- The instructions are the same as in Homework-0, 1, 2.

There are 6 questions for a total of 120 points.

1. (*Please note that this question will be counted towards Homework-3.*)

   Consider two binary strings $x = x_1, ...x_n$ and $y = y_1, ..., y_m$. An interleaving of x and y is a strings $z = z_1, ..., z_{n+m}$ so that the bit positions of z can be partitioned into two disjoint sets X and Y , so that looking only at the positions in X, the sub-sequence of z produced is x and looking only at the positions of Y , the sub-sequence is y. For example, if $x = 1010$ and $y = 0011$, then $z = 10001101$ is an interleaving because the odd positions of z form x, and the even positions form y. The problem is: given x, y and z, determine whether z is an interleaving of x and y.

   Here is a simple back-tracking recursive algorithm for this problem, based on the two cases: If z is an interleaving, either the first character in z is either copied from x or from y.

```
BTInter($x_1, ..., x_n$ ; $y_1, ..., y_m$; $z_1, ..., z_{n+m}$):
  - IF $n = 0$ THEN IF $y_1, ..., y_m = z_1, ..., z_m$ THEN return True ELSE return False
  - IF $m = 0$ THEN IF $x_1, ..., x_n = z_1, ..., z_n$ THEN return True ELSE return False
  - IF $x_1 = z_1$ AND BTInter($x_2, ..., x_n$, $y_1, ..., y_m$; $z_2, ..., z_{n+m}$) return True
  - If $y_1 = z_1$. AND BTInter($x_1, ..., x_n$, $y_2, ..., y_m$; $z_2, ..., z_{n+m}$) return True
  - Return False
```

   Answer the parts below:

   (a) (3 points) Show the tree of recursions the above algorithm would make on the above example.

   Answer.
   Let each recursive call be represented as
   BTIter($x_i x_{i+1}...x_n$,$y_j y_{j+1}...y_m$,$z_k...z_{n+m}$) = BT(i,j,k) wrt to the original binary strings

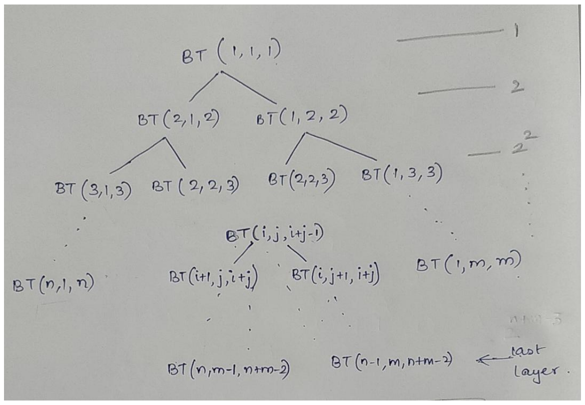


Figure 1: Recursion Tree

   (b) (3 points) Give an upper bound on the total number of recursive calls this algorithm might make in terms of n and m.

   Answer
   From the above recursion tree we can see that the tree grows exponentially in the power of 2. The total number of layers = maximum depth = 2^{n+m-2}. This is because the longest depth will be when we traverse both the bit strings x and y completely,i.e BT(n,m-1,n+m-2) and BT(n-1,m,n+m-2). Therefore, the upper-bound for the given recursion tree based on the recurrence relation T(i,j) =

T(i+1,j) + T(i,j+1)+1 where i and j are the first indices of x and y wrt the original bit strings and $1 <= i <= n$, $1 <= j <= m$, is given as $1 + 2 + 2^2 + ... + 2^{n+m-3} = 2^{m+n-2} - 1 = O(2^{m+n})$
The number of nodes will be lower than this as the base case will be reached for some nodes beyond the depths m and n, when the indices i and j reach the values m and n. This value will be upper-bounded by $O(2^{m+n})$.

(c) (3 points) Which distinct sub-problems can arise in the recursive calls for this problem?
Answer
The distinct recursive subproblems can be uniquely characterised based on the starting indices, say i and j, of both the substrings of x and y. The indices are in the range $1 <= i <= n$, and $1 <= j <= m$ .This implies that there are (n*m) distinct recursive sub-problems for this problem.

(d) (7 points) Translate the recursive algorithm into an equivalent DP algorithm, using your answer to part (c).
Answer:
The redundant recursive calls can be eliminated by storing all the m*n distinct recursive calls in an (n x m) 2D array. If we construct an array where dp where $dp[i][j] =$ BTIter($x_i x_2 ... x_n$,$y_j y_2 ... y_m$,$z_{i+j-1} ... z_{n+m}$) , the DP algorithm will be

```
procedure BTIter-dp(x,y,z):
    n = len(x)
    m = len(y)
    initialize 2D array dp[n][m] to False for all indices (1-indexed)
    for i from 1 to n:
        if(x[i:]==z[i+m:]):
            dp[i][m]=True
    for j from 1 to m:
        if(y[j:]==z[j+n:]):
            dp[n][j]= True
     for j from m-1 to 1:
        for i from n-1 to 1:
            if((x[i]==z[i+j-1]) and (dp[i+1][j])):
                dp[i][j] = True
            elif ((y[j]==z[i+j=1]) and dp[i][j+1]):
                dp[i][j] = True
return dp[1][1]

Final Answer = BTIter-dp(x,y,z)
```

(e) (4 points) Give a time analysis for your DP algorithm

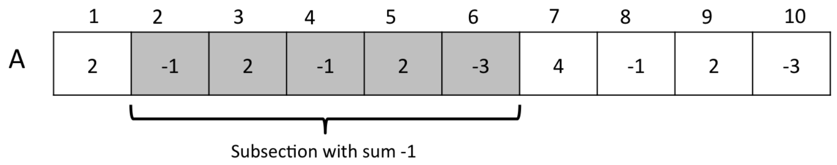
Answer:
The above DP algorithm fills each of the m*n entries of the dp array in O(1) time. Therefore, the total time complexity of the algorithm is O(m*n).

NOTE: For the remaining questions, structure your answer in the following format. You should explicitly give:

1. Description of sub-problems (2 points)
2. Base Case(s) (2 points)
3. Recursion with justification. (*A complete proof by induction is NOT required. However, you should explain why the recursion makes sense and how it covers all possibilities*) (6 points)

4. Order in which sub-problems are solved (2 point)
5. Form of output (how do we get the final answer?) (2 point)
6. Pseudocode (3 points)
7. Runtime analysis (3 points)
8. A small example explained using an array or matrix as in the previous questions (Optional)

2. (20 points) Given a sequence of integers (positive or negative) in an array $A[1...n]$, the goal is to find a *subsection* of this array such that the sum of integers in the subsection is maximized. A subsection is a contiguous sequence of indices in the array. (*For example, consider the array and one of its subsection below. The sum of integers in this subsection is* -1.)



Let us call a subsection that maximizes the sum of integers, a *maximum subsection*. Design a DP algorithm for this problem that output the sum of numbers in a maximum subsection.

Answer:

1. **Description of subproblem:**
We define S[k] as the sum of numbers in a maximum subsection of the subarray A containing k elements from the beginning, and maxEndingAtidx[k], as the sum of numbers in a maximum subsection, ending at kth element.
2. **Base Case:**
The trivial case is when the array is of length 0, i.e. n=0. In that case, the sum of numbers in the contiguous subarray will be 0, as well as the maxEndingAtidx[0] will be 0 as there are no elements in the subsequence.
3. **Recursion with justification:**
We define:
maxEndingAtidx[k] = max(A[k] , maxEndingAtidx[k-1] + A[k])
S[k] = max(S[k-1] , maxEndingAtidx[k])
For the update of maxEndingAtidx, we can simply see the logic that the sum of maximum subsection ending at the kth element can be the maximum of only the kth element or sum of kth element with sum of maximum subsection ending at (k-1)th element. So, it covers both the possibilities that the subsection might be continuing from earlier one(taking new element along with previous sum), or a new subsection formed with just kth element(this is the case when previous sum might be negative, which is true because we leave earlier one only if it decreases our total sum).
For the update of S[k], we take the maximum over the subsection ending at kth element, or the maximum sum upto (k-1)th element. This makes sense because the maximum subsection might be the one that ended earlier(S[k-1]), or the one still continuing(maxEndingAtidx[k]). This covers both include and exclude case for the element, to get the maximum yet.
4. **Order in which sub-problems are solved:**
We see that the update for S[k], requires S[k-1]. Hence, we will start from k=0, and move forward, so that we won't have to recurse.
We set S[0] = 0, maxEndingAtidx[0] = 0.
Then, maxEndingAtidx[1] = (A[1] , maxEndingAtidx[0] + A[1]), and S[1] = max(S[0], maxEndingAtidx[1]), and we similarly go from k=0,1,....n, to get S[n].
5. **Form of output:**
The output we want is the sum of maximum subsection over the array A, containing all n elements, which is nothing other than S[n]. So, our required output is S[n].
6. **Pseudocode:**
The required pseudocode is as follows:

```
maxSubsectionSum(A):
        - initialize the arrays S and maxEndingAtidx
        - S[0] = 0
        - maxEndingAtidx[0] = 0
        - for i in range(1,n):
```

```
        - maxEndingAtidx[i] = max(A[i] , maxEndingAtidx[i-1] + A[i])
        - S[i] = max(S[i-1] , maxEndingAtidx[i])
    - return S[n]
```

7. **Runtime Analysis:**
   Total running time = n * (time to compute S[i])
   = n* O(1) (only 2 operations are required)
   = O(n)
   The overall algorithm just needs a pass over the given array A and S, and hence we can conclude that the model is linear time, i.e. O(n).

3. (20 points) Let $p(1), \ldots, p(n)$ be prices of a stock for n consecutive days. A k-block strategy is a collection of m pairs of days $(b_1, s_1), \ldots, (b_m, s_m)$ with $0 \leq m \leq k$ and $1 \leq b_1 < s_1 < \cdots < b_m < s_m \leq n$. For each pair of days (b_i, s_i), the investor buys 100 shares of stock on day b_i for a price of $p(b_i)$ and then sells them on day s_i for a price of $p(s_i)$ with a total return of:

$$100 \sum_{1 \leq i \leq m} p(s_i) - p(b_i)$$

Design a DP algorithm that takes as input a positive integer k and the prices of the n consecutive days, $p(1), \ldots, p(n)$ and computes the maximum return among all k-block strategies.
Answer

1. Description of subproblem :

We have n days and a total of at most k transactions to be made. We note that 2*k $\leq$ n.
We represent the problem in an array called DP. DP[j,k] denotes the sub-problem of maximum profit until day j (including the selling on day j) after doing at most k transactions (1 transaction means 1 act of buying and selling).

2. Base Case :

There are 2 trivial cases :
Case1 : k = 0
In this case, there cannot be any transactions, hence we should set DP[j,0] = 0 for all j ϵ [1,2,...,n].

Case2 : n = 1 or n = 0
Since there is just one day or no day at all, we cannot buy and sell on the same day. Hence, the total return will be zero. So, we should set DP[1,m] = DP[0,m] = 0 for all m ϵ [0,2,...k].

3. Recursion with justification :

Recursion Equation :
c1 = DP[(i-1),m]
prevDifference = max ([(DP[j-1,m-1] - 100*p[j-1]) for j $\leq$ i ϵ 2,3,...n])
c2 = prevDifference + 100*p[i]
DP[i,m] = max(c1 , c2)

Justification :
c1 refers to the case when we are not selling on day i. So, the total number of transactions remains same as day (i-1).
prevDifference is a variable which stores the maximum difference between the profit made on a particular day (at most m-1 transactions) and the price of stock on that day upto the i th day.
So, prevDifference helps us choose the optimal day out of days 1, 2, ... (i-1) on which we should buy the stock and sell that stock on day i. Why the day with maximum prevDifference value is the optimal day is because maximum return by selling a share purchased on day j and sold on day i, will be the profit of this sale ie 100*(p[i] - p[j]) added to the total maximum profit made till day j by selling at most (m-1) shares, which is represented by DP[j,(m-1)]. We also have to choose day j such that the sum DP[j,(m-1)] + 100*(p[i] - p[j]) is maximised. Hence, mathematically we only need a j such that DP[j,(m-1)] - 100*p[j] is maximised.
Now j must be in the range [1,2...(i-1)] according to constraints of the problem, $i > j$. That is why the prevDifference equation above has (j-1) terms since i varies from 2 to n and j varies from 1 to (i-1).
Now prevDifference + 100*p[i] = DP[j,m-1] - 100*p[j] + 100*p[i] = DP[j,m-1] + 100*(p[i] - p[j]) for a suitable j which maximises this value. We note that this is the maximum obtainable profit if we definitely sell on day i.
So, c2 represents the case when we sell a share on day i with maximum possible profit. Logically, DP[i,m] should be the maximum out of c1 and c2, which is what happens in the recursion equation.

4. Order in which sub-problems are solved:

We first set DP[j,0] = 0 for all j ϵ [1,2,...,n].
Then, DP[1,m] = DP[0,m] = 0 for all m ϵ [0,1,2,...k].
After the base cases, we calculate DP[j,m] column-wise starting from column indexed 1 ie DP[2,1], DP[3,1], DP[4,1], ... DP[n,1] then DP[2,2], DP[3,2], DP[4,2], ... DP[n,2] and so on upto DP[2,k], DP[3,k], ... DP[n,k].

5. Form of output :

Answer = max(DP[n,k]). It will be numeric.

6. Pseudocode :

```
procedure maxprofit(p,n,k):

    for m in range [1,2,...k]:

        prevDifference = Minimum Integer Value ## set it as an extremely small value
        for i in range [2,3,...,n] :

            c = DP[i,m-1] - 100*p[i]
            if (c > prevDifference) :
                prevDifference = c

            c1 = DP[i,m-1]
            c2 = 100*p[i] + prevDifference

            DP[i,m] = max(c1,c2)
    return(DP[n,k])
```

Final Answer = maxprofit(p,n,k)

7. Runtime analysis :

Total time taken = time for initialisation + n*k*(Time taken to calculate DP[i,m]
= O(n) + O(k) + n*k*O(1) [since only a finite number of operations to calculate c, prevDifference, c1, c2]
= O(n+k) + O(n*k)
= O(n*k)

4. (20 points) You are given an $n \times 5$ matrix A consisting of integers (positive or negative). Your goal is to find a set S of tuples (i, j) indicating locations of the 2-D matrix A such that:

   1. $\sum_{(i,j)\in S} A[i, j]$ is maximized, and
   2. For all pairs of tuples $(i_1, j_1), (i_2, j_2) \in S$, $(i_2, j_2) \notin \{(i_1 - 1, j_1), (i_1 + 1, j_1), (i_1, j_1 - 1), (i_1, j_1 + 1)\}$.

   (*For example, consider the* 2×5 *matrix below. The set of locations that satisfies (1) and (2) above are indicated by shading these locations in the matrix.*)

-1	2	▨ 3	-4	▨ 5
▨ 2	-5	3	▨ 5	6

   Design a DP algorithm for this problem that outputs the maximum possible sum attainable.
   Answer

   For simplicity of representation, we define some encodings on the 5 elements in each layer. These encodings are binary-valued strings having 5 5 bits (eg : 01010) and for layer l (indexed from 1 to n), this encoding would mean that l[2], l[4] are included in the summation, rest are excluded. In other words, "1" means include and "0" means exclude. According to the constraints given in the problem, we can have 13 different encodings and we will enumerate them as follows :
   encode(1) = "00000"
   encode(2) = "00001"
   encode(3) = "00010"
   encode(4) = "00100"
   encode(5) = "01000"
   encode(6) = "10000"
   encode(7) = "00101"
   encode(8) = "01010"
   encode(9) = "10100"
   encode(10) = "01001"
   encode(11) = "10010"
   encode(12) = "10001"
   encode(13) = "10101"

   We also define a function to check if a pair of these enumerated encodings can apply to 2 consecutive rows in A , given the constraints. For example : 6 and 9 cannot apply to consecutive rows since both include row[1]. 1 and 2 can apply to consecutive rows and so can 2 and 3. To test this, we can simply simply take element-wise "AND" and check if it results in "00000".

1. Description of subproblem :

We will use a 2-D array called DP for this problem. DP[i,j] will represent the subproblem which is the maximum possible summation for the sub-matrix having first i rows, with encoding for the i th row being j.

2. Base Case :

When A is a 1x5 matrix, then we can select elements only from the first and the only row. So, depending on the encoding used, the summation will be the maximum possible summation. DP[1,j] = encode(j) * A[1]. Where * refers to element-wise multiplication.

3. Recursion with justification :

Recursion Equation :
c = [DP[i-1,k2] + encode(k)*A[i] for all k2 ϵ such that k,k2 can apply to consecutive rows]
DP[i,k] = max(c)

Justification :
c is a temporary list which stores the value DP[i-1,k2] + encode(k) * A[i] for all values of k2 in range 1,2,..,13 such that k and k2 can apply to consecutive rows.
So, if the (i-1)th row has encoding k2 and the i th row has encoding k, the total sum of this configuration is DP[i-1,k2] + encode(k) * A[i]. Then, max(c) is the maximum of all these configuration values, which is stored in DP[i,k]. Hence, DP[i,k] is the maximum possible summation for the sub-matrix having first i rows, with encoding for the i th row being j.
We note that for k = 1, we are also able to cover the case where encode(k) = "00000" which means no element from i th row is selected.

4. Order in which sub-problems are solved:

Starting with i = 1, we first set DP[1,k] = encode(k) * A[1] for k ϵ 1,2,...,13.
We then calculate DP[i,k] for all values of i, starting from 2 upto n, using the recursion equation. This is the order in which sub-problems are solved.

5. Form of output :

The output (final answer) = max(DP) which is the maximum element in the 2-D array. It will be an integer value since all elements in A are integers and for calculating each DP[i,k], we use elements of A multiplied by 0 or 1.

6. Pseudocode :

```
procedure isCompatible(k1,k2):
        d = encode(k1) AND encode(k2) ## element-wise AND operation
        return(d == "00000")
```

```
procedure findMax(A):
        n = A.shape[0] ## number of rows in A
        DP = newArray(n,k) ## Initialise a new 2d array of dimension nxk
        for i in range [2,3,...n]:

                for k in range [1,2,...,13]:

                        C = [] ## Initialise an empty list
                        for k2 in range [1,2,...13]:
                                if (isCompatible(k,k2)) :
                                        m = DP[i-1,k2] + encode(k)*A[i]
                                        C.append(m)

                        DP[i,k] = max(C)

        return max(DP[n]) ## the maximum entry in the 2d array DP

Final Answer = findMax(A)
```

7. Runtime analysis :

Total running time = Time to initialise DP + n*(Time to calculate DP[i])
= O(n*1) + n*(13*(Time to calculate DP[i,k]))
= O(n) + n*13*(O(13)) [asuuming checking compatibilty and appending to a list takes O(1) time and max(C) takes O(13) time]
= O(n) + O(n)
= O(n)

5. (20 points) A town has n residents labelled $1, ..., n$. All n residents live along a single road. The town authorities suspect a virus outbreak and want to set up k testing centers along this road. They want to set up these k testing centers in locations that minimises the sum total of distance that all the residents need to travel to get to their nearest testing center. You have been asked to design an algorithm for finding the optimal locations of the k testing centers.

   Since all residents live along a single road, the location of a resident can be identified by the distance along the road from a single reference point (which can be thought of as the starting point of the town). As input, you are given integer n, integer k, and the location of the residents in an integer array $A[1...n]$ where $A[i]$ denotes the location of resident i. Moreover, $A[1] \leq A[2] \leq A[3] \leq ... \leq A[n]$. Your algorithm should output an integer array $C[1...k]$ of locations such that the following quantity gets minimised:

   $$\sum_{i=1}^{n} D(i), \text{ where } D(i) = \min_{j \in \{1,...,k\}} |A[i] - C[j]|$$

   Here $|x - y|$ denotes the absolute value of the difference of numbers x and y. Note that $D(i)$ denotes the distance resident i has to travel to get to the nearest testing center out of centers at $C[1], ..., C[k]$.

   (*For example, consider* $k = 2$ *and* $A = [1, 2, 3, 7, 8, 9]$*. A solution for this case is* $C = [2, 8]$*. Note that for testing centers at locations* 2 *and* 8*, the total distance travelled by residents will be* (1+0+1+1+0+1) = 4.)

Design a DP algorithm for this problem that outputs the minimum achievable value of the total distance.
Answer:

1. **Description of subproblem:**
For this problem, we define L[i,j] as the minimum achievable distance of 1,2..i residents, given j test centres are placed.

2. **Base Case:**
One base case is when no stations are set up, i.e. j=0. The required sum will be infinite in this case.
Another one is that the sum with just one resident, i.e. i=1, given any number of test centres j (j belongs to 1...k) is 0, i.e. L[1,j]=0.
Assuming the value of k will be greater than or equal to 1, the base case is when only one test centre is placed, i.e. j=1 in L[i,j].
We claim that L[i,j] = median of A[1], A[2]...A[i] (Given that both middle elements are considered as median in case of n being even)
Proof: Let us assume m to be the median of the resident locations, and let us say the distance sum from all residents is d. Let us take the following two cases:

    - n is odd: The number of elements on the left and right of median value are equal. Now, let us say if we were to shift the centre by one unit to the left, then the distance will increase by 1 unit for all the right elements as well the median, while it will decrease by 1 unit for all elements left of median. So, the new sum will be d+(n+1)/2-(n-1)/2 = d + 1. We can say similar if it is shifted to right by a unit. Similarly, the distance sum will increase if we shift even further. Therefore, the minimum distance is when the test centre is at median.
    - n is even: First of all, the sum at median1, and median2 will be same. Now, similarly as before, if we try shifting the median by a unit left for median1, the distance will increase by a unit for each element of the right half of array including the median1, and will decrease by 1 unit for left half of array excluding the median1. Hence, we can see the total distance sum increases. Same is the case with median 2. The sum at any point between median1 and median2, including the medians is the same(Proof for this can be found in the recursive case). Therefore, we can conclude that the minimum is when the test centre is at any of the medians.

3. **Recursion with justification:**
Now, for the transition relation, we can write it as:

```
for j in (1,k):
        for x=i to 1:
                L[i,j]=min(L[i,j],L[x,j-1] + sum of distances of points A[x+1],..A[i] from their median)
```

This update makes sense as what we are doing is that for getting the minimum sum of distances upto resident i, setting up j test centers is to find the minimum of such sum upto residents x¡i, setting j-1 test centers, and setting up the remaining one at the median of remaining residents upto i.
First of all, we claim that the minimum distance sum can be found if the test centres are placed in integer positions.
Proof by induction on number of test centres:
Base case:
When just one test centre is placed, we already proved that it is best to set them at the median values, whether the value of n is odd or even. In case when n is odd, the median already comes out to be integer only(as resident positions are integers), and in case of even n, we can claim that if we chose any point between the (n/2)th and (n/2+1)th element, the sum will be same only.
Without loss of generality, let us assume we set up the test centre at (n/2)th resident position. Now, the number of residents on its left are (n/2-1), and those on its left are (n/2) residents on its

right. Let us say if we shift the centre by a fractional value, f to the right, such that f is less than 1. Now, we can say the distances for all elements on left will increase by f, and the distance of (n/2)th resident also increases by f while the distance of n/2 citizens on right decrease by f, which nulls out the effect of increase, and hence our claim that its same between both median elements is same.
Induction hypothesis:
Let us suppose the test centres are at integer positions for number if test centres 1,2...k. We want to prove that it is integer for k+1 test centres.
Induction Step :
For calculating for k+1 test centres, we take minimum of i residents with k test centres and add the sum of remaining residents from their median, where last test centre is placed. As we proved that the median value is an integer and k test centres were at integer positions according to our hypothesis, therefore, we can say that test centres are placed at integer positions only.
Now, some think it might be the case that the jth test centre need not be the closest one to the residents A[x+1]..A[i]. So, if see the algorithm carefully, we can say that the algorithm already covered the case when the value of x was more than that of the case argued for. Hence, we can say that our algorithm covers all the possible cases.

4. **Order in which subproblems are solved:**
We can see in above update code that calculation of L[i,j] requires the values of L[i',j], where i' belongs to 1,...i-1. Therefore, we will solve the problems starting from L[1,j] (j varies from 1 to k), and increasing the value of i from that to n.

5. **Form of output:**
The final output will be when i will reach the number of residents, i.e. the answer will be L[n,k].

6. **Pseudocode:**
For the pseudocode, we need to do something to find the sum of distances from the median without taking a lot of time. For that, we will use the method of prefixes.

```
calcMedianSum(prefix[0..n],start,end,A[1..n]):
        mid = (start + end)/2
        sum_left = (mid - left)*A[mid] + prefix[start-1] - prefix[mid -1]
        sum_right = prefix[end] - prefix[mid] -(end - mid - 1)*A[mid]
        total_sum = sum_left + sum_right
        return total_sum

getDistanceSum(A[1...n],n,k):
        - initialize the array prefixes to 0 for all n
        - initialize L[i,j] to infinity for all values i belonging to 1 to n, and j belonging to 1 to k.
        - if k is 0, return MAXINT ## a very large value
        - for i in range 1 to n:
                - prefix[i] = prefix[i-1] + A[i]
        - for j in range 1 to k:
                - L[1,j] = 0
        - for i in range 2 to n:
                - for j in range 1 to k:
                        - for x = i to 1:
                                - L[i,j] = min (L[i,j], L[x,j-1] + calcMedianSum(prefix, x+1, i, A))
        - return L[n][k]
```

7. **Running Time:**
Let us discuss the running time for calculating median first.
The making the prefix array takes O(n) time, and using that, sum with median can be calculated in just O(1). Therefore, overall sum with median as test centre can be calculated in O(n).

Now, for the update, we see there is a for loop going from 1 to k, another from 1 to n, and the innermost one from some i to n, which can be worst case 1 to n. Hence, the overall time for completing the matrix is O(n^2k).

6. (20 points) You have a set of n integers $\{x_1, ..., x_n\}$ such that $\forall j, x_j \in \{0, 1, ..., K\}$. You have to design an algorithm to partition the set $\{1, ..., n\}$ into two disjoint sets I_1 and I_2 such that such that $|S_1 - S_2|$ is minimized, where $S_1 = \sum_{j \in I_1} x_j$ and $S_2 = \sum_{j \in I_2} x_j$. Design an algorithm that outputs the minimum value of $|S_1 - S_2|$ achievable. The running time of your algorithm should be polynomial in n and K.

   Answer:
   Let TSum be the total sum of all the elements $\{x_1, ..., x_n\}$

   1. **Description of subproblem:**
      The problem can be split into finding if there exists a subset of elements with the sum i for all $00 <= i <= TSum$. So we define a dp array dp[n+1][TSum+1] where dp[i][j] = True if any subset of elements $\{x_1, ..., x_i\}$ have the sum j, False otherwise.

   2. **Base Case:**
      The base cases are

      (a) dp[0 to n][0] = True, since the sum = 0 can be achieved using any subset of elements. (By taking empty subset)

      (b) dp[0][1 to TSum] = False since we cannot have sum of elements ¿ 0 with zero elements, except for sum = 0 (which is not included).

   3. **Recursion with Justification:**
      dp[i][j] = dp[i-1][j] or dp[i-1][j-x_i]
      To find if there exists a subset s.t the sum equals j, we can split it into 2 cases where x_i is included and where x_i is not included in the subset that gives the sum.
      If x_i is not included then considering $\{x_1, ..., x_i\}$ will be same as considering $\{x_1, ..., x_{i-1}\}$. Therefore , dp[i][j] = dp[i-1][j].
      If x_i is included, then we need to check if there exists a subset of $\{x_1, ..., x_{i-1}\}$ s.t they give the sum (j-x_i). Therefore, dp[i][j] = dp[i-1][j-x_i].
      Finally, dp[i][j] is true if either of the cases is true. Therefore we take the or of 2 subproblems.

   4. **Order in which subproblems are solved.**
      According to the above recursion, we need to solve the above subproblems in increasing order of i. The order of j does not matter. Therefore, the recursion can be solved by 2 nested for loops where i goes from 1 to n and j goes from 1 to TSum.

   5. **Form of output:**
      From the final dp array, to find the answer of the minimum difference, we need to find the largest j s.t dp[n][j] = True for some i where $j < sum//2$. This implies that there exists a subset of $\{x_1, ..., x_n\}$ s.t sum = j. This implies that the sum of the remaining elements = TSum - j. Therefore, the difference = TSum - 2*j. To maximise this difference we need to maximise j ensuring that the difference remains positive. Therefore, we iterate through the dp array and find the greatest j s.t dp[i][j] = True from $0 <= j <= TSum//2$.

6. **Pseudocode:**

```
procedure FindMinDiff({$x_1, ..., x_n$}):
    TSum = 0
    for i from 1 to n:
        TSum += $x_i$
    initialize 2D array dp[n+1][TSum+1]
    for i from 0 to n:
        dp[i][0] = True
    for j from 1 to TSum:
        dp[0][j] = False
    for i from 1 to n:
        for j from 1 to TSum:
            dp[i][j] = dp[i-1][j]
            if($x_i$ ¡ j):
                dp[i][j] = dp[i][j] or dp[i-1][j-$x_i$]
    Diff = INT-MAX
    for j from int[TSum/2] to 0: ## int[x] means integer part of number x
        if(dp[n][j]==True):
            Diff = TSum - 2*j
            break
    return Diff
```

7. **Running time:**
The size of the array in terms of input size variables n and K is (n*TSum). We know that $x_i < K$ and hence $TSum < K * n$. This implies that size of array is $O(n^2K)$.The complexity of initialising the array is hence $O(n^2K)$.Each element of the dp array is filled in O(1) time. The last loop is $O(TSum) = O(nK)$. Therefore the total time complexity is $O(2n^2K + nK) = O(n^2K)$