

# CS4999 – Lorem Ipsum

1. Algorithm Analysis and Data Structures .....	1
1.1. Complexity Theory .....	1
1.2. Sorting Algorithms .....	2
1.3. Graph Theory .....	3
1.4. Dynamic Programming .....	4
1.5. Conclusion .....	5
2. Index .....	6

## 1. Algorithm Analysis and Data Structures

### 1.1. Complexity Theory

#### 1.1.1. Big O Notation

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed do eiusmod tempor incididunt labore et dolore magna aliqua. The **time complexity** can be expressed as:

$$O(n) = \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$$

Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Using **amortized analysis** we can determine the **average cost of operations**.

#### Key Properties:

- Transitivity: If  $f(n) = O(g(n))$  and  $g(n) = O(h(n))$ , then  $f(n) = O(h(n))$
- Sum rule:  $O(f(n)) + O(g(n)) = O(\max(f(n), g(n)))$
- Product rule:  $O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n))$

#### Complexity Classes

Pellentesque habitant morbi tristique senesque et netus et malesuada fames ac turpis egestas. Common complexity classes include:

- P: Problems solvable in polynomial time
- NP: Problems verifiable in polynomial time
- NP-Complete: Hardest problems in NP

#### 1.1.2. Binary Search Implementation

Duis aute irure dolor in **reprehenderit** in voluptate velit esse cillum dolore eu fugiat nulla pariatur. This algorithm uses the **divide and conquer** strategy:

```
1 def binary_search(arr, target):
2     left, right = 0, len(arr) - 1
3
4     while left <= right:
5         mid = (left + right) // 2
6
7         if arr[mid] == target:
8             return mid
9         elif arr[mid] < target:
```

```
10 |         left = mid + 1
11 |     else:
12 |         right = mid - 1
13 |
14 |     return -1
```

### ⚠ Time Complexity Analysis

Mauris blandit aliquet elit, at hendrerit urna semper vel. Binary search achieves  $O(\log n)$  time complexity by eliminating half the search space in each iteration. Curabitur aliquet quam id dui posuere blandit.

## 1.2. Sorting Algorithms

### 1.2.1. QuickSort Analysis

Lorem ipsum dolor sit amet, the **average case time complexity** is  $O(n \log n)$ , sed consectetur adipiscing elit. The recurrence relation demonstrates the **divide and conquer** approach:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

Vestibulum ante ipsum primis in faucibus orci luctus:

```
1 void quickSort(int arr[], int low, int high) {
2     if (low < high) {
3         int pi = partition(arr, low, high);
4
5         quickSort(arr, low, pi - 1);
6         quickSort(arr, pi + 1, high);
7     }
8 }
9
10 int partition(int arr[], int low, int high) {
11     int pivot = arr[high];
12     int i = low - 1;
13
14     for (int j = low; j < high; j++) {
15         if (arr[j] < pivot) {
16             i++;
17             swap(arr[i], arr[j]);
18         }
19     }
20     swap(arr[i + 1], arr[high]);
21     return i + 1;
22 }
```

### ☰ Master Theorem Application

Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia curae. For recurrences of the form  $T(n) = aT\left(\frac{n}{b}\right) + f(n)$ :

1. If  $f(n) = O(n^{\log_b a - \varepsilon})$  for some  $\varepsilon > 0$ , then  $T(n) = \Theta(n^{\log_b a})$
2. If  $f(n) = \Theta(n^{\log_b a})$ , then  $T(n) = \Theta(n^{\log_b a} \log n)$
3. If  $f(n) = \Omega(n^{\log_b a + \varepsilon})$  for some  $\varepsilon > 0$ , then  $T(n) = \Theta(f(n))$

Proin eget tortor risus, donec sollicitudin **molestie malesuada**.

## 1.3. Graph Theory

### 1.3.1. Dijkstra's Algorithm

Sed porttitor lectus nibh, cras ultricies ligula **sed magna dictum porta**. We can represent the graph using an **adjacency matrix**:

```

1 function dijkstra(graph, start) {
2   const distances = {};
3   const visited = new Set();
4   const pq = new PriorityQueue();
5
6   for (let node in graph) {
7     distances[node] = Infinity;
8   }
9   distances[start] = 0;
10  pq.enqueue(start, 0);
11
12  while (!pq.isEmpty()) {
13    const current = pq.dequeue();
14
15    if (visited.has(current)) continue;
16    visited.add(current);
17
18    for (let neighbor in graph[current]) {
19      const distance = distances[current] + graph[current][neighbor];
20      if (distance < distances[neighbor]) {
21        distances[neighbor] = distance;
22        pq.enqueue(neighbor, distance);
23      }
24    }
25  }
26
27  return distances;
28 }
```

#### Aside 1.1: Historical Context

Vivamus magna justo, lacinia eget consectetur sed, convallis at tellus. Edsger Dijkstra developed this algorithm in 1956. Quisque velit nisi, pretium ut lacinia in, elementum id enim.

### 1.3.2. Minimum Spanning Trees

Praesent sapien massa, convallis a pellentesque nec, egestas non nisi. **Kruskal's algorithm** complexity runs in **polynomial time**:

$$T(n) = O(E \log V)$$

Where  $E$  represents edges and  $V$  represents vertices. Nulla porttitor accumsan tincidunt. The **greedy approach** is employed here.

#### Greedy Algorithms

Donec rutrum congue leo eget malesuada. Both Kruskal's and Prim's algorithms use the greedy approach:

- At each step, make the locally optimal choice
- For MST: Always select the minimum weight edge that doesn't create a cycle
- Greedy choice leads to globally optimal solution

Curabitur non nulla sit amet nisl tempus convallis quis ac lectus.

## 1.4. Dynamic Programming

### 1.4.1. Longest Common Subsequence

Vivamus magna justo, lacinia eget consectetur sed, convallis at tellus. The DP table construction uses **memoization**:

```

1 def lcs(X, Y):
2     m, n = len(X), len(Y)
3     dp = [[0] * (n + 1) for _ in range(m + 1)]
4
5     for i in range(1, m + 1):
6         for j in range(1, n + 1):
7             if X[i-1] == Y[j-1]:
8                 dp[i][j] = dp[i-1][j-1] + 1
9             else:
10                dp[i][j] = max(dp[i-1][j], dp[i][j-1])
11
12    return dp[m][n]
```

Nulla porttitor accumsan tincidunt. Space complexity can be **optimized** to  $O(\min(m, n))$  using rolling arrays. This demonstrates **optimal substructure**.

#### Optimal Substructure

Vestibulum ac diam sit amet quam vehicula elementum sed sit amet dui. A problem exhibits optimal substructure if an optimal solution can be constructed from optimal solutions of its subproblems. Pellentesque in ipsum id orci porta dapibus.

### 1.4.2. Knapsack Problem

Donec sollicitudin molestie malesuada. The recurrence relation shows **optimal substructure**:

$$K(i, w) = \max(K(i-1, w), v_i + K(i-1, w - w_i))$$

Proin eget tortor risus, where  $v_i$  is value and  $w_i$  is weight of item  $i$ .

### 0/1 Knapsack Implementation

Lorem ipsum dolor sit amet, consectetur adipiscing elit:

```
1 def knapsack(weights, values, capacity):
2     n = len(weights)
3     dp = [[0] * (capacity + 1) for _ in range(n + 1)]
4
5     for i in range(1, n + 1):
6         for w in range(capacity + 1):
7             if weights[i-1] ≤ w:
8                 dp[i][w] = max(values[i-1] + dp[i-1][w-weights[i-1]],
9                 ||||| dp[i-1][w])
10            else:
11                dp[i][w] = dp[i-1][w]
12
13     return dp[n][capacity]
```

Sed porttitor lectus nibh, time complexity is  $O(nW)$  where  $W$  is capacity.

## 1.5. Conclusion

Sed porttitor lectus nibh. **Cras ultricies ligula** sed magna dictum porta. Quisque velit nisi, pretium ut lacinia in, elementum id enim. Donec rutrum congue leo eget malesuada.

## 2. Index

**adjacency matrix:** Quisque velit nisi pretium ut lacinia in elementum. A square matrix used to represent a finite graph with entries indicating edge presence. [3](#)

**amortized analysis:** Proin eget tortor risus donec sollicitudin molestie. A method for analyzing the average performance of a sequence of operations over time. [1](#)

**divide and conquer:** Vivamus magna justo lacinia eget consectetur sed. An algorithm design paradigm that recursively breaks down a problem into subproblems. [1](#), [2](#)

**greedy approach:** Vestibulum ac diam sit amet quam vehicula elementum. A strategy that makes the locally optimal choice at each step with the hope of finding a global optimum. [4](#)

**memoization:** Pellentesque habitant morbi tristique senectus et netus. An optimization technique that stores the results of expensive function calls. [4](#)

**optimal substructure:** Curabitur non nulla sit amet nisl tempus convallis. A property where an optimal solution contains optimal solutions to its subproblems. [4](#), [4](#)

**polynomial time:** Donec rutrum congue leo eget malesuada. An algorithm runs in polynomial time if its running time is bounded by a polynomial function of the input size. [4](#)