_k\}$ as intermediate vertices do use $v_k$. Then

$$D(k)[i, j] = D(k-1)[i, k] + D(k-1)[k, j].$$

**We conclude:**

$$D(k)[i, j] = \min \{ D(k-1)[i, j], D(k-1)[i, k] + D(k-1)[k, j] \}$$

**Algorithm:**

**Algorithm Floyd(W[1..n, 1..n])**

// Implements Floyd’s algorithm

// Input: Weight matrix W

// Output: Distance matrix of shortest paths’ length

D W

for $k \rightarrow 1$ to $n$ do

  for $i \rightarrow 1$ to $n$ do

    for $j \rightarrow 1$ to $n$ do

      $D[i, j] \rightarrow \min \{ D[i, j], D[i, k] + D[k, j] \}$

return D

**Example:**

Find All pairs shortest paths for the given weighted connected graph using Floyd’s algorithm.

![Graph](image)

**Solution:**

$$D(0) =$$

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>B</td>
<td>4</td>
<td>0</td>
<td>$\infty$</td>
</tr>
<tr>
<td>C</td>
<td>$\infty$</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>
### 4.4 0/1 Knapsack Problem

**Memory function**

- **D(0)**
  - \( k = 1 \)
  - **Vertex 1** can be an intermediate node
  
  \[
  \begin{array}{ccc}
  A & B & C \\
  0 & 2 & 5 \\
  4 & 0 & \infty \\
  \infty & 3 & 0 \\
  \end{array}
  \]

  \[
  D(0)[2,3] = \min \{ D(0)[2,3], D(0)[2,1] + D(0)[1,3] \} = \min \{ \infty, (4 + 5) \} = 9
  \]

- **D(1)**
  - \( k = 2 \)
  - **Vertex 1, 2** can be intermediate nodes
  
  \[
  \begin{array}{ccc}
  A & B & C \\
  0 & 2 & 5 \\
  4 & 0 & 9 \\
  \infty & 3 & 0 \\
  \end{array}
  \]

  \[
  D(1)[3,1] = \min \{ D(1)[3,1], D(1)[3,2] + D(1)[2,1] \} = \min \{ \infty, (4 + 3) \} = 7
  \]

- **D(2)**
  - \( k = 3 \)
  - **Vertex 1, 2, 3** can be intermediate nodes
  
  \[
  \begin{array}{ccc}
  A & B & C \\
  0 & 2 & 5 \\
  4 & 0 & 9 \\
  7 & 3 & 0 \\
  \end{array}
  \]

- **D(3)**
  - NO Change
  
  \[
  \begin{array}{ccc}
  A & B & C \\
  0 & 2 & 5 \\
  4 & 0 & 9 \\
  7 & 3 & 0 \\
  \end{array}
  \]

**ALL PAIRS SHORTEST PATHS for the given graph**

---

**4.4 0/1 Knapsack Problem Memory function**

Dept of CSE, SJBIT
**Definition:**
Given a set of n items of known weights w1, . . . , wn and values v1, . . . , vn and a knapsack of capacity W, the problem is to find the most valuable subset of the items that fit into the knapsack.

Knapsack problem is an OPTIMIZATION PROBLEM.

**Dynamic programming approach to solve knapsack problem**

**Step 1:**
Identify the smaller sub-problems. If items are labeled 1 . . . n, then a sub-problem would be to find an optimal solution for Sk = {items labeled 1, 2, . . . k}.

**Step 2:**
Recursively define the value of an optimal solution in terms of solutions to smaller problems.

Initial conditions:

\[ V[0, j] = 0 \quad \text{for } j \geq 0 \]
\[ V[i, 0] = 0 \quad \text{for } i \geq 0 \]

Recursive step:

\[ V[i, j] = \begin{cases} 
\max \{ V[i-1, j], vi + V[i-1, j - wi] \} & \text{if } j - wi \geq 0 \\
V[i-1, j] & \text{if } j - wi < 0 
\end{cases} \]

**Step 3:**
Bottom up computation using iteration.

**Question:**
Apply bottom-up dynamic programming algorithm to the following instance of the knapsack problem Capacity W = 5.

<table>
<thead>
<tr>
<th>Item #</th>
<th>Weight (Kg)</th>
<th>Value (Rs.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

**Solution:**
Using dynamic programming approach, we have:
<table>
<thead>
<tr>
<th>Step</th>
<th>Calculation</th>
<th>Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><strong>Initial conditions:</strong>&lt;br&gt;V[0, j] = 0 for j ≥ 0&lt;br&gt;V[i, 0] = 0 for i ≥ 0</td>
<td><img src="image1" alt="Table 1" /></td>
</tr>
<tr>
<td>2</td>
<td>W1 = 2,&lt;br&gt;Available knapsack capacity = 1&lt;br&gt;W1 &gt; WA, CASE 1 holds:&lt;br&gt;V[i, j] = V[i-1, j]&lt;br&gt;V[1,1] = V[0, 1] = 0</td>
<td><img src="image2" alt="Table 2" /></td>
</tr>
<tr>
<td>3</td>
<td>W1 = 2,&lt;br&gt;Available knapsack capacity = 2&lt;br&gt;W1 = WA, CASE 2 holds:&lt;br&gt;V[i, j] = max { V[i-1, j], vi + V[i-1, j - wi] }&lt;br&gt;V[1,2] = max { V[0, 2], 3 + V[0, 0] }&lt;br&gt;= max { 0, 3 + 0 } = 3</td>
<td><img src="image3" alt="Table 3" /></td>
</tr>
<tr>
<td>4</td>
<td>W1 = 2,&lt;br&gt;Available knapsack capacity = 3,4,5&lt;br&gt;W1 &lt; WA, CASE 2 holds:&lt;br&gt;V[i, j] = max { V[i-1, j], vi + V[i-1, j - wi] }&lt;br&gt;V[1,3] = max { V[0, 3], 3 + V[0, 1] }&lt;br&gt;= max { 0, 3 + 0 } = 3</td>
<td><img src="image4" alt="Table 4" /></td>
</tr>
<tr>
<td>5</td>
<td>W2 = 3,&lt;br&gt;Available knapsack capacity = 1&lt;br&gt;W2 &gt; WA, CASE 1 holds:&lt;br&gt;V[i, j] = V[i-1, j]&lt;br&gt;V[2,1] = V[1, 1] = 0</td>
<td><img src="image5" alt="Table 5" /></td>
</tr>
</tbody>
</table>
### Case 1
For $W_2 = 3$, available knapsack capacity = 2

$W_2 > W_A$, CASE 1 holds:

$$V[i, j] = V[i-1, j]$$

$V[2,2] = V[1,2] = 3$

<table>
<thead>
<tr>
<th>$j$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i=0$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$i=1$</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>$i=2$</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>$i=3$</td>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$i=4$</td>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Case 2
For $W_2 = 3$, available knapsack capacity = 3

$W_2 = W_A$, CASE 2 holds:

$$V[i, j] = \max \{ V[i-1, j], \, \, \, v_i + V[i-1, j - w_i] \}$$

$V[2,3] = \max \{ V[1,3], \, \, \, 4 + V[1,0] \} = \max \{ 3, 4 + 0 \} = 4$

<table>
<thead>
<tr>
<th>$j$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i=0$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$i=1$</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>$i=2$</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>$i=3$</td>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$i=4$</td>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For $W_2 = 3$, available knapsack capacity = 4

$W_2 < W_A$, CASE 2 holds:

$$V[i, j] = \max \{ V[i-1, j], \, \, \, v_i + V[i-1, j - w_i] \}$$

$V[2,4] = \max \{ V[1,4], \, \, \, 4 + V[1,1] \} = \max \{ 3, 4 + 0 \} = 4$

<table>
<thead>
<tr>
<th>$j$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i=0$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$i=1$</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>$i=2$</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>$i=3$</td>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$i=4$</td>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For $W_2 = 3$, available knapsack capacity = 5

$W_2 < W_A$, CASE 2 holds:

$$V[i, j] = \max \{ V[i-1, j], \, \, \, v_i + V[i-1, j - w_i] \}$$

$V[2,5] = \max \{ V[1,5], \, \, \, 4 + V[1,2] \} = \max \{ 3, 4 + 3 \} = 7$

<table>
<thead>
<tr>
<th>$j$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i=0$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$i=1$</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>$i=2$</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>$i=3$</td>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$i=4$</td>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For $W_3 = 4$, available knapsack capacity = 1,2,3

$W_3 > W_A$, CASE 1 holds:

$$V[i, j] = V[i-1, j]$$

### Table

<table>
<thead>
<tr>
<th>$j$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i=0$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$i=1$</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>$i=2$</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>$i=3$</td>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$i=4$</td>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Case</td>
<td>W3</td>
<td>Available knapsack capacity</td>
<td>W3 = WA</td>
<td>Case 2 holds</td>
<td>( V[i,j] )</td>
<td>( j = 0 )</td>
</tr>
<tr>
<td>------</td>
<td>----</td>
<td>-----------------------------</td>
<td>--------</td>
<td>--------------</td>
<td>----------------</td>
<td>-------</td>
</tr>
<tr>
<td>11</td>
<td>4</td>
<td>4</td>
<td>WA</td>
<td>CASE 2 holds</td>
<td>( V[3,4] ) = \max { \ V[2,4], \ 5 + V[2,0] } }</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>( V[3,4] ) = \max { \ V[2,4], \ 5 + V[2,0] } }</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>( V[3,4] ) = \max { \ V[2,4], \ 5 + V[2,0] } }</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>( V[3,4] ) = \max { \ V[2,4], \ 5 + V[2,0] } }</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>( V[3,4] ) = \max { \ V[2,4], \ 5 + V[2,0] } }</td>
<td>4</td>
</tr>
<tr>
<td>12</td>
<td>5</td>
<td>5</td>
<td>WA</td>
<td>CASE 2 holds</td>
<td>( V[3,5] ) = \max { \ V[2,5], \ 5 + V[2,1] } }</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>( V[3,5] ) = \max { \ V[2,5], \ 5 + V[2,1] } }</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>( V[3,5] ) = \max { \ V[2,5], \ 5 + V[2,1] } }</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>( V[3,5] ) = \max { \ V[2,5], \ 5 + V[2,1] } }</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>( V[3,5] ) = \max { \ V[2,5], \ 5 + V[2,1] } }</td>
<td>4</td>
</tr>
<tr>
<td>13</td>
<td>5</td>
<td>1,2,3,4</td>
<td>WA</td>
<td>CASE 1 holds</td>
<td>( V[4,5] ) = \max { \ V[3,5], \ 6 + V[3,0] } }</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>( V[4,5] ) = \max { \ V[3,5], \ 6 + V[3,0] } }</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>( V[4,5] ) = \max { \ V[3,5], \ 6 + V[3,0] } }</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>( V[4,5] ) = \max { \ V[3,5], \ 6 + V[3,0] } }</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>( V[4,5] ) = \max { \ V[3,5], \ 6 + V[3,0] } }</td>
<td>4</td>
</tr>
</tbody>
</table>

Maximal value is \( V[4,5] = 7/- \)

What is the composition of the optimal subset?
The composition of the optimal subset if found by tracing back the computations for the entries in the table.
Re

marks


ITEM 4 NOT included in the subset


ITEM 3 NOT included in the subset

\[ V[2, 5] \neq V[1, 5] \]

ITEM 2 included in the subset

\[ V[1, 2] \neq V[0, 2] \]

ITEM 1 included in the subset

Optimal subset: \{ item 1, item 2 \}

Total weight is: 5kg \ (2kg + 3kg)
Total profit is: 7/- \ (3/- + 4/-)

### Table 1

<table>
<thead>
<tr>
<th>Step</th>
<th>Table</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>j=0</td>
<td>1 2 3 4 5</td>
</tr>
<tr>
<td>i=0</td>
<td>0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0 0 3 3 3 3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0 0 3 4 4 7</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0 0 3 4 5 7</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0 0 3 4 5 7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>j=0</td>
<td>1 2 3 4 5</td>
</tr>
<tr>
<td>i=0</td>
<td>0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0 0 3 3 3 3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0 0 3 4 4 7</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0 0 3 4 5 7</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0 0 3 4 5 7</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>[ V[i,j] ]</td>
</tr>
<tr>
<td></td>
<td>j=0</td>
<td>1 2 3 4 5</td>
</tr>
<tr>
<td>i=0</td>
<td>0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0 0 3 3 3 3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0 0 3 4 4 7</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0 0 3 4 5 7</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0 0 3 4 5 7</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>[ V[i,j] ]</td>
</tr>
<tr>
<td></td>
<td>j=0</td>
<td>1 2 3 4 5</td>
</tr>
<tr>
<td>i=0</td>
<td>0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0 0 3 3 3 3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0 0 3 4 4 7</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0 0 3 4 5 7</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0 0 3 4 5 7</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>[ V[i,j] ]</td>
</tr>
<tr>
<td></td>
<td>j=0</td>
<td>1 2 3 4 5</td>
</tr>
<tr>
<td>i=0</td>
<td>0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0 0 3 3 3 3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0 0 3 4 4 7</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0 0 3 4 5 7</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0 0 3 4 5 7</td>
<td></td>
</tr>
</tbody>
</table>
Efficiency:
- Running time of Knapsack problem using dynamic programming algorithm is: \((n \times W)\)
- Time needed to find the composition of an optimal solution is: \(O(n + W)\)

Memory function
- Memory function combines the strength of top-down and bottom-up approaches
- It solves ONLY sub-problems that are necessary and does it ONLY ONCE.

The method:
- Uses top-down manner.
- Maintains table as in bottom-up approach.
- Initially, all the table entries are initialized with special “null” symbol to indicate that they have not yet been calculated.
- Whenever a new value needs to be calculated, the method checks the corresponding entry in the table first:
  - If entry is NOT “null”, it is simply retrieved from the table.
  - Otherwise, it is computed by the recursive call whose result is then recorded in the table.

Algorithm:

Algorithm MFKnap( i, j )
if \(V[i, j] < 0\)
  if \(j < \text{Weights}[i]\)
    value → MFKnap( i-1, j )
  else
    value → max \{MFKnap( i-1, j ),
      \(\text{Values}[i] + \text{MFKnap}( i-1, j - \text{Weights}[i])\)\}

\(V[i, j] → value\)
return \(V[i, j]\)

Example:
Apply memory function method to the following instance of the knapsack problem
Capacity \(W = 5\)

<table>
<thead>
<tr>
<th>Item #</th>
<th>Weight (Kg)</th>
<th>Value (Rs.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

Solution:
Using memory function approach, we have:
Initially, all the table entries are initialized with special “null” symbol to indicate that they have not yet been calculated. Here null is indicated with -1 value.

\[
V[i,j] = \begin{cases} 
0 & j=0 \\
1 & j=1 \\
2 & j=2 \\
3 & j=3 \\
4 & j=4 \\
5 & j=5 
\end{cases}
\]

\[
\begin{array}{ccccccc}
  i=0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  1 & 0 & -1 & -1 & -1 & -1 & -1 \\
  2 & 0 & -1 & -1 & -1 & -1 & -1 \\
  3 & 0 & -1 & -1 & -1 & -1 & -1 \\
  4 & 0 & -1 & -1 & -1 & -1 & -1 \\
\end{array}
\]

\[
V[1,5] = 3
\]

\[
\begin{array}{ccccccc}
  i=0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  1 & 0 & -1 & -1 & -1 & -1 & 3 \\
  2 & 0 & -1 & -1 & -1 & -1 & -1 \\
  3 & 0 & -1 & -1 & -1 & -1 & -1 \\
  4 & 0 & -1 & -1 & -1 & -1 & -1 \\
\end{array}
\]

\[
V[1,2] = 3
\]

\[
\begin{array}{ccccccc}
  i=0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  1 & 0 & -1 & 3 & -1 & -1 & 3 \\
  2 & 0 & -1 & -1 & -1 & -1 & -1 \\
  3 & 0 & -1 & -1 & -1 & -1 & -1 \\
  4 & 0 & -1 & -1 & -1 & -1 & -1 \\
\end{array}
\]

\[
V[2,5] = 7
\]

\[
\begin{array}{ccccccc}
  i=0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  1 & 0 & -1 & 3 & -1 & -1 & 3 \\
  2 & 0 & -1 & -1 & -1 & -1 & 7 \\
  3 & 0 & -1 & -1 & -1 & -1 & -1 \\
  4 & 0 & -1 & -1 & -1 & -1 & -1 \\
\end{array}
\]
Efficiency:
- Time efficiency same as bottom up algorithm: $O(n \times W) + O(n + W)$
- Just a constant factor gain by using memory function
- Less space efficient than a space efficient version of a bottom-up algorithm
UNIT-5
DECREASE-AND-CONQUER APPROACHES, SPACE-TIME TRADEOFFS

5.1 Decrease and conquer: Introduction

Decrease & conquer is a general algorithm design strategy based on exploiting the relationship between a solution to a given instance of a problem and a solution to a smaller instance of the same problem. The exploitation can be either top-down (recursive) or bottom-up (non-recursive).

The major variations of decrease and conquer are

1. Decrease by a constant: (usually by 1):
   a. insertion sort
   b. graph traversal algorithms (DFS and BFS)
   c. topological sorting
   d. algorithms for generating permutations, subsets

2. Decrease by a constant factor (usually by half)
   a. binary search and bisection method

3. Variable size decrease
   a. Euclid’s algorithm

Following diagram shows the major variations of decrease & conquer approach.

Decrease by a constant: (usually by 1):
Decrease by a constant factor (usually by half)

5.2 Insertion sort

Description:
Insertion sort is an application of decrease & conquer technique. It is a comparison based sort in which the sorted array is built on one entry at a time.

\[ A[0] \leq \ldots \leq A[j] < A[j + 1] \leq \ldots \leq A[i - 1] \mid A[i] \ldots A[n - 1] \]

smaller than or equal to \( A[i] \)  
greater than \( A[i] \)

Algorithm:

**ALGORITHM Insertionsort(A [0 … n-1])**
//sorts a given array by insertion sort
//i/p: Array A[0…n-1]
//o/p: sorted array A[0…n-1] in ascending order

for i → 1 to n-1
  V → A[i]
  j → i-1
  while j ≥ 0 AND A[j] > V do
    j → j – 1
  A[j + 1] → V

Analysis:

- **Input size:** Array size, \( n \)
- **Basic operation:** key comparison
- **Best, worst, average case exists**
  Best case: when input is a sorted array in ascending order:
Worst case: when input is a sorted array in descending order:

- Let $C_{\text{worst}}(n)$ be the number of key comparison in the worst case. Then

\[
C_{\text{worst}}(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = \frac{(n - 1)n}{2} \in \Theta(n^2).
\]

- Let $C_{\text{best}}(n)$ be the number of key comparison in the best case. Then

\[
C_{\text{best}}(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n).
\]

**Example:**
Sort the following list of elements using insertion sort:
89, 45, 68, 90, 29, 34, 17

<table>
<thead>
<tr>
<th>89</th>
<th>45</th>
<th>68</th>
<th>90</th>
<th>29</th>
<th>34</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>89</td>
<td>68</td>
<td>90</td>
<td>29</td>
<td>34</td>
<td>17</td>
</tr>
<tr>
<td>45</td>
<td>68</td>
<td>89</td>
<td>90</td>
<td>29</td>
<td>34</td>
<td>17</td>
</tr>
<tr>
<td>45</td>
<td>68</td>
<td>89</td>
<td>90</td>
<td>29</td>
<td>34</td>
<td>17</td>
</tr>
<tr>
<td>29</td>
<td>45</td>
<td>68</td>
<td>89</td>
<td>90</td>
<td>34</td>
<td>17</td>
</tr>
<tr>
<td>29</td>
<td>34</td>
<td>45</td>
<td>68</td>
<td>89</td>
<td>90</td>
<td>17</td>
</tr>
<tr>
<td>17</td>
<td>29</td>
<td>34</td>
<td>45</td>
<td>68</td>
<td>89</td>
<td>90</td>
</tr>
</tbody>
</table>

**Advantages of insertion sort:**
- Simple implementation. There are three variations
  - Left to right scan
  - Right to left scan
o Binary insertion sort
  • Efficient on small list of elements, on almost sorted list
  • Running time is linear in best case
  • Is a stable algorithm
  • Is a in-place algorithm

5.3 Depth-first search (DFS) and Breadth-first search (BFS)

DFS and BFS are two graph traversing algorithms and follow decrease and conquer approach – decrease by one variation to traverse the graph

Some useful definition:
  • Tree edges: edges used by DFS traversal to reach previously unvisited vertices
  • Back edges: edges connecting vertices to previously visited vertices other than their immediate predecessor in the traversals
  • Cross edges: edge that connects an unvisited vertex to vertex other than its immediate predecessor. (connects siblings)
  • DAG: Directed acyclic graph

Depth-first search (DFS)
Description:
  • DFS starts visiting vertices of a graph at an arbitrary vertex by marking it as visited.
  • It visits graph’s vertices by always moving away from last visited vertex to an unvisited one, backtracks if no adjacent unvisited vertex is available.
  • Is a recursive algorithm, it uses a stack
  • A vertex is pushed onto the stack when it’s reached for the first time
  • A vertex is popped off the stack when it becomes a dead end, i.e., when
there is no adjacent unvisited vertex
- “Redraws” graph in tree-like fashion (with tree edges and back edges
  for undirected graph)

**Algorithm:**

**ALGORITHM DFS (G)**

//implements DFS traversal of a given graph
//i/p: Graph G = { V, E}
//o/p: DFS tree

```
Mark each vertex in V with 0 as a mark of being “unvisited”

count→ 0
for each vertex v in V do
  if v is marked with 0
    dfs(v)

dfs(v)
  count→ count + 1
mark v with count
for each vertex w in V adjacent to v do
  if w is marked with 0
    dfs(w)
```

**Example:**
Starting at vertex A traverse the following graph using DFS traversal method:

![Graph Diagram]

**Solution:**

<table>
<thead>
<tr>
<th>Step</th>
<th>Graph</th>
<th>Remarks</th>
</tr>
</thead>
</table>
| 1    | ![Step 1 Graph](image1) | Insert A into stack  
A(1) |
| 2    | ![Step 2 Graph](image2) | Insert B into stack  
B (2)  
A(1) |
| 3    | ![Step 3 Graph](image3) | Insert F into stack  
F (3)  
B (2)  
A(1) |
Design and Analysis of Algorithms 10CS43

Dept of CSE,SJBIT

Page 67
Insert C into stack

Insert D into stack

Insert H into stack

NO unvisited adjacent vertex for H, backtrack

Delete H from stack
12  NO unvisited adjacent vertex for D, backtrack
Delete D from stack
H (8, 3)
D (7, 4)
E (4, 1)
F (3, 2)
B (2)
A(1)

13  NO unvisited adjacent vertex for C, backtrack
Delete C from stack
H (8, 3)
D (7, 4)
E (4, 1)
F (3, 2)
B (2)
A(1)

14  NO unvisited adjacent vertex for G, backtrack
Delete G from stack
H (8, 3)
D (7, 4)
E (4, 1)
F (3, 2)
B (2)
A(1)  

15  NO unvisited adjacent vertex for B, backtrack  

Delete B from stack  
- H (8, 3)  
- D (7, 4)  
- E (4, 1)  
- C (6, 5)  
- F (3, 2)  
- G (5, 6)  
- B (2, 7)  
- A(1)  

16  NO unvisited adjacent vertex for A, backtrack  

Delete A from stack  
- H (8, 3)  
- D (7, 4)  
- E (4, 1)  
- C (6, 5)  
- G (5, 6)  
- B (2, 7)  
- A(1, 8)  

Stack becomes empty. Algorithm stops as all the nodes in the given graph are visited.

The DFS tree is as follows: (dotted lines are back edges)
Applications of DFS:
- The two orderings are advantageous for various applications like topological sorting, etc.
- To check connectivity of a graph (number of times stack becomes empty tells the number of components in the graph)
- To check if a graph is acyclic. (no back edges indicates no cycle)
- To find articulation point in a graph

Efficiency:
- Depends on the graph representation:
  - Adjacency matrix: $\Theta(n^2)$
  - Adjacency list: $\Theta(n + e)$

Breadth-first search (BFS)
Description:
- BFS starts visiting vertices of a graph at an arbitrary vertex by marking it
as visited.
• It visits graph’s vertices by across to all the neighbors of the last visited vertex
• Instead of a stack, BFS uses a queue
• Similar to level-by-level tree traversal
• “Redraws” graph in tree-like fashion (with tree edges and cross edges for undirected graph)

Algorithm:

ALGORITHM BFS (G)
//implements BFS traversal of a given graph
//i/p: Graph G = { V, E}
//o/p: BFS tree/forest
Mark each vertex in V with 0 as a mark of being “unvisited”
count→ 0
for each vertex v in V do
  if v is marked with 0
    bfs(v)

bfs(v)
count→ count + 1
mark v with count and initialize a queue with v
while the queue is NOT empty do
  for each vertex w in V adjacent to front’s vertex v do
    if w is marked with 0
      count→ count + 1
      mark w with count
      add w to the queue
  remove vertex v from the front of the queue

Example:
Starting at vertex A traverse the following graph using BFS traversal method:
Solution:

<table>
<thead>
<tr>
<th>Step</th>
<th>Graph</th>
<th>Remarks</th>
</tr>
</thead>
</table>
| 1    | ![Graph](image.png) | Insert A into queue  
A(1) |
| 2    | ![Graph](image.png) | Insert B, E into queue  
A(1), B(2), E(3)  
B(2), E(3) |
| 3    | ![Graph](image.png) | Insert F, G into queue  
B(2), E(3), F(3), G(4) |
E(3), F(3), G(4)

4  | NO unvisited adjacent vertex for E, backtrack | Delete E from queue  
   |                                           | F(3), G(4)          
5  | NO unvisited adjacent vertex for F, backtrack | Delete F from queue  
   |                                           | G(4)               
6  | Insert C, H into queue                      |                  
7  | G(4), C(5), H(6)                            | C(5), H(6)         
8  | Insert D into queue                         |                  
9  | NO unvisited adjacent vertex for H, backtrack | Delete H from queue  
   |                                           | D(7)               
   | NO unvisited adjacent vertex for D, backtrack | Delete D from queue  
   |                                           |                  

Queue becomes empty. Algorithm stops as all the nodes in the given graph are visited

The BFS tree is as follows: (dotted lines are cross edges)
Applications of BFS:
- To check connectivity of a graph (number of times queue becomes empty tells the number of components in the graph)
- To check if a graph is acyclic. (no cross edges indicates no cycle)
- To find minimum edge path in a graph

Efficiency:
- Depends on the graph representation:
  - Array: $\Theta(n^2)$
  - List: $\Theta(n + e)$

Difference between DFS & BFS:
### DFS vs BFS

<table>
<thead>
<tr>
<th>Data structure</th>
<th>DFS</th>
<th>BFS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td></td>
<td>Queue</td>
</tr>
<tr>
<td>No. of vertex orderings</td>
<td>2 orderings</td>
<td>1 ordering</td>
</tr>
<tr>
<td>Edge types</td>
<td>Tree edge</td>
<td>Tree edge</td>
</tr>
<tr>
<td></td>
<td>Back edge</td>
<td>Cross edge</td>
</tr>
<tr>
<td>Applications</td>
<td>Connectivity</td>
<td>Connectivity</td>
</tr>
<tr>
<td></td>
<td>Acyclicity</td>
<td>Acyclicity</td>
</tr>
<tr>
<td></td>
<td>Articulation points</td>
<td>Minimum edge paths</td>
</tr>
</tbody>
</table>

- **Efficiency for adjacency matrix**: \( \Theta(n^2) \) for both DFS and BFS.
- **Efficiency for adjacency lists**: \( \Theta(n + e) \) for both DFS and BFS.

### 5.4 Topological Sorting

**Description:**
Topological sorting is a sorting method to list the vertices of the graph in such an order that for every edge in the graph, the vertex where the edge starts is listed before the vertex where the edge ends.

**NOTE:** There is no solution for topological sorting if there is a cycle in the digraph. [MUST be a DAG]

Topological sorting problem can be solved by using
1. DFS method
2. Source removal method

**DFS Method:**
- Perform DFS traversal and note the order in which vertices become dead
ends (popped order)
• Reverse the order, yield the topological sorting.

**Example:**
Apply DFS – based algorithm to solve the topological sorting problem for the given graph:

![Graph with nodes C1, C2, C3, C4, C5]

<table>
<thead>
<tr>
<th>Step</th>
<th>Graph</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><img src="C1" alt="Graph" /></td>
<td>Insert C1 into stack</td>
</tr>
<tr>
<td></td>
<td><img src="C1" alt="Graph" /></td>
<td>C1(1)</td>
</tr>
<tr>
<td>2</td>
<td>![Graph](C1 C3)</td>
<td>Insert C2 into stack</td>
</tr>
<tr>
<td></td>
<td>![Graph](C1 C3)</td>
<td>C2 (2)</td>
</tr>
<tr>
<td></td>
<td>![Graph](C1 C3)</td>
<td>C1(1)</td>
</tr>
<tr>
<td>3</td>
<td>![Graph](C1 C3 C4)</td>
<td>Insert C4 into stack</td>
</tr>
<tr>
<td></td>
<td>![Graph](C1 C3 C4)</td>
<td>C4 (3)</td>
</tr>
<tr>
<td></td>
<td>![Graph](C1 C3 C4)</td>
<td>C2 (2)</td>
</tr>
<tr>
<td>Step</td>
<td>Action</td>
<td>Operation</td>
</tr>
<tr>
<td>------</td>
<td>--------</td>
<td>-----------</td>
</tr>
<tr>
<td></td>
<td>Insert C5 into stack</td>
<td>C5 (4)</td>
</tr>
<tr>
<td></td>
<td>Delete C5 from stack</td>
<td>C5 (4, 1)</td>
</tr>
<tr>
<td></td>
<td>NO unvisited adjacent vertex for C5, backtrack</td>
<td>C4 (3)</td>
</tr>
<tr>
<td></td>
<td>Delete C5 from stack</td>
<td>C4 (3)</td>
</tr>
<tr>
<td></td>
<td>NO unvisited adjacent vertex for C4, backtrack</td>
<td>C2 (2)</td>
</tr>
<tr>
<td></td>
<td>Delete C5 from stack</td>
<td>C2 (2)</td>
</tr>
<tr>
<td></td>
<td>NO unvisited adjacent vertex for C3, backtrack</td>
<td>C1(1)</td>
</tr>
<tr>
<td></td>
<td>Delete C5 from stack</td>
<td>C1(1)</td>
</tr>
<tr>
<td></td>
<td>NO unvisited adjacent vertex for C1, backtrack</td>
<td>Delete C1 from stack</td>
</tr>
</tbody>
</table>

Dept of CSE, SJBIT
Stack becomes empty, but there is a node which is unvisited, therefore start the DFS again from arbitrarily selecting a unvisited node as source.

<table>
<thead>
<tr>
<th>Node</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>C5</td>
<td>Insert C2 into stack</td>
</tr>
<tr>
<td>C4</td>
<td>C2 (3, 2)</td>
</tr>
<tr>
<td>C2</td>
<td>C2 (2, 3)</td>
</tr>
<tr>
<td>C1</td>
<td>C1 (1, 4)</td>
</tr>
<tr>
<td>C2</td>
<td>C2 (5)</td>
</tr>
</tbody>
</table>

Insert C2 into stack

Insert C2 into stack

Insert C2 into stack

Delete C2 from stack

Delete C2 from stack

Stack becomes empty, NO unvisited node left, therefore algorithm stops.

The popping–off order is:
C5, C4, C3, C1, C2,

Topologically sorted list (reverse of pop order):
C2, C1, C3, C4, C5

**Source removal method:**

Dept of CSE, SJB IT
- Purely based on decrease & conquer
- Repeatedly identify in a remaining digraph a source, which is a vertex with no incoming edges
- Delete it along with all the edges outgoing from it.

**Example:**

Apply Source removal – based algorithm to solve the topological sorting problem for the given graph:

```
Original Graph:

C1 -> C3 -> C4
C2 -> C3 -> C5

Solution:

Delete C1

C1 -> C3 -> C4
C2 -> C3 -> C5

Delete C2

C1 -> C3 -> C4
C4 -> C5

Delete C3

C1 -> C4
C4 -> C5

```

Dept of CSE, SJBIT
The topological order is C1, C2, C3, C4, C5

5.5 Space-Time Tradeoffs: Introduction

Two varieties of space-for-time algorithms:

- **input enhancement** — preprocess the input (or its part) to store some info to be used later in solving the problem
  - counting sorts
  - string searching algorithms

- **pre structuring** — preprocess the input to make accessing its elements easier
5.6 SORTING BY COUNTING

Assume elements to be sorted belong to a known set of small values between \( l \) and \( u \), with potential duplication.

Constraint: we cannot overwrite the original list Distribution Counting: compute the frequency of each element and later accumulate sum of frequencies (distribution)

Algorithm:

```plaintext
for j ← 0 to u-l do D[j] ← 0 // init frequencies
for i ← n-l downto 0 do
    j ← A[i] - 1
    D[j] ← D[j] - 1
return S
```
Example: $A = \begin{bmatrix} 13 & 11 & 12 & 13 & 12 & 12 \\ \end{bmatrix}$

<table>
<thead>
<tr>
<th>Array Values</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequencies</td>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Distribution</td>
<td>1</td>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>13</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>5</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>5</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>13</td>
</tr>
</tbody>
</table>

Efficiency: $\Theta(n)$

Best so far but only for specific types of input
5.7 INPUT ENHANCEMENT IN STRING MATCHING

Horspool’s Algorithm

A simplified version of Boyer-Moore algorithm:

- preprocesses pattern to generate a shift table that determines how much to shift the pattern when a mismatch occurs
- always makes a shift based on the text’s character \( c \) aligned with the last compared (mismatched) character in the pattern according to the shift table’s entry for \( c \)

### How far to shift?

Look at first (rightmost) character in text that was compared:

- The character is not in the pattern
  
  \[ \text{BAOBAB} \]
  
  \( c \) not in pattern

- The character is in the pattern (but not the rightmost)
  
  \[ \text{BAOBAB} \]
  
  \( O \) occurs once in pattern

  \[ \text{BAOBAB} \]
  
  \( A \) occurs twice in pattern

- The rightmost characters do match
  
  \[ \text{BAOBAB} \]
Shift table

- Shift sizes can be precomputed by the formula:
  \[ d_{\text{shift}}(c) = \begin{cases} 
  \text{distance from } c \text{'s rightmost occurrence in pattern among its first } m-1 \text{ characters to its right end} \\
  \text{pattern's length } m, \text{ otherwise} 
\end{cases} \]

by scanning pattern before search begins and stored in a table called *shift table*. After the shift, the right end of pattern is \( d_{\text{shift}}(c) \) positions to the right of the last compared character in text.

- Shift table is indexed by text and pattern alphabet.
  *E.g.*, for \texttt{BAOBAB}:

|        | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
| Index  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10| 11| 12| 13| 14| 15| 16| 17| 18| 19| 20| 21| 22| 23| 24| 25|

Example of Horspool’s algorithm

|        | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | _ |
| Index  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10| 11| 12| 13| 14| 15| 16| 17| 18| 19| 20| 21| 22| 23| 24| 25|

\texttt{BAOBAB (unsuccessful search)}

If \( k \) characters are matched before the mismatch, then the shift distance is \( d_{\text{shift}} = d_{\text{shift}}(c) - k \).

\[ \begin{array}{c}
\cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \text{...bzyx} \end{array} \]

Note that the shift could be negative!

*E.g.*, if text = ...\texttt{ABAB} B...

\[ \begin{array}{c}
\cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \text{...bzyx} \end{array} \]