# IDAA432C Assignment-7 (Group: 41)


Aditya Vallabh IIT2016517 | Kiran Velumuri ITM2016009 | Neil Syiemlieh IIT2016125 | Tauhid Alam BIM2015003



***Abstract***

**There are n engineers and n jobs. Each engineer can get only one job. The respective liking for each job according to each engineer is given. Suggest an allotment such that the average liking of all the jobs is maximized.**


## I. Introduction

In this problem, we are given the number of engineers and jobs - $n$. Then, we generate an $n$ x $n$ random matrix consisting of the respective liking for each of the $n$ jobs according to each engineer. The aim of the problem is to allot the given $n$ jobs to the $n$ engineers such that that the average liking of all the jobs is maximized. Our approach to solving this problem is to convert the given problem into a minimization problem and then solving the minimization problem.

Using the relevant test cases, we obtain the time complexity of the algorithms for each of best, average and worst cases. Then we plot the graphs for the above mentioned cases.

## II. Algorithm Design

### A. Conversion to minimization problem

The given problem is a slight variation of the famous **assignment problem** where we are supposed to minimize the overall cost. However in this question we are required to maximize the average liking of all jobs so we first convert the maximization problem to a minimization one by subtracting all the elements from the maximum element in the given matrix. This way we would be minimizing the 'disliking' towards each job.

This is illustrated in Fig. 1 and Fig. 2, where the numbers represent the engineers and the letters represent the jobs. The maximum element in the matrix is 41(at position [4][C]) and we subtract all other elements from 41 to convert into minimization problem.

| | A | B | C | D | E |
|---|---|---|---|---|---|
| **1** | 32 | 38 | 40 | 28 | 40 |
| **2** | 40 | 24 | 28 | 21 | 36 |
| **3** | 41 | 27 | 33 | 30 | 37 |
| **4** | 22 | 38 | 41 | 36 | 36 |
| **5** | 29 | 33 | 40 | 35 | 39 |

Fig. 1. Allocation matrix

| | A | B | C | D | E |
|---|---|---|---|---|---|
| **1** | 9 | 3 | 1 | 13 | 1 |
| **2** | 1 | 17 | 13 | 20 | 5 |
| **3** | 0 | 14 | 8 | 11 | 4 |
| **4** | 19 | 3 | 0 | 5 | 5 |
| **5** | 12 | 8 | 1 | 6 | 2 |

Fig. 2. Conversion into minimization problem

**Algorithm 1** convertToMin method

Input: $costMatrix, n$
**for** $i \leftarrow 0$ *to* $n$ **do**
  **for** $j \leftarrow 0$ *to* $n$ **do**
    $max \leftarrow$MAX(max, costMatrix[i][j])
**for** $i \leftarrow 0$ *to* $n$ **do**
  **for** $j \leftarrow 0$ *to* $n$ **do**
    $costMatrix[i][j] \leftarrow$max - $costMatrix[i][j]$
**return** max

### B. The Assignment Algorithm

The problem can be approached in a bottom-up manner with the help of **dynamic programming** and **bit-masking**. The mask is going to represent a binary number whose $i^{th}$ bit indicates whether the $i^{th}$ job has been assigned or not. The state of our $dp$ is going to be this $mask$ and $k$ where $k$ denotes the number of engineers who have been alloted a job. Now suppose we have the answer $dp[k, mask]$ we can assign a task $i$ to person $k$ if the $i^{th}$ bit is not set. Then
$dp[k+1, mask\&(1 \ll i)] \leftarrow MIN(dp[k+1, mask\&(1 \ll i)], dp[mask] + cost[k][i])$
However $k$ here is unecessary as we can get it by counting the number of set bits in $mask$. So our $dp$ reduces to a single state. Following is the implementation of the algorithm.

**Algorithm 2** Allot Method

```
Input: costMatrix, n, dp
for mask ← 0 to 2^n do
    dp[mask].val ← inf
dp[0].val ← 0
for mask ← 0 to 2^n do
    x ← countSetBits(mask)
    for j ← 0 to n do
        if mask & (1 ≪ j) == 0 then
            a ← dp[mask|(1 ≪ j)]
            b ← dp[mask] + costMatrix[x][j]
            idx ← mask|(1 ≪ j)
            if a ≤ b then
                dp[idx].val ← a
            else
                dp[idx].val ← b
                for i ← 0 to n do
                    dp[idx].allots[i] ← dp[mask].allots[i]
                dp[idx].allots[x] ← j
return dp[2^n − 1]
```

### C. The countSetBits Algorithm

$countSetBits(x)$ returns the number of 1-bits set in an integer x. So this is an algorithm to determine how populated an integer is.

For example, say we have an integer x with value equal to 12. 12 in binary is just 1100, and the rest of the digits are just 0's. So, $countSetBits(x)$ returns the value 2.

**Algorithm 3** countSetBits Method

```
Input: n
if n == 0 then
    return 0
return (n&1) + countSetBits(n ≫ 1)
```

### D. Main Method

**Algorithm 4** Main Method

```
Input: costmatrix, n
max ← convertToMin(costMatrix, n)
solution ← allot(costmatrix, n, dp)
print(max * n − solution.val)
print(solution.allots)
```

## III. Analysis

### A. Time Complexity

*1) convertToMin:* The $convertToMin$ method traverses through all the cells in the input costMatrix twice – first to find the maximum value, then to convert the matrix to a "job dislike" matrix. Hence, the time taken is of the expression $T(n) = 2 * n^2$, with a time complexity of $O(n^2)$.

*2) countSetBits:* The computation time of the $countSetBits$ method on a number of value $n$ is recursive, and is given below:

$$T(n) = T(\frac{n}{2}) + O(1)$$
$$T(1) = O(1)$$

Here, the $T(\frac{n}{2})$ is the computation time of the same method, called recursively, on $n$ right shifted by one bit. This is equivalent to dividing $n$ by 2. The following is a derivation of the non-recursive form

of the expression.

$$\begin{aligned} T(n) &= T(\frac{n}{2}) + 1 \\ &= [T(\frac{n}{4}) + 1] + 1 \\ &= T(\frac{n}{4}) + 2 \\ &= [T(\frac{n}{8}) + 1] + 2 \\ &\vdots \\ T(n) &= T(\frac{n}{2^k}) + k \end{aligned}$$

For $T(1)$:

$$\begin{aligned} &\implies \frac{n}{2^k} = 1 \\ &\implies 2^k = n \end{aligned}$$

Entering the value of $k$ in the equation above, we obtain the solution:

$$T(n) = \log_2(n) + O(1)$$

Hence, the time complexity of this method is $O(\log n)$.

*3) allot:* On analysing the steps involved in the $allot$ method, we can derive the following expression:

$$T(n) = ((1+n) * n + log_2 n) * 2^n + 2^n$$

The $log_2 n$ term above is the time taken of the $countSetBits$ methods discussed in the previous section. For large values of $n$, we would take only the dominant terms into consideration to find the overall growth of the function. On doing so, we obtain the time complexity of the $allot$ method to be $O(n^2 \times 2^n)$.

Therefore, the overall time complexity of the implemented algorithm is $O(n^2 \times 2^n)$.

### B. Space Complexity

The data structure we used to implement the algorithm is a struct which contains an integer and an array of size $n$. Moreover this struct $dp$ is of size $2^n$ hence the space complexity of our algorithm is $O(n \times 2^n)$.

## IV. Experimental Analysis

The algorithm was run against various inputs randomly generated for values of $n$ ranging from 0 to 20. The total number of computations have been plotted against the size of the input in Fig. 3.

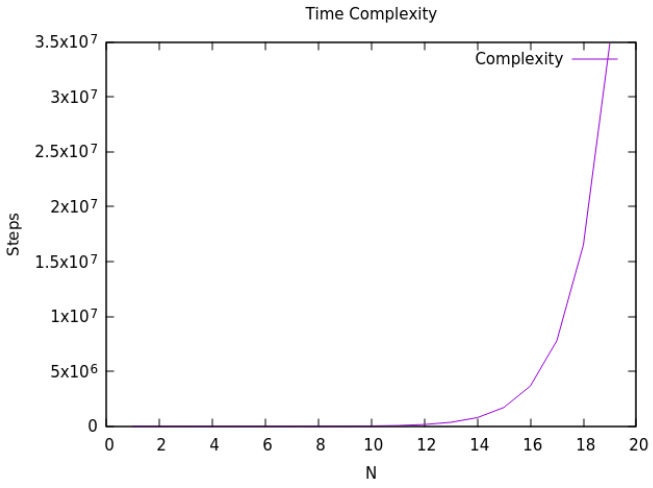


Fig. 3. Time complexity of finding filled cells with recursion

## V. Discussion

In order to solve the problem, there is another approach which uses brute force. In this approach, we try every possible assignment of the jobs to the engineers and find the one which maximizes the average liking of all the jobs. Since there are $n$ jobs to be assigned, the total number of ways in which they can be rearranged for allotment is $n!$. So, the time complexity for this approach is of the order $O(n!)$, which is clearly not good.

## VI. Conclusion

Through this paper we proposed the algorithm to assign $n$ jobs to $n$ engineers such that the average liking of all the jobs is maximized by using dynamic programming and bit masking techniques. This helped us achieve a better time complexity than the brute force method which is an $O(n!)$ implementation. Our algorithm has time complexity of $O(n^2 * 2^n)$ and space complexity of $O(n * 2^n)$.

## References

[1] Introduction to Algorithms Book by Charles E. Leiserson, Clifford Stein, Ronald Rivest, and Thomas H. Cormen