Chapter 5

Dynamic Programming

LEARNING OBJECTIVES

- Dynamic programming
- Multi-stage graph
- All pairs shortest path problem
- Hashing methods
- Mid-square method
- Folding method
- Resolving collisions
- Matrix chain multiplication
- Longest common subsequence
- Optimal substructure of LCS
- NP-hard and NP-complete
- P-problem
- NP-problem
- P, NP, and Co-NP
- Cooks theorem
- Non-deterministic search

Dynamic Programming

Dynamic programming is a method for solving complex problems by breaking them down into simpler sub problems. It is applicable to problems exhibiting the properties of overlapping sub problems which are only slightly smaller, when applicable; the method takes far less time than naive method.

- The key idea behind dynamic programming is to solve a given problem, we need to solve different parts of the problem (sub problems) then combine the solutions of the sub problems to reach an overall solution. Often, many of these sub problems are the same.
- The dynamic programming approach seeks to solve each sub problem only once, thus reducing the number of computations. This is especially useful when the number of repeating sub problems is exponentially large.
- There are two key attributes that a problem must have in order for dynamic programming to be applicable 'optimal sub structure' and 'overlapping sub-problems'. However, when the overlapping problems are much smaller than the original problem, the strategy is called 'divide-and-conquer' rather than 'dynamic programming'. This is why merge sort-quick sort are not classified as dynamic programming problems.

Dynamic programming is applied for:

- Multi stage graph
- All pairs shortest path

Principle of Optimality

It states that whatever the initial state is, remaining decisions must be optimal with regard to the state following from the first decision.

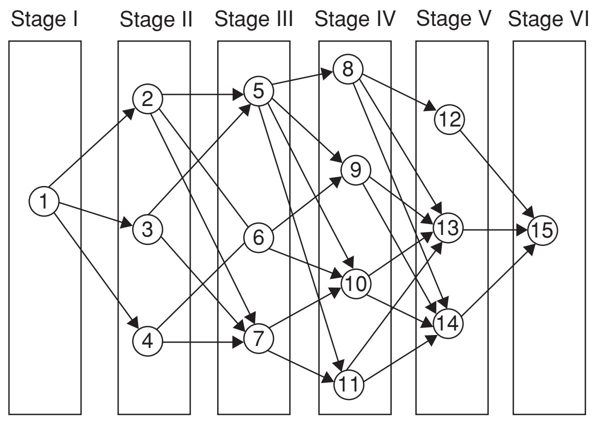
To solve a problem using dynamic programming strategy, it must observe the principle of optimality.

Multi-stage Graph

A multi-stage graph is a graph

- $G = (V, E)$ with V partitioned into $K >= 2$ disjoint subsets such that if (a, b) is in E, then a is in V_i, and b is in V_{i+1} for some sub sets in the partition;
- $|V_1| = |V_K| = 1$ the vertex S in V_1 is called the source; the vertex t is called the sink.
- G is usually assumed to be a weighted graph.
- The cost of a path from node V to node W is sum of the costs of edges in the path.
- The 'multi-stage graph problem' is to find the minimum cost path from S to t.

Example:



Costs of edges

1 – 2	$\rightarrow$	10
1 – 3	$\rightarrow$	20

1 – 4	→	30
2 – 5	→	10
2 – 6	→	20
2 – 7	→	30
3 – 5	→	40
3 – 7	→	50
4 – 6	→	40
4 – 7	→	30
5 – 8	→	10
5 – 9	→	20
5 – 10	→	30
5 – 11	→	40
6 – 9	→	20
6 – 10	→	30
7 – 10	→	30
7 – 11	→	20
8 – 12	→	10
8 – 13	→	20
8 – 14	→	30
9 – 13	→	20
9 – 14	→	10
10 – 13	→	10
10 – 14	→	20
11 – 13	→	10
11 – 14	→	30
12 – 15	→	20
13 – 15	→	10
14 – 15	→	30

Solution Using Backward Cost

Format: COST (Stage, node) = minimum cost of travelling to the node in stage from the source node (node 1)

Step I:

Cost (I, 1) = 0

Step II:

Cost (II, 2) = cost (I, 1) + cost (1, 2) = 0 + 10 = 10
Cost (II, 3) = cost (I, 1) + cost (1, 3) = 0 + 20 = 20
Cost (II, 4) = cost (I, 1) + cost (1, 4) = 0 + 30 = 30

Step III:

Cost (III, 5) = min {cost (II, 2) + cost (2, 5),
cost (II, 3) + cost (3, 5),
cost (II, 4) + cost (4, 5)
= min {10 + 10, 20 + 40, 30 + ∞}
= 20 → Via path 1 – 2 – 5

Cost (III, 6) = min {cost (II, 2) + cost (2, 6),
cost (II, 3) + cost (3, 6),
cost (II, 4) + cost (4, 6)}
= min {10 + 20, 20 + ∞, 30 + 40}
= 30 → via the path 1 – 2 – 6

Cost (III, 7) = min {cost(II, 2) + cost (2, 7),
Cost (II, 3) + cost (3, 7),
Cost (II, 4) + cost (4, 7)}
= min {10 + 30, 20 + 50, 30 + 30}
= 40 → Via the path 1 – 2 – 7

Step IV:

Cost (IV, 8) = min {cost (III, 5) + cost (5, 8),
Cost (III, 6) + cost (6, 8),
Cost (III, 7) + cost (7, 8)}
= min {20 + 10, 30 + ∞, 40 + ∞}
= 30 → Via path 1 – 2 – 5 – 8

Cost (IV, 9) = min {cost (III, 5) + cost (5, 9),
Cost (III, 6) + cost (6, 9),
Cost (III, 7) + cost (7, 9)}
= min {20 + 20, 30 + 20, 40 + ∞}
= 40 → Via the path 1 – 2 – 5 – 9

Cost (IV, 10) = min {cost (III, 5) + cost (5, 10),
Cost (III, 6) + cost (6, 10),
Cost (III, 7) + cost (7, 10}
= min {20 + 30, 30 + 30, 40 + 30}
= 50 → Via the path 1 – 2 – 5 – 10

Cost (IV, 11) = min {cost (III, 5) + cost (5, 11)
Cost (III, 6) + cost (6, 11),
Cost (III, 7) + cost (7, 11)}
= min {20 + 40, 30 + ∞, 40 + 20}
= 60 → Via the path 1 – 2 – 5 – 11
or Via the path 1 – 2 – 7 – 11

Step V:

Cost (V, 12) = min {cost (IV, 8) + cost (8, 12)
Cost (IV, 9) + cost (9, 12),
Cost (IV, 10) + cost (10, 12),
Cost (IV, 11) + cost (11, 12)}
= min {30 + 10, 40 + ∞, 50 + ∞, 60 + ∞}
= 40 → Via the path 1 – 2 – 5 – 8 – 12

Cost (V, 13) = min {cost (IV, 8) + cost (8, 13)
Cost (IV, 9) + cost (9, 13),
Cost (IV, 10) + cost (10, 13),
Cost (IV, 11) + cost (11, 13)}
= min {30 + 20, 40 + 20, 50 + 10, 60 + 10}
= 50 → Via the path 1 – 2 – 5 – 8 – 13

Cost (V, 14) = min {cost (IV, 8) + cost (8, 14)
Cost (IV, 9) + cost (9, 14),
Cost (IV, 10) + cost (10, 14),
Cost (IV, 11) + cost (11, 14)}
= min {30 + 30, 40 + 10, 50 + 20, 60 + 30}
50 → Via the path 1 – 2 – 5 – 9 – 14

Step VI:

Cost (VI, 15) = min {cost (V, 12) + cost (12, 15),
Cost (V, 13) + cost (13, 15),
Cost (V, 14) + cost (14, 15)}
= min {40 + 20, 50 + 10, 50 + 30}
= 60 → Via the path 1 – 2 – 5 – 8 – 13 – 15
(or) 1 – 2 – 5 – 8 – 12 – 15

All Pairs Shortest Path Problem (Floyd–Warshall Algorithm)

A weighted graph is a collection of points (vertices) connected by lines (edges), where each edge has a weight (some real number) associated with it.

Example: A graph in the real world is a road map. Each location is a vertex and each road connecting locations is an edge. We can think of the distance travelled on a road from one location to another as the weight of that edge.

- The Floyd–Warshall algorithm determines the shortest path between all pairs of vertices in a graph.
- The vertices in a graph be numbered from 1 to n. Consider the subset $\{1, 2, \ldots K\}$ of these n vertices.
- Finding the shortest path from vertex i to vertex j that uses vertex in the set $\{1, 2, \ldots K\}$ only. There are two situations.

  1. K is an intermediate vertex on the shortest path.
  2. K is not an intermediate vertex on the shortest path.

In the first situation, we can break down our shortest path into two paths: i to K and then K to j. Note that all the vertices from i to K are from the set $\{1, 2, \ldots K-1\}$ and that all the intermediate vertices from K to j are from the set $\{1, 2, \ldots K-1\}$. Also in the second situation, we simply have that all intermediate vertices are from the set $\{1, 2, \ldots K-1\}$. Now define the function D for a weighted graph with the vertices $\{1, 2, \ldots n\}$ as follows.

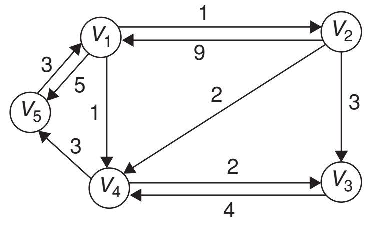
$D(i, j, K)$ = the shortest distance from vertex i to vertex j using the intermediate vertices. In the set $\{1, 2, \ldots K\}$

Using the above idea, we can recursively define the function D.

$$\boxed{\begin{array}{l} D(i, j, K) = W(i, j) \text{ if } K = 0 \\ \min(D(i, j, K-1), D(i, K, K-1) + D(K, j, K-1)) \text{ if } K > 0 \end{array}}$$

- The first line says that if we do not allow intermediate vertices, then the shortest path between two vertices is the weight of the edge that connects them. If no such weightexists, we usually define this shortest path to be of length infinity.
- The second line pertains to allowing intermediate vertices. It says that the minimum path from i to j through vertices $\{1, 2, \ldots K\}$ is either the minimum path from i to j through vertices $\{1, 2, \ldots K-1\}$ OR the sum of the minimum path from vertex i to K through $\{1, 2, \ldots K-1\}$ plus the minimum path from vertex K to j through $\{1, 2, \ldots K-1\}$. Since this is the case, we compute both and choose the smaller of these.

Example:



The weight matrix will be

	1	2	3	4	5
1	0	1	∞	1	5
2	9	0	3	2	∞
3	∞	∞	0	4	∞
4	∞	∞	2	0	3
5	3	∞	∞	∞	0

Let $D^{(K)}[i, j]$ = weight of a shortest path from v_i to v_j using only vertices from $\{v_1, v_2, \ldots v_k\}$ as intermediate vertices in the path.

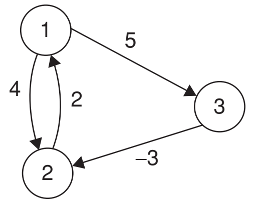
- $D^{(0)} = W$
- $D^{(n)} = D$ which is the goal matrix.

How to compute $D^{(K)}$ from $D^{(K-1)}$?

Case I: 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 II: 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 use V_K. Then $D^{(K)}[i, j] = D^{(K-1)}[i, K] + D^{(K-1)}[K, j]$
Since $D^{(K)}[i, j] = D^{(K-1)}[i, j]$
or $D^{(K)}[i, j] = D^{(K-1)}[i, K] + D^{(K-1)}[K, j]$
We conclude: $D^{(K)}[i, j] = \min\{D^{(K-1)}[i, j], D^{(K-1)}[i, K] + D^{(K-1)}[K, j]\}$

Example: 1



$$W = D^{\circ} = \begin{array}{c|ccc} & 1 & 2 & 3 \\ \hline 1 & 0 & 4 & 5 \\ 2 & 2 & 0 & \infty \\ 3 & \infty & -3 & 0 \end{array}$$

$$P = \begin{array}{c|ccc} & 1 & 2 & 3 \\ \hline 1 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 \end{array}$$

$K = 1$, vertex 1 can be intermediate node

$$\begin{aligned} D^1[2,3] &= \min(D^\circ[2,3], D^\circ[2,1] + D^\circ[1,3]) \\ &= \min(\infty, 7) \\ &= 7 \end{aligned}$$

$$\begin{aligned} D^1[3,2] &= \min(D^\circ[3,2], D^\circ[3,1] + D^\circ[1,2]) \\ &= \min(-3, \infty) \\ &= -3 \end{aligned}$$

$$D^1 = \begin{array}{c|c|c|c|} & 1 & 2 & 3 \\ \hline 1 & 0 & 4 & 5 \\ \hline 2 & 2 & 0 & 7 \\ \hline 3 & \infty & -3 & 0 \\ \hline \end{array}$$

$$P = \begin{array}{c|c|c|c|} & 1 & 2 & 3 \\ \hline 1 & 0 & 0 & 0 \\ \hline 2 & 0 & 0 & 1 \\ \hline 3 & 0 & 0 & 0 \\ \hline \end{array}$$

$K = 2$, vertices 1, 2 can be intermediate nodes,

$$\begin{aligned} D^2[1,3] &= \min(D'[1,3], D'[1,2] + D'[2,3]) \\ &= \min(5, 4 + 7) = 5 \end{aligned}$$

$$\begin{aligned} D^2[3,1] &= \min(D'[3,1], D'[3,2] + D'[2,1]) \\ &= \min(\infty, -3 + 2) \\ &= -1 \end{aligned}$$

$$D^2 = \begin{array}{c|c|c|c|} & 1 & 2 & 3 \\ \hline 1 & 0 & 4 & 5 \\ \hline 2 & 2 & 0 & 7 \\ \hline 3 & -1 & -3 & 0 \\ \hline \end{array}$$

$$P = \begin{array}{c|c|c|c|} & 1 & 2 & 3 \\ \hline 1 & 0 & 0 & 0 \\ \hline 2 & 0 & 0 & 1 \\ \hline 3 & 2 & 0 & 0 \\ \hline \end{array}$$

$K = 3$ vertices 1, 2, 3 can be intermediate

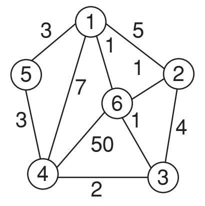
$$\begin{aligned} D^3[1,2] &= \min(D^2[1,2], D^2[1,3] + D^2[3,2]) \\ &= \min(4, 5 + (-3)) \\ &= 2 \end{aligned}$$

$$\begin{aligned} D^3[2,1] &= \min(D^2[2,1], D^2[2,3] + D^2[3,1]) \\ &= \min(2, 7 + (-1)) \\ &= 2 \end{aligned}$$

$$D^3 = \begin{array}{c|c|c|c|} & 1 & 2 & 3 \\ \hline 1 & 0 & 2 & 5 \\ \hline 2 & 2 & 0 & 7 \\ \hline 3 & -1 & -3 & 0 \\ \hline \end{array}$$

$$P = \begin{array}{c|c|c|c|} & 1 & 2 & 3 \\ \hline 1 & 0 & 3 & 0 \\ \hline 2 & 0 & 0 & 1 \\ \hline 3 & 2 & 0 & 0 \\ \hline \end{array}$$

Example 2:



The final distance matrix and P

$$D^6 = \begin{array}{c|c|c|c|c|c|c|} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & 0 & 2(6) & 2(6) & 4(6) & 3 & 1 \\ \hline 2 & 2(6) & 0 & 2(6) & 4(6) & 5(6) & 1 \\ \hline 3 & 2(6) & 2(6) & 0 & 2 & 5(4) & 1 \\ \hline 4 & 4(6) & 4(6) & 2 & 0 & 3 & 3(3) \\ \hline 5 & 3 & 5(4) & 5(4) & 3 & 0 & 4(1) \\ \hline 6 & 1 & 1 & 1 & 3(3) & 4(1) & 0 \\ \hline \end{array}$$

The values in parenthesis are the non-zero P values.

Table 1 *Divide and conquer vs dynamic programming.*

1. This design strategy divides the problem into sub problems, conquer the each sub problem recursively, finally combine all the sub problem solutions, for the original problem.	1. This design strategy chooses an optimal solution for the problem, by recursively defining the value of optimal solution, these values are computed in bottom up fashion or top down fashion.
2. each sub problem is solved recursively, and consumes more time at each sub problem	2. Each sub problem is solved only once and is stored in table
3. Sub problems are independent of each other e.g., Binary search	3. The sub problems are dependent e.g., Traveling sales person problem

Dynamic Programming vs Greedy Method

The main difference between greedy method (*GM*) and dynamic programming (*DP*) methodology is, *DP* considers all possible solutions for the given problem and picks the optimal one. Where as greedy, considers only one set of solutions to the problem.

The other difference between *GM* and *DP* is that, *GM* considers the choice, which is best at that step, which is done at each level of the sub problem. That is, it won't reconsider its choice. The choices reflect only present, won't consider the future choices, where as *DP* tries out all the best alternatives and finds the optimal solution. It implements principle of optimality. At each stage of the problem, it decides based on the previous decision made in the previous stage.

Hashing Methods

Uniform Hash Function

If the keys, K, are integers randomly distributed in $[0, r]$ then hash function $H(K)$ is given as

$$H(K) = \left\lfloor \frac{mk}{r} \right\rfloor$$

$H(K)$ is a uniform hash function

Uniform hashing function should ensure

$$\sum_{K|h(K)=0} P(K) = \sum_{K|h(K)=1} P(K) = \cdots \sum_{K|h(K)=0} P(K) = \frac{1}{m}$$

$P(K)$ = probability that a key K, occurs that is the number of keys that map to each slot is equal.

Division method

Hashing an integer x is to divide x by M and then to use the remainder modulo M. This is called the division method of hashing. In this case the hash function is

$$\boxed{h(x) = x \bmod M}$$

Generally this approach is quite good for just about any value of M. However, in certain situations some extra care is needed in the selection of a suitable value for M. For example, it is often convenient to make M an even number. But this means that $h(x)$ is even if x is even, and $h(x)$ is odd of x is odd. If all possible keys are equiprobable, then this is not a problem. However, if say even keys are more likely than odd keys, the function $h(x) = x \bmod M$ will not spread the hashed values of those keys evenly.

- Let M be a power of two, i.e., $M = 2^k$ for some integer $k > 1$. In this case, the hash function $h(x) = x \bmod 2^k$ simply extracts the bottom k-bits of the binary representation of x. While this hash function is quite easy to compute, it is not a desirable function because it does not depend on all the bits in the binary representation of x.
- For these reasons M is often chosen to be a prime number. Suppose there is bias in the way the keys are created that makes it more likely for a key to be a multiple of some small constant, say two or three. Then making M a prime increases the likelihood that those keys are spread out evenly. Also if M is a prime number, the division of x by that prime number depends on all the bits of x, not just the bottom k-bits, for some small constant k.

Example: Hash table size = 10
Key value = 112
Hash function = $h(k) = k \bmod M$
$= 112 \bmod 10 = 2$

Disadvantage: A potential disadvantage of the division method is due to the property that consecutive keys map to consecutive hash values.

$h(i) = i$
$h(i + 1) = i + 1 \pmod M$
$h(i + 2) = i + 2 \pmod M$
.
.
.

While this ensures that consecutive keys do not collide, it does not mean that consecutive array locations will be occupied. We will see that in certain implementations this can lead to degradation in performance.

Multiplication method

A variation on the middle-square method that alleviates its deficiencies is called, multiplication hashing method. Instead of multiplying the key x by itself, we multiply the key by a carefully chosen constant 'a' and then extract the middle k bits from the result. In this case, the hashing function is

$$\boxed{h(x) = \left\lfloor \frac{M}{W}(ax \bmod W) \right\rfloor}$$

if we want to avoid the problems that the middle-square method encounters with keys having a large number of leading (or) trailing zero's then we should choose an 'a' that has neither leading nor trailing zero's.

Furthermore, if we, choose an 'a' that is relatively prime to W, then there exists another number 'a' such that $aa' = 1 \pmod W$. Such a number has the nice property that if we take a key x, and multiply it by 'a' to get ax, we can recover the original key by multiplying the product again by a', since $a \times a' = aa'x = 1x$.

The multiplication method for creating a hash function operates in two steps:

Step 1: Multiply the key K by a constant A in the range $0 < A < 1$ and extract the fractional part of KA.
Step 2: Multiply this value by M and take the floor of the result.

In short the hash function is

$$h(k) = \lfloor M \cdot (KA \bmod 1) \rfloor$$

Where (KA mod 1) denotes the fractional part of KA, that is $KA \quad \lfloor KA \rfloor$

Example:
Let $m = 10000$, $K = 123456$ and $A = \frac{\sqrt{5} - 1}{2}$
$= 0.618033$

Then $h(k) = \lfloor 10000 \cdot (123456 \cdot 0.61803 \bmod 1) \rfloor$

$= \lfloor 10000 \cdot (76300.00412 \bmod 1) \rfloor$

$= \lfloor 10000 \cdot 0.00412 \rfloor = 41$

Practical issues

- Easy to implement
  –On most machines multiplication is faster than division.
  –We can substitute one multiplication by shift operation.
  –We don't need to do floating-point operations.
- If successive keys have a large interval, $A = 0.6125423371$ can be recommended.

Mid-square method

A good hash function to use with integer key values is the mid-square method. The mid-square method squares the key value, and then takes out the middle 'r' bits of the result, giving a value in the range 0 to $2^r - 1$. This works well because most (or) all bits of the key value contribute to the result.

Example:
Consider records whose keys are 4-digit numbers in base 10. The goal is to hash these key values to a table of size 100(i.e., a range of 0 to 99).

This range is equivalent to two digits in base 10.

That is $r = 2$. If the input is the number 4567, squaring yields an 8-digit number, 20857489. The middle two digits of this result are 57. All digits of the original key value (equivalently, all bits when the number is viewed in binary) contribute to the middle two digits of the squared value. Thus, the result is not dominated by the distribution of the bottom or the top digit of the original key value. Of course, if the key values all tend to be small numbers, then their squares will only affect the low order digits of the hash value.

Example: To map the key 3121 into a hash table of size 1000, we square it $(3121)^2 = 97\underline{406}41$ and extract 406 as the hash value.

Folding method

The folding method breaks up a key into precise segments that are added to form a hash value, and still another technique is to apply a multiplicative hash function to each segment individually before folding.

Algorithm $H(x) = (a + b + c) \bmod m$. Where a, b, and c represent the preconditioned key broken down into three parts, m is the table size, and mod stands for modulo. In other words: The sum of three parts of the pre conditioned key is divided by the table size. The remainder is the hash key.

Example:
Fold the key 123456789 into a hash table of ten spaces (0 through 9)

We are given $x = 123456789$ and the table size (i.e., $m = 10$)

Since we can break x into three parts any way, we will break it up evenly.

Thus $a = 123$, $b = 456$ and $c = 789$

$H(x) = (a + b + c) \bmod M$

$H(123456789) = (123 + 456 + 789) \bmod 10$

$= 1368 \bmod 10 = 8$

123456789 are inserted into the table at address 8.

The folding method is distribution independent.

Resolving collisions In collision resolution strategy algorithms and data structures are used to handle two hash keys that hash to the same hash keys. There are a number of collision resolution techniques, but the most popular are open addressing and chaining.

- Chaining: An array of linked list, Separate chaining
- Open Addressing: Array based implementation:
  – Linear probing (Linear Search)
  – Quadratic probing (non-linear search)
  – Double hashing (use two hash functions)

Separate chaining Every linked list has each element that collides to the similar slot. Insertion need to locate the accurate slot and appending to any end of the list in that slot wherever, deletion needs searching the list and removal.

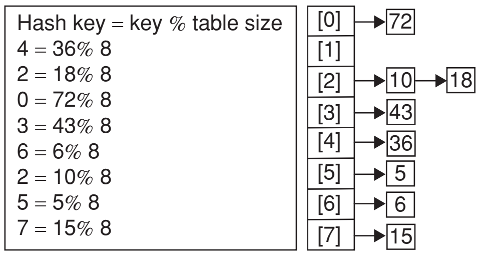


Figure 1 Separate chaining

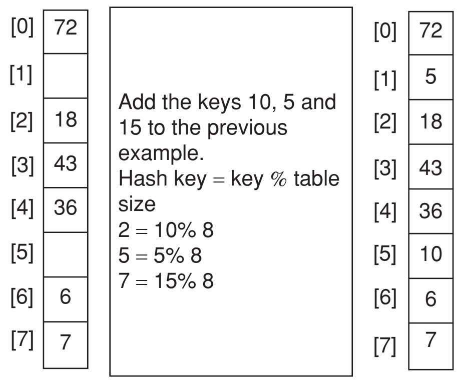
Open addressing Open addressing hash tables are used to stock up the records straight inside the array. This approach is also known as closed hashing. This procedure is based on probing. Well known probe sequence include:

- Linear probing: In which the interval between probes is fixed often at 1.
- Quadratic probing: In which the interval between probes increases proportional to the hash value (the interval thus increasing linearly and the indices are described by a quadratic function).
- Double hashing: In which the interval between probes is computed by another hash function.

(i) **Linear probing:** Linear probing method is used for resolving hash collisions of values of hash functions by sequentially searching the hash table for a free location. The item will be stored in the next available slot in the table in linear probing. Also an assumption is made that the table is not already full.

This is implemented via a linear search for an empty slot, from the point of collision.

If the physical end of table is reached during the linear search, the search will again get start around to the beginning of the table and continue from there. The table is considered as full, if an empty slot is not found before reaching the point of collision.

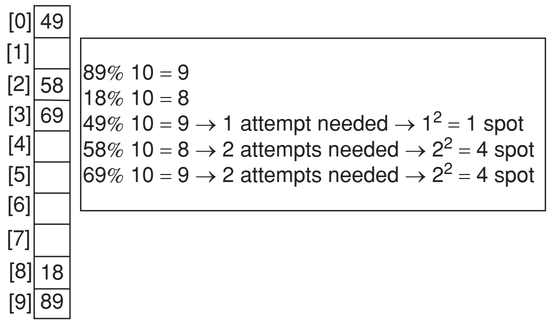


Add the keys 10, 5 and 15 to the previous example.
Hash key = key % table size
2 = 10% 8
5 = 5% 8
7 = 15% 8

Figure 2 Linear probing

Limitation: A problem with linear probe method is primary clustering. In primary clustering blocks of data may possibly be able to form collision. Several attempts may be required by any key that hashes into the cluster to resolve the collision.

(ii) Quadratic probing: To resolve the primary clustering problem, quadratic probing can be used. With quadratic probing, rather than always moving one spot, move i^2 spots from the point of collision where i is the number of attempts needed to resolve the collision.



89% 10 = 9
18% 10 = 8
49% 10 = 9 → 1 attempt needed → 1^2 = 1 spot
58% 10 = 8 → 2 attempts needed → 2^2 = 4 spot
69% 10 = 9 → 2 attempts needed → 2^2 = 4 spot

Limitation: Maximum half of the table can be used as substitute locations to resolve collisions. Once the table gets more than half full, its really hard to locate an unfilled spot. This new difficulty is recognized as secondary clustering because elements that hash to the same hash key will always probe the identical substitute cells.

(iii) Double hashing: Double hashing uses the idea of applying a second hash function to the key when a collision occurs, the result of the second hash function will be the number of positions from the point of collision to insert. There are some requirements for the second function:

1. It must never evaluate to zero
2. Must make sure that all cells can be probed.

A popular second hash function is:
Hash(key) = R-(Key mod R) where R is a prime number smaller than the size of the table.

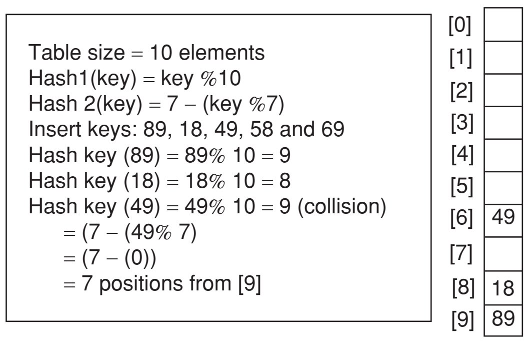
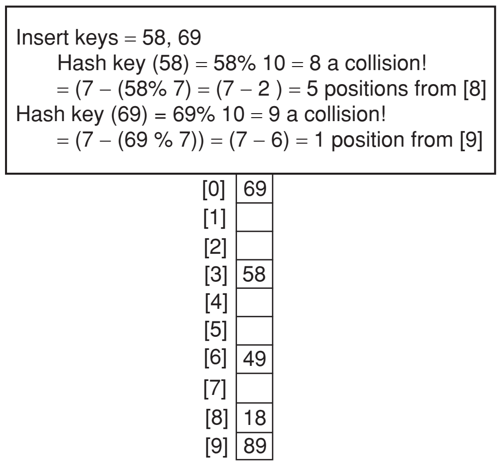


Table size = 10 elements
Hash1(key) = key %10
Hash 2(key) = 7 – (key %7)
Insert keys: 89, 18, 49, 58 and 69
Hash key (89) = 89% 10 = 9
Hash key (18) = 18% 10 = 8
Hash key (49) = 49% 10 = 9 (collision)
= (7 – (49% 7)
= (7 – (0))
= 7 positions from [9]

Figure 3 Double hashing



Insert keys = 58, 69
Hash key (58) = 58% 10 = 8 a collision!
= (7 – (58% 7) = (7 – 2) = 5 positions from [8]
Hash key (69) = 69% 10 = 9 a collision!
= (7 – (69 % 7)) = (7 – 6) = 1 position from [9]

Figure 4 Double hashing

Matrix-chain Multiplication

We are given a sequence of n matrices $m_1, m_2 \ldots m_n$ to be multiplied. If the chain matrices is $< m_1, m_2, m_3, m_4>$, the product m_1, m_2, m_3, m_4 can be fully parenthesized in 5 distinct ways:

1. $(m_1 (m_2 (m_3\, m_4)))$
2. $(m_1 ((m_2\, m_3)\, m_4))$
3. $((m_1\, m_2)\, (m_3\, m_4))$
4. $((m_1 (m_2\, m_3))\, m_4)$
5. $(((m_1\, m_2)\, m_3)\, m_4)$

The way we parenthesize a chain of matrices can have a dramatic impact on the cost of evaluating the product. We can multiply 2 matrices A and B only if they are compatible i.e., the number of columns of A must equal the number of rows of B. If A is a $(p \times q)$ matrix and B is a $(q \times r)$ matrix, the resulting matrix C is a $(p \times r)$ matrix. The time to compute C is the number of scalar multiplications, which is (pqr).

Example: Consider the problem of a chain $<m_1, m_2, m_3>$ of three matrices. Suppose that the dimensions of the matrices are (6 × 8), (8 × 14), (14 × 20) respectively. Which parenthesization will give least number of multiplications?

Soluation:

(i) $((m_1\ m_2)\ m_3)$

$[m_1]_{6\times 8} \times [m_2]_{8\times 14} = [m_1\ m_2]_{6\times 14}$

Number of multiplications performed

$= 6 \times 8 \times 14 = 672$

$[m_1\ m_2]_{6\times 14} \times [m_3]_{14\times 20} = ((m_1\ m_2)\ m_3)_{6\times 20}$

Number of multiplications performed

$= 6 \times 14 \times 20 = 1680$

Total number of multiplications

$= 672 + 1680 = 2352$

(ii) $(m_1\ (m_2\ m_3))$

$[m_2]_{8\times 14} \times [m_3]_{14\times 20} = [m_2\ m_3]_{8\times 20}$

Number of multiplications performed

$= 8 \times 14 \times 20 = 2240$

$[m_1]_{6\times 8} \times [m_2\ m_3]_{8\times 20} = (m_1\ (m_2\ m_3))_{6\times 20}$

Number of multiplications performed

$= 6 \times 8 \times 20 = 960$

Total number of multiplications = 960 + 2240 = 3200

$\therefore$ $((m_1\ m_2)\ m_3)$ gives least number of multiplications.

We need to define the cost of an optimal solution recursively in terms of the optimal solutions to sub problems. For Matrix-chain multiplication problem, we pick as our sub problem the problems of determining the minimum cost of a parenthesization of $A_i\ A_{i+1} \ldots A_j$ for $1 \le i \le j \le n$ let $m[i, j]$ be the minimum number of scalar multiplications needed to compute the matrix $A_i \ldots {}_j$; for the full problem, the cost of a cheapest way to compute $A_1 \ldots {}_N$ would be $m[1, n]$. We can define $m[i, j]$ recursively as follows:

$$\boxed{m\,[i, j] = m\,[i, k] + m\,[k + 1, j] + P_{i-1}\,P_k\,P_j}$$

If $i = j$, the problem is trivial. The chain consists of just one matrix $A_i \ldots {}_i = A_i$, so that no scalar multiplications are necessary to compute the product.

Minimum cost of parenthesizing the product $A_i\ A_{i+1} \ldots A_j$ becomes

$$m[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min\{m[i,k] + m[k+1,j] \\ + p_{i-1}p_k p_i\} & \text{if } i < j, i \le k < j \end{cases}$$

The $m[i, j]$ values give the costs of optimal solutions to sub problems.

At this point, to write a recursive algorithm based on recurrence to compute the minimum cost $m[1, n]$ for multiplying $A_1\ A_2 \ldots A_n$. However, this algorithm takes exponential time, which is not better than the brute force method of checking each way of parenthesizing the product. The important observation we can make at this point is that we have relatively few sub problems, one problem for each choice of i and j satisfying $1 \le i \le j \le n$ (or) $\left(\frac{n}{2}\right) + n = \theta(n^2)$ in all. The property of overlapping sub problems is the second hallmark of the applicability of dynamic programming.

The first hall mark being optimal substructure.

Algorithm

```
1.  $n \leftarrow$ length $[p] - 1$
2.  for $i \leftarrow 1$ to $n$
3.  do $m[i, i] \leftarrow 0$
4.  for $i \leftarrow 2$ to $n$
5.  do for $i \leftarrow 1$ to $n - i + 1$
6.  do $j \leftarrow i + i - 1$
7.  $m[i, j] \leftarrow \infty$
8.  for $k \leftarrow i$ to $j - 1$
9.  do $q \leftarrow m\,[i, k] + m\,[k + 1, j] + P_{i-1}\,P_k\,P_j$
10. if $q < m\,[i, j]$
11. then $m\,[i, j] \leftarrow q$
12. $S[i, j] \leftarrow k$
13. return $m$ and $S$
```

It first computes $m[i, j] \leftarrow 0$ for $i = 1, 2 \ldots n$ (the minimum costs for chains of length 1). To compute $m[i, i + 1]$ for $i = 1, 2, \ldots n - 1$(the minimum costs for chains of length $\lambda = 2$ and so on). At each step, the $m[i, j]$ cost computed depends only on table entries $m[i, k]$ and $m[k + 1, j]$ already computed. An entry $m[i, j]$ is computed using the products $P_{i-1}\ P_k\ P_j$ for $k = i, i + 1, \ldots j - 1$. A simple inspection of the nested loop structure of the above algorithm yields a running time of $O(n^3)$ for the algorithm.

Longest Common Subsequence

A sub sequence of a given sequence is just the given sequence with 0 or more elements left out. Formally, given a sequence $x = \langle x_1, x_2 \cdots x_m \rangle$, another sequence $z = \langle z_1, z_2 \cdots z_k \rangle$ is a subsequence of x if there exists a strictly increasing sequence $\langle i_1, i_2 \ldots i_k \rangle$ of indices of x such that for all $j = 1, 2 \cdots k$, we have $x_{ij} = z_j$

Example: $z = \langle B, C, D, B \rangle$ is a subsequence of $x = \langle A, B, C, B, D, A, B \rangle$ with corresponding index sequence <2, 3, 5, 7>

Example: Given 2 sequences x and y, we say that a sequence z is a common sub sequence of x and y if z is a sub sequence of both x and y.

If $x = \langle A, B, C, B, D, A, B \rangle$

$y = \langle B, D, C, A, B, A \rangle$

The sequence $\langle B, C, A \rangle$ is a common subsequence of both x and y.

The sequence $\langle B, C, A \rangle$ is not a longest common subsequence (LCS) of x and y since it has length '3' and the sequence $\langle B, C, B, A \rangle$, which is also common to both x and y, has length 4. The sequence $\langle B, C, B, A \rangle$ is an LCS of x and y, as is the sequence $\langle B, D, A, B \rangle$, since there is no common subsequence of length 5 or greater.

- In the longest-common-sub sequence problem, we are given 2 sequences $x = <x_1, x_2, x_3 \ldots x_m>$ and $y = <y_1, y_2 \ldots y_n>$ and wish to find a maximum length common subsequence of x and y.
- LCS problem can be solved efficiently using dynamic programming.
- A brute force approach to solve the LCS problem is to enumerate all subsequences of x and check each subsequence to see if it is also a subsequence of y, keeping track of the longest subsequence found. Each subsequence of x corresponds to a subset of the indices $\{1, 2 \ldots m\}$ of x. There are 2^m subsequences of x, so this approach requires exponential time, making it impractical for long sequences.
- The classes of sub problems correspond to pairs of 'pre fixes' of 2 input sequences:

Given a sequence $x = <x_1, x_2 \cdots x_m>$, we define the ith prefix of x, for $i = 0, 1, \ldots m$, as

$$x_i = <x_1\ x_2 \ldots x_i>$$

Example: If $x = <A, B, C, B, D, A, D>$, then $x_4 = <A, B, C, B>$ and x_0 is the empty sequence. LCS problem has an optimal sub-structure property.

Optimal Substructure of LCS

Let $x = <x_1, x_2 \ldots x_m>$ and $y = <y_1, y_2 \ldots y_n>$ be sequences and let $z = <z_1, z_2 \ldots z_k>$ be any LCS of x and y then

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and z_{k-1} is an LCS of x_{m-1} and y_{n-1}.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that z is an LCS of x_{m-1} and y.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that z is an LCS of x and y_{n-1}.

NP-HARD AND *NP*-COMPLETE

A mathematical problem for which, even in theory, no shortcut or smart algorithm is possible that would lead to a simple or rapid solution. Instead the only way to find an optimal solution is a computationally intensive, exhaustive analysis in which all possible outcomes are tested. Examples of NP-hard problems include the travelling salesman problem.

P-problem

A problem is assigned to the *P*(polynomial time) class if there exists at least one algorithm to solve that problem, such that number of steps of the algorithm is bounded by a polynomial in n, where n is the length of the input.

NP-problem

A problem is assigned to the *NP* (non-deterministic polynomial time) class if it is solvable in polynomial time by a non-deterministic turing machine.

A *P*-problem (whose solution time is bounded by a polynomial) is always also *NP*. If a problem is known to be *NP*, and a solution to the problem is somehow known, then demonstrating the correctness of the solution can always be reduced to a single *P* (polynomial time) verification. If *P* and *NP* are not equivalent then the solution of *NP*-problems requires (in the worst case) an exhaustive search.

A problem is said to be *NP*-hard, if an algorithm for solving it can be translated into one for solving any other *NP*-problem. It is much easier to show that a problem is *NP* than to show that it is *NP*-hard. A problem which is both *NP* and *NP*-hard is called an *NP*-complete problem.

P versus NP-problems

The *P* versus *NP* problem is the determination of whether all *NP*-problems are actually *P*-problems, if *P* and *NP* are not equivalent then the solution of *NP*-problem requires an exhaustive search, while if they are, then asymptotically faster algorithms may exist.

NP-complete problem

A problem which is both *NP* (verifiable in non-deterministic polynomial time) and *NP*-hard (any *NP*-problem can be translated into this problem). Examples of *NP*-hard problems include the Hamiltonian cycle and travelling sales man problems.

Example:

Circuit satisfiability is a good example of problem that we don't know how to solve in polynomial time. In this problem, the input is a Boolean circuit. A collection of and, or and not gates connected by wires. The input to the circuit is a set of m Boolean (true/false) values $x_1 \ldots x_m$. The output is a single Boolean value. The circuit satisfiability problem asks, given a circuit, whether there is an input that makes the circuit output TRUE, or conversely, whether the circuit always outputs FLASE. Nobody knows how to solve this problem faster than just trying all 2^m possible inputs to the circuit but this requires exponential time.

P, NP, and Co-NP

- *P* is a set of yes/no problems that can be solved in polynomial time. Intuitively *P* is the set of problems that can be solved quickly.
- *NP* is the set of yes/no problems with the following property: If the answer is yes, then there is a proof of this fact that can be checked in polynomial time. Intuitively *NP* is the set of problems where we can verify a YES answer quickly if we have the solution in front of us.

Example: The circuit satisfiability problem is in *NP*.

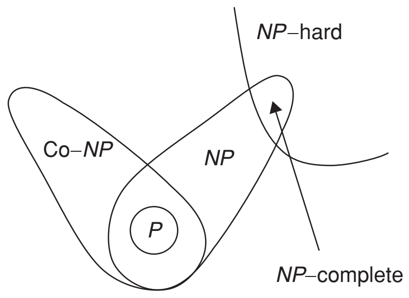
If the answer is yes, then any set of m input values that produces TRUE output is a proof of this fact, we can check the proof by evaluating the circuit in polynomial time.

- Co-*NP* is the exact opposite of *NP*. If the answer to a problem in co-*NP* is no, then there is a proof of this fact that can be checked in polynomial time.

- π is *NP*-hard $\Rightarrow$ if π can be solved in polynomial time, then $P = NP$.

This is like saying that if we could solve one particular *NP*-hard problem quickly, then we could solve any problem whose solution is easy to understand, using the solution to that one special problem as a subroutine. *NP*-hard problems are atleast as hard as any problem in *NP*.

- Saying that a problem is *NP*-hard is like saying 'If I own a dog, then it can speak fluent English'. You probably don't know whether or not I own a dog, but you're probably pretty sure that I don't own a talking dog. Nobody has a mathematical proof that dogs can't speak English. The fact that no one has ever heard a dog speak English is evidence as per the hundreds of examinations of dogs that lacked the proper mouth shape and brain power, but mere evidence is not a proof nevertheless, no sane person would believe me if I said I owned a dog that spoke fluent English. So the statement 'If I own a dog then it can speak fluent English' has a natural corollary: No one in their right mind should believe that I own a dog ! Likewise if a problem is *NP*-hard no one in their right mind should believe it can be solved in polynomial time.



Cooks Theorem

Cook's theorem states that CNFSAT is *NP*-Complete

It means, if the problem is in *NP*, then the deterministic Turing machine can reduce the problem in polynomial time.

The inference that can be taken from these theorems is, if deterministic polynomial time algorithm exists for solving satisfiability, then to all problems present in *NP* can be solved in polynomial time.

Non-deterministic Search

Non-deterministic algorithms are faster, compared to deterministic ones. The computations are fast as it always chooses right step

The following functions are used to specify these algorithms

1. Choice (A), which chooses a random element from set A
2. Failure (A), specifies failure
3. Success ( ), Specifies success

The non-deterministic search is done as follows.

Let us consider an array $S[1 \ldots n]$, $n \geq 1$ we need to get the indice of 'i' such that $S[i] = t$ (or) $i = 0$. The algorithm is given below.

Steps:

```
1. i = Choice (1, n);
2. if S[i] = t, then
     (i) Print (i);
    (ii) Success ( );
3. Print (0)
   failure
4. Stop.
```

If the search is successful it returns the indice of array '*S*', otherwise it returns '0', the time complexity is $\Omega(n)$.

Exercises

Practice Problems I

Directions for questions 1 to 15: Select the correct alternative from the given choices.

1. Hash the keys 12, 44, 13, 88, 23, 94, 11, 39, 20 using the hash function with chaining $(2k + 5) \bmod 11$, which of the following slots are empty?
   (A) 0, 1, 2, 3, 4 (B) 0, 2, 3, 4, 8, 10
   (C) 0, 1, 2, 4, 8, 10 (D) 0, 1, 2, 4, 8
2. Using linear probing on the list given in the above question with the same hash function, which slots are not occupied?
   (A) 3, 4 (B) 4, 5
   (C) 3, 6 (D) 4, 6
3. In hashing, key value 123456 is hashed to which address using multiplication method ($m = 10^4$)?
   (A) 40 (B) 41
   (C) 42 (D) 44
4. Insert element 14 into the given hash table with double hashing? $h_1(k) = k \bmod 13$, $h_2(k) = 1 + (k \bmod 11)$. The element will occupy, which slot?

0	
1	79
2	
3	
4	69
5	98
6	
7	72
8	
9	
10	
11	50
12	

(A) 7th (B) 8th
(C) 2nd (D) 9th

5. Consider the below given keys:

257145368, 25842354, 12487654, 248645452. Find the hash values of keys using shift folding method?

(A) 770, 221, 153, 345 (B) 221, 770, 153, 345
(C) 760, 770, 153, 345 (D) 815, 770, 153, 345

6. Consider the following two problems on unidirected graphs.

β : Given $G(V, E)$, does G have an independent set of size $|V|-4$?

α : Given $G(V, E)$, does G have an independent set of size 5?

Which of the following is true?

(A) β is in P and α is in NP-Complete
(B) β is in NP-Complete and α is in P
(C) Both α and β are NP-Complete
(D) Both α and β are in P

7. Let S be an NP-complete problem and Q and R be two other problems not known to be in NP. Q is polynomial-time reducible to S and S is polynomial-time reducible to R. Which one of the following statements is true?

(A) R is NP-Complete (B) R is NP-Hard
(C) Q is NP-Complete (D) Q is NP-Hard

8. Let FHAM$_3$ be the problem of finding a Hamiltonian cycle in a graph $G = (V, E)$ with $|V|$ divisible by 3 and DHAM$_3$ be the problem of determining if a Hamiltonian cycle exists in such graphs. Which of the following is true?

(A) Both FHAM$_3$ and DHAM$_3$ are NP-hard
(B) FHAM$_3$ is NP-hard but DHAM$_3$ is not
(C) DHAM$_3$ is NP-hard but FHAM$_3$ is not
(D) Neither FHAM$_3$ nor DHAM$_3$ is NP-hard

9. Consider a hash table of size 7, with starting index '0' and a hash function $(3x + 4)$ mod 7. Initially hash table is empty. The sequence 1, 3, 8, 10 is inserted into the table using closed hashing then what is the position of element 10?

(A) 1st (B) 2nd
(C) 6th (D) 0th

10. Place the given keys in the hash table of size 13, index from '0' by using open hashing, hash function is $h(k)$ mod 13.

Keys: A, FOOL, HIS, AND

(hint : Add the positions of a word's letters in the alphabet, take $A \to 1, B \to 2, C \to 3. D \to 4 \ldots Z \to 26$).

Which of the following shows the correct hash addresses of keys?

(A) A – 1, FOOL – 10, HIS – 9, AND – 6
(B) A – 1, FOOL – 9, HIS – 10, AND – 6
(C) A – 0, FOOL – 6, HIS – 10, AND – 9
(D) A – 0, FOOL – 9, HIS – 9, AND – 6

11. Consider the following input (322, 334, 471, 679, 989, 171, 173, 199) and the hash function is x mod 10 which statement is true?

I. 679, 989, 199 hash to the same value

II. 471, 171, hash to the same value

III. Each element hashes to a different value

IV. All the elements hash to the same value

(A) I Only (B) II Only
(C) I and II (D) III

12. For the input 30, 20, 56, 75, 31, 19 and hash function $h(k) = k$ mod 11, what is the largest number of key comparisons in a successful search in the open hash table.

(A) 4 (B) 3
(C) 5 (D) 2

13. The keys 12, 18, 13, 2, 3, 23, 5 and 15 are inserted into an empty hash table of length 10 using open addressing with hash function, $h(k) = k$ mod 10 and linear probing.

Which is the resultant hash table?

(A)

	0
	1
2	2
23	3
13	4
15	5
	6
	7
	8
	9

(B)

	0
3	1
12	2
13	3
	4
15	5
	6
	7
	8
	9

(C)

	0
	1
12	2
13	3
2	4
3	5
23	6
5	7
18	8
15	9

(D)

	0
	1
2	2
3	3
12	4
13	5
23	6
5	7
18	8
15	9

14. Which one of the following is correct?

(A) Finding shortest path in a graph is solvable in poly-nomial time.
(B) Finding longest path from a graph is solvable in poly-nomial time.
(C) Finding longest path from a graph is solvable in polynomial time, if edge weights are very small values.
(D) Both (A) and (B) are correct

15. In the following pair of problems

$$\underbrace{\text{2 CNF Satisfiability}}_{\text{I}} \text{ Vs } \underbrace{\text{3 CNF Satisfiability}}_{\text{II}}.$$

(A) I is solvable in polynomial time, II is NP complete problem.
(B) II is solvable in polynomial time, I is NP complete problem.
(C) Both are solvable in polynomial time
(D) None can be solved in polynomial time.

Practice Problems 2

Directions for questions 1 to 15: Select the correct alternative from the given choices.

1. For *NP*-complete problems
(A) Several polynomial time algorithms are available
(B) No polynomial time algorithm is discovered yet
(C) Polynomial time algorithms exist but not discovered
(D) Polynomial time algorithms will not exist, hence cannot be discovered

2. In the division method for creating hash functions, we map a key k into one of m slots by taking the remainder of k divided by m. That is, the hash function is
(A) $h(k) = m \bmod k$
(B) $h(k) = m \bmod m/k$
(C) $h(k) = k \bmod m$
(D) $h(k) = mk \bmod k$

3. In the division method for creating hash function, which of the following hash table size is most appropriate?
(A) 2
(B) 7
(C) 4
(D) 8

4. Which of the following techniques are commonly used to compute the probe sequence required for open addressing?
(A) Linear probing
(B) Quadratic probing
(C) Double hashing
(D) All the above

5. Which of the following problems is not *NP*-hard?
(A) Hamiltonian circuit problem
(B) The 0/1 knapsack problem
(C) The graph coloring problem
(D) None of these

6. For problems x and y, y is *NP*-complete and x reduces to y in polynomial time. Which of the following is true?
(A) If x can be solved in polynomial time, then so can y
(B) x is *NP*-hard
(C) x is *NP*-complete
(D) x is in *NP*, but not necessarily *NP*-complete

7. If P_1 is *NP*-complete and there is a polynomial time reduction of P_1 to $P_{2,}$ then P_2 is
(A) NP-complete
(B) Not necessarily NP-complete
(C) Cannot be *NP*-complete
(D) None of these

8. A problem is in *NP*, and as hard as any problem in *NP*. The given problem is
(A) *NP* hard
(B) *NP* complete
(C) *NP*
(D) *NP*-hard $\cap$ *NP*-complete

9. Which of the following is TRUE?
(A) All *NP*-complete problems are *NP*-hard.
(B) If an *NP*-hard problem can be solved in polynomial time, then all *NP*-complete problems can be solved in polynomial time.
(C) *NP*-hard problems are not known to be *NP*-complete.
(D) All the above

10. If a polynomial time algorithm makes polynomial number of calls to polynomial time subroutines, then the resulting algorithm runs in
(A) Polynomial time
(B) No-polynomial time
(C) Exponential time
(D) None of these

11. If a polynomial time algorithm makes atmost constant number of calls to polynomial time subroutines, then the resulting algorithm runs in
(A) Polynomial time
(B) No-polynomial time
(C) Exponential time
(D) None of these

12. When a record to be inserted maps to an already occupied slot is called
(A) Hazard
(B) Collision
(C) Hashing
(D) Chaining

13. Worst-case analysis of hashing occurs when
(A) All the keys are distributed
(B) Every key hash to the same slot
(C) Key values with even number, hashes to slots with even number
(D) Key values with odd number hashes to slots with odd number

14. Main difference between open hashing and closed hashing is
(A) Closed hashing uses linked lists and open hashing does not.
(B) Open hashing uses linked list and closed hashing does not
(C) Open hashing uses tree data structure and closed uses linked list
(D) None of the above

15. The worst case scenario in hashing occurs when
(A) All keys are hashed to the same cell of the hash table
(B) The size of hash table is bigger than the number of keys
(C) The size of hash table is smaller than the number of keys
(D) None of the above

Previous Years' Questions

1. Consider a hash table of size seven, with starting index zero, and a hash function $(3x + 4)$ mod7. Assuming the hash table is initially empty, which of the following is the contents of the table when the sequence 1, 3, 8, 10 is inserted into the table using closed hashing? Note that − denotes an empty location in the table. **[2007]**

(A) 8, −, −, −, −, −, 10 (B) 1, 8, 10, −, −, −, 3
(C) 1, −, −, −, −, −, 3 (D) 1, 10, 8, −, −, −, 3

Common data for questions 2 and 3: Suppose the letters a, b, c, d, e, f have probabilities $\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \frac{1}{32}, \frac{1}{32}$, respectively.

2. Which of the following is the Huffman code for the letter a, b, c, d, e, f? **[2007]**

(A) 0, 10, 110, 1110, 11110, 11111
(B) 11, 10, 011, 010, 001, 000
(C) 11, 10, 01, 001, 0001, 0000
(D) 110, 100, 010, 000, 001, 111

3. What is the average length of the correct answer to above question? **[2007]**

(A) 3 (B) 2.1875
(C) 2.25 (D) 1.9375

4. The subset-sum problem is defined as follows: Given a set S of n positive integers and a positive integer W, determine whether there is a subset of S whose elements sum to W.

An algorithm Q solves this problem in $O(nW)$ time. Which of the following statements is false? **[2008]**

(A) Q solves the subset-sum problem in polynomial time when the input is encoded in unary
(B) Q solves the subset-sum problem in polynomial time when the input is encoded in binary
(C) The subset sum problem belongs to the class NP
(D) The subset sum problem is NP-hard

5. Let π_A be a problem that belongs to the class NP. Then which one of the following is TRUE? **[2009]**

(A) There is no polynomial time algorithm for π_A.
(B) If π_A can be solved deterministically in polynomial time, then $P = NP$.
(C) If π_A is NP-hard, then it is NP-complete.
(D) π_A may be undecidable.

6. The keys 12, 18, 13, 2, 3, 23, 5 and 15 are inserted into an initially empty hash table of length 10 using open addressing with hash function $h(k) = k$ mod 10 and linear probing. What is the resultant hash table? **[2009]**

(A)

0	
1	
2	12
3	23
4	
5	15
6	
7	
8	18
9	

(B)

0	
1	
2	12
3	13
4	
5	5
6	
7	
8	18
9	

(C)

0	
1	
2	12
3	13
4	2
5	3
6	23
7	5
8	18
9	15

(D)

0	
1	
2	12,2
3	13,3,23
4	
5	5,15
6	
7	
8	18
9	

Common data for questions 7 and 8: A sub-sequence of a given sequence is just the given sequence with some elements (possibly none or all) left out. We are given two sequences $X[m]$ and $Y[n]$ of lengths m and n, respectively, with indexes of X and Y starting from 0.

7. We wish to find the length of the longest common subsequence (LCS) of $X[m]$ and $Y[n]$ as $l(m, n)$, where an incomplete recursive definition for the function $l(i, j)$ to compute the length of the LCS of $X[m]$ and $Y[n]$ is given below:

$I(i, j) = 0$, if either $i = 0$ or $j = 0$
= expr1, if $i, j > 0$ and $X[i - 1] = Y[j - 1]$
= expr2, if $i, j > 0$ and $X[i - 1] \neq Y[j - 1]$

Which one of the following options is correct? **[2009]**

(A) expr1 $\equiv I(i - 1, j) + 1$
(B) expr1 $\equiv I(i, j - 1)$
(C) expr2 $\equiv \max(I(i - 1, j), I(i, j - 1))$
(D) expr2 $\equiv \max(I(i - 1, j - 1), I(i, j))$

8. The values of $l(i, j)$ could be obtained by dynamic programming based on the correct recursive definition of $l(i, j)$ of the form given above, using an array $L[M, N]$, where $M = m + 1$ and $N = n + 1$, such that $L[i, j] = l(i, j)$.
Which one of the following statements would be TRUE regarding the dynamic programming solution for the recursive definition of $l(i, j)$? **[2009]**

(A) All elements of L should be initialized to 0 for the values of $l(i, j)$ to be properly computed.

(B) The values of $l(i, j)$ may be computed in a row major order or column major order of $L(M, N)$.
(C) The values of $l(i, j)$ cannot be computed in either row major order or column major order of $L(M, N)$.
(D) $L[p, q]$ needs to be computed before $L[r, s]$ if either $p < r$ or $q < s$.

9. The weight of a sequence $a_0, a_1, \cdots, a_{n-1}$ of real numbers is defined as $a_0 + a_1/2 + \cdots + a_{n-1}/2^{n-1}$. A subsequence of a sequence is obtained by deleting some elements from the sequence, keeping the order of the remaining elements the same. Let X denote the maximum possible weight of a subsequence of $a_0, a_1, \ldots, a_{n-1}$. Then X is equal to **[2010]**
(A) $\max(Y, a_0 + Y)$ (B) $\max(Y, a_0 + Y/2)$
(C) $\max(Y, a_0 + 2Y)$ (D) $a_0 + Y/2$

10. Four matrices M_1, M_2, M_3 and M_4 of dimensions $p \times q$, $q \times r$, $r \times s$ and $s \times t$ respectively, can be multiplied in several ways with different number of total scalar multiplications. For example when multiplied as $((M_1 \times M_2) \times (M_3 \times M_4))$, the total number of scalar multiplications is $pqr + rst + prt$. When multiplied $(((M_1 \times M_2) \times M_3) \times M_4)$ the total number of scalar multiplications is $pqr + prs + pst$.

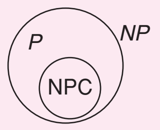
If $p = 10$, $q = 100$, $r = 20$, $s = 5$ and $t = 80$, then the minimum number of scalar multiplications needed is **[2011]**

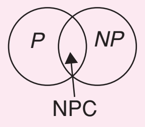
(A) 248000
(B) 44000
(C) 19000
(D) 25000

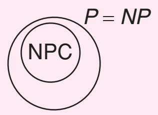
11. Assuming $P \neq NP$, which of the following is **TRUE**? **[2012]**
(A) NP-complete $= NP$
(B) NP-complete $\cap P = \varnothing$
(C) NP-hard $= NP$
(D) $P = NP$-complete

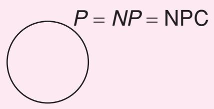
12. Which of the following statements are TRUE?
(i) The problem of determining whether there exists a cycle in an undirected graph is in P.
(ii) The problem of determining whether there exists a cycle in an undirected graph is in NP.
(iii) If a problem A is NP-Complete, there exists a non-deterministic polynomial time algorithm to solve A. **[2013]**
(A) 1, 2 and 3 (B) 1 and 2 only
(C) 2 and 3 only (D) 1 and 3 only

13. Suppose a polynomial time algorithm is discovered that correctly computes the largest clique in a given graph. In this scenario, which one of the following represents the correct Venn diagram of the complexity classes P, NP and NP-complete (NPC)? **[2014]**

(A) 

(B) 

(C) 

(D) 

14. Consider a hash, table with 9 slots. The hash function is $h(K) = K \bmod 9$. The collisions are resolved by chaining. The following 9 keys are inserted in the order: 5, 28, 19, 15, 20, 33, 12, 17, 10. The maximum, minimum, and average chain lengths in the hash table, respectively, are **[2014]**
(A) 3, 0 and 1 (B) 3, 3 and 3
(C) 4, 0 and 1 (D) 3, 0 and 2

15. Consider two strings A = 'qpqrr' and B = 'pqprqrp'. Let x be the length of the longest common subsequence (not necessarily contiguous between A and B and let y be the number of such longest common subsequences between A and B. then $x + 10y =$ ______ **[2014]**

16. Suppose you want to move from 0 to 100 on the number line. In each step, you either move right by a unit distance or you take a *shortcut*. A shortcut is simply a pre-specified pair of integers i, j with $i < j$. Given a shortcut i, j if you are at position i on the number line, you may directly move to j. Suppose $T(k)$ denotes the smallest number of steps needed to move from k to 100. Suppose further that there is at most 1 shortcut involving any number, and in particular from 9 there is a shortcut to 15. Let y and z be such that $T(9) = 1 + \min(T(y), T(z))$. Then the value of the product yz is ______ **[2014]**

17. Consider the decision problem 2CNFSAT defined as follows: **[2014]**

$\{\phi \mid \phi$ is a satisfiable propositional formula in CNF with at most two literals per clause$\}$

For example, $\phi = (x_1 \vee x_2) \wedge (x_1 \vee x_3) \wedge (x_2 \vee x_4)$ is a Boolean formula and it is in 2CNFSAT.

The decision problem 2CNFSAT is
(A) NP-complete
(B) Solvable in polynomial time by reduction to directed graph reach ability.
(C) Solvable in constant time since any input instance is satisfiable.
(D) NP-hard, but not NP-complete

18. Consider a hash table with 100 slots. Collisions are resolved using chaining. Assuming simple uniform hashing, what is the probability that the first 3 slots are unfilled after the first 3 insertions? **[2014]**

(A) $(97 \times 97 \times 97)/100^3$
(B) $(99 \times 98 \times 97)/100^3$
(C) $(97 \times 96 \times 95)/100^3$
(D) $(97 \times 96 \times 95)/(3! \times 100^3)$

19. Match the following **[2014]**

(P) prim's algorithm for minimum spanning tree	(i) Backtracking
(Q) Floyd-Warshall algorithm for all pairs shortest paths	(ii) Greedy method
(R) Mergesort	(iii) Dynamic programming
(S) Hamiltonian circuit	(iv) Divide and conquer

(A) P–iii, Q–ii, R–iv, S–i
(B) P–i, Q–ii, R–iv, S–iii
(C) P–ii, Q–iii, R–iv, S–i
(D) P–ii, Q–i, R–iii, S–iv

20. Given a hash table T with 25 slots that stores 2000 elements, the load factor $\propto$ for T is ________ **[2015]**

21. Language L_1 is polynomial time reducible to language L_2. Language L_3 is polynomial time reducible to L_2, which in turn is polynomial time reducible to language L_4. Which of the following is/are true? **[2015]**

(1) if $L_4 \in P$, then $L_2 \in P$
(2) if $L_1 \in P$ or $L_3 \in P$, then $L_2 \in P$
(3) $L_1 \in P$, if and only if $L_3 \in P$
(4) if $L_4 \in P$, then $L_1 \in P$ and $L_3 \in P$

22. The Floyd - Warshall algorithm for all -pair shortest paths computation is based on **[2016]**

(A) Greedy paradigm
(B) Divide-and-Conquer paradigm
(C) Dynamic Programming paradigm
(D) Neither Greedy nor Divide-and-Conquer nor Dynamic Programming paradigm.

23. Let A_1, A_2, A_3, and A_4 be four matrices of dimensions 10×5, 5×20, 20×10, and 10×5, respectively. The minimum number of scalar multiplications required to find the product $A_1 A_2 A_3 A_4$ using the basic matrix multiplication method is _____ . **[2016]**

24. Consider the following table:

Algorithms	Design Paradigms
(P) Kruskal	(i) Divide and Conquer
(Q) Quicksort	(ii) Greedy
(R) Floyd-Warshall	(iii) Dynamic Programming

Match the algorithms to the design paradigms they are based on. **[2017]**

(A) (P) ↔ (ii), (Q) ↔ (iii),(R) ↔ (i)
(B) (P) ↔ (iii), (Q) ↔ (i), (R) ↔ (ii)
(C) (P) ↔ (ii), (Q) ↔ (i), (R) ↔ (iii)
(D) (P) ↔ (i), (Q) ↔ (ii), (R) ↔ (iii)

25. Assume that multiplying a matrix G_1 of dimension $p \times q$ with another matrix G_2 of dimension $q \times \underline{r}$ requires pqr scalar multiplications. Computing the product of n matrices $G_1G_2G_3, \ldots, G_n$ can be done by parenthesizing in different ways. Define $G_i\ G_{i+1}$ as an explicitly computed pair for a given paranthesization if they are directly multiplied. For example, in the matrix multiplication chain $G_1G_2G_3G_4G_5G_6$ using parenthesization $(G_1(G_2G_3))(G_4(G_5G_6))$, G_2G_3 and G_5G_6 are the only explicitly computed pairs.

Consider a matrix multiplication chain $F_1F_2F_3F_4F_5$, where matrices F_1, F_2, F_3, F_4, and F_5 are of dimensions 2×25, 25×3, 3×16, 16×1 and 1×1000, respectively. In the parenthesization of $F_1F_2F_3F_4F_5$ that minimizes the total number of scalar multiplications, the explicitly computed pairs is/are: **[2018]**

(A) F_1F_2 and F_3F_4 only
(B) F_2F_3 only
(C) F_3F_4 only
(D) F_1F_2 and F_4F_5 only

26. Consider the weights and values of items listed below. Note that there is only one unit of each item.

Item no.	Weight (in Kgs)	Value (in Rupees)
1	10	60
2	7	28
3	4	20
4	2	24

The task is to pick a subset of these items such that their total weight is no more than 11 kgs and their total value is maximized. Moreover, no item may be split. The total value of items picked by an optimal algorithm is denoted by V_{opt}. A greedy algorithm sorts the items by their value-to-weight ratios in descending order and packs them greedily, starting from the first item in the ordered list. The total value of items picked by the greedy algorithm is denoted by V_{greedy}. The value of $V_{\text{opt}} - V_{\text{greedy}}$ is ______. **[2018]**

Answer Keys

Exercises

Practice Problems 1

1. B	**2.** A	**3.** B	**4.** D	**5.** A	**6.** C	**7.** C	**8.** A	**9.** B	**10.** B
11. C	**12.** B	**13.** C	**14.** A	**15.** A					

Practice Problems 2

1. B	**2.** C	**3.** B	**4.** D	**5.** B	**6.** C	**7.** A	**8.** B	**9.** D	**10.** C
11. A	**12.** B	**13.** B	**14.** B	**15.** A					

Previous Years' Questions

1. B	**2.** A	**3.** D	**4.** B	**5.** C	**6.** C	**7.** C	**8.** B	**9.** C	**10.** B
11. B	**12.** A	**13.** D	**14.** A	**15.** 34	**16.** 150	**17.** B	**18.** A	**19.** C	**20.** 80
21. C	**22.** C	**23.** 1500	**24.** C	**25.** C	**26.** 16				