[CS203] Design and Analysis of Algorithms

Course Instructor: Dr. Dibyendu Roy — Autumn 2023-24
Scribed by: Your name (Roll Number) — Lecture (Weak #)

1 QuickSort Example

Let's illustrate the QuickSort algorithm with a sample array. We want to sort the following array in ascending order:

$$A = [6, 10, 13, 5, 8, 3, 2, 11]$$

1.1 Initial Array

The initial array is:

$$A = [6, 10, 13, 5, 8, 3, 2, 11]$$

1.2 Select Pivot

We'll choose the last element, which is 11, as the pivot:

$$A = [6, 10, 13, 5, 8, 3, 2, \mathbf{11}]$$

1.3 Partition the Array

Now, we'll partition the array. The pivot is 11. The partitioning process starts with i at -1. We compare each element $A[j]$ with the pivot, and if $A[j] \leq$ pivot, we increment i and swap $A[i]$ with $A[j]$. The array is rearranged as follows:

$$A = [6, 10, 5, 8, 3, 2, 13, 11]$$

The pivot element (11) is now in its correct position. Elements on the left are less than or equal to the pivot, and elements on the right are greater.

1.4 Recursion

We will now apply the QuickSort algorithm to the subarrays on the left and right of the pivot:

1.4.1 Left Subarray

The left subarray is:

$$[6, 10, 5, 8, 3, 2]$$

We'll apply the QuickSort algorithm to this subarray.

1.4.2 Right Subarray

The right subarray is:

[13]

The right subarray contains only one element, so it's considered sorted.

We'll continue applying the QuickSort algorithm recursively to sort the subarrays until the entire array is sorted.

Best-Case Analysis of QuickSort

To analyze the best-case scenario for QuickSort, let's consider the recurrence relation for the algorithm:

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

This recurrence relation represents the best-case scenario where the pivot chosen at each step happens to be the median, leading to well-balanced partitions.

Master Theorem

We can apply the Master Theorem to analyze the recurrence relation:

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

In this case, we have $a = 2$, $b = 2$, and $f(n) = \Theta(n)$. Comparing $f(n)$ to $n^{\log_b(a)}$, we have:

$$\Theta(n) \text{ vs. } n^{\log_2(2)} = n$$

Since $f(n)$ and $n^{\log_b(a)}$ are of the same order, we are in **Case 2** of the Master Theorem.

Case 2 of the Master Theorem

For **Case 2**, if $f(n)$ is $\Theta(n^{\log_b(a)})$, then the time complexity of the algorithm is:

$$T(n) = \Theta(n^{\log_b(a)} \log n)$$

In our case, $a = 2$ and $b = 2$, so:

$$T(n) = \Theta(n^{\log_2(2)} \log n) = \Theta(n \log n)$$

Therefore, in the best-case scenario, the QuickSort algorithm has a time complexity of $\Theta(n \log n)$.

This means that when QuickSort consistently selects the median as the pivot element, it achieves its best-case time complexity.

Worst-Case Analysis of QuickSort

In this section, we'll analyze the worst-case scenario for the QuickSort algorithm, where the pivot chosen at each step results in the most unbalanced partitions.

Recurrence Relation

The recurrence relation for QuickSort in the worst-case scenario is given by:

$$T(n) = T(1) + (n-1)\Theta(1) + \Theta(n)(n-1)$$

Now, let's expand the recurrence step by step:

$$T(n) = T(1) + (n-1) + \Theta(n^2) - \Theta(n)$$

The term $T(1)$ represents the time taken for subproblems of size 1, which is a constant, so it can be ignored in the big-O analysis. Now, we can focus on the remaining terms:

$$T(n) = \Theta(n^2) - \Theta(n)$$

The dominant term in this expression is $\Theta(n^2)$, and the $\Theta(n)$ term can be considered as a lower-order term in the big-O analysis.

Big-O Analysis

In the big-O notation, we provide an upper bound on the growth rate of a function. In this case, we have:

$$T(n) = \Theta(n^2) - \Theta(n)$$

The worst-case time complexity is dominated by the quadratic term $\Theta(n^2)$. Therefore, in the worst-case scenario, the time complexity of QuickSort is $T(n) = \Theta(n^2)$.

Comparison with Average and Best Case

It's important to note that the worst-case time complexity ($\Theta(n^2)$) is different from the average-case time complexity ($\Theta(n \log n)$) and the best-case time complexity ($\Theta(n \log n)$) of QuickSort.

In the average case, QuickSort performs significantly better due to randomized pivot selection and generally achieves $\Theta(n \log n)$. However, in the worst-case scenario, the time complexity can degrade to $\Theta(n^2)$.

Practical Considerations

While the worst-case time complexity of QuickSort is less favorable, it's essential to consider practical implementations and real-world data distributions. QuickSort's average-case performance often makes it a preferred choice for sorting algorithms due to its efficiency and low constant factors. In practice, the worst-case scenario is rarely encountered.

Why Dynamic Programming?

Dynamic programming (DP) is a powerful technique in algorithm design and optimization. It is often used to solve problems where the solution can be efficiently found by breaking it down into smaller overlapping subproblems. A classic example that illustrates the need for dynamic programming is the computation of Fibonacci numbers.

Let's consider the Fibonacci sequence:

The Fibonacci sequence is defined as follows:

$$F(0) = 0$$
$$F(1) = 1$$
$$F(n) = F(n-1) + F(n-2) \quad \text{for } n > 1$$

To compute Fibonacci numbers, we can use a recursive function:

```
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n-1) + fibonacci(n-2)
```

However, this recursive solution is highly inefficient. When you try to compute, for example, fibonacci(5), the function ends up computing the same Fibonacci numbers multiple times. For example, fibonacci(4) is computed twice, and fibonacci(3) is computed three times, leading to a lot of redundant work.

Dynamic programming provides a way to avoid this redundant computation. Instead of recomputing the same Fibonacci numbers, we can store the results in an array (or a memoization table) as we compute them and reuse those values. This approach significantly improves the efficiency of the algorithm.

In this dynamic programming approach, we create an array `fib` to store the Fibonacci numbers as we compute them. We start from the base cases ($F(0)$ and $F(1)$) and then use a loop to compute the Fibonacci numbers up to n. This way, we only compute each Fibonacci number once.

Rod Cutting Problem

The Rod Cutting problem is a classic optimization problem in computer science. The goal of this problem is to determine the best way to cut a given rod of a certain length into smaller pieces to maximize the total value or profit.

Problem Statement:

You are given a rod of length n inches, and there is a price list that tells you the price for selling pieces of the rod of different lengths. The price list provides the value or revenue you can obtain for each possible piece's length. Your goal is to find the optimal way to cut the rod to maximize the total revenue.

Let's consider an example to understand the problem better. Suppose we have a rod of length 10 inches, and we have the following price list:

Length (in inches)	1	2	3	4	5	6	7	8	9	10
Price (in dollars)	1	5	8	9	10	17	17	20	24	30

In this example, we want to find the optimal way to cut the 10-inch rod to maximize the total revenue. Each piece of the rod we cut has a certain price associated with its length, as specified in the price list. Our task is to determine the lengths at which we should cut the rod to obtain the maximum total revenue.

The optimal solution for this example would be to cut the rod into one piece of length 10 inches. The total revenue for this cutting strategy would be \$30, which is the maximum possible revenue for this rod.

The Rod Cutting problem can be solved using dynamic programming techniques, specifically by finding the optimal way to cut the rod to maximize the total value.

Rod Length	1	2	3	4	5	6	7	8	9	10
Maximum Value	1	5	8	10	13	17	18	22	25	30
Cut Sequence	1	2	3	2+2	2+3	6	7	2+6	3+6	10
Computed Value (rn)	1	5	8	10	13	17	18	22	25	30

$$r_n = \max(P_n, r_1 + r_{n-1}, \ldots, r_i + r_{n-i})$$

The exhaustive approach, which explores all possible combinations for rod cutting, suffers from exponential time complexity due to the extensive number of redundant calculations.

Algorithm 1: Cut-Rod (Top-Down)

Input: Price list p, Rod length n
Output: Maximum revenue q

```
if n = 0 then
    return 0;
q ← −∞;
for i ← 1 to n do
    q ← max(q, p[i] + Cut-Rod(p, n − i));
return q;
```

Time Complexity Analysis

Let's analyze the time complexity of the `Cut-Rod` algorithm using the top-down approach.

The recurrence relation for the time complexity is as follows:

$$T(n) = T(n-1) + T(n-2) + \ldots + T(0)$$

We can observe that this analysis uses weak induction when relating $T(n)$ to $T(n-1)$ and strong induction when relating $T(n)$ to $T(j)$.

Let's calculate the values of $T(1)$, $T(2)$, and so on:

$$\begin{aligned} T(1) &= T(0) = 1 \\ T(2) &= T(1) + T(0) = 2 \\ &\vdots \\ T(n) &= T(n-1) + T(n-2) + \ldots + T(0) = 2^n \end{aligned}$$

Therefore, the time complexity of the `Cut-Rod` algorithm using the top-down approach is $T(n) = 2^n$.

Heap-Increase-Key

The 'Heap-Increase-Key' operation is used to increase the key of a particular element in the heap and maintain the max-heap property.

Here's how the 'Heap-Increase-Key' operation works:

Algorithm 2: Heap-Increase-Key

```
0: procedure HEAP-INCREASE-KEY(A, i, key) if key < A[i] then
0:
   error "New key is smaller"
0:
0:    A[i] = key while i > 1 and A[parent[i]] < A[i] do
0:
   EXCHANGE(A[i], A[parent[i]])
0:    i = parent[i]
0:
0:    =0
```

1. If the new key is smaller than the current key of element $A[i]$, it raises an error because the operation should only increase the key.

2. It sets the new key in element $A[i]$.

3. The algorithm then checks the max-heap property by comparing the key of $A[i]$ with its parent's key ($A[\text{parent}[i]]$). If the parent's key is smaller, it swaps the two elements. This process continues up the tree until the max-heap property is satisfied.

The time complexity of 'Heap-Increase-Key' is $O(\log n)$ because the height of the heap is $\log n$, and in the worst case, we may need to traverse the entire height of the heap.

This operation is crucial for maintaining the max-heap property and is used in various algorithms like Heap Sort and Priority Queues.

Algorithm 3: Memorized-Cut-Rod(P, n)

```
0: r[0, 1, ..., n] ← an array of size n + 1 for i ← 0 to n do
0:
   r[i] ← −∞
0:
0: return MEMORIZED-CUT-ROD-AUX(P, n, r) =0
```

Algorithm 4: Memorized-Cut-Rod-Aux(P, n, r)

```
1 if r[n] ≥ 0 then-
  0:
2return r[n]
0: if n = 0 then
0:
   q ← 0 else
0:
   q ← −∞ for i ← 1 to n do
0:
   q ← max(q, p[i] + Memorized-Cut-Rod-Aux(P, n − i, r))
0:
0:
0: r[n] ← q
0: return q =0
```

Algorithm 5: Bottom-Up-Cut-Rod(P, n)

$r[0, 1, \ldots, n] \leftarrow$ an array of size $n + 1$
$r[0] \leftarrow 0$ **for** $j \leftarrow 1$ *to* n **do**

$q \leftarrow -\infty$ **for** $i \leftarrow 1$ *to* j **do**

$q \leftarrow \max(q, p[i] + r[j - i])$

$r[j] \leftarrow q$

return $r[n]$ =0

Matrix Chain Multiplication - Bottom-Up Approach

Given:

- A sequence of matrices A, where $A = A_1 A_2 A_3 \ldots A_n$
- $p(n)$ is the number of multiplications required for matrix A with dimensions $A_1 A_2 \ldots A_n$
- $p(1) = 0$ (no multiplication required for a single matrix)
- The cost $C(i, j)$ is the minimum number of multiplications required to compute the product of matrices from A_i to A_j

The goal is to find $C(1, n)$, the optimal way to parenthesize the entire sequence of matrices.
Mathematical Formulation:
The cost $C(i, j)$ of multiplying matrices from A_i to A_j can be defined recursively as follows:

$$C(i,j) = \begin{cases} 0, & \text{if } i = j, \\ \min_{i \le k < j} \left(C(i,k) + C(k+1,j) + p_{i-1} \cdot p_k \cdot p_j \right), & \text{if } i < j. \end{cases}$$

The base case is when $i = j$, in which case there is no multiplication required. The main recurrence formula expresses the cost $C(i, j)$ in terms of subproblems by considering all possible split points k within the range from i to $j - 1$ and finding the split point that minimizes the total cost.

We compute $C(1, n)$ to find the minimum number of multiplications needed to compute the product of all matrices $A_1 A_2 \ldots A_n$.

Using this mathematical formulation, we can compute $C(1, n)$ efficiently to find the optimal way to parenthesize the sequence of matrices to minimize the number of multiplications.