CSE2046 – Analysis of Algorithms
Modified Coin-collection Problem

Supervisor:
Assistant Prog. Ömer KORÇAK
Senior Lecturer
Computer Science Engineering Department, Marmara University
e-mail: omer.korcak@marmara.edu.tr

Supervisees:
Bilgehan NAL
Student ID: 150114038
Computer Science Engineering Department, Bachelor's Degree
e-mail: bilgehan.nal@marun.edu.tr

Oğuzhan BÖLÜKBAŞ
Student ID: 150114022
Computer Science Engineering Department, Bachelor's Degree
e-mail: oguzhanbolukbas@marun.edu.tr

Table of Contents

1. Abstract

This is the example of solving and simulating the wifi connection problem. We assume a system laptops can communicate between each other with their wifis and we tried to find out the shortest path of first laptop and other laptops.
This work is made for CSE 246 Lecture Marmara University.
Assignment page:
http://mimoza.marmara.edu.tr/~omer.korcak/courses/CSE246/HW3_Spring2017.pdf

2. Introdunction

In this problem we have n laptops and these laptops can send a message if it connects to other laptop. Firstly we tried to show connectivity map of laptops on a graph. And we tried to solve our problem with a shortest path algorithm. We use Dijkstra's algorithm for this example and we tested this algorithm for different inputs.

3. Theory

3.1.Divide and Conquer Algorithm

In computer science, divide and conquer (D&C) is an algorithm design paradigm based on multi-branched recursion. 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.
This divide and conquer technique is the basis of efficient algorithms for all kinds of problems, such as sorting (e.g., quicksort, merge sort), multiplying large numbers (e.g. the Karatsuba algorithm), finding the closest pair of points, syntactic analysis (e.g., top-down parsers), and computing the discrete Fourier transform (FFTs).
Understanding and designing D&C algorithms is a complex skill that requires a good understanding of the nature of the underlying problem to be solved. As when proving a theorem by induction, it is often necessary to replace the original problem with a more general or complicated problem in order to initialize the recursion, and there is no systematic method for finding the proper generalization. These D&C complications are seen when optimizing the calculation of a Fibonacci number with efficient double recursion.
The correctness of a divide and conquer algorithm is usually proved by mathematical induction, and its computational cost is often determined by solving recurrence relations.[1]

1. Advantages

a) Solving difficult problems

Divide and conquer is a powerful tool for solving conceptually difficult problems: all it requires is a way of breaking the problem into sub-problems, of solving the trivial cases and of combining sub-problems to the original problem. Similarly, divide and conquer only requires reducing the problem to a single smaller problem, such as the classic Tower of Hanoi puzzle, which reduces moving a tower of height n to moving a tower of height $n - 1$.

b) Algorithm efficiency

The divide-and-conquer paradigm often helps in the discovery of efficient algorithms. It was the key, for example, to Karatsuba's fast multiplication method, the quicksort and mergesort algorithms, the Strassen algorithm for matrix multiplication, and fast Fourier transforms.

In all these examples, the D&C approach led to an improvement in the asymptotic cost of the solution. For example, if (a) the base cases have constant-bounded size, the work of splitting the problem and combining the partial solutions is proportional to the problem's size n, and (b) there is a bounded number p of subproblems of size ~ n/p at each stage, then the cost of the divide-and-conquer algorithm will be $O(n \log pn)$.

c) Parallelism

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.

d) Memory access

Divide-and-conquer algorithms naturally tend to make efficient use of memory caches. The reason is that once a sub-problem is small enough, it and all its sub-problems can, in principle, be solved within the cache, without accessing the slower main memory. An algorithm designed to exploit the cache in this way is called cache-oblivious, because it does not contain the cache size as an explicit parameter.[2] Moreover, D&C algorithms can be designed for important algorithms (e.g., sorting, FFTs, and matrix multiplication) to be optimal cache-oblivious algorithms–they use the cache in a probably optimal way, in an asymptotic sense, regardless of the cache size. In contrast, the traditional approach to exploiting the cache is blocking, as in loop nest optimization, where the problem is explicitly divided into chunks of the appropriate size—this can also use the cache optimally, but only when the algorithm is tuned for the specific cache size(s) of a particular machine.

The same advantage exists with regards to other hierarchical storage systems, such as NUMA or virtual memory, as well as for multiple levels of cache: once a sub-problem is small enough, it can be solved within a given level of the hierarchy, without accessing the higher (slower) levels.

e) Roundoff control

In computations with rounded arithmetic, e.g. with floating point numbers, a divide-and-conquer algorithm may yield more accurate results than a superficially equivalent iterative method. For example, one can add N numbers either by a simple loop that adds each datum to a single variable, or by a D&C algorithm called pairwise summation that breaks the data set into two halves, recursively computes the sum of each half, and then adds the two sums. While the second method performs the same number of additions as the first, and pays the overhead of the recursive calls, it is usually more accurate.[3]

2. Implementation issues

f) Recursion

Divide-and-conquer algorithms are naturally implemented as recursive procedures. In that case, the partial sub-problems leading to the one currently being solved are automatically stored in the procedure call stack. A recursive function is a function that calls itself within its definition.

g) Explicit stack

Divide and conquer algorithms can also be implemented by a non-recursive program that stores the partial sub-problems in some explicit data structure, such as a stack, queue, or priority queue. This approach allows more freedom in the choice of the sub-problem that is to be solved next, a feature that is important in some applications — e.g. in breadth-first recursion and the branch and bound method for function optimization. This approach is also the standard solution in programming languages that do not provide support for recursive procedures.

h) Stack size

In recursive implementations of D&C algorithms, one must make sure that there is sufficient memory allocated for the recursion stack, otherwise the execution may fail because of stack overflow. Fortunately, D&C algorithms that are time-efficient often have relatively small recursion depth. For example, the quicksort algorithm can be implemented so that it never requires more than $\log_2 n$nested recursive calls to sort n items.

Stack overflow may be difficult to avoid when using recursive procedures, since many compilers assume that the recursion stack is a contiguous area of memory, and some allocate a fixed amount of space for it. Compilers may also save more information in the recursion stack than is strictly necessary, such as return address, unchanging parameters, and the internal variables of the procedure. Thus, the risk of stack overflow can be reduced by minimizing the parameters and internal variables of the recursive procedure, or by using an explicit stack structure.

i) Choosing the base cases

In any recursive algorithm, there is considerable freedom in the choice of the base cases, the small subproblems that are solved directly in order to terminate the recursion.

Choosing the smallest or simplest possible base cases is more elegant and usually leads to simpler programs, because there are fewer cases to consider and they are easier to solve. For example, an FFT algorithm could stop the recursion when the input is a single sample, and the quicksort list-sorting algorithm could stop when the input is the empty list; in both examples there is only one base case to consider, and it requires no processing.

On the other hand, efficiency often improves if the recursion is stopped at relatively large base cases, and these are solved non-recursively, resulting in a hybrid algorithm. This strategy avoids the overhead of recursive calls that do little or no work, and may also allow the use of specialized non-recursive algorithms that, for those base cases, are more efficient than explicit recursion. A general procedure for a simple hybrid recursive algorithm is short-circuiting the base case, also known as arm's-length recursion. In this case whether the next step will result in the base case is checked before the function call, avoiding an unnecessary function call. For example, in a tree, rather than recursing to a child node and then checking if it is null, checking null before

recursing; this avoids half the function calls in some algorithms on binary trees. Since a D&C algorithm eventually reduces each problem or sub-problem instance to a large number of base instances, these often dominate the overall cost of the algorithm, especially when the splitting/joining overhead is low. Note that these considerations do not depend on whether recursion is implemented by the compiler or by an explicit stack.

Thus, for example, many library implementations of quicksort will switch to a simple loop- based insertion sort (or similar) algorithm once the number of items to be sorted is sufficiently small. Note that, if the empty list were the only base case, sorting a list with n entries would entail maximally n quicksort calls that would do nothing but return immediately. Increasing the base cases to lists of size 2 or less will eliminate most of those do-nothing calls, and more generally a base case larger than 2 is typically used to reduce the fraction of time spent in function-call overhead or stack manipulation.

Alternatively, one can employ large base cases that still use a divide-and-conquer algorithm, but implement the algorithm for predetermined set of fixed sizes where the algorithm can be completely unrolled into code that has no recursion, loops, or conditionals (related to the technique of partial evaluation). For example, this approach is used in some efficient FFT implementations, where the base cases are unrolled implementations of divide-and-conquer FFT algorithms for a set of fixed sizes. Source code generation methods may be used to produce the large number of separate base cases desirable to implement this strategy efficiently.[4]

The generalized version of this idea is known as recursion "unrolling" or "coarsening" and various techniques have been proposed for automating the procedure of enlarging the base case.[5]

j) Sharing repeated subproblems

For some problems, the branched recursion may end up evaluating the same sub-problem many times over. In such cases it may be worth identifying and saving the solutions to these overlapping subproblems, a technique commonly known as memoization. Followed to the limit, it leads to bottom-up divide-and-conquer algorithms such as dynamic programming and chart parsing.

3.2.Dynamic Programming

In computer science, mathematics, management science, economics and bioinformatics, dynamic programming (also known as dynamic optimization) is a method for solving a complex problem by breaking it down into a collection of simpler subproblems, solving each of those subproblems just once, and storing their solutions. The next time the same subproblem occurs, instead of recomputing its solution, one simply looks up the previously computed solution, thereby saving computation time at the expense of a (hopefully) modest expenditure in storage space. (Each of the subproblem solutions is indexed in some way, typically based on the values of its input parameters, so as to facilitate its lookup.) The technique of storing solutions to subproblems instead of recomputing them is called "memoization".

Dynamic programming algorithms are often used for optimization. A dynamic programming algorithm will examine the previously solved subproblems and will combine their solutions to give the best solution for the given problem. In comparison, a greedy algorithm treats the solution as some sequence of steps and picks the locally optimal choice at each step. Using a greedy algorithm does not guarantee an optimal solution, because picking locally optimal choices may result in a bad global

solution, but it is often faster to calculate. Some greedy algorithms (such as Kruskal's or Prim's for minimum spanning trees) are however proven to lead to the optimal solution.

For example, in the coin change problem of finding the minimum number of coins of given denominations needed to make a given amount, a dynamic programming algorithm would find an optimal solution for each amount by first finding an optimal solution for each smaller amount and then using these solutions to construct an optimal solution for the larger amount. In contrast, a greedy algorithm might treat the solution as a sequence of coins, starting from the given amount and at each step subtracting the largest possible coin denomination that is less than the current remaining amount. If the coin denominations are 1,4,5,15,20 and the given amount is 23, this greedy algorithm gives a non-optimal solution of 20+1+1+1, while the optimal solution is 15+4+4.

When to use?

Dynamic programming is used to solve problems which have overlapping sub-problems.
When problem breaks down into recurring small dependent sub-problems.
When the solution can be recursively described in terms of solutions to sub-problems.

How it works?

Dynamic programming algorithm finds solutions to sub-problems and stores them in memory for later use. It is sometimes considered the opposite of recursion. Where a recursive solution starts at the top and breaks the problem down, solving all small problems until the complete problem is solved, a dynamic programming solution starts at the bottom, solving small problems and combining them to form an overall solution to the big problem. It is used to convert algorithm of complexity 2n to O(n3) or O(n2).

Coin Row Problem : There is a row of n coins whose values are some positive integers c1, c2, . . . , cn, not necessarily distinct. The goal is to pick up the maximum amount of money subject to the constraint that no two coins adjacent in the initial row can be picked up.
Or in other words, there is an integer array consisting positive numbers only. Find maximum possible sum of elements such that there are no 2 consecutive elements present in the sum.

For ex: Coins[] = {5, 22, 26, 15, 4, 3,11}
Max Sum = 22 + 15 + 11 = 48

To solve this problem using Dynamic Programming first we will have to define recurrence relation. Let F[n] is the array which will contain the maximum sum at n for any given n. The recurrence relation will be.

$F(n) = \max\{Coins[n] + F[n-2], F[n-1]\}$ for $n > 1$,
$F(0) = 0$, $F(1) = Coins[1]$.

This is very easy to understand. While calculating F[n] we are taking maximum of coins[n]+the previous of preceding element and the previous element.

For example F[2] = max{coins[0]+ F[2-2], F[2-1]} // No consecutive

The algorithm:

```
1  public class CoinRow {
2
3      public static int getMaximumAmount(int[] coins) {
4          assert coins.length > 0 : "no coins to select";
5
6          int[] C = new int[coins.length + 1];
7          for (int i = 0; i < coins.length; i++) {
8              C[i + 1] = coins[i];// make room at first place
9          }
10
11         int[] F = new int[coins.length + 1];
12         F[0] = 0;
13         F[1] = C[1];
14
15         for (int i = 2; i <= coins.length; i++) {
16             F[i] = max(C[i] + F[i - 2], F[i - 1]);
17         }
18         return F[coins.length];
19     }
20
21     private static Integer max(int i, int j) {
22         return i > j ? i : j;
23     }
24
25     public static void main(String[] args) {
26         int[] coins = { 5, 22, 26, 10, 4, 8};
27         System.out.println(getMaximumAmount(coins));
28     }
29 }
30 Output
31 -------
32 40
```

Lets trace the algorithm.

Index	0	1	2	3	4	5	6
C		5	22	26	10	4	8
F	0	5					

F [0] = 0, F [1] = c1 = 5

Index	0	1	2	3	4	5	6
C		5	22	26	10	4	8
F	0	5	22				

F [2] = max {22 + 0, 5} = 22

Index	0	1	2	3	4	5	6
C		5	22	26	10	4	8
F	0	5	22	31			

F [3] = max {26 + 5, 22} = 31

Index	0	1	2	3	4	5	6
C		5	22	26	10	4	8
F	0	5					

F [0] = 0, F [1] = c1 = 5

Index	0	1	2	3	4	5	6
C		5	22	26	10	4	8
F	0	5	22				

F [2] = max {22 + 0, 5} = 22

Index	0	1	2	3	4	5	6
C		5	22	26	10	4	8
F	0	5	22	31			

F [3] = max {26 + 5, 22} = 31

Index	0	1	2	3	4	5	6
C		5	22	26	10	4	8
F	0	5	22	31	32		

F [4] = max {10 + 22, 31} = 32

Index	0	1	2	3	4	5	6
C		5	22	26	10	4	8
F	0	5	22	31	32	35	

F [5] = max {4 + 31, 32} = 35

Index	0	1	2	3	4	5	6
C		5	22	26	10	4	8
F	0	5	22	31	32	35	***40***

F [6] = max {8 + 32, 35} = 40

F[6] will have the final value. [6]

Our problem: We have considered the following modified version of the coin-collecting robot problem. Each cell has an integer value between -5 and 5. If the value of a cell is positive, then the robot gains that amount of coins when it visits that cell. If the value of the cell is negative, then the robot pays that amount of coins if it visits that cell and if it has enough coin. But if it has not collected that amount of coins in the previous cells, then the robot cannot visit a cell with negative value. Also there are some cells on the board that are always inaccessible for the robot.
Again the robot starts at (1,1), it can only move towards right and down and it stops at (n,m). Following corresponds to an example board, where inaccessible cells are shown by X's and values of cells are shown by integers between -5 and 5:

0	-1	5	4	0	0
1	2	0	X	4	0
3	1	3	-1	X	5
0	0	-2	3	-1	5
X	X	X	0	1	1

4. Solving the problem

Dynamic Programming

An important part of given problems can be solved with the help of dynamic programming and we use a dynamic programming algorithm to solve this modified coin collecting problem.

0	1	-2	5	X	4
2	-2	2	4	X	0
3	1	0	-1	1	1
3	-1	-2	Z	2	5
X	X	-5	0	1	1

Basically our dynamic programming approach for this problem is determining the values of all cells and following the largest value of cell until reaching from start point to end point.

To determine the values of cells, we sum the value of the cell with maximum value of left and up cell. The aim of this process is: Our agent can move only right or down and the agent gain or lose some coins. Therefore if we process this process we can determine that how could we gain much more coin coming left or up way and learn largest value of the agent could take largest value on that cell. We take the coordinate of the cell in a tuple object.

In the final part, the agent choose a cell which has maximum value of right and down. Of course if the selected way is closed for the agent. The agent try the other way for the previous move.

Steps of the Algorithm:

1- Read the data from the input file.
2- Convert all values to an integer and keep the data in an array.
3- Determine the values(Best value to reach that cell from the start point) for each cell.
4- Follow and push the maximum value of the cell which has larger value to a stack. (The stack keeps cell with best way to reach end point.)
5- if the selected way is closed for the agent. The agent try the other way for the previous move helping with stack.

Time Complexity Of The Algorithm

Calculating the value of all cells for one time. Therefore time complexity is:

O(n*m)

Determining the value of each cell takes O(n*m) time. (Worst Case)

Recurrence relation for following the maximum value of the cell:

$T(i, j) = T(i-1, j) + 1$ or $T(i, j) = T(i, j-1) + 1$

Following the maximum value of the cell takes $\Theta(n + m)$ time.

Finally, this algorithm's time complexity is $O(n*m) + \Theta(n + m) = O(n*m)$. Therefore the execution time of this algorithm is changing proportionally with number of cell on the grid.

Divide and conquer

An important part of given problems can be solved with the help of divide and conquer techniques. We tested the divide and conquer algorithm for modified coin collecting problem.

When we applicate this algorithm to a problem. We are taking the problem to the sub pieces and solving the smallest parts of the problem then we are collecting them in one piece recursively.

Steps of the Algorithm:

1- Read the data from the input file.
2- Convert all values to an integer and keep the data in an array.
3- Define a function returning the results.
    a. Result: if the agent can take 2 cells(down and right) we return function with parameter of cell which has maximum value + value of that cell.

       Else if the agent can take one cell return function wşth parameter of that cell + value of that cell.
       Else if the agent at the end point, return value of that cell.
4- Returning value is maximum coin collected from the grid.

Time Complexity Of The Algorithm

In the worst case scenario the agent looks every time calls the function for either down or right.

$T(i*j) = T(i*j-i) + T(i*j-j) + 1$ (Worst case time complexity)

$T(0,0) = 1$ (This is the end point.)

If we write variables with one variable (let I equals to j and i*j -> n)

$T(n) = 2T(n-\sqrt{n}) + 1$

Square root of n is constant value for this reccurince relation.

$T(n-\sqrt{n}) = 2T(n-2\sqrt{n}) + 1$

$T(n) = 4\ T(n-2\sqrt{n}) + 3$

$T(n-2\sqrt{n}) = 2T(n-3\sqrt{n}) + 1$

$T(n) = 8T(n - 3\sqrt{n}) + 7$

$T(n) = 2^nT(0) + 2^n-1$

$O(n) = 2^{n+1}$

$O(i, j) = 2^{i*j+1}$

5. Conclusion

We tried to solve modified coin collecting problems in two ways. (Dynamic programming and divide and conqure). According to the real tests and theoretical calculations. Divide and conquer algorithm has more time complexity and when we applicate the divide and conquer, we figured out that divide and conquer way is using system stack a lot because, the recursive program calls the same function with same paramaters again and again. Therefore according to the our tests we can say that for this problem dynamic programming is more efficient.

6. References

[1]: https://en.wikipedia.org/wiki/Dynamic_programming
[2]: M. Frigo; C. E. Leiserson; H. Prokop (1999). "Cache-oblivious algorithms". Proc. 40th Symp. on the Foundations of Computer Science.
[3]: Nicholas J. Higham, "The accuracy of floating point summation", SIAM J. Scientific Computing 14 (4), 783–799 (1993).
[4]: Frigo, M.; Johnson, S. G. (February 2005). "The design and implementation of FFTW3"
[5]: Radu Rugina and Martin Rinard, "Recursion unrolling for divide and conquer programs," in Languages and Compilers for Parallel Computing, chapter 3, pp. 34–48. Lecture Notes in Computer Science vol. 2017 (Berlin: Springer, 2001).
[6]: http://www.basicsbehind.com/tag/dynamic-programming/