**KALINGA UNIVERSITY, NAYA RAIPUR**

**Faculty of Computer Science & IT**

**Programme- Bachelor of Computer Applications (AIML)**

**Course Name- Design and Analysis of Algorithm**

**Course Code- BCAAIML504** **5th Semester**

**Unit-III**

# DYNAMIC PROGRAMMING TECHNIQUE

## DYNAMIC PROGRAMMING

Dynamic programming is a technique for solving problems with overlapping subproblems. Typically, these subproblems arise from a recurrence relating a given problem's solution to solutions of its smaller subproblems. Rather than solving overlapping subproblems again and again, dynamic programming suggests solving each of the smaller subproblems only once and recording the results in a table from which a solution to the original problem can then be obtained.

This technique can be illustrated by revisiting the Fibonacci numbers. The Fibonacci numbers are the elements of the sequence

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots,$$

which can be defined by the simple recurrence

$$F(n) = F(n-1) + F(n-2) \text{ for } n > 1$$

Since a majority of dynamic programming applications deal with optimization problems, we also need to mention a general principle that underlines such applications. Richard Bellman called it the ***principle of optimality***.

## PRINCIPLE OF OPTIMALITY

The principle of optimality in dynamic programming (DAA) states that an optimal solution to a problem can be constructed from optimal solutions to its subproblems. In simpler terms, if you have the best solution for the overall problem, then any part of that solution (a subproblem) must also be the best solution for its corresponding smaller problem. This principle is crucial for breaking down complex problems into manageable, solvable subproblems.

**1.Optimal Substructure:**

The core idea is that the problem exhibits optimal substructure, meaning the optimal solution to the overall problem contains, within it, optimal solutions to smaller subproblems.

**2.Decomposition:**

Dynamic programming leverages this by breaking down a large problem into smaller, overlapping subproblems.

**3.Combining Solutions:**

By solving these subproblems optimally and combining those solutions, we can build up to the optimal solution for the entire problem.

**Example**:

Consider finding the shortest path between two points. The principle of optimality implies that if the shortest path from point A to point C passes through point B, then the subpath from A to B must be the shortest path from A to B, and the subpath from B to C must be the shortest path from B to C.

Essentially, the principle of optimality allows us to avoid recomputing solutions to overlapping subproblems, leading to more efficient algorithms like dynamic programming.

A problem is said to satisfy the Principle of Optimality if the subsolutions of an optimal solution of the problem are themesleves optimal solutions for their subproblems.

Examples:

1. The shortest path problem satisfies the Principle of Optimality.
2. This is because if a,x1,x2,...,xn,b is a shortest path from node a to node b in a graph, then the portion of xi to xj on that path is a shortest path from xi to xj.
3. The longest path problem, on the other hand, does not satisfy the Principle of Optimality. Take for example the undirected graph G of nodes a, b, c, d, and e, and edges (a,b) (b,c) (c,d) (d,e) and (e,a). That is, G is a ring. The longest (noncyclic) path from a to d to a,b,c,d. The sub-path from b to c on that path is simply the edge b,c. But that is not the longest path from b to c. Rather, b,a,e,d,c is the longest path. Thus, the subpath on a longest path is not necessarily a longest path.

## COIN CHANGE PROBLEM

**Objective:** Given a set of coins and amount, Write an algorithm to find out how many ways we can make the change of the amount using the coins given.
This is another problem in which i will show you the advantage of Dynamic programming over recursion.

**Recursive Solution:**

- We can solve it using recursion.
- For every coin we have an option to include it in solution or exclude it.

**Time Complexity : $2^n$**

Given a value N, if we want to make change for N cents, and we have infinite supply of each of S = { S1, S2, .. , Sm} valued coins, how many ways can we make the change? The order of coins doesn't matter.

For example, for N = 4 and S = {1,2,3}, there are four solutions: {1,1,1,1},{1,1,2},{2,2}, {1,3}. So output should be 4. For N = 10 and S = {2, 5, 3, 6}, there are five solutions: {2,2,2,2,2},
{2,2,3,3}, {2,2,6}, {2,3,5} and {5,5}. So the output should be 5.

**Optimal Substructure**

To count the total number of solutions, we can divide all set solutions into two sets.

1) Solutions that do not contain mth coin (or Sm).

2) Solutions that contain at least one Sm.

Let count(S[], m, n) be the function to count the number of solutions, then it can be written as sum of count(S[], m-1, n) and count(S[], m, n-Sm).
Therefore, the problem has optimal substructure property as the problem can be solved using solutions to sub problems.

**Overlapping Sub problems**

Following is a simple recursive implementation of the Coin Change problem. The implementation simply follows the recursive structure mentioned above.

```
int count( int S[], int m, int n )
```

```
{
  // If n is 0 then there is 1 solution
  // (do not include any coin) if (n == 0)
     return 1;
  // If n is less than 0 then no
  // solution exists if (n < 0)
     return 0;
  // If there are no coins and n
  // is greater than 0, then no
  // solution exist
  if (m <=0 && n >= 1) return 0;
  // count is sum of solutions (i)
  // including S[m-1] (ii) excluding S[m-1]
  return count( S, m - 1, n ) + count( S, m, n-S[m-1] );
```

**COMPUTING A BINOMIAL COEFFICIENT**

Computing binomial coefficients is non optimization problem but can be solved using dynamic programming. Binomial coefficients are represented by $C(n, k)$ or $(nk)$ and can be used to represent the coefficients of a binomail:

$$(a + b)^n = C(n, 0)a^n + \ldots + C(n, k)a^{n-k}b^k + \ldots\ldots + C(n, n)b^n$$

The recursive relation is defined by the prior power $C(n, k) = C(n\text{-}1, k\text{-}1) + C(n\text{-}1, k)$ for $n > k > 0$ IC $C(n, 0) = C(n, n) = 1$

Dynamic algorithm constructs a $n$x$k$ table, with the first column and diagonal filled out using the IC.

Construct the table:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | 0 | 1 | $k$<br>2 | ... | $k$-1 | $k$ |

| | | | | | |
|---|---|---|---|---|---|
| | 0 | 1 | | | |
| | 1 | 1 | 1 | | |
| *n* | 2<br>. | 1 | 2 | 1 | |
| | . | | | | |
| | .<br>*k* | 1 | | | 1 |
| | . | | | | |
| | . | | | | |
| | .<br>*n*-1 | 1 | | $C(n\text{-}1, k\text{-}1)$ | |
| | *n* | 1 | | | $C(n, k)$ |

The table is then filled out iteratively, row by row using the recursive relation.

**Algorithm** *Binomial*$(n, k)$

**for** $i \leftarrow 0$ **to** $n$ **do** // fill out the table row wise

**for** $i = 0$ **to** $\min(i, k)$ **do**

**if** $j==0$ or $j==i$ **then** $C[i, j] \leftarrow 1$ // IC

**else** $C[i, j] \leftarrow C[i\text{-}1, j\text{-}1] + C[i\text{-}1, j]$ // recursive relation

**return** $C[n, k]$

The cost of the algorithm is filing out the table. Addition is the basic operation. Because $k \leq n$, the sum needs to be split into two parts because only the half the table needs to be filled

out for $i< k$ and remaining part of the table is filled out across the entire row.

$A(n, k)$ = sum for upper triangle + sum for the lower rectangle

$$= \sum_{i=1}^{k} \sum_{j=1}^{i-1} 1 + \sum_{i=1}^{n} \sum_{j=1}^{k} 1$$

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

$$= (k-1)k/2 + k(n-k) \in \Theta(nk)$$

## FLOYD'S ALGORITHMS

Warshall's algorithm for computing the transitive closure of a directed graph and Floyd's algorithm for the all-pairs shortest-paths problem. These algorithms are based on essentially the same idea: exploit a relationship between a problem and its simpler rather than smaller version.

### Floyd's Algorithm for the All-Pairs Shortest-Paths Problem

Given a weighted connected graph (undirected or directed), the ***all-pairs shortest paths problem*** asks to find the distances—i.e., the lengths of the shortest paths— from each vertex to all other vertices. It is convenient to record the lengths of shortest paths in an $n \times n$ matrix $D$ called the ***distance matrix***: the element $dij$ in the $i$th row and the $j$th column of this matrix indicates the length of the shortest path from the $i$th vertex to the $j$th vertex. For an example, see Figure 8.14. We can generate the distance matrix with an algorithm that is very similar to Warshall's algorithm.

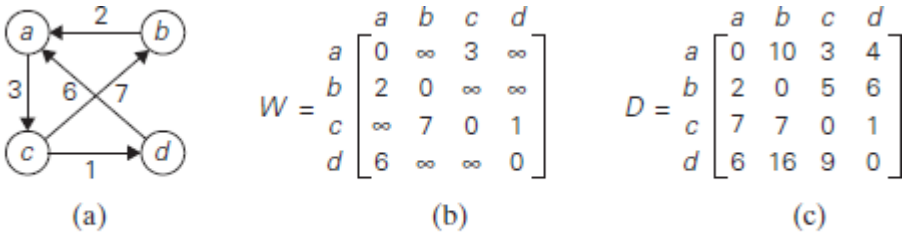



**FIGURE 8.14** (a) Digraph. (b) Its weight matrix. (c) Its distance matrix.

Floyd's algorithm computes the distance matrix of a weighted graph with $n$ vertices through a series of $n \times n$ matrices:

D(0), . . . , D(k−1), D(k), . . . , D(n).

we can compute all the elements of each matrix $D(k)$ from its immediate predecessor$D(k-1)$ in series. Let $d(k)\ ij$ be the element in the $i$th row and the $j$th column of matrix $D(k)$. This means that $d(k)\ ij$ is equal to the length of the shortest path among all paths from the $i$th vertex $vi$ to the $j$th vertex $vj$ with their intermediate vertices numbered not higher than $k$:

***vi,* a list of intermediate vertices each numbered not higher than *k*, *vj* . *vi,* vertices numbered ≤ *k* − 1, *vk,* vertices numbered ≤ *k* − 1, *vj* .**

The application of Floyd's algorithm to the graph in Figure 8.14 is illustrated in Figure 8.16.

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

//Implements Floyd's algorithm for the all-pairs shortest-paths problem

//Input: The weight matrix *W* of a graph with no negative-length cycle

//Output: The distance matrix of the shortest paths' lengths

$D \leftarrow W$ //is not necessary if *W* can be overwritten

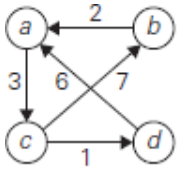
**for** $k \leftarrow 1$ **to** *n* **do**

    **for** $i \leftarrow 1$ **to** *n* **do**

        **for** $j \leftarrow 1$ **to** *n* **do**

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

**return** *D*




$D^{(0)} =$

| | a | b | c | d |
|---|---|---|---|---|
| a | 0 | ∞ | 3 | ∞ |
| b | 2 | 0 | ∞ | ∞ |
| c | ∞ | 7 | 0 | 1 |
| d | 6 | ∞ | ∞ | 0 |

Lengths of the shortest paths with no intermediate vertices ($D^{(0)}$ is simply the weight matrix).

$D^{(1)} =$

| | a | b | c | d |
|---|---|---|---|---|
| a | 0 | ∞ | 3 | ∞ |
| b | 2 | 0 | **5** | ∞ |
| c | ∞ | 7 | 0 | 1 |
| d | 6 | ∞ | **9** | 0 |

Lengths of the shortest paths with intermediate vertices numbered not higher than 1, i.e., just *a* (note two new shortest paths from *b* to *c* and from *d* to *c*).

$D^{(2)} =$

| | a | b | c | d |
|---|---|---|---|---|
| a | 0 | ∞ | 3 | ∞ |
| b | 2 | 0 | 5 | ∞ |
| c | **9** | 7 | 0 | 1 |
| d | 6 | ∞ | 9 | 0 |

Lengths of the shortest paths with intermediate vertices numbered not higher than 2, i.e., a and *b* (note a new shortest path from *c* to *a*).

$D^{(3)} =$

| | a | b | c | d |
|---|---|---|---|---|
| a | 0 | **10** | 3 | **4** |
| b | 2 | 0 | 5 | **6** |
| c | 9 | 7 | 0 | 1 |
| d | 6 | **16** | 9 | 0 |

Lengths of the shortest paths with intermediate vertices numbered not higher than 3, i.e., a, *b*, and *c* (note four new shortest paths from *a* to *b* from a to *d*, from *b* to *d*, and from *d* to *b*).

$D^{(4)} =$

| | a | b | c | d |
|---|---|---|---|---|
| a | 0 | 10 | 3 | 4 |
| b | 2 | 0 | 5 | 6 |
| c | **7** | 7 | 0 | 1 |
| d | 6 | 16 | 9 | 0 |

Lengths of the shortest paths with intermediate vertices numbered not higher than 4, i.e., a, *b*, *c*, and *d* (note a new shortest path from *c* to *a*).

## MULTI-STAGE GRAPH PROBLEM

1. A multistage graph G = (V,E) is a directed graph in which the vertices are portioned into K
   > = 2 disjoint sets Vi, 1 <= i<= k.
2. In addition, if < u,v > is an edge in E, then u < = Vi and V □ Vi+1 for some i, 1<= i < k.
3. If there will be only one vertex, then the sets Vi and Vk are such that [Vi]=[Vk] = 1.
4. Let 's' and 't' be the source and destination respectively.
5. The cost of a path from source (s) to destination (t) is the sum of the costs of the edger on the path.
6. The *MULTISTAGE GRAPH* problem is to find a minimum cost path from 's' to 't'.
7. Each set Vi defines a stage in the graph. Every path from 's' to 't' starts in stage-1, goes to stage-2 then to stage-3, then to stage-4, and so on, and terminates in stage-k.

## BACKWARD METHOD

- if there one 'K' stages in a graph using back ward approach. we will find out the cost of each & every vertex starting from 1st
  stage to the kth stage.
- We will find out the minimum cost path from destination to source (ie)[from stage k to stage 1]

## PROCEDURE:

1. It is similar to forward approach, but differs only in two or three ways.
2. Maintain a cost matrix to store the cost of every vertices and a distance matrix to store the minimum distance vertex.
3. Find out the cost of each and every vertex starting from vertex 1 up to vertex k.
4. To find out the path star from vertex 'k', then the distance array D (k) will give the minimum cost neighbor vertex which in turn gives the next nearest neighbor vertex and proceed till we reach the destination.

**STEP:**

Cost(1) = 0 => D(1)=0 Cost(2) = 9 => D(2)=1 Cost(3) = 7 => D(3)=1 Cost(4) = 3 => D(4)=1 Cost(5) = 2 => D(5)=1

Cost(6) =min(c (2,6) + cost(2),c (3,6) + cost(3)) =min(13,9) cost(6) = 9 =>D(6)=3

Cost(7) =min(c (3,7) + cost(3),c (5,7) + cost(5) ,c (2,7) + cost(2))

=min(14,13,11) cost(7) = 11 =>D(7)=2

Cost(8) =min(c (2,8) + cost(2),c (4,8) + cost(4) ,c (5,8) +cost(5))

=min(10,14,10) cost(8) = 10 =>D(8)=2

Cost(9) =min(c (6,9) + cost(6),c (7,9) + cost(7))

=min(15,15) cost(9) = 15 =>D(9)=6

Cost(10)=min(c(6,10)+cost(6),c(7,10)+cost(7)),c(8,10)+cost(8))

=min(14,14,15) cost(10)= 14 =>D(10)=6

Cost(11) =min(c (8,11) + cost(8)) cost(11) = 16 =>D(11)=8

cost(12)=min(c(9,12)+cost(9),c(10,12)+cost(10),c(11,12)+cost(11))

=min(19,16,21) cost(12) = 16 =>D(12)=10

*PATH:*

Start from vertex-12 D(12) = 10

D(10) = 6

D(6) = 3

D(3) = 1

So the minimum cost path is,

1 → 3 → 6 → 10 → 12

The cost is 16.

**ALGORITHM** : BACKWARD METHOD

**Algorithm BGraph (G,k,n,p)**

```
// The I/p is a k-stage graph G=(V,E) with 'n' vertex.
// Indexed in order of stages E is a set of edges.
// and c[i,J] is the cost of<i,j>,p[1:k] is a minimum cost path.
{
    bcost[1]
     =0.0;
     for j=2 to n do
  {
     //compute bcost[j],
    // let 'r' be the vertex such that <r,j> is an edge of 'G' &
   // bcost[r]+c[r,j] is minimum.

   bcost[j] = bcost[r] + c[r,j]; d[j] =r;
  }
 // find a minimum cost path.
```

P[1]=1;

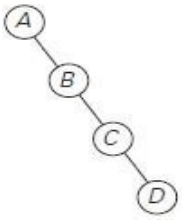
P[k]=n;

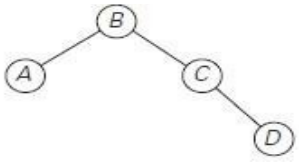
For j= k-1 to 2 do P[j]=d[p[j+1]]

;

## OPTIMAL BINARY SEARCH TREES

A binary search tree is one of the most important data structures in computer science. One of its principal applications is to implement a dictionary, a set of elements with the




operations of searching, insertion, and deletion. If probabilities of searching for elements of a set are known—e.g., from accumulated data about past searches—it is natural to pose a question about an optimal binary search tree for which the average number of comparisons in a search is the smallest possible.

As an example, consider four keys *A, B*, *C*, and *D* to be searched for with probabilities 0.1, 0.2, 0.4, and 0.3, respectively. Figure 8.7 depicts two out of 14 possible binary search trees containing these keys. The average number of comparisons in a successful search in the first of these trees is 0.1 . 1+ 0.2 . 2 + 0.4 . 3+ 0.3 . 4 = 2.9, and for the second one it is

0.1 . 2 + 0.2 . 1+

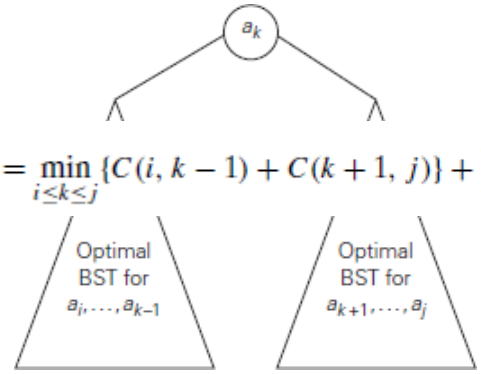
0.4 . 2 + 0.3 . 3= 2.1. Neither of these two trees is, in fact, optimal.

For our tiny example, we could find the optimal tree by generating all 14 binary search trees with these keys. As a general algorithm, this exhaustive-search approach is unrealistic: the total number of binary search trees with *n* keys is equal to the *n*th ***Catalan***

$$c(n) = \frac{1}{n+1}\binom{2n}{n} \quad \text{for } n > 0, \quad c(0) = 1,$$

***number***,

To derive a recurrence underlying a dynamic programming algorithm, we will consider all possible ways to choose a root *ak* among the keys *ai, . . . , aj* . For such a binary search tree (Figure 8.8), the root contains key *ak,* the left subtree *T k*−1 *i* contains keys *ai, . . . , ak*−1 optimally arranged, and the right subtree *Tj k*+1 contains keys *ak*+1, *. . . , aj* also optimally arranged.



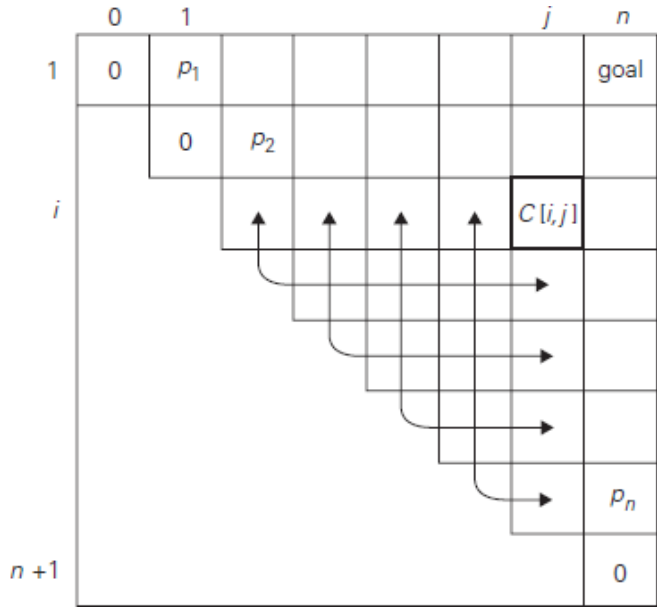

$$C(i, j) = \min_{i \le k \le j} \{C(i, k-1) + C(k+1, j)\} + \sum_{s=i}^{j} p_s \quad \text{for } 1 \le i \le j \le n. \quad \textbf{(8.8)}$$

If we count tree levels starting with 1 to make the comparison numbers equal the keys' levels, the following recurrence relation is obtained:

We assume in formula (8.8) that *C(i, i* − 1*)* = 0 for 1≤ *i* ≤ *n* + 1, which can be interpreted as the number of comparisons in the empty tree. Note that this formula implies that

$$C(i, i) = p_i \quad \text{for } 1 \le i \le n,$$

as it should be for a one-node binary search tree containing *ai* .



The two-dimensional table in Figure 8.9 shows the values needed for computing *C(i, j)* by formula they are in row *i* and the columns to the left of column *j* and in column *j* and the rows below row *i*. The arrows point to the pairs of entries whose sums are computed in order to find the smallest one to be recorded as the value of *C(i, j)*. This suggests filling the table along its diagonals, starting with all zeros on the main diagonal and given probabilities *pi,* $1 \le i \le n$, right above it and moving toward the upper right corner.

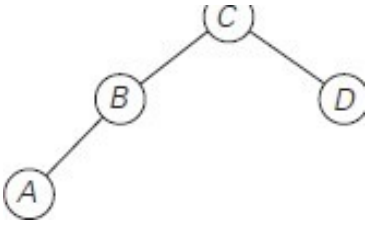
| key | *A* | *B* | *C* | *D* |
|---|---|---|---|---|
| probability | 0.1 | 0.2 | 0.4 | 0.3 |

main table

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 1 | 0 | 0.1 | 0.4 | 1.1 | 1.7 |
| 2 | | 0 | 0.2 | 0.8 | 1.4 |
| 3 | | | 0 | 0.4 | 1.0 |
| 4 | | | | 0 | 0.3 |
| 5 | | | | | 0 |

root table

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 1 | | 1 | 2 | 3 | 3 |
| 2 | | | 2 | 3 | 3 |
| 3 | | | | 3 | 3 |
| 4 | | | | | 4 |
| 5 | | | | | |



**ALGORITHM** *OptimalBST(P* [1..*n*]*)*

**for** *i* ←1 **to** *n* **do**

*C*[*i*, *i* − 1]←0 *C*[*i*,

*i*]←*P*[*i*]

*R*[*i*, *i*]←*i C*[*n* + 1, *n*]←0

**for** *d* ←1 **to** *n* − 1 **do** //diagonal count

**for** *i* ←1 **to** *n* − *d* **do**

*j* ←*i*

+ *d minv al*←

∞

**for** *k*←*i* **to** *j* **do**

**if** *C*[*i*, *k* − 1]+ *C*[*k* + 1, *j*]< *minval*

*minval*←*C*[*i*, *k* − 1]+ *C*[*k* + 1, *j*]; *kmin*←*k*

*R*[*i*, *j* ]←*kmin*

*sum*←*P*[*i*]; **for** *s* ←*i* + 1 **to** *j* **do** *sum*←*sum* + *P*[*s*]

*C*[*i*, *j* ]←*minval* + *sum*

**return** *C*[1, *n*], *R*

The algorithm's space efficiency is clearly quadratic; the time efficiency of this version of the algorithm is cubic.

## KNAPSACK PROBLEM USING DYNAMIC PROGRAMMING

**Knapsack problem:** given $n$ items of known weights $w1, \ldots, wn$ and values $v1, \ldots, vn$ and a knapsack of capacity $W$, find the most valuable subset of the items that fit into the knapsack.

To design a dynamic programming algorithm, we need to derive a recurrence relation that expresses a solution to an instance of the knapsack problem in terms of solutions to its smaller subinstances. Let us consider an instance defined by the first $i$ items, $1 \le i \le n$, with weights $w1, \ldots, wi$, values $v1, \ldots, vi$ , and knapsack capacity $j$, $1 \le j \le W$. Let $F(i, j)$ be the value of an optimal solution to this instance, i.e., the value of the most valuable subset of the first $i$ items

that fit into the knapsack of capacity $j$.We can divide all the subsets of the first $i$ items that fit the knapsack of capacity $j$ into two categories: those that do not include the $i$th item and those that do. Note the following:

1. Among the subsets that do not include the $i$th item, the value of an optimal subset is, by definition, $F(i - 1, j)$.
2. Among the subsets that do include the $i$th item (hence, $j - wi \ge 0$), an optimal subset is made up of this item and an optimal subset of the first $i - 1$ items that fits into the knapsack of capacity $j - wi$ . The value of such an optimal subset is $vi + F(i - 1, j - wi)$.

$$F(i, j) = \begin{cases} \max\{F(i-1, j),\ v_i + F(i-1, j-w_i)\} & \text{if } j - w_i \ge 0, \\ F(i-1, j) & \text{if } j - w_i < 0. \end{cases}$$

It is convenient to define the initial conditions as follows:

$$F(0, j) = 0 \text{ for } j \ge 0 \quad \text{and} \quad F(i, 0) = 0 \text{ for } i \ge 0.$$

Our goal is to find $F(n, W)$, the maximal value of a subset of the $n$ given items that fit into the knapsack of capacity $W$, and an optimal subset itself.

| | | 0 | $j - w_i$ | $j$ | $W$ |
|---|---|---|---|---|---|
| 0 | | 0 | 0 | 0 | 0 |
| $i-1$ | | 0 | $F(i-1, j-w_i)$ | $F(i-1, j)$ | |
| $w_i$, $v_i$ | $i$ | 0 | | $F(i, j)$ | |
| $n$ | | 0 | | | goal |

| item | weight | value |
|---|---|---|
| 1 | 2 | \$12 |
| 2 | 1 | \$10 |
| 3 | 3 | \$20 |
| 4 | 2 | \$15 |

capacity $W = 5$.

| | | capacity $j$ | | | | | |
|---|---|---|---|---|---|---|---|
| | $i$ | 0 | 1 | 2 | 3 | 4 | 5 |
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $w_1 = 2, v_1 = 12$ | 1 | 0 | 0 | 12 | 12 | 12 | 12 |
| $w_2 = 1, v_2 = 10$ | 2 | 0 | 10 | 12 | 22 | 22 | 22 |
| $w_3 = 3, v_3 = 20$ | 3 | 0 | 10 | 12 | 22 | 30 | 32 |
| $w_4 = 2, v_4 = 15$ | 4 | 0 | 10 | 15 | 25 | 30 | **37** |

Thus, the maximal value is $F(4, 5) = \$37$. We can find the composition of an optimal subset by backtracing the computations of this entry in the table. Since $F(4, 5) > F(3, 5)$, item 4 has to be included in an optimal solution along with an optimal subset for filling $5 - 2 = 3$ remaining units of the knapsack capacity. The value of the latter is $F(3, 3)$. Since $F(3, 3) = F(2, 3)$, item 3 need not be in an optimal subset. Since $F(2, 3) > F(1, 3)$, item 2 is a part of an optimal selection, which leaves element $F(1, 3 - 1)$ to specify its remaining composition. Similarly, since $F(1, 2) > F(0, 2)$, item 1 is the final part of the optimal solution {item 1, item 2, item 4}. The

time efficiency and space efficiency of this algorithm are both in $\Theta(nW)$.

Given a set of items, each with a weight and a value, determine a subset of items to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

The knapsack problem is in combinatorial optimization problem.

**Applications**

In many cases of resource allocation along with some constraint, the problem can be derived in a similar way of Knapsack problem. Following is a set of example.

- Finding the least wasteful way to cut raw materials
- portfolio optimization
- Cutting stock problems

Based on the nature of the items, Knapsack problems are categorized as

- Fractional Knapsack
- Knapsack

**Fractional Knapsack**

In this case, items can be broken into smaller pieces, hence the thief can select fractions of

items.

According to the problem statement,

- There are **n** items in the store
- Weight of $\mathbf{i^{th}}$ item wi>0wi>0
- Profit for $\mathbf{i^{th}}$ item pi>0pi>0 and
- Capacity of the Knapsack is **W**

In this version of Knapsack problem, items can be broken into smaller pieces. So, the thief may take only a fraction $\boldsymbol{xi}$ of **ith** item.

$$0 \leqslant xi \leqslant 10 \leqslant xi \leqslant 1$$

The $\mathbf{i^{th}}$ item contributes the weight xi.wixi.wi to the total weight in the knapsack and profit xi.pixi.pi to the total profit.

Hence, the objective of this algorithm is to

maximize$\sum n=1n(xi.pi)$maximize$\sum n=1n(xi.pi)$

subject to constraint,

$\sum n=1n(xi.wi) \leqslant W \sum n=1n(xi.wi) \leqslant W$

It is clear that an optimal solution must fill the knapsack exactly, otherwise we could add a fraction of one of the remaining items and increase the overall profit.

Thus, an optimal solution can be obtained by

$$\sum n=1n(xi.wi)=W \sum n=1n(xi.wi)=W$$

In this context, first we need to sort those items according to the value of piwipiwi, so that pi+1wi+1pi+1wi+1 ≤ piwipiwi . Here, $\boldsymbol{x}$ is an array to store the fraction of items.

**Algorithm: Greedy-Fractional-Knapsack (w[1..n], p[1..n], W)**

```
for i = 1 to n do x[i] = 0
weight = 0 for i = 1 to n
  if weight + w[i] ≤ W then x[i] = 1
    weight = weight + w[i] else
    x[i] = (W - weight) / w[i] weight = W
```

break return x

**Analysis**

If the provided items are already sorted into a decreasing order of piwipiwi, then the whileloop takes a time in ***O(n)***; Therefore, the total time including the sort is in ***O(n logn)***.

**Example**

Let us consider that the capacity of the knapsack $W = 60$ and the list of provided items are shown in the following table −

| **Item** | **A** | **B** | **C** | **D** |
|---|---|---|---|---|
| Profit | 280 | 100 | 120 | 120 |
| Weight | 40 | 10 | 20 | 24 |
| Ratio (piwi)(piwi) | 7 | 10 | 6 | 5 |

As the provided items are not sorted based on piwipiwi. After sorting, the items are as shown in the following table.

| **Item** | **B** | **A** | **C** | **D** |
|---|---|---|---|---|
| Profit | 100 | 280 | 120 | 120 |
| Weight | 10 | 40 | 20 | 24 |
| Ratio (piwi)(piwi) | 10 | 7 | 6 | 5 |

**Solution**

After sorting all the items according to piwipiwi. First all of ***B*** is chosen as weight of ***B*** is less than the capacity of the knapsack. Next, item ***A*** is chosen, as the available capacity of the knapsack is greater than the weight of ***A***. Now, ***C*** is chosen as the next item. However, the whole item cannot be chosen as the remaining capacity of the knapsack is less than the weight of ***C***.

Hence, fraction of ***C*** (i.e. (60 − 50)/20) is chosen.

Now, the capacity of the Knapsack is equal to the selected items. Hence, no more item can be selected.

The total weight of the selected items is **10 + 40 + 20 * (10/20) = 60**

And the total profit is **100 + 280 + 120 * (10/20) = 380 + 60 = 440**

This is the optimal solution. We cannot gain more profit selecting any different combination of items.

## MEMORY FUNCTIONS IN DYNAMIC PROGRAMMING

Dynamic programming deals with problems whose solutions satisfy a recurrence relation with overlapping subproblems. The direct top-down approach to finding a solution to such a recurrence leads to an algorithm that solves common subproblems more than once and hence is very inefficient. The classic dynamic programming approach, on the other hand, works bottom up: it fills a table with solutions to *all* smaller subproblems, but each of them is solved only once. An unsatisfying aspect of this approach is that solutions to some of these smaller subproblems are often not necessary for getting a solution to the problem given. Since this drawback is not present in the top-down approach, it is natural to try to combine the strengths of the top-down and bottom-up approaches. The goal is to get a method that solves only subproblems that are

necessary and does so only once. Such a method exists; it is based on using ***memory functions***. This method solves a given problem in the top-down manner but, in addition, maintains a table. **ALGORITHM** *MFKnapsack(i, j )*

**if** $F[i, j] < 0$

**if** $j < Weights[i]$

$value \leftarrow MFKnapsack(i - 1, j)$

**else**

$value \leftarrow \max(MFKnapsack(i - 1, j), Values[i] + MFKnapsack(i - 1, j - Weights[i]))$

$F[i, j] \leftarrow value$

**return** $F[i, j]$

| | | capacity $j$ | | | | | |
|---|---|---|---|---|---|---|---|
| | $i$ | 0 | 1 | 2 | 3 | 4 | 5 |
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $w_1 = 2,\ v_1 = 12$ | 1 | 0 | 0 | 12 | 12 | 12 | 12 |
| $w_2 = 1,\ v_2 = 10$ | 2 | 0 | — | 12 | 22 | — | 22 |
| $w_3 = 3,\ v_3 = 20$ | 3 | 0 | — | — | 22 | — | 32 |
| $w_4 = 2,\ v_4 = 15$ | 4 | 0 | — | — | — | — | **37** |

The table in Figure 8.6 gives the results. Only 11 out of 20 nontrivial values (i.e., not those in row 0 or in column 0) have been computed. Just one nontrivial entry, *V (*1, 2*),* is retrieved rather than being recomputed.

## GREEDY TECHNIQUE OVERVIEW

The greedy technique in Design and Analysis of Algorithms (DAA) is a problem-solving approach that makes the locally optimal choice at each stage with the hope of finding a global optimum. It focuses on making the best immediate choice without considering the long-term consequences. This method is often used for optimization problems, where the goal is to find the best solution (either maximum or minimum) among many possible solutions.

Key Concepts:

- **Local Optimality:** At each step, the algorithm selects the option that appears best at that moment, without looking ahead to see if it leads to the best overall solution.
- **No Backtracking:** Once a choice is made, it is not reconsidered or changed in later steps.
- **Optimization Problems:** Greedy algorithms are particularly well-suited for problems where the objective is to maximize or minimize some value, subject to certain constraints.
- **Feasible Solutions:** A feasible solution is one that satisfies all the problem's constraints.
- **Optimal Solution:** The optimal solution is the best feasible solution, either maximizing or minimizing the objective function.

How it works:

1. **Start with a partial solution:** Begin with a basic, feasible solution.
2. **Iteratively build the solution:** At each step, choose the best option based on the current state of the solution and the problem constraints.
3. **Repeat until a complete solution is reached:** Continue making greedy choices until the problem is solved.

Example:

Consider the coin change problem. Given a set of coin denominations and an amount, the goal is to find the minimum number of coins to make up that amount. A greedy approach would be to repeatedly select the largest denomination coin that is less than or equal to the remaining amount.

Advantages:

- **Simplicity:** Greedy algorithms are relatively easy to understand and implement.
- **Efficiency:** They often have good time complexity, making them suitable for large datasets.

Disadvantages:

- **Not always optimal:**

The greedy choice at each step might not lead to the globally optimal solution.

- **Requires specific problem structure:**

Greedy algorithms work best when the problem exhibits optimal substructure, meaning the optimal solution can be built from optimal solutions to subproblems.

Examples of Greedy Algorithms:

- Dijkstra's algorithm for finding the shortest path in a graph.
- Kruskal's and Prim's algorithms for finding the minimum spanning tree.
- Huffman coding for data compression.

***Change-making problem:*** give change for a specific amount $n$ with the least number of coins of the denominations $d1 > d2 > \ldots > dm$ used in that locale.

For example, the widely used coin denominations in the United States are $d1 = 25$ (quarter), $d2 = 10$ (dime), $d3 = 5$ (nickel), and $d4 = 1$ (penny). How would you give change with coins of these denominations of, say, 48 cents?

If you came up with the answer 1 quarter, 2 dimes, and 3 pennies. ―Greedy‖ thinking leads to giving one quarter because it reduces the remaining amount the most, namely, to 23 cents. In the second step, you had the same coins at your disposal, but you could not give a quarter, because it would have violated the problem's constraints. So your best selection in this step was one dime, reducing the remaining amount to 13 cents. Giving one more dime left you with 3 cents to be given with three pennies. The greedy approach suggests constructing a solution through a sequence of steps, each expanding a partially constructed solution obtained so far, until a complete solution to the problem is reached. On each step the choice made must be:

- *feasible*, i.e., it has to satisfy the problem's constraints
- *locally optimal*, i.e., it has to be the best local choice among all feasible choices available on that step
- *irrevocable*, i.e., once made, it cannot be changed on subsequent steps of the algorithm.

**CONTAINER LOADING PROBLEM:**

The Container Loading Problem (CLP) is a type of combinatorial optimization problem, specifically a three-dimensional bin packing problem, where the goal is to efficiently pack a set of rectangular items (boxes) into a larger rectangular container, maximizing the volume utilization while preventing overlap and staying within the container's dimensions. It's a computationally challenging problem, often requiring heuristic or approximation algorithms for practical solutions.

**Key Aspects:**

- **Objective:** Maximize the volume of items packed into the container.
- **Constraints:** Items cannot overlap and must fit within the container's boundaries (length, width, height).
- **Variations:** The problem can be simple (single container, same size boxes) or more complex (heterogeneous items, multiple containers, specific loading rules).

**Relevance to Design and Analysis of Algorithms (DAA):**

- **Computational Complexity:**

CLP is known to be NP-hard, meaning finding the absolute optimal solution is computationally very expensive for larger problem instances.

- **Algorithm Design:**

DAA focuses on developing efficient algorithms, often using heuristics, approximation algorithms, or metaheuristics, to find near-optimal solutions within reasonable time.

- **Greedy Algorithms:**

A common approach involves greedy algorithms, which make locally optimal choices at each step, potentially leading to good, but not necessarily the best, solutions.

- **Other Techniques:**

Other techniques like branch and bound, simulated annealing, and genetic algorithms can also be applied to CLP.

Common Approaches:

- **Greedy Algorithms:**

Construct solutions layer by layer, prioritizing the placement of boxes based on factors like size, weight, or evaluation functions.

- **Heuristic Algorithms:**

Employ rules and shortcuts to guide the search for good solutions, often based on observations of practical packing scenarios.

- **Metaheuristics:**

Use more sophisticated techniques like genetic algorithms or simulated annealing to explore the solution space and find better solutions.

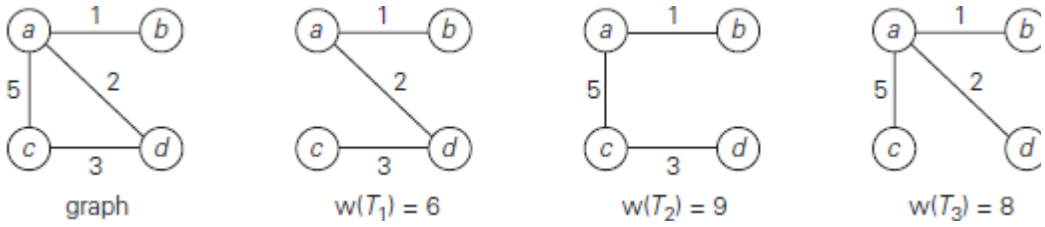
Example Scenario:

Imagine loading a cargo ship with containers. The goal is to fit as many containers as possible onto the ship, maximizing the cargo transported. This is a CLP where containers are the "boxes" and the ship's cargo space is the "container".

## PRIM'S ALGORITHM

The following problem arises naturally in many practical situations: given *n* points, connect them in the cheapest possible way so that there will be a path between every pair of points.

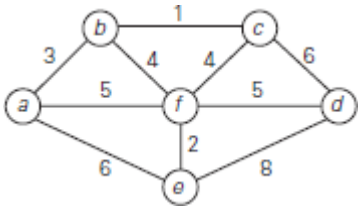
A ***spanning tree*** of an undirected connected graph is its connected acyclic subgraph (i.e., a tree) that contains all the vertices of the graph. If such a graph has weights assigned to its edges, a ***minimum spanning tree*** is its spanning tree of the smallest weight, where the ***weight*** of a tree is defined as the sum of the weights on all its edges. The ***minimum spanning tree problem*** is the problem of finding a minimum spanning tree for a given weighted connected graph.

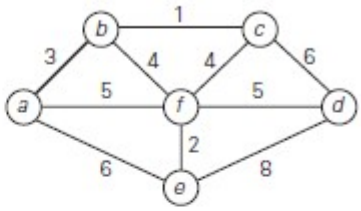
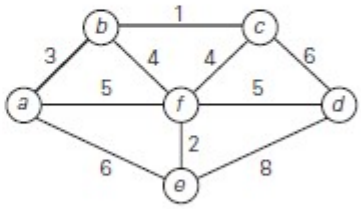
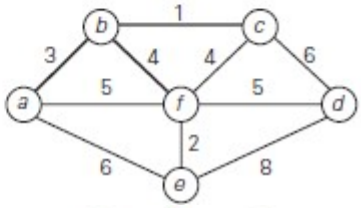
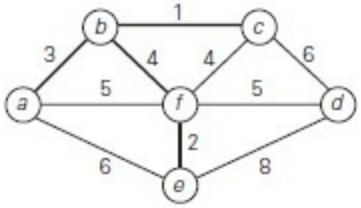
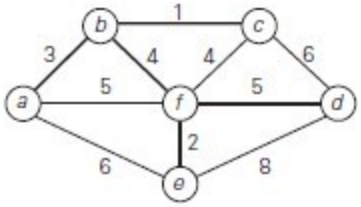



Prim's algorithm constructs a minimum spanning tree through a sequence of expanding subtrees. The initial subtree in such a sequence consists of a single vertex selected arbitrarily from the set $V$ of the graph's vertices. On each iteration, the algorithm expands the current tree in the greedy manner by simply attaching to it the nearest vertex not in that tree. After we have identified a vertex $u^*$ to be added to the tree, we need to perform two operations: Move $u^*$ from the set $V - VT$ to the set of tree vertices $VT$. For each remaining vertex $u$ in $V - VT$ that is connected to $u^*$ by a shorter edge than the $u$'s current distance label, update its labels by $u^*$ and the weight of the edge between $u^*$ and $u$, respectively.

**ALGORITHM** *Prim(G)*

//Prim's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph $G = \langle V, E \rangle$
//Output: $E_T$, the set of edges composing a minimum spanning tree of $G$
$V_T \leftarrow \{v_0\}$ //the set of tree vertices can be initialized with any vertex
$E_T \leftarrow \varnothing$
**for** $i \leftarrow 1$ **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 $V - V_T$
    $V_T \leftarrow V_T \cup \{u^*\}$
    $E_T \leftarrow E_T \cup \{e^*\}$
**return** $E_T$



| Tree vertices | Remaining vertices | Illustration |
|---|---|---|
| a(−, −) | **b(a, 3)** c(−, ∞) d(−, ∞) e(a, 6) f(a, 5) |  |
| b(a, 3) | **c(b, 1)** d(−, ∞) e(a, 6) f(b, 4) |  |
| c(b, 1) | d(c, 6) e(a, 6) **f(b, 4)** |  |
| f(b, 4) | d(f, 5) **e(f, 2)** |  |
| e(f, 2) | **d(f, 5)** |  |
| d(f, 5) | | |

## KRUSKAL'S ALGORITHM

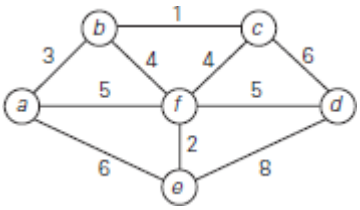
Kruskal's algorithm looks at a minimum spanning tree of a weighted connected graph

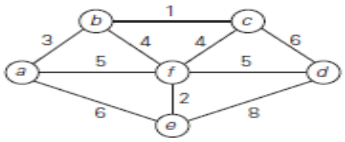
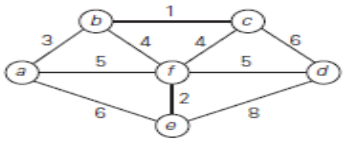
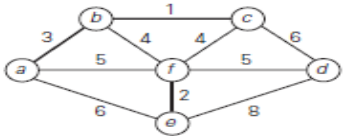
$G$= $\{V, E\}$ as an acyclic subgraph with $|V| - 1$ edges for which the sum of the edge weights is the smallest. Consequently, the algorithm constructs a minimum spanning tree as an expanding sequence of subgraphs that are always acyclic but are not necessarily connected on the intermediate stages of the algorithm.

The algorithm begins by sorting the graph's edges in nondecreasing order of their weights. Then, starting with the empty subgraph, it scans this sorted list, adding the next edge on the list to the current subgraph if such an inclusion doesnot create a cycle and simply skipping the edge otherwise.

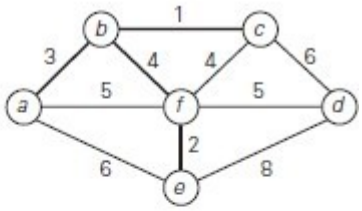
**ALGORITHM** *Kruskal(G)*

```
//Kruskal's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph G = ⟨V, E⟩
//Output: E_T, the set of edges composing a minimum spanning tree of G
sort E in nondecreasing order of the edge weights w(e_{i_1}) ≤ ··· ≤ w(e_{i_|E|})
E_T ← ∅;  ecounter ← 0      //initialize the set of tree edges and its size
k ← 0                       //initialize the number of processed edges
while ecounter < |V| − 1 do
    k ← k + 1
    if E_T ∪ {e_{i_k}} is acyclic
        E_T ← E_T ∪ {e_{i_k}};  ecounter ← ecounter + 1
return E_T
```

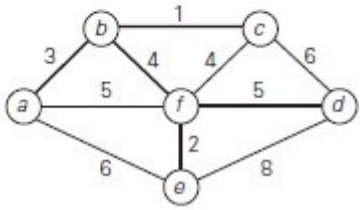


| Tree edges | Sorted list of edges | Illustration |
|---|---|---|
| | **bc** ef ab bf cf af df ae cd de<br>1 2 3 4 4 5 5 6 6 8 |  |
| bc<br>1 | bc **ef** ab bf cf af df ae cd de<br>1 2 3 4 4 5 5 6 6 8 |  |
| ef<br>2 | bc ef **ab** bf cf af df ae cd de<br>1 2 3 4 4 5 5 6 6 8 |  |

| ab | bc | ef | ab | **bf** | cf | af | df | ae | cd | de |
|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 1 | 2 | 3 | 4 | 4 | 5 | 5 | 6 | 6 | 8 |




| bf | bc | ef | ab | bf | cf | af | **df** | ae | cd | de |
|---|---|---|---|---|---|---|---|---|---|---|
| 4 | 1 | 2 | 3 | 4 | 4 | 5 | 5 | 6 | 6 | 8 |




df
5