CS 218, Spring 2017 Homework 7
Abhishek Kumar SRIVASTAVA May 31, 2017
Student ID: 861307778

I certify that this submission represents my own original work

---

Solution 1. Let $P(i, j)$ be the minimum cost of sequential bin packing of i elements into j bins. When bin count is 1 it is nothing but sum of all the items which is the base cases. When bin count is more than 1 we will build upon previous case of when they have 1 bin by considering multiple combinations and dividing among possible bins. we will calculate the maximum weight for different combinations possible and then choosing the minimum value among that. Therefore the $P(i, j)$ can be recursively defined as follows:

$$P(i,j) = \begin{cases} s_1 & i = 1, j = 1 \\ P(i-1,j) + s_i & i > 1, j = 1 \\ \min_{1 \le k \le i-1}(\ max\{P(k, j-1), P(i,1) - P(k,1)\}) & i > 1, j > 1 \end{cases}$$

Algorithm 1

```
function SequentialBinMatching(S[ ],k)
    P[ ][ ]                                              ▷ To store computed values
    for i = 1 to length(S) do
        for j = 1 to k do
            if j == 1 then
                if i == 1 then
                    P[i][j] ← S[1]                                   ▷ Base Case
                else
                    P[i][j] ← P[i − 1][j] + S[i]                     ▷ Base Case
                end if
            else
                for l = 1 to i − 1 do                                ▷ Recursive Case
                    min_max_wt ← min( min_max_wt, max(P[k][j − 1], P[i][1] − P[k][1]) )
                end for
                P[i][j] ← min_max_wt
            end if
        end for
    end for
    return P[length(S)][k]
end function
```

The outermost loop takes $\mathbf{O(n)}$ time to iterate over list of n items. Its inner loop takes $\mathbf{O(k)}$ to iterate over k bins. The innermost for loop takes $\mathbf{O(n)}$ time to compute the min_max_wt. Therefore the overall time complexity of the algorithm is $\mathbf{O(n^2k)}$. We are also storing the results of all the sub-problems in a table (of dimension $n \times k$). Therefore the **space complexity** of the algorithm is $\mathbf{O(nk)}$.

Solution 2. Let $P(i, j)$ be the minimum weight of cross-free matching where i is i^{th} element in X *i.e* x_i & j is j^{th} element in Y *i.e* y_j in $G = (X, Y, X \times Y)$ for $X = \{x_1, ..., x_i\}$, $Y = \{y_1, ..., y_j\}$. When there is only 1 element in X it is a base case and we will simply assign that weight value. For the case when $i > j$ weight cost for it will be infinite since we are assuming only cases where edges are not crossed. And for the last case we consider edge which result in min weight cost. Therefore we can define $P(x_i, y_j)$ recursively as follows:

$$P(i,j) = \begin{cases} w_{1,1} & i = 1 \\ \infty & i > j \\ \min\{P(i-1, j-1) + w_{i,j} \ , \ P(i, j-1)\} & i \leq j \end{cases}$$

Algorithm 2

```
function FindCrossFreeMatching(X[ ], Y[ ], w[ ][ ])
    P[ ][ ]                                  ▷ Stores the previously computed values
    M                                        ▷ Stores the edges part of the matching
    for i in length(X) do
        for j in length(Y) do
            if i == 1 then
                P[i][j] ← w[1][1]
            end if
            if i > j then
                P[i][j] ← ∞
            else
                if P[i − 1][j − 1] + w[i][j] < P[i][j − 1] then
                    P[i][j] ← P[i − 1][j − 1] + w[i][j]
                    M.add((X[i], Y[j]))
                else
                    P[i][j] ← P[i][j − 1]
                end if
            end if
        end for
    end for
    return M
end function
```

As we are iterating over X in the outer loop and over Y in the inner loop, the **time complexity** of the above algorithm is **O**(**nm**). We are also storing the results of all the sub-problems in a table (of dimension $n \times m$). Therefore the **space complexity** of the algorithm is **O**(**nm**).

Solution 3. Part 1. Let $P(i, j)$ be the probability of obtaining j heads when i coins are tossed independently at random. When $i < j$ probability will be 0. When $j = 0$ it is multiplication of $1 - p_i$. When $j \neq 0$, j^{th} coin is tossed it can be either be heads with p_j probability or be tails with $(1 - p_j)$ probability. Therefore the $P(i, j)$ can be recursively defined as follows:

$$P(x_i, y_j) = \begin{cases} 0 & i < j \\ 1 - p_1 & i = 1, j = 0 \\ P(i-1, j) \cdot (1 - p_i) & i > 1, j = 0 \\ P(i-1, j) \cdot (1 - p_i) + P(i-1, j-1) \cdot (p_i) & i \geq j, j \geq 0 \end{cases}$$

Algorithm 3

```
function FindProbability(p[ ], k)
    P[ ][ ]                                  ▷ Stores the previously computed values
    for i = 1 to length(p) do
        for j = 0 to k do
            if i < j then
                P[i][j] ← 0
            else
                if j == 0 then
                    if i == 1 then
                        P[i][j] ← (1 − p[1])
                    else
                        P[i][j] ← P[i − 1][j] · (1 − p[i])
                    end if
                else
                    P[i][j] ← P[i − 1][j] · (1 − p[i]) + P[i − 1][j − 1] · p[i]
                end if
            end if
        end for
    end for
    return P[length(p)][k]
end function
```

Now the outer loop executes n times and the inner loop executes k time where $k \in [0, n]$. Therefore the worst case complexity of the above algorithm will be $\mathbf{O(n^2)}$ when $\mathbf{k = n}$.

2. Consider p_i to be probability that the i^{th} coin comes up heads and $q_i = 1 - p_i$ to be the probability that the i^{th} coin comes up tails. Now $g(x)$ can be represented as following product.

$$\begin{aligned} g(x) &= \prod_{k=1}^{n} (p_k x + q_k) \\ &= \prod_{k=1}^{n/2} (p_k x + q_k) * \prod_{k=n/2+1}^{n} (p_k x + q_k) \end{aligned}$$

The product can be computed recursively by dividing it in two subproblems of half the input size and multiplying the result of the two sub-problems to get the answer to the original problem. Time complexity for multiplication of two polynomials of degree $\frac{n}{2}$ using **FFT** is $\mathbf{O(\frac{n}{2} * log(\frac{n}{2})) \leq O(n * log\ n)}$. Therefore recurrence relation can be represented as follows:

$$T(n) = 2 \cdot T(\frac{n}{2}) + O(n * log\ n)$$

$$\therefore T(n) = O(n \cdot log^2 n) \qquad \text{MT case(II) with a=2, b=2, k=1, f(n) = O(n*log n)}$$