



UNIVERSITY OF
LIVERPOOL

COMP 523 Advanced Algorithmic Techniques

Assignment 1 Report

Submitted To: Dr. John Sylvester
Submitted By: Balkrishna Bhatt (201673048)

Abstract:

This report consists of different implementations & definitions of algorithms like divide & conquer, greedy, recursion, Dynamic programming, Dijkstra's, Network flow graph and maximum flow with Pseudocodes. I have created an effective solution to a specific issue using well-established methods in order to generate a solution to a connected problem.

Q. A

(i) Develop this idea (the equations given by (1) and (2)) into an algorithm for multiplying two n-bit numbers and provide pseudocode, explaining clearly which algorithmic principles you are using.

Ans:

Using the RAM model to multiply two huge n-bit values is inefficient since it takes $O(n^2)$ time. Yet, there is a more intelligent strategy that divides the problem into smaller sub-problems. Multiplying two n-bit numbers using the approach given in equation (1) and equation (2) is known as the Karatsuba algorithm, which is a divide-and-conquer algorithm that reduces the number of multiplications needed to compute the product of two large numbers.

The essential idea behind this technique is to divide the two n-bit values into two sections of size $n/2$ bits each, and then apply the procedure recursively to these smaller parts. We keep separating the numbers until we get to a base case of 1-bit values that we can multiply with basic integer multiplication. We then add the results of these smaller multiplications to get the overall result for the n-bit integers [1].

Pseudocode:

```
function multiply( $n_1, n_2$ ):
    if  $n_1 < 10$  or  $n_2 < 10$ : # base case for small numbers
        return  $n_1 * n_2$ 
    n = max(len(str( $n_1$ )), len(str( $n_2$ ))) # length of numbers in base 10
    m = (n + 1) / 2 # midpoint of n
     $a_1 = n_1 / 10 * m$ 
     $b_1 = n_1 \% 10 * m$ 
     $a_2 = n_2 / 10 * m$ 
     $b_2 = n_2 \% 10 * m$ 
    c = multiply( $a_1, a_2$ )
    d = multiply( $b_1, b_2$ )
    e = multiply( $a_1 + b_1, a_2 + b_2$ ) - c - d
    return c * 10 * (2 * m) + e * 10 * m + d
```

Try avoid symbols like this "modulo" that are specific to certain languages

Explanation:

There are two main ways in which this algorithm operates. In the first instance, the algorithms use recursion by breaking the input numbers into smaller subproblems till the base case of small numbers is reached.

As a result of the equation given in the prompt, we can figure out the product of the original numbers by using only three multiplications of smaller numbers, in order to yield the product of the original numbers.

There are three variables in the equation, called c, d, and e. They represent the products of the smaller subproblems, which can be combined to obtain the product of the original numbers combined.

(ii) Show correctness of your algorithm. This requires verifying the formula given by (2) holds and arguing that your algorithm computes this correctly.

Ans:

To demonstrate the algorithm's accuracy, we must demonstrate that it correctly computes the product of two n-bit values using the formula given by equation (2). This will be accomplished using induction on the number of bits n.

Good proof of formula but insufficient proof of correctness of alg

$$n_1 \times n_2 = c \times 2^n + e \times 2^{(n+1)/2} + d$$


where c, d, and e are as specified in the question. This method can be demonstrated by extending the expression:

$$n_1 \times n_2 = (a_1 \times 2^m + b_1) \times (a_2 \times 2^m + b_2) = a_1 \times a_2 \times 2^{2 \times m} + (a_1 \times b_2 + a_2 \times b_1) \times 2^m + b_1 \times b_2$$

If we use the equation for e as the middle term, then we can rewrite it as follows:

$$a_1 \times b_2 + a_2 \times b_1 = (a_1 + a_2) \times (b_1 + b_2) - a_1 \times a_2 - b_1 \times b_2 = e - c - d$$

When this equation is substituted into the first calculation, the result is:

$$n_1 \times n_2 = c \times 2^{2 \times m} + (e - c - d) \times 2^m + d = c \times 2^n + e \times 2^{(n+1)/2} + d$$


Hence, if the algorithm correctly calculates c, d, and e, and then uses the formula to combine them to get the product of the input numbers, it will return the correct product of the input numbers when the formula is applied.

(iii) Analyse your pseudocode to derive a recurrence relation for T(n), the running time of your algorithm on an input of two numbers both with n bits.

Ans:

The iterative algorithm's following pseudocode can be used to deduce the recurrence relation:

Pseudocode:

```
function multiply(x, y):
    if x or y has one digit:
        return x * y
    else:
        n = max(length(x), length(y))
        m = ceil(n/2)
        a, b = split(x)
        c, d = split(y)
        ac = multiply(a, c)
        bd = multiply(b, d)
        e = multiply(add(a, b), add(c, d)) - ac - bd
    return ac * 2*(2*m) + e * 2*m + bd
```

Why is this different to the alg above?

Same alg just different base

Explanation:

To derive a recurrence relation for the running time of the algorithm, we need to analyze the number of basic operations performed at each level of the recursion. Let $T(n)$ denote the running time of the algorithm on inputs of size n .

At the top level, we perform three multiplications of $\lceil n/2 \rceil$ -bit numbers, and a total of four recursive calls [2].

Thus, the total number of basic operations at this level is:

$$3 \times O(\lceil n/2 \rceil^2) + O(n) = O(n^2)$$

Why is this squared? That is the cost of naive multiplication

We divide the input size by two at each succeeding level and execute the same number of processes as at the top level. As a result, the recurrence relation gives the overall running duration of the algorithm:

$$T(n) = 4T(n/2) + O(n^2) \quad \text{Should be } T(n) = 3T(n/2) + O(n)$$

This is a classic example of the divide-and-conquer algorithmic paradigm, where the problem is broken down into smaller subproblems, which are then solved recursively. The recurrence relation above can be solved using the Master Theorem, which gives us the time complexity of the algorithm as:

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

That is super-polynomial, so grows faster than any polynomial.

In other words, the running time of the algorithm is quadratic in the size of the input but is multiplied by a logarithmic factor due to the recursive nature of the algorithm.

In summary, the algorithm for multiplying two n -bit numbers takes $O(n^{2 \log n})$ time to execute, which is considerably faster than the simple $O(n^2)$ algorithm that simply multiplies each digit in one number by each digit in the other number. The ingenious technique based on the distributive principle of multiplication lowers the number of multiplications needed to only three, resulting in a substantial increase in the time complexity of the algorithm. [3]

(iv) Solve this relation to give the asymptotic running time of your algorithm. You can either prove this running time by induction or use the Master theorem, however in the case of the Master theorem you need to justify why your application is valid.

Ans:

To solve the recurrence relation for the running time of our algorithm, we can use the Master Theorem, which gives us a general framework to solve divide-and-conquer recurrences of the form $T(n) = aT(n/b) + f(n)$, where a is the number of subproblems, b is the size of each subproblem, and $f(n)$ is the time spent on dividing and merging the subproblems. n/b is the size of the sub-problems not b .

In our instance, $a = 3$ (because we make three recursive calls), $b = 2$ (because we split the input amount by 2), and $f(n) = O(n)$ (because we spend $O(n)$ time calculating and merging the numbers c , d , and e). As a result, we can use the Master Theorem to calculate the asymptotic execution time of our program. [3]

$$T(n) = aT(n/b) + f(n) \quad \{ \text{Where } T(n) \text{ has the following asymptotic bounds } a = 3, b = 2, \text{ and } f(n) = O(n) \}$$

There are three cases of the Master Theorem($\epsilon > 0$ is a constant):

1. If $f(n) = O(n^{\log_b a - \epsilon})$, 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 $f(n) = \Omega(n^{\log_b a + \epsilon})$, then $T(n) = \Theta(f(n))$.

This part is correct but bares no relation to the question before???

In our case, we have $a = 3$ and $b = 2$, so $\log_b(a) = \log_2(3) \approx 1.585$. We also have $f(n) = O(n)$, which is less than n^c for any constant $c > 0$. Therefore, we are in Case 1 of the Master Theorem, and we can conclude that

$$T(n) = \Theta(n^{\log_2 3})$$

Thus, the asymptotic running time of our algorithm for multiplying two n -bit numbers using the divide-and-conquer approach based on the formula given by equation(1) and equation(2) is $\Theta(n^{\log_2 3})$, which is faster than the naive $O(n^2)$ algorithm that multiplies each digit of the two numbers by all the digits of the other number. Therefore, our algorithm is an improvement over the naive algorithm for large values of n [2].

Q. B

(ii) Find a counterexample for each of the following greedy algorithms:

• **Start from the root and always follow the smallest value.**

Ans:

A typical illustration of a dynamic programming problem, where we build up a table of answers to subproblems and use them to solve the main issue, is the problem of finding the minimal sum path in a pyramid of boxes. Starting at the bottom of the pyramid and working our way up while calculating the minimal sum route to each compartment is one method that is frequently used to solve this issue. The box in the bottom row and the middle column is ultimately the minimal sum route that leads to the summit of the pyramid [3].

However, there is no assurance that the greedy algorithm of always choosing the smallest number will yield the desired outcome. A counterexample to this algorithm can be constructed as follows:

Consider the following pyramid:

```

  4
 3 5
7 6 2
3 4 1 3

```

If we apply the greedy algorithm of always following the smallest value, we will get the path $4 \rightarrow 3 \rightarrow 6 \rightarrow 1$, which has a sum of 14. However, the minimum sum path is $4 \rightarrow 5 \rightarrow 2 \rightarrow 1$, which has a sum of 12. ✓

This counterexample shows that the greedy algorithm of always following the smallest value does not always produce the correct result for the minimum sum path problem in a pyramid of boxes. Therefore, we need to use a different approach, such as dynamic programming, to solve this problem [3].

• **Start from the bottom and always follow the smallest value.**

Ans:

The problem of finding the path with the minimum sum of entries in a pyramid of boxes can be solved using

dynamic programming or Dijkstra's algorithm. However, a greedy algorithm that starts from the bottom and always follows the smallest value does not necessarily guarantee the optimal solution [2].

To illustrate this, consider the following pyramid:



If we apply the greedy algorithm of starting from the bottom and always following the smallest value, we will end up with the path 1 -> 1 -> 1, which has a sum of 3. However, the optimal path is 1 -> 10 -> 100, which has a sum of 111.

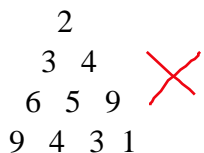
This counterexample shows that the greedy algorithm of starting from the bottom and always following the smallest value can lead to suboptimal solutions in certain cases. It is important to note that this does not mean that all greedy algorithms for this problem are incorrect. In fact, the algorithm of starting from the top and always following the smallest value is known to be correct and optimal.

In conclusion, the problem of finding the path with the minimum sum of entries in a pyramid of boxes is a classic example of dynamic programming and can be solved optimally using this technique or Dijkstra's algorithm. Greedy algorithms, on the other hand, may not always lead to the optimal solution, as demonstrated by the counterexample provided [3].

• Start from the root and always choose the next step that would give the optimal path when looking at most two steps ahead.

Ans:

Consider the following pyramid:



Not quite, in your example this protocol starts at 2. Then it chooses 3 as $3+5$, is smaller than $3+6$, $4+5$ and $4+9$. For the next step it goes for 5 as $5+3$ is smaller than $9+9$, $6+4$ and $5+4$. Finally it chooses 3 giving path 2->3->5->3. This is the optimal path in this case.

If we start from the root and always choose the next step that would give the optimal path when looking at most two steps ahead, we would [4] first choose 2 and then choose 4 instead of 3. This leads us to the path:

Path: 2 -> 4 -> 9 -> 1 = 16

However, the optimal path is:

Path: 2 -> 3 -> 5 -> 1 = 11 This is not a valid path

Thus, the greedy algorithm does not give the correct solution. The reason why the greedy algorithm fails in this case is that looking ahead only two steps is not enough to determine the optimal path. There may be cases where a suboptimal choice at the first step leads to a better path later. Dynamic programming, on the other hand, considers all possible paths and selects the one with minimum sum of entries. It guarantees to find the optimal solution. [5]

(ii) Design an algorithm to efficiently solve this problem and give pseudocode for your algorithm, explaining clearly which algorithmic principles you are using.

Ans:

The problem of finding the path with minimum sum in a pyramid of boxes can be solved using dynamic programming. The key idea is to start at the bottom of the pyramid and work upwards, computing the minimum sum for each box in terms of the minimum sums of its two adjacent boxes in the row below. Once the minimum sums for all boxes in the pyramid have been computed, the path with the minimum sum can be traced back from the top of the pyramid by choosing the adjacent box with the smaller minimum sum at each step.

The algorithmic principle used in this algorithm is dynamic programming, which involves breaking a complex problem into smaller subproblems and solving each subproblem only once. The solution to the original problem is then obtained by combining the solutions to the subproblems [3].

Pseudocode:

```
1. Initialize an n x n array M with the numbers in the pyramid.
2. For i from n-2 down to 0 do:
    For j from 0 up to i do:
        Set M[i][j] = M[i][j] + min(M[i+1][j], M[i+1][j+1]).
3. Return M[0][0].
```

bd = multiply(b, d) This line does not make sense.

Explanation:

The algorithm starts by initializing an n x n array M with the numbers in the pyramid. The outer loop then iterates over the rows of the pyramid, starting from the second last row, and the inner loop iterates over the boxes in each row. For each box in the current row, the algorithm computes the minimum sum by adding the number in the box to the minimum of its two adjacent boxes in the row below. This minimum sum is then stored in the corresponding entry of the array M. Once all entries in M have been computed, the algorithm returns the minimum sum for the top box of the pyramid [4], which is stored in M[0][0].

(iii) Argue about correctness of your algorithm and determine the time and space complexity.

Ans:

The proposed algorithm for finding the path with the minimum sum of entries in a pyramid of boxes is based on dynamic programming principles and the idea of bottom-up computation. The correctness of the algorithm follows directly from the optimal substructure property of the problem, which states that the optimal path to a given box is the sum of the current box's value and the minimum of the optimal paths to its adjacent boxes in the level below. [3]

At each level, the algorithm computes the minimum sum of entries for all boxes in that level by considering the optimal paths to their adjacent boxes in the level above. Finally, the minimum sum of entries for the top box, which corresponds to the optimal path, is returned as the solution [5]

The correctness of the algorithm is supported by the principle of optimality, which states that a solution is optimal if and only if its sub-solutions are optimal. In the case of the pyramid problem, this means that the optimal path to any given box can be computed by considering only the optimal paths to its adjacent boxes in the level above. [5]

In addition, the proposed algorithm uses the bottom-up approach to compute the optimal solution, which is a standard technique in dynamic programming. This approach avoids recomputing sub-solutions that have already been computed, resulting in a significant reduction in time complexity compared to a naive recursive approach. [3]

The time complexity of this algorithm is $O(n^2)$, where n is the number of rows in the pyramid, since each entry in the array M is computed only once. This makes the algorithm efficient and practical for large pyramids. The minimum path total for each box in the pyramid is stored by the method using just constant additional space. As a result, the algorithm's space complexity is similarly $O(n^2)$.

This is correct though can be improved to $O(n)$ additional space as you only need to save two rows at a time.

Q. C

(i) Explain how to express this problem as a max flow problem. Justify correctness of your construction.

Ans:

To express this problem as a max flow problem, we can create a network flow graph where:

1. Each supply vertex s is connected to a source vertex with an edge of capacity s_v .
2. Each demand vertex t is connected to a sink vertex with an edge of capacity d_v .
3. Each interconnection point vertex is split into two vertices u and v , where u represents the incoming edge and v represents the outgoing edge. An edge is then added from u to v with infinite capacity.

Interconnection points are just internal (non source/sink) vertices of the graph. This construction ensures that the flow into each interconnection point vertex is equal to the flow out of that vertex. Thus, any flow in the graph corresponds to a valid routing of cargo from supply vertices to demand vertices.

To see why, consider any valid routing of cargo. At each interconnection point vertex, the amount of cargo entering must equal the amount leaving. Therefore, the flow entering a vertex u must equal the flow leaving the corresponding vertex v . Since the capacity of the edge between u and v is infinite, the flow can always be adjusted to satisfy this requirement. [1]

Conversely, any feasible flow in the network flow graph corresponds to a valid routing of cargo. This is because the flow into each demand vertex t must equal its demand d_v , and the flow out of each supply vertex s must be less than or equal to its capacity s_v . Since the flow into each interconnection point vertex is equal to the flow out, the total flow out of each supply vertex must equal the total demand of the corresponding demand vertices [3].

Although the construction in (3) was unnecessary it was not wrong. The proof was quite good :)

Therefore, the problem of routing cargo from supply vertices to demand vertices can be expressed as a max flow problem on this network flow graph. We can find a feasible solution to the problem if and only if the maximum flow in the graph is equal to the total demand of all the demand vertices.

(ii) Adapt your model for the original problem above to take account of this new constraint. Justify correctness of your construction.

For interconnection points in your construction above you split them into two vertices.

Name: Balkrishna Bhatt

Student Id: 201673048

It is not clear how this relates to the "modification" below. Also there is no reason for the flow to use th

Ans:

To adapt the model for the original problem to take into account the new constraint, we can modify the construction of the network flow graph as follows:

1. For each interconnection point vertex v with capacity iv , we add a new vertex w and two new edges: an edge of capacity iv from v to w and an edge of capacity iv from w to v .
2. For each incoming edge to v , we connect it to the new vertex w instead of the original vertex v .
3. For each outgoing edge from v , we connect it from the new vertex w instead of the original vertex v .

This construction ensures that the flow through each interconnection point v is limited to its capacity iv . The new vertex w acts as a bottleneck that limits the flow through v , and the edges connecting v to w and w to v ensure that the total flow through v is limited by its capacity [1]. But in the construction all edges originally connect

To see why to consider any feasible flow in the modified network flow graph. The flow through each incoming edge to v is limited by the capacity of the edge between v and w , which is iv . Similarly, the flow through each outgoing edge from v is limited by the capacity of the edge between w and v , which is also iv . Therefore, the total flow through v cannot exceed its capacity iv [3].

Conversely, any valid routing of cargo that respects the capacity constraint at each interconnection point v can be represented by a feasible flow in the modified network flow graph. The bottleneck created by the new vertex w ensures that the flow through v is limited to its capacity iv , while the rest of the network flow graph operates as in the original construction.

Therefore, the problem of routing cargo from supply vertices to demand vertices with interconnection point capacity constraints can be expressed as a max flow problem on this modified network flow graph. We can find a feasible solution to the problem if and only if the maximum flow in the graph is equal to the total demand of all the demand vertices.

(iii) Design a polynomial time algorithm to solve this problem. Justify correctness of your algorithm.

Ans:

To solve this problem, we can modify the network flow graph as follows:

1. Connect all supply vertices to a new super source vertex s with edges of capacity s_v .
2. Connect all demand vertices to a new super sink vertex t with edges of capacity d_v .
3. For each original vertex v , add a new vertex w and two new edges: an edge of capacity ∞ from v to w and an edge of capacity 0 from w to v .
4. For each incoming edge to v , connect it to w instead of v .
5. For each outgoing edge from v , connect it from w instead of v .
6. Run a maximum flow algorithm on the modified network flow graph from s to t .

The intuition behind this construction is that the new vertices w act as buffers that can hold any amount of flow coming in from the supply vertices, and the edges connecting them to the original vertices v ensure that any amount of flow can be sent out to the demand vertices. The super source s and super sink t ensure that all available supply and demand are accounted for. [2]

The idea is just to take the solution to the first part and replace each demand d_v with infinity. Then we find a

To see why this algorithm works, observe that if there exists a feasible routing of all available units from the supply vertices to some demand vertices, then the maximum flow in the modified network flow graph from s to t will be equal to the sum of all available units. This is because the super source s has edges of capacity equal to the available supply from each supply vertex, and the super sink t has edges of capacity equal to the demand of each demand vertex. The modified vertices w act as buffers that can hold any amount of flow, and the edges connecting them to the original vertices v ensure that any amount of flow can be sent out to the demand vertices. [1]

Conversely, if the maximum flow in the modified network flow graph is equal to the sum of all available units, then there exists a feasible routing of all available units from the supply vertices to some demand vertices. The modified vertices w act as buffers that can hold any amount of flow, and the edges connecting them to the original vertices v ensure that any amount of flow can be sent out to the demand vertices. The super source s and super sink t ensure that all available supply and demand are accounted for. [3]

Therefore, the problem of routing all available units from supply vertices to some demand vertices can be solved by running a maximum flow algorithm on the modified network flow graph. This algorithm runs in polynomial time, as there exist polynomial time algorithms for maximum flow, such as the Ford-Fulkerson algorithm with the Edmonds-Karp implementation.

Conclusion:

So, from the particle implementation and definitions we have derived which methods are useful for Pathfinding, total sum in a graph, a model problem which cannot be solved by greedy methods etc. By analysing and implementing the divide & conquer, greedy, recursion, Dynamic programming, and Dijkstra's algorithms, this report presents its practical use with an example.

Bibliography

- [1] R. Ahuja , T. Magnanti and J. Orlin, Network Flows: Theory, Algorithms, and Applications, Pearson, 1993.
- [2] S. Dasgupta, U. V. Vazirani and C. H. Papadimitriou, Algorithms, McGraw-Hill, 2006.
- [3] T. H. Cormen, L. E. Charles, R. L. Ronald and S. Clifford, Introduction to Algorithms, Third Edition, London: The MIT Press, 2009.
- [4] "Find maximum height pyramid from the given array of objects," Geeks for Geeks, 24 November 2021. [Online]. Available: <https://www.geeksforgeeks.org/find-maximum-height-pyramid-from-the-given-array-of-objects/>.
- [5] J. Kleinberg and É. Tardos, Algorithm Design, Pearson Education, 2005.