A Handbook on
Computer Science

5

Algorithms

CONTENTS

Analysis and Asymptotic Notations 174

Divide-and-Counquer 180

Greedy Approach 183

Dynamic Programming 186

Graph Traversals, Hashing and Sorting 190

Complexity Classes: P, NP, NPH and NPC 194

Analysis and Asymptotic Notations

1

Algorithm

An algorithm is the step by step instructions to solve a given problem.

Asymptotic Notation

Big-O Notation	Comparison Notation	Limit Definition
$f \in o(g)$	$f \lessdot g$	$\lim_{x\to\infty} \frac{f(x)}{g(x)} = 0$
$f \in O(g)$	$f \leqslant g$	$\lim_{x\to\infty} \frac{f(x)}{g(x)} < \infty$
$f \in \Theta(g)$	$f = g$	$\lim_{x\to\infty} \frac{f(x)}{g(x)} \in R > 0$
$f \in \Omega(g)$	$f \geqslant g$	$\lim_{x\to\infty} \frac{f(x)}{g(x)} > 0$
$f \in \omega(g)$	$f \gtrdot g$	$\lim_{x\to\infty} \frac{f(x)}{g(x)} = \infty$

Complexity Types

Complexity	Notation	Description
Constant	O(1)	Constant number of operations, not depending on the input data size n = 1000000 → 1-2 operations
Logarithmic	O(log n)	Number of operations proportional to $\log_2 n$ n = 1000000000 → 30 operations
Linear	O(n)	Number of operations proportional to input data size n = 10000 → 5000 operations
Quadratic	$O(n^2)$	Number of operations proportional to the square of the input data size n = 500 → 250000 operations
Cubic	$O(n^3)$	Number of operations proportional to the cube of the input data size n = 200 → 8000000 operations
Exponential	$O(2^n)$, $O(k^n)$, O(n!)	Exponential number of operations, fast growing n = 20 → 1048576 operations

Properties of Asymptotic Notation

- **Reflexive Property:** If $f(n) = O(f(n))$ and $f(n) = \Omega(f(n))$ then $f(n) = \Theta(f(n))$.
- **Transitive Property:**
  (*i*) If $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$ then $f(n)$ is $\Theta(h(n))$.
  (*ii*) If $f(n) = O(g(n))$ and $g(n) = O(h(n))$ then $f(n)$ is $O(h(n))$.
  (*iii*) If $f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n))$ then $f(n)$ is $\Omega(h(n))$.
- **Symmetric Property:** $f(n) = \Theta(g(n))$ **if and only if** $g(n) = \Theta(f(n))$
- **Transpose Symmetry:** $f(n) = O(g(n))$ iff $g(n) = \Omega(f(n))$
- If $f(n) = O(g(n))$ and $e(n) = O(h(n))$ then
  (a) $f(n) + e(n) = \mathbf{O(g(n) + h(n))}$ **or** $\mathbf{O(\max(g(n), h(n)))}$
  (b) $f(n).e(n) = \mathbf{O(g(n).h(n))}$
  (c) If $f(n) = O(g(n))$ then $\mathbf{h(n).f(n) = O(h(n).g(n))}$

Important Formulae

1. $\log x^y = y \log x$
2. $\log^k n = (\log n)^k$
3. $\log_b x = \log_a x / \log_a b$
4. $\log_b x = \log x^a / \log b^a$
5. $\log(xy) = \log x + \log y$
6. $\log(x/y) = \log x - \log y$
7. $n! = O(n^n)$
8. $2^n = O(n^n)$
9. $\sum_{i=1}^{n} 2^i = O(2^n)$
10. $\sum_{k=1}^{n} \log k = O(n \log n)$
11. $\sum_{k=1}^{n} \frac{1}{k} \log k = O(n \log n)$
12. $\sum_{i=1}^{n} i^2 = O(n^3)$
13. $\sum_{k=1}^{n} k^p = \frac{1}{p+1} n^{p+1}$
14. $\sum_{i=1}^{n} i = 1 + 2 + 3 \ldots + n = O(n^2)$
15. $\sum_{k=0}^{n} x^k = 1 + x + x^2 + x^3 + \ldots x^n = \frac{x^{n+1}}{x-1}; (x \neq 1)$

Asymptotic Analysis

Loops

The running time of a loop, is atmost the running time of the statements inside the loop (including tests) multiplied by the number of iterations.

Example:

```
main( )                          ↣   1
{
    x = y + z;                   ↣   1
    for (i =1; i <= n; i++)      ↣   2n+1
    {
        x = y + z;               ↣   n
    }
}
Total time                       ↣   = cn = O(n)
```

Nested Loop

Analyse from inner loop to outer loop. Total running time is the product of the sizes of all the loops

Example:

```
for (i = 1, i <= n, i++)         ↣   2n+1
{
    for (j = 1, j <= n, j++)     ↣   O(n²)
    k = k + 1;                   ↣   O(n²)
}
Total time                       ↣   = cn² = O(n²)
```

Consecutive for Statements

Add the time complexities of for statements.

```
x = x + 1;                       ↣   O(1)
for (i = 1; i < n; i++)          ↣   O(n)
    m = m + 2;                   ↣   O(n)
for (i = 1; i ≤ n; i++)          ↣   O(n)
{
    for (j = 1; j < n; k++)      ↣   O(n²)
    k = k + 1;                   ↣   O(n²)
}
Total time                       ↣   = O(n²)
```

If-then-else Statement

Worst case running time is either **then** part or **else** part (whichever is the larger).

Example:

```
if (length() = = 0)                          ↣   O(1)
{
    return false;                            ↣   O(1)
}
else
{
    for (n = 0; n < length(); n ++)          ↣   O(n)
    {
        if (! list [n] = (otherlist [n]))    ↣   O(n)
        return false;                        ↣   O(n)
    }
}
Total Time                                   ↣   = O(n)
```

Logarithmic Complexity

An algorithm is O(log$n$) if it takes constant time to cut the problem size by a fraction (usually by 1/2)

Example:

```
for (i = 1; i < n; )                         ↣   O(log n)
{
    i = i * 2;                               ↣   O(log n)
}
Total Time                                   ↣   = O(log n)
```

Recurrence Relations

Substitution Method

Substituting the given function again and again until given function is removed.

1. $T(n) = \begin{cases} T(n-1) + n & \text{if } n > 1 \\ 1 & \text{if } n = 1 \end{cases}$ ↣ $O(n^2)$

2. $T(n) = \begin{cases} T(n-1) * n & \text{if } n > 1 \\ 1 & \text{if } n = 1 \end{cases}$ ↣ $O(n^n)$

3. $T(n) = \begin{cases} T(n/2) + c & \text{if } n > 1 \\ 1 & \text{if } n = 1 \end{cases}$ ↣ $O(\log_2 n)$

4. $T(n) = \begin{cases} T(n/2) + n & \text{if } n > 1 \\ 1 & \text{if } n = 1 \end{cases} \rightarrowtail O(n)$

5. $T(n) = \begin{cases} T(n-1) + \log n & \text{if } n > 1 \\ 1 & \text{if } n = 1 \end{cases} \rightarrowtail O(n \log n)$

Recursive Tree Method

If $T(n)$ can be divided into parts then recursive tree method is used.

1. $T(n) = \begin{cases} C & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{otherwise} \end{cases} \rightarrowtail O(n \log_2 n)$
2. $T(n) = T(n/3) + T(2n/3) + cn \rightarrowtail O(n \log n)$
3. $T(n) = 3T(n/4) + cn^2 \rightarrowtail O(n^2)$

Master Method

Let $f(n)$ be a function and $T(n)$ be defined on the non-negative integers by recurrence relation: $T(n) = aT(n/b) + f(n)$ where $a \geq 1$ and $b > 1$. Then $T(n)$ can be bounded asymptotically as follows:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$ then $T(n) = \Theta(n^{\log_b a})$
2. If $f(n) = \Theta\left(n^{\log_b a} \log^k n\right)$ $k =$ constant $\forall k \geq 0$ then

   $T(n) = \Theta\left(n^{\log_b a} \cdot \log^{k+1} n\right)$

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

Extended Master Theorem:

Recurrence: $T(n) = aT(n/b) + \Theta(n^k \log^p n)$

where $a \geq 1$, $b > 1$, $k \geq 0$ and p is a real number then,

(1) If $a > b^k$ then $T(n) = Q(n^{\log_b a})$

(2) If $a = b^k$

if $p > -1$, then $T(n) = \Theta(n^{\log_b a} \log^{p+1} n)$

if $p = -1$, then $T(n) = \Theta(n^{\log_b a} \log \log n)$

if $p < -1$, then $T(n) = \Theta(n^{\log_b a})$

(3) If $a < b^k$

if $p \geq 0$, then $T(n) = \Theta(n^k \log^p n)$

if $p < 0$, then $T(n) = O(n^k)$

Muster Theorem for substract and conquer recurrences:

Let $T(n)$ be a function defined on possible n:

$$T(n) = \begin{cases} aT(n-b)+f(n), & \text{if } n > 1 \\ c, & n \le 1 \end{cases}$$

for some constant c, $a > 0$, $b > 0$, $d \ge 0$ and function $f(n) = O(n^d)$.

(1) $T(n) = O(n^d)$, if $a < 1$.

(2) $T(n) = O(n^{d+1})$, if $a = 1$.

(3) $T(n) = O(n^d a^{n/b})$, if $a > 1$.

■■■

Insertion is better than merge for small input

2 Divide-and-Counquer

Divide and Conquer (DAC):

A divide and conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.

1. Divide and conquer algorithms are naturally adapted for execution in multi-processor machines, especially shared-memory systems where the communication of data between processors does not need to be planned in advance, because distinct sub-problems can be executed on different processors.
2. Divide-and-conquer algorithms naturally tend to make efficient use of memory caches.
3. Divide-and-conquer algorithms are naturally implemented as recursive procedures.

```
DAC (P)
{
    if (P is small) return solution (P);
    else
    {
        k = Divide (P);
        return (combine (DAC (P1), DAC (P2),...DAC (Pk)));
    }
}
```

Divide and Conquer Algorithms

Maximum Minimum

Find Maximum and minimum element of an array using minimum number of comparisons.

- **Recurrence Relation:** $T(n) = \begin{cases} 0; & \text{if } n = 1 \\ 1; & \text{if } n = 2 \\ T(n/2) + T(n/2) + 1; & \text{if } n > 2 \end{cases}$

  Time complexity with DAC = $O(n)$

  (*i*) Using linear search = $O(n)$

(ii) **Using Tournament method:** Time complexity = $O(n)$
Number of Comparison: When n is power of 2 then
$T(n) = 2T(n/2) + 2 = (3n/2) - 2$ else more than $(3n/2) - 2$.

(iii) **Compare in Pairs:** Time complexity = $O(n)$
Number of comparison
(a) When n is even = $(3n/2) - 2$, (b) Else n is odd = $(3n-1)/2$.

Merge Sort

Comparison based sorting.
Stable sorting algorithm but **outplace**.

Recurrence Relation: $T(n) = \begin{cases} c \text{ if } n = 1 \\ T(n/2) + T(n/2) + cn \text{ if } n > 1 \end{cases}$

Time complexity: **(i) Worst case** = $O(n\log n)$, **(ii) Best case** = $O(n\log n)$,
(iii) **Average case** = $O(n\log n)$.
Space complexity: $S(n) = 2n + \log n + c$; **Worst case** = $O(n)$

Quick Sort

Comparison based sorting.
2 to 3 times faster than merge and heap sort.
Not stable sorting algorithm but **inplace**.
Choosing pivot is most important factor.
As Start element $T(n) = O(n^2)$, As End element $T(n) = O(n^2)$
As Middle element $T(n) = O(n^2)$, As Median element $T(n) = O(n^2)$
As Median of Median $T(n) = O(n \log n)$
Time complexity:

(i) **Best case:** each partition splits array into two halves then
$T(n) = T(n/2) + T(n/2) + n = O(n \log n)$

(ii) **Worst case:** each partition gives unbalanced splits we get
$T(n) = T(n-k) + T(k-1) + cn = O(n^2)$

(iii) **Average case:** In the average case, we donot know where the split happens for this reason we take all the possible value of split locations.

$$T(n) = \frac{1}{n}\sum_{i=1}^{n} T(i-1) + T(n-i) + n + 1 = O(n \log n)$$

Space complexity: $S(n) = 2n = n + \log n = O(n)$.
Randomized quick sort choose pivot from random places make worst case $O(n \log n)$ but for array with same element worst case $O(n^2)$.

Matrix Multiplication

- Using DAC: $T(n) = \begin{cases} O(1), & \text{for } n = 1 \\ 8T(n/2) + O(n^2), & \text{for } n > 1 \end{cases}$

  Time complexity = $O(n^3)$

- Strassen's Matrix Multiplication:

$$T(n) = \begin{cases} O(1) & , \text{ for } n = 1 \\ 7T(n/2) + an^2 & , \text{ for } n > 1 \end{cases}$$

  Time complexity = $O(n^{2.81})$ by Strassen's.
  Time complexity = $O(n^{2.37})$ by Coppersmith and Winograd.

- Karatsuba algorithm for fast multiplication of two n-digit numbers required exactly $n^{\log_2 3}$ **(when n is a power of 2)** using Divide and Conquer.

Power of an Element

- $T(n) = \begin{cases} 1 & , \text{ if } n = 0 \\ a & , \text{ if } n = 1 \\ T(n/2) + c, & \text{otherwise} \end{cases}$

  Time complexity = $O(\log_2 n)$

Binary Search based on decrease and conquer approach

- Recurrence Relation: $T(n) = \begin{cases} 1 \text{ if } n = 1 \\ c + T(n/2) \text{ if } n > 2 \end{cases}$
- Best case = $O(1)$
- Average case = worst case = $O(\log_2 n)$

Closest Pair of Points

The problem is to find the closest pair of points in a set of points in x-y plane.

- **Without Divide and Conquer:** The problem can be solved in $O(n^2)$ time by calculating distances of every pair of points and comparing the distances to find the minimum.
- **With Divide and Conquer:** The Divide and Conquer algorithm solves the problem in $O(n\log_2 n)$ time.

■■■

Greedy Approach

3

Greedy Algorithms

A greedy algorithm always makes the choice that seems to be the best at that moment. i.e. locally-optimal choice in the hope that it will lead to a globally-optimal solution. A Greedy algorithm makes greedy choices at each step to ensure that the objective function is optimized.

A problem must comprise two components for a greedy algorithm to work:

- It has **optimal substructures**. The optimal solution for the problem contains optimal solutions to the sub-problems.
- It has a **greedy property**, if you make a choice that seems the best at the moment and solve the remaining sub-problems later, you still reach an optimal solution. You will never have to reconsider your earlier choices.

Job Sequencing with Deadline

Given an array of jobs where every job has a deadline and associated profit if the job is finished before the deadline. It is also given that every job takes single unit of time, so the minimum possible deadline for any job is 1. Our task is to maximize total profit if only one job can be scheduled at a time. **Steps to solve problem:**

(*i*) Sort the given list of *n* jobs in ascending order of profits

(*ii*) Find maximum deadline in the given array of *n*-deadlines and take the array of maximum deadlines size.

(*iii*) For every slot *i* apply linear search to find a job which repeat this step for every slot.

Time complexity = $O(n^2)$

Activity Selection Problem

You are given *n* activities with their start and finish times. Select the maximum number of activities that can be performed by a single person, assuming that a person can only work on a single activity at a time. **Steps to solve problem:**

(*i*) Sort the activities according to their finishing time.

(*ii*) Select the first activity from the sorted array and print it.

(*iii*) For sorted array check if the start time of this activity is greater than or equal to the finish time of previously selected activity then select this activity and print it.

Time Complexity:

(a) It takes $O(n \log n)$ time if input activities may not be sorted.

(b) It takes $O(n)$ time when it is given that input activities are always sorted.

Huffman Coding

Assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters. The most frequent character gets the smallest code and the least frequent character gets the largest code. **Steps to solve problem:**

1. Build a Huffman Tree (using Min Heap is used as a priority queue) from input characters.
   (*i*) Initially, the least frequent character is at root.
   (*ii*) Extract two nodes with the minimum frequency from the min heap.
   (*iii*) Create a new internal node with frequency equal to the sum of the two nodes frequencies.
   (*iv*) Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.
2. Traverse the Huffman Tree and assign codes to characters.

- Has property that no code is prefix of any other code
  Time complexity: $O(n \log n)$
- **The number of bits per message:**
  Σ(frequency of external node 'i') × (Number of bits required for q_i)
- **Weigthed external path length** = $\Sigma q_i d_i$ (d_i = distance from root to external nodes 'i').

Fractional Knapsack Problem

Steps to solve problem:

(*i*) for ($i = 1$; $i \le n$; i++)
$a[i] = P_i/w_i$ **(Profit to weight ratio for each item)**

(*ii*) Arrange **array a** in ascending order

(*iii*) Take one by one object from a and keep in Knapsack untill Knapsack becomes full.

Time complexity = $O(n \log n) + O(n) = O(n \log n)$

Kruskal's Minimum Spanning Tree Algorithm

Steps of Kruskal's Algorithm:

(*i*) Sort all the edges from low weight to high

(*ii*) Take the edge with the lowest weight and add it to the spanning tree. If adding the edge created a cycle, then reject this edge.

(*iii*) Keep adding edges until we reach all vertices.

Time Complexity: O(E log E) or O(E log V).

- It starts with edge (min cost edge) and intermediate may results either tree or forest.

Prim's Minimum Spanning Tree

The idea behind Prim's algorithm is simple, a spanning tree means all vertices must be connected.

Steps of Prim's Algorithm:

(*i*) Initialize the minimum spanning tree with a vertex chosen at random.

(*ii*) Find all the edges that connect the tree to new vertices, find the minimum and add it to the tree.

(*iii*) Keep repeating step 2 until we get a minimum spanning tree.

- Always a connected graph.

Single Source Shortest Path (Dijkstra's Algorithm)

- **Time complexity:** O((V + E) log V) using binary min heap.
- Drawback of Dijkstra's Algorithm: It will not give shortest path for some vertices if graph contain negative weight cycle.
- Time complexity of Dijkstra's and Prim's algorithm using various data structure:
  - (*i*) **Using Binary Heap:** O((V + E) log V)
  - (*ii*) **Fibonacci heap:** O((V logV + E))
  - (*iii*) **Binomial Heap:** O((V + E) log V)
  - (*iv*) **Array:** $O(V^2 + E)$

Bellman Ford Algorithm

- It finds shortest path from source to every vertex, if the graph doesn't contain negative weight cycle.
- If graph contain negative weight cycle, it don't compute shortest path from source to all other vertices but it will report saying "negative weight cycle exists".

  Time complexity = O(VE) when dense graph $E = V^2$ and for sparse graph E = V.

■■■

Dynamic Programming

4

Dynamic Programming Algorithms

Longest Common Subsequence

Given two sequences, find the length of longest subsequence present in both of them. A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous.

- **Recurrence Relation:**

$$LCS(i, j) = \begin{cases} 0; & \text{if } i = 0 \text{ (or) } j = 0 \\ 1 + LCS(i-1, j-1); & \text{if } x[i-1] = y[j-1] \\ \max[LCS(i-1, j), LCS(i, j-1)]; & \text{if } x[i] \neq y[j] \end{cases}$$

Time complexity: by **Brute force** = $O(2^m)$, by **Dynamic programming** = $O(m \times n)$.

Space complexity = $O(mn)$; where m and n are length of two given sequences.

0/1 Knapsack Problem

Given weights and values of n items, select items to put items in a Knapsack of capacity W to maximise the total value in the Knapsack. You cannot break an item, either pick the complete item, or don't pick it (0 to 1 property).

- $$0/1\ KS(M, N) = \begin{cases} 0; & \text{if } M = 0 \text{ or } N = 0 \\ 0/1\ KS(M, N-1); & \text{if } w[n] > M \\ \max \begin{cases} 0/1\ KS(M - W[n], N-1) + P[n] \\ 0/1\ KS(M, N-1) \end{cases}; & \text{otherwise} \end{cases}$$

Time complexity = $O(MN)$: If M value is very large then it behaves like an exponential and **NP-complete problem.**

Travelling Salesperson Problem

Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point. In other words find the minimum cost Hamiltonian cycle.

$$TSP(A, R) = \begin{cases} C(A,S) \text{ if } R = \phi \\ \min\{(C[A,K] + TSP(K, R-K)) \forall K \in R\} \end{cases}$$

Time:
Time complexity: without dynamic programming = $O(n^n)$, **with** dynamic programming = $O(2^n \cdot n^2)$
Space complexity: without dynamic programming = $O(n^2)$, **with** dynamic programming = $O(n^n)$.
It is one of the NP Hard problem.

Matrix Chain Multiplication

Given a sequence of matrices, find the most efficient order to multiply these matrices together in order to minimise the number of multiplications.

$$MCM = \begin{cases} 0 & \text{if } i = j \\ \min\left\{ mcm(i, K) + mcm(K+1, j) + Pi \times Pj \times Pk \right\} & \text{where } i \le K < j \text{ or } i \le K \le j-1 \end{cases}$$

Number of ways to multiply matrix: $T(n) = \sum_{i=1}^{n-1} T(i).T(n-i)$

Time complexity: without dynamic programing = $O(n^n)$, **with** dynamic programing = $O(n^3)$.
Space complexity: without dynamic programing = $O(n)$, **with** dynamic programing = $O(n^2)$.

Sum of Subset Problem

Find if there is a subset of the given set, sum of whose elements is equal to given sum.

- **Recurrence Relation:**

$$SoS(M, N, S) = \begin{cases} \text{return}(S); & M = 0 \\ \text{return}(-1); & N = 0 \\ SoS(M, N-1, S); & \text{if } w[N] > M \\ \text{Min}\left\{ SoS(M - w[N], N-1, S \cup w[N]) \right\}; \text{otherwise} \\ \qquad \left\{ SoS(M, N-1, S) \right\} \end{cases}$$

Time complexity by Brute force = $O(MN)$

Floyd-Warshall's: All Pair Shortest Path

For finding shortest paths between all pairs of vertices in a weighted graph with positive or negative edge weights (but with no negative cycles).

- $A^k(i, j)$ = the min cost required to go from i to j by considering all intermediate vertices are numbered not greater than k.

$$A^0(i, j) = C(i, j)$$

$$A^k(i, j) = \min\left\{A^{k-1}(i, j), A^{k-1}(i, k) + A^{k-1}(k, j)\right\}$$

Time complexity = $O(n^3)$ with heap = $O(n^2 \log n)$.

- Warshall's algorithm will not work for negative edge cycle.

Optimal Binary Search Tree

Given a sorted array keys[0.. $n-1$] of search keys and an array frequency[0.. $n-1$] of frequency counts, where frequency[i] is the number of searches to keys[i]. Construct a binary search tree of all keys such that the total cost of all the searches is as small as possible.

- **It is lexically ordered tree:**

$$\text{cost}(i, j) = \min_{i<k\le j} \{\text{cost}(i, k-1) + \text{cost}(k, j) + w(i, j)\}$$

$$w(i, j) = p(j) + q(i) + w(i, j-1)$$
$$w(i, i) = q(i) \text{ and cost } (i, i) = 0$$

$$\text{cost} = \underbrace{\sum_{i}^{n} p_i \times \text{level } a_i}_{\text{(successful)}} + \underbrace{\sum_{i}^{n+1} q_i \,(\text{level } E_i - 1)}_{\text{(unsuccessful)}}$$

Time complexity is $O(n^3)$.

Multistage Graph

A Multistage graph is a directed graph in which the nodes can be divided into a set of stages such that all edges are from a stage to next stage only.

Task is to find shortest path from source to destination using the special property of multistage graph.

Shortcut: Travels back from destination to source with every time min cost selection.

$$MSG \quad \underset{\substack{\downarrow \\ \text{stage,}}}{(S_i,} \quad \underset{\substack{\downarrow \\ \text{vertex}}}{V_j)} = \begin{cases} 0 \text{ if } s_i = F \,\&\&\, V_j = 0 \\ \min \begin{cases} ((V_a, K) + MSG(s_i + s, K)) \\ \forall K \in s_i + 1 \,\&\&\, (V_j, K) \in E \end{cases} \end{cases}$$

Time complexity: without dynamic programing = $O(2^n)$, **with** dynamic programing = $O(V + E)$.

Space complexity: without dynamic programing = $O(v^2)$, **with** dynamic programing = $O(v^2)$.

■■■

Graph Traversals, Hashing and Sorting

5

Graph traversals

Depth First Search

1. **Preorder traversal** or postorder or inorder of ordered tree [$O(n + e)$: Average case]
   [$O(n^2)$: Worst case adjacency matrix].
2. Stack and backtracking technique.
3. Computes the number of paths between two vertices.
4. Computing a cycle and to find articulation points.
5. Biconnected components and strong components.
6. Euler paths and number of connected components.
7. Whether the graph is connected or not connected.

Breadth First Search

1. Level by level traversal of ordered tree [$O(n + e)$]
2. Queue and greedy technique.
3. Used for Topological sort and dijkstra's Algorithm
4. Prim's Algorithm.
5. Whether the graph is connected or not connected.
6. Number of connected components.
7. Transitive closure of a graph.
8. Computing a cycle.

Hashing

Hashing is a searching technique in which the searching is done in constant O(1). It is based on indexing mechanism. It uses a function called HASH FUNCTION.

- **Components in hashing:** Hash table, Hash functions, Collisions and Collision Resolution Techniques.
- Load factor = $\frac{\text{Number of elements in hash table}}{\text{Hash table size}}$

Direct address table: In direct address table key is the address without any manipulation. Even though number of keys are very less but one of key may contain 64 bits then size of hash table should be $2^{64} - 1$. The range of keys determines the size of the direct address table.

- **Types of Hash Functions:**
  (*i*) Division modulo method:
  $H(X) = X\% \ m$, where m = Hash table size and X = Key.
  (*ii*) Digit extraction method (Truncation method)
  (*iii*) Mid Square method
  (*iv*) Folding Method: Fold shifting method and Fold boundary
- **Collision Resolution Techniques:**
  (*i*) Open addressing
  (*ii*) Chaining.

Collision Resolution Techniques

Collision will take place if a new element is getting mapped where an element is already present.

- **Open Addressing:**
  (*i*) Linear Probing: If there is a collision at location '*L*' then look for empty location successively.
  Disadvantage:
  Primary clustering: The trend is for long sequence of preoccupied positions still become longer, primarily at one place.
  (*ii*) Quadratic Probing: Hash function = $h+i^2$ [Quadratic in nature]
  Disadvantage: Secondary clustering.
  (a) The maximum number of comparisons performed for successful search = size of largest cluster + 1
  (b) In Quadratic probing, all buckets are not compared because of quadratic nature.
  (*iii*) Random Probing: RNG = Random Number Generator.
  No number should be repeated within random space.

$$Y_{new} = Y_{old} + C \bmod m$$

where Y = Seed value
C = Constant
m = Hash table size

Disadvantage: Random clustering.

(*iv*) *Double Hashing/Rehashing:* '*C*' is changed in random probing in order to get new sequence.

Advantage: Clustering is reduced.

Disadvantage: Clustering is not avoided.

Note:

- Load Factor (α) = n/m

  n = number of elements present in hash table

  m = Hash table size
- If α is less, collisions are less
- $1 - \alpha$ = % of free space
- In open addressing scheme $0 \le \alpha \le 1$

- **Chaining:**
  (*i*) Chaining uses linked implementation for hashing.
  (*ii*) Number of elements mapped to a location = length of the linked list.
  (*iii*) Complexity of deleting an element in the chaining = deletion of a node in single linked list.
  (*iv*) The maximum number of comparisons performed for successful search in chaining = size of largest linked list.
  (*v*) There is no overflow problem.
  (*vi*) Load factor $\alpha = n/m$, where $\alpha \ge 0$.
- Average number of probes (comparisons) in successful search

Successful Seach	Unsuccessful Seach
Linear probing: $\frac{1}{2}+\frac{1}{2(1-\alpha)}$	$\frac{1}{2}+\frac{1}{2(1-\alpha)^2}$
Chaining: $(\alpha+1)/2$	$1+\alpha/2$

Searching and Sorting

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quick sort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n^2)$	$O(\log(n))$
Merge sort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(n)$
Time sort	$\Omega(n)$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(n)$
Heap sort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(1)$
Bubble sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Insertion sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Selection sort	$\Omega(n^2)$	$\Omega(n^2)$	$O(n^2)$	$O(1)$
Tree sort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n^2)$	$O(n)$
Shell sort	$\Omega(n \log(n))$	$\Theta(n \log(n))^2)$	$O(n \log(n))^2)$	$O(1)$
Bucket sort	$\Omega(n + k)$	$\Theta(n + k)$	$O(n^2)$	$O(n)$
Radix sort	$\Omega(nk)$	$\Theta(nk)$	$O(nk)$	$O(n + k)$
Counting sort	$\Omega(n + k)$	$\Theta(n + k)$	$O(n + k)$	$O(k)$
Cube sort	$\Omega(n)$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(n)$

■■■

Complexity Classes: P, NP, NPH and NPC

6

Complexity Theory

Complexity theory is mainly used to recognize whether the problem that is decidable is easy or hard. This can be calculated by means of time complexity and space complexity.

Complexity Classes

P-space Problem

The language in P-space represents the polynomial time and space complexity of the language.

NP-space Problem:

The language in NP space represents the non polynomial space complexity of the language.

P-problem

P is the class of problems that can be solved by deterministic algorithms in a time that is polynomially related to the size of the respective problem instance.

It consists of all languages accepted by some deterministic turing machine that runs in polynomial amount of time, as a function of input length.

Examples: 1-SAT, 2-SAT, Unary partition, Shortest path, 2-colourability, Enter cycle, Equivalence of DFA, Min cut, Shortest cycle in a graph, Sorting.

NP-problem

NP is the class of problems that can be solved by nondeterministic algorithms in a time that is polynomially related to the size of the respective problem instance.

It consists of all languages that are accepted by non deterministic turing machines with a polynomial bound on the time taken along any sequence of non deterministic choices.

Examples: Traveling sales person problem and Sub graph isomorphism.

NP-Hard Problem

If there is a language X such that every language Y in NP can be polynomially reducible to X and we cannot prove that X is in NP then X is said to be NP-Hard problem.

A problem X is NPH iff $\forall Y \in NP, Y \leq_p X$.

Examples: Turing machine halting problem.

NP-Complete Problem

If there is a language X such that every language Y in NP can be polynomially reducible to X and X is in NP then X is said to be NP-Hard problem. If NP hard problem is present in NP then it is called NP-complete problem.

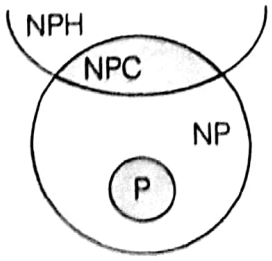
A problem X is NPC iff $\forall Y \in NP, Y \leq_p X, X \in NP$

Examples: 3-SAT problem, Traveling sales man problem, Hamilton circuit problem, Vertex cover problem, Independent set problem, Chromatic number problem, Coloring problem, Subgraph isomorphism problem, Edge cove problem, Knapsack problem, Max cut.

Reduction

$P_1 \leq P_2$: Problem P_1 is reducible to a problem P_2. It means problem P_2 is atleast as hard as problem P_1.

- If $P_1 \leq P_2$ and P_1 is undecidable then P_2 is also undecidable.
- If $P_1 \leq P_2$ and P_2 is decidable then P_1 is also decidable.
- If $P_1 \leq P_2$ and P_2 is recursive then P_1 is also recursive.
- If $P_1 \leq P_2$ and P_2 is recursive enumerable then P_1 is also recursive enumerable.
- If $P_1 \leq P_2$ and P_2 is P-problem then P_1 is also P-problem.
- If $P_1 \leq P_2$ and P_2 is NP-problem then P_1 is also NP-problem.
- If NP-complete is in P then P = NP.



■■■