FEDERAL STATE AUTONOMOUS EDUCATIONAL INSTITUTION
OF HIGHER EDUCATION
ITMO UNIVERSITY

Report
on the practical task No. 8
“Practical analysis of advanced algorithms”

Performed by
Zakhar Pinaev
J4132c
Accepted by
Dr Petr Chunaev

St. Petersburg
2021

Goal

Practical analysis of advanced algorithms.

Problems and methods

I. Choose two algorithms (interesting to you and not considered in the course) from the Thomas H. Cormen Charles E. Leiserson Ronald L. Rivest Clifford Stein Introduction to Algorithms Third Edition, 2009.

II. Analyze the chosen algorithms in terms of time and space complexity, design technique used, etc. Implement the algorithms and produce several experiments. Analyze the results.

Brief theoretical part

An Algorithm is a sequence of steps that describe how a problem can be solved. Every computer program that ends with a result is basically based on an Algorithm. Algorithms, however, are not just confined for use in computer programs; these can also be used to solve mathematical problems and on many matters of day-to-day life. There are many types of Algorithms, but the fundamental types of Algorithms are:

1. Recursive Algorithm
   This is one of the most interesting Algorithms as it calls itself with a smaller value as inputs which it gets after solving for the current inputs. In more simpler words, It's an Algorithm that calls itself repeatedly until the problem is solved. Problems such as the Tower of Hanoi or DFS of a Graph can be easily solved by using these Algorithms.
2. Divide and Conquer Algorithm
   This is another effective way of solving many problems. In Divide and Conquer algorithms, divide the algorithm into two parts; the first parts divide the problem on hand into smaller subproblems of the same type. Then, in the second part, these smaller problems are solved and then added together (combined) to produce the problem's final solution. Merge sorting, and quick sorting can be done with divide and conquer algorithms.
3. Dynamic Programming Algorithm
   These algorithms work by remembering the results of the past run and using them to find new results. In other words, a dynamic programming algorithm solves complex problems by breaking them into multiple simple subproblems and then it solves each of them once and then stores them for future use. Fibonacci sequence is a good example for Dynamic Programming algorithms
4. Greedy Algorithm
   These algorithms are used for solving optimization problems. In this algorithm, we find a locally optimum solution (without any regard for any consequence in future) and hope to find the optimal solution at the global level. The method does not guarantee that we will be able to find an optimal solution. Huffman Coding and Dijkstra's algorithm are two prime examples where the Greedy algorithm is used.
5. Brute Force Algorithm
   This is one of the simplest algorithms in the concept. A brute force algorithm blindly iterates all possible solutions to search one or more than one solution that may solve a function.

6. Backtracking Algorithm
   Backtracking is a technique to find a solution to a problem in an incremental approach. It solves problems recursively and tries to solve a problem by solving one piece of the problem at a time. If one of the solutions fail, we remove it and backtrack to find another solution. In other words, a backtracking algorithm solves a subproblem, and if it fails to solve the problem, it undoes the last step and starts again to find the solution to the problem. N Queens problem is one good example to see Backtracking algorithm in action.

Results

As the first algorithm, the Divide and Conquer algorithm was chosen to solve the problem of finding the maximum sum of a subarray of the original array. The choice was made primarily in favor of learning something new. In addition, the choice was made on the basis that the algorithm is very general (not narrowly specialized), and therefore may be useful in the future in different areas. For example, now this algorithm finds application in such areas as genomic sequence analysis and computer vision. In particular, in computer vision, maximum-subarray algorithms are used on bitmap images to detect the brightest area in an image.

Thus, the Divide and Conquer algorithm was first implemented to solve the problem of finding the maximum sum of a subarray of the original array. In order to check the operation of the implemented algorithm, it was decided to feed it an array from the example in the book mentioned above. An example from the book is shown in Figure 1.

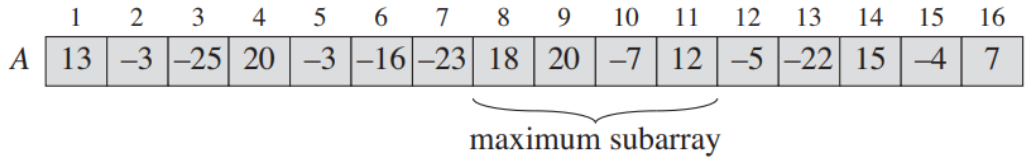

Figure 1 – an example of the original array given in the book and its corresponding maximum subarray, the sum of which is 43

To solve the problem, two algorithms were implemented - naive and divide and conquer. The array from the example was passed to the input of the implemented algorithms. The result of the programs is shown in Figure 2.

```
Maximum contiguous sum using D&C is  43
Maximum contiguous sum using Naive is  43
```

Figure 2 – the result of the implemented algorithm using Divide & Conquer and naïve approaches on the input array from the example in the book

As can be seen from Figures 1, 2, the correct value is obtained at the output of the algorithms, therefore, the implemented algorithms work correctly, which was proved by a series of experiments.

Further, the study of the time complexity of the implemented algorithms for solving the problem under consideration was carried out. In order to obtain the average execution time of the program, measurements for each step were carried out 6 times and then averaged. Figure 3 shows the dependence of the execution time of the algorithms depending on the size of the input data.

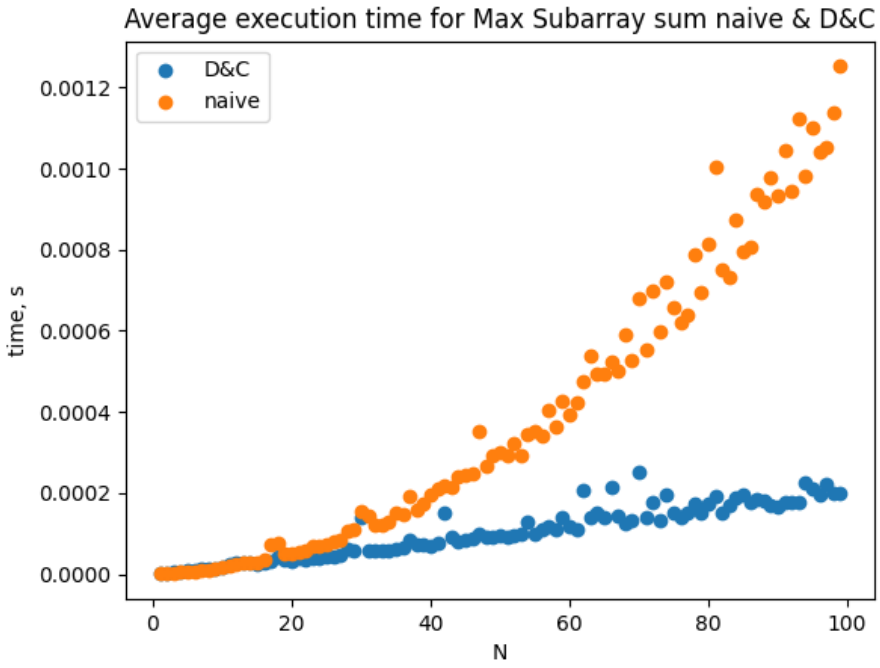

Figure 3 – graph of the dependence of the program execution time on the size of the input data

As can be seen from Figure 3, the naive approach has a complexity of $O(n^2)$, while the D&C approach has a complexity closer to O(nLogn). The conclusions drawn about the complexity of the implemented algorithms agree with the theory.

As the second considered algorithm, the algorithm for Cutting a rog was chosen (for the same reasons). The algorithm was again implemented using two approaches - naive and dynamic. As the second considered algorithm, the algorithm for Cutting a rog was chosen (for the same reasons). The algorithm was again implemented using two approaches - naive and dynamic. First of all, the work of each of the algorithms was checked - Figure 4 shows the output of programs that solve the Cutting a rog problem using the two mentioned methods.

```
Maximum Obtainable Value using Naive is 22
Maximum Obtainable Value using Dynamic is 22
```

Figure 4 – outputting programs for solving the Cutting a rog problem using a naive and dynamic method

During a series of tests, it was revealed that both algorithms produce equally correct solutions, which indicates the correctness of the implementation. Further, a study was also carried out on the time complexity of the algorithms depending on the size of the input data. And again, in order to get the average values of the program execution, the execution time at each step was measured 6 times, and then averaged. The research results are shown in Figure 5.

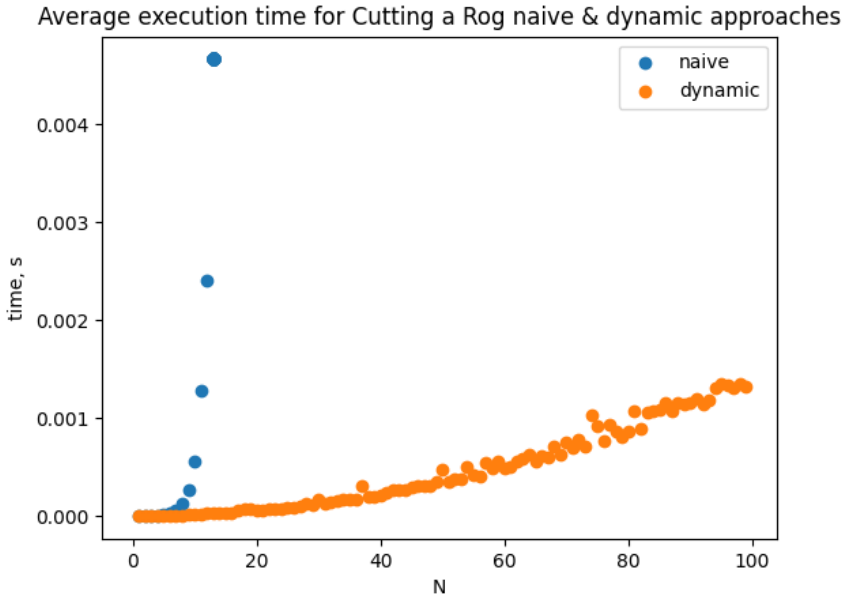

Figure 5 – graph of the dependence of the program execution time on the size of the input data

According to the theory, a naive approach to solving this problem has a time complexity of $O(2^{N-1})$, while a dynamic one is $O(N^2)$. Looking at Figure 5, one can be convinced that the obtained results of the estimate of the time complexity correspond to the theory, since the graph of the dependence of the elapsed time on the amount of input data for the naive approach grows dramatically faster than for the dynamic one.

Conclusion

Thus, in the course of the work, modern algorithms were considered and analyzed. Divide and conquer and the dynamic approach in solving the problems of finding the maximum sum of subarrays and Cutting a rog, respectively, were considered as the approaches under consideration. For each of the approaches, a series of experiments were carried out, which showed that the algorithms work correctly and give correct results in the output. And finally, for each of the problems, studies of the time complexity of the algorithms for its solution were carried out, and for clarity, the corresponding graphs were built. According to the results of the study, it was concluded that the results obtained in assessing the time complexity of the algorithms completely agree with the theory, which indicates the correctness of the implementation of the algorithms and the conduct of experiments.

Appendix

```
import math

import matplotlib.pyplot as plt
import numpy as np
import sys
import random
import timeit

INT_MIN = -32767
```

```
def maxCrossingSum(arr, l, m, h):
    sm = 0
    left_sum = -10000
    for i in range(m, l - 1, -1):
        sm = sm + arr[i]

        if (sm > left_sum):
            left_sum = sm
    sm = 0
    right_sum = -1000
    for i in range(m + 1, h + 1):
        sm = sm + arr[i]

        if (sm > right_sum):
            right_sum = sm
    return max(left_sum + right_sum, left_sum, right_sum)


def maxSubArraySum(arr, l, h):
    if (l == h):
        return arr[l]
    m = (l + h) // 2
    return max(maxSubArraySum(arr, l, m),
               maxSubArraySum(arr, m + 1, h),
               maxCrossingSum(arr, l, m, h))


def cutRod_naive(price, n):
    if (n <= 0):
        return 0
    max_val = -sys.maxsize - 1

    # Recursively cut the rod in different pieces
    # and compare different configurations
    for i in range(0, n):
        max_val = max(max_val, price[i] +
                      cutRod_naive(price, n - i - 1))
    return max_val


def cutRod_dynamic(price, n):
    val = [0 for x in range(n + 1)]
    val[0] = 0

    for i in range(1, n + 1):
        max_val = INT_MIN
        for j in range(i):
            max_val = max(max_val, price[j] + val[i - j - 1])
        val[i] = max_val
    return val[n]


def maxSubArray_naive(arr):
  maximum = -math.inf
  for i in range(0, len(arr)):
    sum=0
    for j in range(i, len(arr)):
      sum += arr[j]
      maximum = max(sum, maximum) #compare the resulting sum with the
existing maximum value
  return maximum
```

```
def main():

    # MAXIMUM SUBARRAY ALG

    # Driver Code
    arr = [13, -3, -25, 20, -3, -16, -23, 18, 20, -7, 12, -5, -22, 15, -4, 7]
    n = len(arr)

    max_sum = maxSubArraySum(arr, 0, n - 1)
    print("Maximum contiguous sum using D&C is ", max_sum)

    max_sum = maxSubArray_naive(arr)
    print("Maximum contiguous sum using Naive is ", max_sum)

    # CUTTING A ROD ALGORITM

    arr_2 = [1, 5, 8, 9, 10, 17, 17, 20]
    size = len(arr_2)
    print("Maximum Obtainable Value using Naive is", cutRod_naive(arr_2,
size))

    print("Maximum Obtainable Value using Dynamic is", cutRod_dynamic(arr_2,
size))

    N = list(range(1, 100))

    time_1 = [0] * len(N)
    time_2 = [0] * len(N)
    time_3 = [0] * len(N)
    time_4 = [0] * len(N)

    for n in N:
        v = [random.randrange(-50, 50, 1) for r in range(n)]
        v2 = [random.randrange(1, 100, 1) for r in range(n)]

        for k in range(6):

            start = timeit.default_timer()
            temp = maxSubArraySum(v, 0, n - 1)
            time_1[n - 1] = time_1[n - 1] + (timeit.default_timer() - start)
/ 6


            start = timeit.default_timer()
            temp = maxSubArray_naive(v)
            time_4[n - 1] = time_4[n - 1] + (timeit.default_timer() - start)
/ 6

            if n < 14:
                start = timeit.default_timer()
                temp = cutRod_naive(v2, n)
                time_2[n - 1] = time_2[n - 1] + (timeit.default_timer() -
start) / 6


            else:
                time_2[n - 1] = time_2[12]

            start = timeit.default_timer()
            temp = cutRod_dynamic(v2, n)
            time_3[n - 1] = time_3[n - 1] + (timeit.default_timer() - start)
/ 6
```

```
    # graph for max subarray alg
    plt.figure(1)
    plt.plot(N, time_1)
    plt.xlabel('N')
    plt.ylabel('time, s')
    plt.title('Average execution time for Max Subarr Sum using D&C alg')

    # graph for cutting a Rod problem Naive approach
    plt.figure(2)
    plt.plot(N[0:13], time_2[0:13])
    plt.xlabel('N')
    plt.ylabel('time, s')
    plt.title('Average execution time for Cutting a Rog using naive
approach')

    # graph for cutting a Rod problem Naive approach
    plt.figure(3)
    plt.plot(N, time_3)
    plt.xlabel('N')
    plt.ylabel('time, s')
    plt.title('Average execution time for Cutting a Rog using dynamic
approach')

    N2 = [0] * len(N)
    print(len(N), len(N2))
    for i in range(len(N)-1):
        if i < 13:
            N2[i] = N[i]
        else:
            N2[i] = N[12]

    # graph for cutting a Rod problem Naive approach
    plt.figure(4)
    #plt.plot(N, time_2, N, time_3)
    plt.scatter(N2, time_2)
    plt.scatter(N, time_3)
    plt.xlabel('N')
    plt.ylabel('time, s')
    plt.title('Average execution time for Cutting a Rog naive & dynamic
approaches')
    plt.legend(['naive', 'dynamic'], loc='best')

    # graph for cutting a Rod problem Naive approach
    plt.figure(5)
    # plt.plot(N, time_2, N, time_3)
    plt.scatter(N, time_1)
    plt.scatter(N, time_4)
    plt.xlabel('N')
    plt.ylabel('time, s')
    plt.title('Average execution time for Max Subarray sum naive & D&C')
    plt.legend(['D&C', 'naive'], loc='best')


    plt.show()

if __name__ == "__main__":
    main()
```