Name: Ananye Agarwal **Entry Number:** 2017CS10326
Name: Kabir Tomer **Entry Number:** 2017CS50410
Name: Rajat Jaiswal **Entry Number:** 2017CS50415

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.

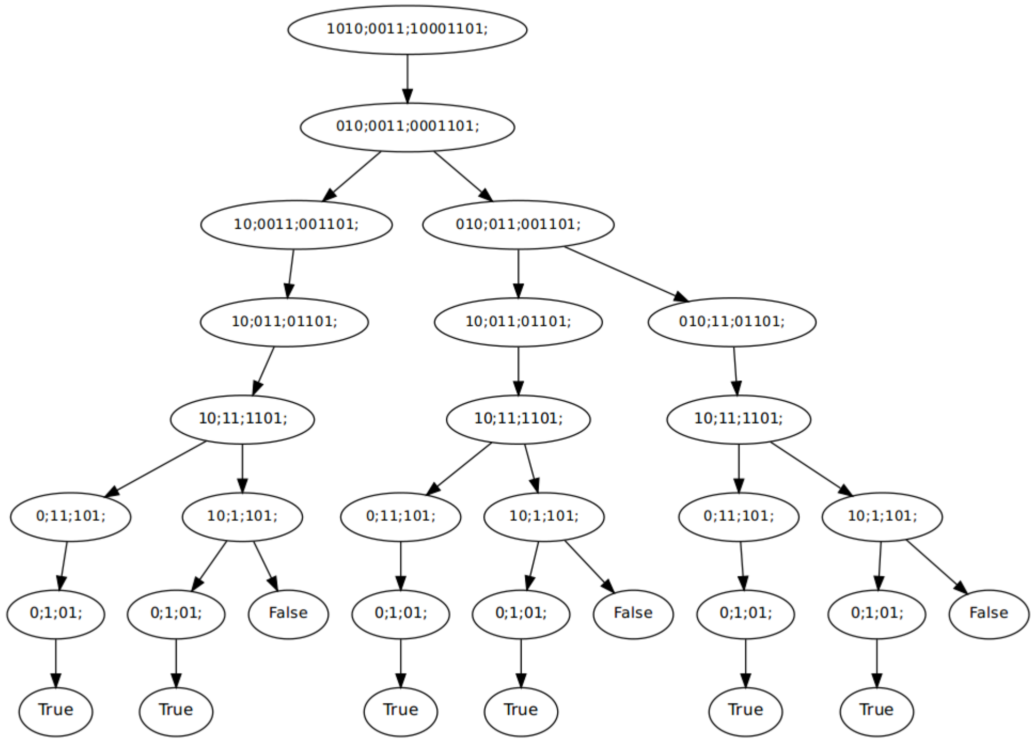
Here is a simple back-tracking recursive algorithm for this problem, based on the two cases: If z is an interleaving, either the first character in z is either copied from x or from y.

```
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.

Solution: Assuming lazy evaluation of AND, the recursion tree (with function calls identified by parameters) is as follows.



The actual evaluation path is a DFS path that stops when it reaches True, i.e. only the leftmost path will actually be followed during evaluation.

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

Solution: Each function call can make up to 2 recursive calls, and in each call either m or n decreases by 1. Therefore, the recursion depth can be at most $m+n-1$. A reasonable upper bound is thus $\mathcal{O}(2^{m+n})$ (or 2^{m+n} if a concrete value is required).

(c) (3 points) Which distinct sub-problems can arise in the recursive calls for this problem?

Solution: Each recursive call to $BTInter$ can be characterised by the lengths of the first two inputs, i.e., by m and n. This is because the final argument is always the last $m+n$ bits of the string z, and the first two inputs are the last m and n bits of x and y. These represent a possible mn subproblems that may arise.

(d) (7 points) Translate the recursive algorithm into an equivalent DP algorithm, using your answer to part (c).

Solution: DP[a][b] represents the subproblem where lengths are a and b as described above. DP[a][b] requires either DP[a-1][b] or DP[a][b-1] to be calculated. Therefore we may fill each row from left to right and choose rows to fill from top to bottom.

Let $A[-p]$ be syntactic sugar for $A[length(A)-p]$, i.e., pth from the last bit of A

```
BTInterDP($x = x_1, ..., x_n$ ; $y = y_1, ..., y_m$; $z = z_1, ..., z_{n+m}$):
  - $DP \leftarrow$ 2D array of dimensions $n + 1 * m + 1$, initialised to False.
  - $DP[0][0] \leftarrow True$
  - FOR $1 \leq j \leq m$
    - IF $y[-j] = z[-j]$ AND $DP[0][j-1]$THEN $DP[0][j] \leftarrow True$    - END FOR
  - FOR $1 \leq i \leq n$
    - IF $x[-i] = z[-i]$ AND $DP[i-1][0]$THEN $DP[i][0] \leftarrow True$    - END FOR
  - FOR $1 \leq i \leq n$
    - FOR $1 \leq j \leq m$
      - IF $x[-i] = z[-(i+j)]$ AND $DP[i-1][j]$ THEN $DP[i][j] \leftarrow True$
      - IF $y[-j] = z[-(i+j)]$ AND $DP[i][j-1]$ THEN $DP[i][j] \leftarrow True$
    - END FOR
  - END FOR
  - Return $DP[n][m]$
```

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

Solution: Initialising $DP[i][j]$ is a constant time operation for each i, j. Therefore the two unnested for loops run in time $\mathcal{O}(m)$ and $\mathcal{O}(n)$, while the nested loops together run in $\mathcal{O}(mn)$. Runtime is therefore $\mathcal{O}(mn)$

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.

Solution: We solve a slightly different general problem, and use it to construct our solution.

- **Subproblems:** For a given index i and fixed array A, consider the problem of finding the subsection with the maximum sum that ends (inclusive) at index i. Let $DP[i]$ denote the solution for the subproblem, i.e., the sum for the max sum subsection. We assume subsection cannot be empty (else return empty if sum is negative).
- **Base Case:** $DP[0] = A[0]$, since there is only one subsection ending on the first element.
- **Justification of recursion:** For the sub-problem $DP[i]$, we consider two cases. Either the subsection is just the i'th element, in which case $DP[i] = A[i]$, or it contains more that 1 element. We claim that in this case the subsection formed by appending $A[i]$ to the subsection given by $DP[i-1]$ will be the solution for $DP[i]$, i.e. $DP[i] = A[i] + DP[i-1]$. Consider some subsection S of more than 1 element that ends at $A[i]$. The subsection S' formed by removing the last element ends on $A[i-1]$. Therefore the subsection given by $DP[i-1]$ has a sum greater than or equal to S'. This shows that $A[i]$ as defined would be the maximum subsection ending at i. The two cases can thus be computed using only $DP[i-1]$ and then directly compared to see which to set as $DP[i]$.
- **Order of Sub-problems:** The $DP[.]$ array gets filled in from smaller indices to larger indices, with first element filled separately. As seen in the justification above, $DP[i]$ depends on the cell $DP[i-1]$, so if smaller indices are filled first, all required information shall be available.
- **Form of Output:** The algorithm will return the maximum sum, i.e., max over DP. If required the subsection itself may also be returned. This is correct since the global maximum subsection shall be stored in $DP[i]$ for some i.

Algorithm 1 Return maximum subsection sum

```

function MaxSubsection($A[1, 2, .., n]$)

    $DP$ ←Array of size n
    $DP[0] \leftarrow A[0]$
    for i = 1 to $n$ do
        if $0 > DP[i-1]$ then
            $DP[i] \leftarrow A[i]$
        else
            $DP[i] \leftarrow A[i] + DP[i-1]$
        end if
    end for
    return $max(DP)$.
end function

```

- **Runtime analysis:** The For-loop takes $\mathcal{O}(n)$ time, as well as finding the maximum. Therefore overall runtime is $\mathcal{O}(n)$.
- **Example:** Let price array be $A = [3, -2, 4, 3, -6, -3, 5]$. This gives us a $DP = [3, 1, 5, 8, 2, -1, 5]$, and a maximum sum of 8, corresponding to the subsection $[3, -2, 4, 3]$.

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.

Solution:

- **Subproblems:** Let $DP[i][j]$ denote the solution for the subproblem which is the maximum return among all i-block strategies, given the prices of first j consecutive days, $p(1), \ldots, p(j)$.

- **Base Case:** $DP[i][0] = 0$, for all $0 \leq i \leq k$, because given prices of 0 days, there is no transaction(buy or sell) that can happen. Similarly, $DP[i][1] = 0$, for all $0 \leq i \leq k$, because given price of just 1 day, buying and selling can't happen on single day, hence no transaction. $DP[0][j] = 0$, for all $0 \leq j \leq n$, because 0-block strategy means no collection of pairs and hence 0 return.

- **Justification of recursion:** For the sub-problem, DP[i][j], there can be two cases, one where there is a selling on day j and one where there's no selling on day j.
  - **Selling on day j:** If there's a selling on day j then prior to that there has to be a buying on day x, where $1 \leq x \leq j-1$. If this was the case then you receive $100 \cdot (P[j] - P[x])$ by the transaction of this pair. And since this was 1 pair, you can have a collection of maximum $i-1$ pairs in the initial $x-1$ days. And the maximum return in first $x-1$ days with a $i-1$-block strategy is the value $DP[i-1][x-1]$. Since we don't have an idea that which is the best day(x) to buy for the sale to happen on day j, so we iterate over all possible values of x from 1 to $j-1$, and take the maximum possible value. Hence we get the value of $DP[i][j]$ in this case as $\max_{1 \leq x \leq j-1}(100 \cdot (P[j] - P[x]) + DP[i-1][x-1])$.
  - **No Selling on day j:** If there's no selling on day j, then we skip the day j and calculate the maximum return in first $j-1$ days with a i-block strategy, which is precisely the value $DP[i][j-1]$. Hence we get the value of $DP[i][j]$ in this case as $DP[i][j-1]$.

  Since we also do not know whether there's a selling on day$j$ or not, so we take the maximum of both the cases discussed above. So finally the value of $DP[i][j]$ comes out to be $max\{(DP[i], j-1]), \max_{1 \leq x \leq j-1}(100 \cdot (P[j] - P[x]) + DP[i-1][x-1])\}$, which is precisely what the algorithm is calculating in iterations of the two outermost loops.

- **Order of Sub-problems:** The $DP[.][.]$ table gets filled in a top to bottom and left to right manner. As seen in the justification above, each of the cell $DP[i][j]$ depends on the cell $DP[i][j-1]$ and $DP[i-1][x-1]$, where $1 \leq x \leq j-1$. So basically the value $DP[i][j]$ depends on the rows that are above it and the columns that are to the left of it. So filling the table in the manner described above makes sense. We have already been provided the value for the base cases before entering the nested loop. So when filling the table for the first row all the values in 0th row has already been filled. And similarly, when going through second column, all the values in columns $0, 1$ are already filled. Hence this order works.

- **Form of Output:** The algorithm will return the value of $DP[k][n]$. By the definition of the sub-problems, $DP[k][n]$ denotes the maximum return among all k-block strategies, given the prices of n consecutive days, $p(1), \ldots, p(n)$, which is precisely what algorithm has to compute.

Algorithm 2 Maximum return among all k-block strategies given prices of n consecutive days.

```
function K-BLOCK-STRATEGY($P[1,2,..,n], k$)

    $DP[k+1][n+1] \leftarrow$ Initialise an array of size $(k+1) \times (n+1)$
    $Val[k+1][n+1] \leftarrow$ Initialise an array of size $(k+1) \times (n+1)$

    for i = 0 to $k$ do
        $DP[i][0] \leftarrow 0$; $Val[i][0] \leftarrow 0$
        $Val[i][0] \leftarrow 0$; $Val[i][1] \leftarrow -100 \cdot P[1]$
    end for

    $DP[0][1] \leftarrow 0$
    for j = 2 to $n$ do
        $DP[0][j] \leftarrow 0$; $Val[0][j] \leftarrow max(Val[0][j-1], -100 \cdot P[j])$
    end for

    for j = 2 to $n$ do
        for i = 1 to $k$ do
            $v \leftarrow 100 \cdot P[j] + Val[i-1][j-1]$.
            $DP[i][j] \leftarrow max(v, DP[i][j-1])$
            $Val[i][j] \leftarrow max(Val[i][j-1], DP[i][j-1] - 100 \cdot P[j])$
        end for
    end for

    return $DP[k][n]$.
end function
```

- **Runtime analysis:** Line 7-10 takes $O(k)$ time. Line 13-15 takes $O(n)$ time. In line 17-23, the two outer for loops runs $O(nk)$ iterations in total, and in each iteration the 3 statements takes $O(1)$ time in the worst case. All other operations also take $O(1)$ time. So, the overall running time of the algorithm is $\mathbf{O(nk)}$ time.

  To reduce the running time, we have used memoization. $\max_{1\leq x\leq j-1}(100 \cdot (P[j] - P[x]) + DP[i-1][x-1])$ is equivalent to $100 \cdot P[j] + \max_{1\leq x\leq j-1}(DP[i-1][x-1] - 100 \cdot P[x])$. We have stored the second term in the Val table. $Val[i][j]$ stores $\max_{1\leq x\leq j}(DP[i][x-1] - 100 \cdot P[x])$. Hence the recursion formed would be $Val[i][j] = max(Val[i][j-1], DP[i][j-1] - 100 \cdot P[j])$, $0 \leq i \leq k$ & $2 \leq j \leq n$.. The second term in DP recursion requires $Val[i-1][j-1]$. Since $Val[i][j]$ depends only on $Val[i][j-1]$ and $DP[i][j-1]$, filling the table from left to right and top to bottom will do the job, which is exactly what we are doing here. The base cases for Val table are $Val[i][0]$, $0 \leq i \leq k$, will be 0 because $x \geq 1$ in computation of $Val[i][j]$. $Val[i][1]$, $0 \leq i \leq k$, is $-100 \cdot P[1]$, since $DP[i][0]$ is 0. $Val[0][j]$, can be computed since $DP[0][.]$ is 0 & $Val[0][1]$ is known.(Propagation).

- **Example:** Let price array be $P = [2, 3, 1, 5, 6]$ & $k = 2$. The computed values are correct since in the given example the indices pairs are (1,2) & (3,5) evaluating to 600. The DP & Val matrices are as follows:

	0	**1**	**2**	**3**	**4**	**5**
0	0	0	0	0	0	0
1	0	0	100	100	400	500
2	0	0	100	100	500	600

DP matrix.

	0	**1**	**2**	**3**	**4**	**5**
0	0	-200	-200	-100	-100	-100
1	0	-200	-200	0	0	0
2	0	-200	-200	0	0	0

Val matrix.

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.

Solution: Consider a row i of the input which has 5 values stored in it, indexed from 1 to 5. If we take a set of values from this row to be considered in the set S, then with the constraints defined in the problem, there could be a total of 13 possible combinations/subsets of these indices 1 to 5 that can be considered in set S and not violate the constraint. We index these combinations from 1 to 13. `Table 1(dict)` below summarizes the different subset of indices from 1 to 5 corresponding to these combination numbers. If an index j of row i is considered in the set S, then the same index j of row $i - 1$ can't be considered in the set S. Hence `Table 2(skip)` below summarizes the set of subsets/combinations of row $i - 1$ that can't be considered when the combination number x of row i is taken in set S. The table has been carefully curated since if an index j appears in a combination x, then all other combinations y where j occurs have to be skipped and is present in the table corresponding to the entry x. Empty set means nothing has to be skipped or nothing is present in the combination.

x	Subset/Combination
1	{}
2	{1}
3	{2}
4	{3}
5	{4}
6	{5}
7	{1, 3}
8	{1, 4}
9	{1, 5}
10	{2, 4}
11	{2, 5}
12	{3, 5}
13	{1, 3, 5}

Table 1(dict): Different subset of indices(1-5).

x	Subset of combinations
1	{}
2	{2, 7, 8, 9, 13}
3	{3, 10, 11}
4	{4, 7, 12, 13}
5	{5, 8, 10}
6	{6, 9, 11, 12, 13}
7	{2, 4, 7, 8, 9, 12, 13}
8	{2, 5, 7, 8, 9, 10, 13}
9	{2, 6, 7, 8 9, 11, 12, 13}
10	{3, 5, 8, 10, 11}
11	{3, 6, 9, 10, 11, 12, 13}
12	{4, 6, 7, 9, 11, 12, 13}
13	{2, 4, 6, 7, 8, 9, 11, 12, 13}

Table 2(skip): Combination numbers to skip.

- **Sub-problems:** Let $DP[i][j]$ denote the maximum possible attainable sum for first i rows of the input with the constraint that the $A[i][indices]$ ($indices$ in $dict[j]$(defined in `Table 1`)) must be included in the set S.
- **Base Case:** $DP[0][j] = 0$, for all $1 \leq j \leq 13$, because the first 0 rows means there is no row and hence no value or indices to compute. So the maximum possible sum that is attainable in this case will be 0.

- **Justification of recursion:** For the sub-problem, DP[i][j], we have to include all the elements which are $A[i][indices]$, where $indices \in dict[j]$. We take the sum of these elements. For all $indices \in dict[j]$, we have to skip all the combination numbers y such that $indices \in y$. This is precisely what *skip* dictionary stores. And for all other combination numbers z, such that any index $indices \in dict[j]$ and $indices \notin z$, we take one such z, for which $DP[i-1][z]$ is maximum.

  As explained above, any element index *indices* of the ith row can be included in the set S or not. If it is included then with the constraints given $(indices - 1)$ and $(indices + 1)$ of the ith row can't be there in the set S. This is what is stored in the dictionary *dict* that if we are including *indices* then what all other index can ocuur with it without violating constraints.

  Similarly, if we are including the *indices* of $i$the row in set S, then we can't include *indices* of $(i-1)$th row. We also can't include *indices* of $(i+1)$th row. But that will be taken care of when we are in recursion for *indices* of $(i+1)$th row, in that case $(i+1)$th will become ith, and ith will become $(i-1)$th. This is what is stored in dictionary *skip*, that if you have a particular combination of indices as per the dictionary *dict*, then what other combinations you can't take from $(i-1)$th row.

  Hence when we are considering $DP[i][j]$, we take the combination number $dict[j]$ or ith row, and skip all the combinations of $(i-1)$th row given by $skip[j]$. The maximum value of these combinations of ith row i.e. $\max_{1 \leq x \leq 13}(DP[i][x])$, will give us the maximum possible attainable sum for the first i rows of input.

- **Order of Sub-problems:** he $DP[.][.]$ table gets filled in a top to bottom manner. The column-wise order doesn't really matter. As seen in the justification above, each of the cell $DP[i][j]$ depends on the cell $DP[i-1][x]$, where $1 \leq x \leq 13$. So basically the value $DP[i][j]$ depends on the row that is just above it and all the columns of that row. So filling the table in the manner described above makes sense. We have already been provided the value for the base cases before entering the nested loop. So when filling the table for the first row all the values in 0th row has already been filled. Hence this order works.

- **Form of Output:** The algorithm will return the value $\max_{1 \leq x \leq 13}(DP[n][x])$. By the definition of the sub-problems, $DP[n][x]$ denotes the maximum possible sum for the input with the constraint that the $A[n][indices]$ [*indices* in $dict[x]$(defined in `Table 1`)] must be included in the set S. As discussed before the column indices $1 \leq x \leq 13$, cover all possible combinations of a particular row that can be included in the set S, including no values to be chosen from the row. Hence the maximum value over all possible combinations will give us the answer, which is precisely what the algorithm computes.

- **Example:** Let us take the same example as given in the question and fill the DP array for the same. The DP array comes out to be as following:

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	-1	2	3	-4	5	2	-5	4	-2	7	8	7
2	8	1	3	10	13	9	12	**15**	11	8	4	11	13

  This is correct since in the given example the the set of tuples would be (1, 3), (1, 5), (2, 1), and (2, 4) which evaluates to 15, which is exactly the value returned by the algorithm.

- **Pseudocode:**

Algorithm 3 Maximum possible sum attainable.

```
 1:
 2: function MAX-SUM($A[[1_1, 1_2, 1_3, 1_4, 1_5], ...[n_1, n_2, n_3, n_4, n_5]]$)
 3:
 4:     $DP[n+1][13] \leftarrow$ Initialise an array of size $(n+1)X(13)$
 5:
 6:     for i = 1 to 13 do
 7:         $DP[0][i] \leftarrow 0$
 8:     end for
 9:
10:     dict ←{1: {}, 2:{1}, 3: {2}, 4: {3}, 5: {4}, 6: {5}, 7: {1, 3}, 8: {1, 4},
11:             9: {1, 5}, 10: {2, 4}, 11: {2, 5}, 12:, {3, 5}, 13: {1, 3, 5}}
12:
13:     skip ←{1: {}, 2:{2, 7, 8, 9, 13}, 3: {3, 10, 11}, 4: {4, 7, 12, 13}, 5: {5, 8, 10},
14:             6: {6, 9, 11, 12, 13}, 7: {2, 4, 7, 8, 9, 12, 13}, 8: {2, 5, 7, 8, 9, 10, 13},
15:             9: {2, 6, 7, 8 9, 11, 12, 13}, 10: {3, 5, 8, 10, 11}, 11: {3, 6, 9, 10, 11, 12, 13},
16:             12: {4, 6, 7, 9, 11, 12, 13}, 13: {2, 4, 6, 7, 8, 9, 11, 12, 13}}
17:
18:     for i = 1 to $n$ do
19:         for j = 1 to 13 do
20:             $v \leftarrow 0$
21:             for x in $dict[j]$ do
22:                 $v \leftarrow v + A[i][x]$
23:             end for
24:             $prev \leftarrow 0$
25:             for x = 1 to 13 do
26:                 if x not in $skip[j]$ then
27:                     $prev \leftarrow max(prev, DP[i-1][x])$
28:                 end if
29:             end for
30:             $DP[i][j] \leftarrow v + prev$
31:         end for
32:     end for
33:
34:     $v \leftarrow 0$
35:     for i = 1 to 13 do
36:         $v \leftarrow max(v, DP[n][i])$
37:     end for
38:
39:     return $v$.
40: end function
41:
```

- **Running time analysis:** Line 6-8 takes constant amount of time. The for-loop at line 18 runs for n iterations, all the inner statements of this loop takes constant amount of time. The inner for-loop at line 19, runs for constant time, since the column size of the $DP[.][.]$ table is fixed as well. The for-loop at line 21 runs for constant time as well, since the maximum size of $dict[j]$ can be 5. *dict* and *skip* are dictionary of sets and are of constant size hence the lookup can be done in $O(1)$ time. The for-loop at line 25 runs for constant time as well. Hence, the overall running time of the algorithm is $O(n)$.

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.

Solution: We start by proving the following lemma

Lemma 5.1: The minimum value of the function $f(x) = |x - x_1| + |x - x_2| + \ldots + |x - x_n|$ occurs when x equals $x_{\lceil \frac{n}{2} \rceil}$.

Proof: Assume WLOG that $x_1 \leq x_2 \leq \ldots \leq x_n$. There are two cases

Case 1: $n = 2k$. Consider the following inequality,

$$f(x) = \sum_{i=1}^{2k} |x - x_i| = \sum_{i=1}^{k} |x - x_i| + |x_{k+i} - x| \geq \sum_{i=1}^{k} |(x - x_i) + (x_{k+i} - x)| = \sum_{i=1}^{k} x_{k+i} - x_i$$

This is derived from the triangle inequality $|x| + |y| \geq |x + y|$. Notice that in the above equation equality holds when $x = x_k = x_{\lceil \frac{n}{2} \rceil}$. Indeed, for $x = x_k$,

$$\sum_{i=1}^{2k} |x - x_i| = \sum_{i=1}^{k} (x_k - x_i) + (x_{k+i} - x_k) = \sum_{i=1}^{k} x_{k+i} - x_i$$

Case 2: $n = 2k + 1$. Again, consider the following inequalities,

$$f(x) = \sum_{i=1}^{2k+1} |x - x_i| = |x - x_{k+1}| + \sum_{i=1}^{k} |x - x_i| + |x_{k+i+1} - x| \geq |x - x_{k+1}| + \sum_{i=1}^{k} |(x - x_i) + (x_{k+i+1} - x)|$$

$$\implies |x - x_{k+1}| + \sum_{i=1}^{k} |(x - x_i) + (x_{k+i+1} - x)| \geq |x - x_{k+1}| + \sum_{i=1}^{k} (x_{k+i+1} - x_i) \geq \sum_{i=1}^{k} x_{k+i+1} - x_i$$

where we have used the fact that $|z| \geq 0$ always. Notice that here, equality holds when $x = x_{k+1} = x_{\lceil \frac{n}{2} \rceil}$. Indeed, for $x = x_{k+1}$,

$$\sum_{i=1}^{2k+1} |x - x_i| = \sum_{i=1}^{k} (x_{k+1} - x_i) + (x_{k+i+1} - x_{k+1}) = \sum_{i=1}^{k} x_{k+i+1} - x_i$$

Thus, in either case, equality is achieved when $x = x_{\lceil \frac{n}{2} \rceil}$ completing the proof of the lemma. □
Now, we will try to solve a superset of the original problem. Instead of just finding locations $C[1..k]$ of the testing stations, we also output for each resident i the index of the testing station they must go to. Call these indices $T[1..n]$. The total distance travelled by i^{th} resident $D'(i)$ is

$$D'(i) = |A[i] - C[T[i]]|$$

and we wish to find T, C such that $\sum_i^n D'(i)$ is minimized. Notice that solving this problem is equivalent to solving the original problem since for any given C, the values of T that minimize distances will always be those given in the statement of the problem. Thus, it is sufficient to solve this modified problem and return only the array C.

1. **Description of sub-problems:** Let BestPlacements(i, j) be the problem of placing and assigning j testing centers for the residents $A[i..n]$ such that the total distance to testing centres is minimized. Notice that the solution to the original problem is simply BestPartition$(1, k)$. We use two arrays to memoize – DP[i, j] stores the minimum distance and NUM[i, j] stores the number of residents assigned to the leftmost testing station. We assume that $i = n + 1$ corresponds to an empty array.
2. **Base Case(s):** The base cases are when $i = n + 1$ or $j = 0$. When $i = n + 1$, since there are no residents, the distances DP$[i, j] = 0$ and since no residents are assigned to the leftmost center NUM$[i, j] = 0$ as well. If $j = 0$ but $i < n + 1$ then there are no testing centers whereas the array A is non-empty. Therefore, in this case, DP$[i, j] = \infty$ and NUM$[i, j] = 0$.
3. **Recursion with justification:** When solving BestPlacements(i, j) we must choose how many residents we will assign to the leftmost testing center. Suppose we choose $0 \le l \le n - i + 1$ residents. Using Lemma 5.1, we get that the optimal placement of the leftmost testing center for the array $\{x_i, x_{i+1}, \ldots, x_{i+l-1}\}$ is $x_{\lceil \frac{2i+l-2}{2} \rceil}$. Once we have made this choice, we must assign $j - 1$ testing centers to the remaining $A[(i + l)..n]$ residents. This is exactly the problem BestPlacements$(i + l, j - 1)$. Therefore, the optimal placements is the minimum distance over all values l. This gives us the following recursion

$$\text{DP}[i, j] = \min_{1 \le l \le n-i+1} \left\{ \sum_{r=i}^{i+l-1} \left| x_r - x_{\lceil \frac{2i+l-2}{2} \rceil} \right| + \text{DP}[i + l, j - 1] \right\}$$

$$\text{NUM}[i, j] = \operatorname*{argmin}_{1 \le l \le n-i+1} \left\{ \sum_{r=i}^{i+l-1} \left| x_r - x_{\lceil \frac{2i+l-2}{2} \rceil} \right| + \text{DP}[i + l, j - 1] \right\}$$

4. **Order in which sub-problems are solved:** Notice that from the recurrence we infer that we can solve BestPlacements(i, j) if know the solutions to BestPlacements(i', j') for all $i' > i$. Thus, we can solve the subproblems in lexicographically decreasing order of (i, j).
5. **Form of the output:** The output is an array C of the positions of the testing centers. To find C, we can start with the solution to BestPlacements$(1, k)$ and then look at NUM$[1, k]$ to see how many people must be assigned to the leftmost testing camp. Using lemma 5.1, this will give us the location of the leftmost testing camp. We can then look at the remaining subproblem and recursively find the location of the leftmost testing camp using a similar procedure and continue until we have found all the locations. This is outlined in the BestPlacements function below.

6. **Pseudocode:** The BestPlacements function below solves this problem.

Algorithm 4 Best placements.

```
function BestPlacements($(A, n, k)$)
    Declare 2D empty array DP
    Declare 1D empty array NUM

    for $j$ from 0 to $k$ do                                          ▷ Base case
        DP$[n+1, j] = j$
        NUM$[n+1, j] = 0$
    end for


    for $i$ from 1 to $n$ do                                          ▷ Base case
        DP$[i, 0] = \infty$
        NUM$[n+1, j] = 0$
    end for

    for $i$ from $n$ to 1 do                                          ▷ Recursion
        for $j$ from 1 to $k$ do
            DP$[i, j] = \infty$
            for $l$ from 1 to $n - i + 1$ do
                tmp $= \sum_{r=i}^{i+l-1} \left| A[r] - A\left[\lceil \frac{2i+l-2}{2} \rceil\right] \right| +$ DP$[i+l, j-1]$

                if DP$[i, j] >$ tmp then
                    DP$[i, j] =$ tmp
                    NUM$[i, j] = l$
                end if
            end for
        end for
    end for

    Declare empty array $C$ initialized with zeros
    $j = k$
    $i = 1$

    while $j \geq 0$ do
        $l =$ NUM$[i, j]$
        $C[k - j + 1] = A\left[\lceil \frac{2i+l-2}{2} \rceil\right]$
        $j = j - 1$
        $i = i + l$
    end while

    return $C$
end function
```

7. **Runtime Analysis:** The lines 2-3 are constant time. The for loop 5-8 runs $O(k)$ times, and the loop 11-14 runs $O(n)$ times. There are three nested loops 16-28, 17-27 and 18-26 that run respectively $O(n), O(k), O(n)$ times. Since line 20 takes $O(n)$ time, the overall runtime is $O(n^3k)$. The final while loop runs in $O(k)$ time. Therefore, the overall runtime is $O(n^3k)$.

   We note that the runtime can be improved to $O(n^2k)$ by employing several tricks as outlined below. Accordingly, we present an optimized $O(n^2k)$ version of the above function called BestPlacementsOptimized.

- Suppose an array SuffixDistanceSum is defined such that

$$\text{SuffixDistanceSum[i]} = \sum_{j=i}^{n}(x_n - x_i)$$

Notice that the array SuffixDistanceSum can be computed in $O(n)$ by an iterative procedure (outlined in pseudocode). Using this, we can compute in $O(1)$ the following sum as shown,

$$\text{DistanceSum}(l, r) = \sum_{j=l}^{r}(x_r - x_j) = \sum_{j=l}^{r}(x_n - x_j) + \sum_{j=l}^{r}(x_r - x_n)$$

$$\implies \text{DistanceSum}(l, r) = \left[\sum_{j=l}^{n}(x_n - x_j) - \sum_{j=r+1}^{n}(x_n - x_j)\right] + (r - l + 1)(x_r - x_n)$$

$$\implies \text{DistanceSum}(l, r) = (\text{SuffixDistanceSum}[l] - \text{SuffixDistanceSum}[r + 1]) + (r - l + 1)(x_r - x_n$$

- Using the previous observation, we can compute in $O(1)$ for any subarray of people $\{x_l, \ldots, x_r\}$ the minimum distance to an optimally placed testing centre. From lemma 5.1, we know that the optimal location is x_p where $p = \lceil \frac{r+l-1}{2} \rceil$. Call the sum of distances to this MinDist(l, r). This means that,

$$\text{MinDist}(l, r) = \sum_{i=l}^{r} |x_p - x_i| = \sum_{i=l}^{p}(x_p - x_i) + \sum_{i=p}^{r}(x_i - x_p)$$

$$\implies \text{MinDist}(l, r) = \text{DistanceSum}(l, p) + \left[\sum_{i=p}^{r}(x_i - x_r) + (x_r - x_p)(r - p + 1)\right]$$

$$\implies \text{MinDist}(l, r) = \text{DistanceSum}(l, p) + \text{DistanceSum}(p, r) + (x_r - x_p)(r - p + 1)$$

Therefore, we can replace the summation in line 20 with a call to MinDist which makes the overall runtime $O(n^2k)$.

Algorithm 5 Optimized best placements

```
function PREPROCESS($A, n, k$)
    Declare global array SuffixDistanceSum
    SuffixDistanceSum[$n$] = 0
    for $i$ from $n - 1$ to 1 do
        SuffixDistanceSum[$i$] = SuffixDistanceSum[$i + 1$] + $(x_n - x_i)$
    end for
end function

function DISTANCESUM($l, r, A, n, k$)
    return SuffixDistanceSum[$l$] - SuffixDistanceSum[$r + 1$] + $(r - l + 1)(A[r] - A[n])$
end function
(contd..)
```

```
function MinDist($l, r, A, n, k$)
    $p = \lceil \frac{l+r-1}{2} \rceil$

    return DistanceSum($l, p, A, n, k$) + DistanceSum($p, r, A, n, k$) + $(A[r] - A[p])(r - p + 1)$
end function

function BestPlacementsOptimized($A, n, k$)
    Preprocess($A, n, k$)
    Declare 2D empty array DP
    Declare 1D empty array NUM

    for $j$ from 0 to $k$ do                                          ▷ Base case
        DP$[n+1, j] = j$
        NUM$[n+1, j] = 0$
    end for


    for $i$ from 1 to $n$ do                                          ▷ Base case
        DP$[i, 0] = \infty$
        NUM$[n+1, j] = 0$
    end for

    for $i$ from $n$ to 1 do                                          ▷ Recursion
        for $j$ from 1 to $k$ do
            DP$[i, j] = \infty$
            for $l$ from 1 to $n - i + 1$ do
                tmp = MinDist($i, i + l - 1, A, n, k$) +DP$[i + l, j - 1]$

                if DP$[i, j] >$ tmp then
                    DP$[i, j] =$ tmp
                    NUM$[i, j] = l$
                end if
            end for
        end for
    end for

    Declare empty array $C$ initialized with zeros
    $j = k$
    $i = 1$
    while $j \geq 0$ do
        $l =$ NUM$[i, j]$
        $C[k - j + 1] = A\left[\lceil \frac{2i+l-2}{2} \rceil\right]$
        $j = j - 1$
        $i = i + l$
    end while

    return $C$
end function
```

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.

Solution: Let the subproblem's result be BestPartition(i, j) and be stored in the array DP$[i, j]$.

1. **Description of sub-problems:** BestPartition(i, j) is the minimum possible value of $|S_1 + j - S_2|$ where S_1, S_2 is the sum of elements in I_1, I_2 which in turn is a partition of $\{x_i, x_{i+1}, \ldots, x_n\}$. When $i = n + 1$, we assume that I_1, I_2 is a partition of the empty set. Notice that the solution to the original problem is simply BestPartition$(1, 0)$.
2. **Base Case(s):** The base cases are when $i = n + 1$. In this case, clearly BestPartition$(i, j) = j$ since I_1, I_2 are always empty and therefore $S_1 = S_2 = 0$ implying $|S_1 + j - S_2| = j$ always.
3. **Recursion with justification:** The recursion for $i < n + 1$ is

$$\text{BestPartition}(i, j) = \min(\text{BestPartition}(i + 1, j + x_i), \text{BestPartition}(i + 1, j - x_i))$$

   To justify this, suppose that we wish to solve the subproblem BestPartition(i, j) by partitioning the array $\{x_i, x_{i+1}, \ldots, x_n\}$ into two sets I_1, I_2. When placing the element x_i there are two choices

   Choice 1: x_i is placed in I_1. In this case, we wish to divide the remaining elements $\{x_{i+1}, x_{i+2}, \ldots, x_n\}$ into two sets such that $|S_1 + j - S_2|$ is minimized. We can write,

$$|S_1 + j - S_2| = |(S_1 - x_i) + x_i + j - S_2|$$

   Notice that $S_1 - x_i$ is the sum of all elements in I_1 apart from x_i. Since $I_1 \setminus \{x_i\}, I_2$ is a partition of $\{x_{i+1}, \ldots, x_n\}$, we see that the problem of dividing the remaining elements is the same as partitioning $\{x_{i+1}, x_{i+2}, \ldots, x_n\}$ into I_1', I_2' such that $|S_1' + j + x_i - S_2'|$ is minimized where S_1', S_2' are the sum of elements in I_1', I_2' respectively. This is exactly the problem BestPartition$(i + 1, j + x_i)$.

   Choice 2: x_i is placed in I_2. In this case, we wish to divide the remaining elements $\{x_{i+1}, x_{i+2}, \ldots, x_n\}$ into two sets such that $|S_1 + j - S_2|$ is minimized. We can write,

$$|S_1 + j - S_2| = |S_1 - x_i + j - (S_2 - x_i)|$$

   Notice that $S_2 - x_i$ is the sum of all elements in I_2 apart from x_i. From a similar reasoning as in the previous case we see that the minimum possible value of this is BestPartition$(i + 1, j - x_i)$.

   From the above analysis, we conclude that upon placing x_i into I_1, the best partition we obtain has value BestPartition$(i + 1, j + x_i)$ and upon placing it into I_2 the best value is BestPartition$(i + 1, j - x_i)$. Clearly, the best way of partitioning $\{x_i, \ldots, x_n\}$ is to choose the smaller of the two, giving us the desired recurrence relation.
4. **Order in which sub-problems are solved:** From the recurrence we notice that BestPartition(i, j) can be computed if we know the values of BestPartition$(i + 1, l)$ for all l. Thus, we can solve the problems in lexicographically decreasing order of (i, j).
5. **Form of the output:** The output is an integer, the minimum possible value of $|S_1 - S_2|$.

6. **Pseudocode:** The BestPartition function below solves this problem. Notice that we can restrict j to lie between $-nK$ and nK since the index j is a sum of the form $(\pm x_1) + (\pm x_2) \ldots + (\pm x_m)$ for $1 \le m \le n$ and each element lies between 0 and K.

Algorithm 6 Best partition.

```
function BestPartition($A, n, K$)
    Declare 2D empty array DP

    for $j$ from $-nK$ to $nK$ do                                        ▷ Base case
        DP[$n + 1, j$] = $j$
    end for

    for $i$ from $n$ to 1 do                                             ▷ Recursion
        for $j$ from $-nK$ to $nK$ do
            DP[$i, j$] = min(DP[$i + 1, j + A[i]$], DP[$i + 1, j - A[i]$])
        end for
    end for

    return DP[1, 0]
end function
```

7. **Runtime Analysis:** The for loop in 4-6 runs $2nK$ times and line 5 runs in $O(1)$ time therefore the overall time is $O(nK)$. Likewise, the loop 8-12 runs n times while for each iteration of the outer loop, the inner one runs $2nK$ times. This means line 10 runs $O(n^2K)$ times. Since each run of line 10 is $O(1)$, the overall runtime of lines 8-12 is $O(n^2K)$. The overall runtime of BestPartition therefore is $O(n^2K)$.