- Please note that the solution may not be as per the required format for answering questions. Also note that the solution may be giving more than what is required in the question (e.g., algorithm for optimal solution in addition to optimal value).

There are 6 questions for a total of 120 points.

---

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

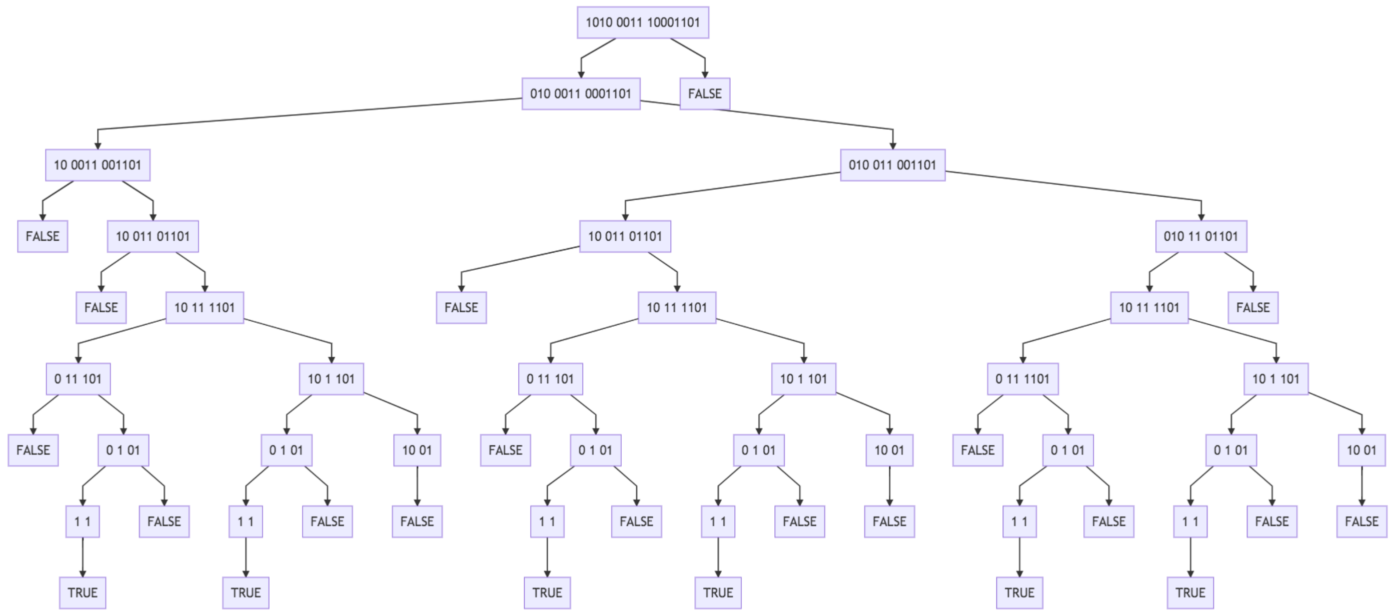
   Consider two binary strings $x = x_1, ...x_n$ and $y = y_1, ..., y_m$. An interleaving of x and y is a strings $z = z_1, ..., z_{n+m}$ so that the bit positions of z can be partitioned into two disjoint sets X and Y, so that looking only at the positions in X, the sub-sequence of z produced is x and 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.



   (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: Note that we might have to perform both recursive calls if $x_1 = y_1 = z_1$. So the above algorithm takes $O(2^{n+m})$ time.

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

Solution: However, as we run through the different recursive calls, we see a pattern: all recursive calls are of the form $RecIL(x_I..x_n, y_J...y_m, z_{I+J-1}...z_{n+m})$, where we use $I = n+1$ to mean the first input is empty, and similarly for $J = m+1$.

Thus, we have only about nm distinct calls.

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

Solution: We can use this to define a matrix of booleans, $IL[1..n+1][1..m+1]$ where we intend $IL[I][J]$ to store the answer to $RecIL(x_I..x_n, y_J..y_m, z_{I+J-1}..z_{n+m})$. The above recursion becomes:

```
1. $IL[n+1, m+1] := True$.
2. For $I = n+1$ and $1 \leq J \leq m$ : IF $y_J..y_m = z_{n+J}..z_{n+m}$ THEN $IL[n+1, J] := True$ ELSE $IL[n+1, J] := False$
3. For $J = m+1$ and $1 \leq I \leq n$ : IF $x_I..x_n = z_{m+I}..z_{n+m}$ THEN $IL[I, m+1] := True$ ELSE $IL[I, m+1] := False$
4. If both $1 \leq I \leq n$ and $1 \leq J \leq m$:
5.      $IL[I, J] := False$
6.    IF $z_{I+J-1} = x_I$ and $IL[I+1, J] = TRUE$ THEN $IL[I, J] := True$.
7.    IF $z_{I+J-1} = y_J$ and $IL[I, J+1] = TRUE$ THEN $IL[I, J] := True$
```

In the recursion, we either increase I or J. So a bottom-up order must decrease both I and J, from the largest values to the smallest.
So we finally get: BTIL$[x_1..x_n, y_1..y_m, z_1..z_{n+m}$

```
1. $IL[n+1, m+1] := True$.
2. FOR $J = 1$ to $m$ do: IF $y_J..y_m = z_{n+J}..z_{n+m}$ THEN $IL[n+1, J] := True$ ELSE $IL[n+1, J] := False$
3. FOR $I = 1$ to $n$ do: IF $x_I..x_n = z_{m+I}..z_{n+m}$ THEN $IL[I, m+1] := True$ ELSE $IL[I, m+1] := False$
4. FOR $I = n$ downto 1 do:
5.    FOR $J = m$ downto 1 do:
6.      $IL[I, J] := False$
7.    IF $z_{I+J-1} = x_I$ and $IL[I+1, J] = TRUE$ THEN $IL[I, J] := True$.
8.    IF $z_{I+J-1} = y_J$ and $IL[I, J+1] = TRUE$ THEN $IL[I, J] := True$
9. Return $IL[1, 1]$.
```

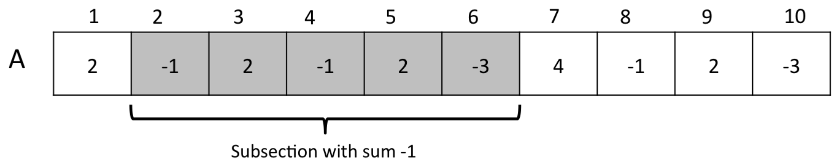
(e) (4 points) Give a time analysis for your DP algorithm

Solution: The bulk of the work is in the nested for loops, the inside of which is constant time. Thus, the overall time is $O(nm)$.

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.

Solution: Let $M(i)$ denote the maximum sum of the subsection of the array $A[1]...A[i]$ that includes $A[i]$. We can write the following recursive formulation for $M(.)$.

$$\forall i > 1, M(i) = \max\{M(i-1) + A[i], A[i]\} \quad \text{and} \quad M(1) = A[1]$$

This follows from the fact that the sum of subsection of $A[1]...A[i]$ that includes $A[i]$ either starts with $A[i]$ or starts at some index $j < i$. In the former case, the sum is $A[i]$ and in the latter case, the maximum sum is the maximum sum of subsection of $A[1]...A[i-1]$ that ends with $A[i-1]$ plus $A[i]$. Taking the maximum of the previous two options gives us the optimal result.

Here is the pseudocode for the algorithm based on the above recursive formulation.

```
MaxSubsectionValue($A, n$)
  - if all elements of array $A$ are negative, then return(0)
  - $M[1] \leftarrow A[1]$
  - $max \leftarrow M[1]$
  - for $i = 2$ to $n$
      - $M[i] \leftarrow \max\{M[i-1] + A[i], A[i]\}$
      - if ($M[i] > max$) $max \leftarrow M[i]$
  - return($max$)
```

Running time: The first line involves making one pass over the array and costs $O(n)$ time. The remaining algorithm has a for loop that runs for $O(n)$ iterations and costs constant time per iteration. So, the running time of the algorithm is $O(n)$.

Suppose you were also asked to output a maximum subsection. Here is how you can do this.

In order to output the subsection that maximizes the sum, we will need to maintain some more information. For every index i, we will store as $P[i]$ the starting index $j < i$ of the subsection of $A[1]...A[i]$ that maximizes the sum and that includes $A[i]$. Here is the pseudocode. The output $(0, 0)$ indicates that all elements of the array are negative and hence an *empty* subsection is a maximum subsection (*you will not lose any points if you have assumed that a subsection has at least one index*).

```
MaxSubsection($A, n$)
  - if all elements of array $A$ are negative, then return($(0, 0)$)
  - $M[1] \leftarrow A[1]$
  - $max \leftarrow M[1]$; $maxIndex \leftarrow 1$
  - $P[1] \leftarrow 1$
  - for $i = 2$ to $n$
      - if $(A[i] > M[i-1] + A[i])$
          - $M[i] \leftarrow A[i]$
          - $P[i] \leftarrow i$
      - else
          - $M[i] \leftarrow M[i-1] + A[i]$
          - $P[i] \leftarrow P[i-1]$
      - if $(M[i] > max)$
          - $max \leftarrow M[i]$
          - $maxIndex \leftarrow i$
  - return($(P[maxIndex], maxIndex)$)
```

Running time: The running time is the same as that of the previous algorithm since again, the algorithm involves a for loop with constant number of operations in every iteration of the loop.

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:

1. Description of sub-problems

   Let $MP(i, j)$ be maximum return among all j-shot strategies for i consecutive days with prices $p(1), ..., p(i)$.

2. Base Case(s)

   $MP(i, 0) = 0$ because a 0-shot strategy means that you never buy or sell. $MP(0, j) = 0$ because there are no days available to buy or sell. $MP(1, j) = 0$ because you cannot buy and sell over the course of only one day.

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)

$$MP(i, j) = \max\left\{MP(i-1, j), \max_{1 \leq h \leq i-1} \{100 \cdot (p(i) - p(h)) + MP(h-1, j-1)\}\right\}$$

   We will consider all possibilities by looking at several cases.

   Case 1: Suppose the maximum return does not require selling on day i. Then:

$$MP(i, j) = MP(i-1, j).$$

   Case 2: if the maximum return does require selling on day i then we must consider all possibilities for what was the last day you bought before you sold on day i. For example, if day h was the last day you bought before you sold on day i then:

$$MP(i, j) = 100 \cdot (p(i) - p(h)) + MP(h-1, j-1).$$

   This is because you are making $100 \cdot (p(i) - p(h))$ for the final buy and sell pair and you are making the maximum profit in the first $h - 1$ days with a $(j - 1)$-shot strategy which is $MP(h - 1, j - 1)$. Since we don't know up front which is the best day to buy before selling on day i, we must take the maximum over all possible days h before i: $1 \leq h \leq i - 1$. So, we get:

$$\max_{1 \leq h \leq i-1} \{100 \cdot (p(i) - p(h)) + MP(h-1, j-1)\}$$

   Finally, we don't know if Case 1 or Case 2 will give us the maximum return. So in the end, we must take the maximum over both cases.

4. Order in which sub-problems are solved

   The cell (i, j) depends on $(i - 1, j)$ and $(h - 1, j - 1)$ for $1 \leq h \leq i - 1$. So if your 2D-array is setup so i runs from 0 to n along the rows top to bottom and j runs from 0 to k along the columnass left to right then we have:

$(0, j-1)$	
$(1, j-1)$	
$(2, j-1)$	
$\vdots$	
	$(i-1, j)$
	(i, j)

and given the base cases, we can be sure to visit each of $(0, j-1)$,...,$(i-2, j-1)$ and $(i-1, j)$ before visiting (i, j) if we run from the top to the bottom and visiting each row from left to right, i.e., this order:

```
for $i = 2...n$:
  for $j = 1...n$:
```

5. Form of output (how do we get the final answer?)

   What the algorithm should output: We can return $MP(n, k)$ because by definition this is the maximum return among all k-shot strategies for i consecutive days with prices $p(1), ..., p(n)$.

6. Pseudocode

```
kShotProfit($p(1), ..., p(n); k$)
  - $MP(i, 0) = 0$ for all $0 \leq i \leq n$
  - $MP(0, j) = 0$ and $MP(1, j) = 0$ for all $0 \leq j \leq k$
  - for $i = 2...n$:
      - for $j = 1...k$:
          - $MP(i, j) = \max\{MP(i-1, j), \max_{1 \leq h \leq i-1}\{100 \cdot (p(i) - p(h)) + MP(h-1, j-1)\}\}$
  - return $MP(n, k)$
```

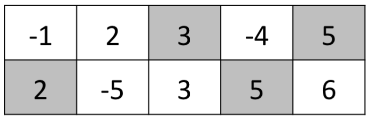
7. Runtime analysis

   This algorithm has a two for loops going through nk iterations with a procedure that takes (worst-case) $O(n)$ time per iteration and so the total runtime is $O(n^2 k)$.

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.

Solution: (*Please note that here we just give an outline of the main ideas which may be converted to a pseudocode.*)

We give a dynamic programming algorithm for this problem. As usual, we will first try to compute the maximum value that can be achieved given the matrix and then discuss how to get the locations that maximize the sum. We will use subproblems of the form $M(i, S)$ where i is an integer between 1 and n while S is a subset of $\{1, 2, 3, 4, 5\}$. Here is the definition of this subproblem:

$M(i, S)$: This is the maximum sum that can be obtained by considering numbers in all locations within rows $i + 1, ..., n$ and in row i, only numbers at columns $j \in S$.

Given this, here is our recursive formulation:

$$M(i,S) = \max_{C \subseteq \{1,...,5\}\setminus S, C \text{does not have neighbor indices}} \left[\sum_{j\in C} A(i,j) + M(i+1,C)\right]$$

$$M(n,S) = \max_{C \subseteq \{1,...,5\}\setminus S, C \text{does not have neighbor indices}} \left[\sum_{j\in C} A(i,j)\right]$$

The solution is given by $M(1, \{\})$. We compute row wise. The size of the table is $n \cdot 2^5$ and filling one table entry involves checking at most 2^5 possibilities. So the total time for filling the table is $O(n)$. For outputting the optimal locations, we can maintain a separate array that stores these locations.

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: For this question, we will first need to obtain an algorithm for $k = 1$. That is, an algorithm to determine the optimal location when only one center is opened. This is given in the following lemma.

Lemma 1: Let $A[1] \leq A[2] \leq ... \leq A[n]$. The index j such that $\sum_{i=1}^{n} |A[i] - A[j]|$ is minimised is the median index $\lceil \frac{n}{2} \rceil$.

Proof. Let $Cost(j)$ be defined as $Cost(j) = \sum_{i=1}^{n} |A[i] - A[j]|$. We will argue that:

$$Cost(1) \geq Cost(2) \geq ... \geq Cost(\lceil n/2 \rceil) \leq Cost(\lceil n/2 \rceil + 1) \leq ... \leq Cost(n).$$

Claim 1: For every $k < \lceil n/2 \rceil$, $Cost(k) \geq Cost(k + 1)$.

Proof. Let $d = A[k + 1] - A[k]$. Then we have:

$$\begin{aligned} Cost(k+1) - Cost(k) &= \left(\sum_{i=1}^{k+1}(A[k+1] - A[i]) + \sum_{i=k+2}^{n}(A[i] - A[k+1])\right) \\ &\quad - \left(\sum_{i=1}^{k}(A[k] - A[i]) + \sum_{i=k+1}^{n}(A[i] - A[k])\right) \\ &= \sum_{i=1}^{k} d + \sum_{i=k+1}^{n}(-d) \\ &\leq 0 \quad (\text{since } k < \lceil n/2 \rceil) \end{aligned}$$

□

Claim 2: For every $k > \lceil n/2 \rceil$, $Cost(k) \geq Cost(k - 1)$.

Proof. Let $d = A[k] - A[k-1]$. Then we have:

$$\begin{aligned}
Cost(k-1) - Cost(k) &= \left(\sum_{i=1}^{k-1}(A[k-1]-A[i]) + \sum_{i=k}^{n}(A[i]-A[k-1])\right) \\
&\quad - \left(\sum_{i=1}^{k}(A[k]-A[i]) + \sum_{i=k+1}^{n}(A[i]-A[k])\right) \\
&= \sum_{i=1}^{k-1}(-d) + \sum_{i=k}^{n} d \\
&\leq 0 \quad (\text{since } k > \lceil n/2 \rceil)
\end{aligned}$$

□

This completes the proof of Lemma 1. □

We will design a Dynamic Programming algorithm for this problem. Let $L(i, j)$ denote the minimum value of total distance travelled by the first i residents (i.e., residents $1, ..., i$) if j testing centers were to be opened **just for them**. (*For example, if* $A = [1, 2, 3, 7, 8, 9]$, *then* $L(4, 1) = 7$.)

First, we note that using the above lemma, we can write $L(i, 1)$ as follows

$$L(i,1) = \sum_{t=1}^{i} |A[t] - A[\lceil i/2 \rceil]|.$$

The location of residents can be put on a number line. Note that any set of k testing locations, partitions the residents on the number line into non-overlapping *interval sets* such that all residents in a partition report to their nearest testing center. From problem 4, we know that the optimal location (the one that minimises the total distance travelled by residents in the interval set) of testing center for an interval set is the median location. Consider the following recursive formulation:

$$L(i,j) = \min_{1 \leq t \leq i} \{L(t-1, j-1) + \sum_{l=t}^{i} |A[l] - A\left[\lfloor (t+i)/2 \rfloor\right]|\}$$

The above recursive relation holds since the optimal cost of covering all residents with j testing centers will be equal to the optimal cost of covering the residents of the last interval set plus the optimal cost of covering the remaining residents with $(j-1)$ centers and we check all possibilities for the last interval set.

The following table-filling algorithm outputs the minimum cost:

```
Cluster-cost($A, n, k$)
  - For $j = 0$ to $k$
    - $L[0, j] \leftarrow 0$
  - For $i = 1$ to $n$
    - $L[i, 1] \leftarrow \sum_{l=1}^{i} |A[l] - A[\lceil i/2 \rceil]|$
  - For $i = 1$ to $n$
    - For $j = 1$ to $k$
      - $L[i, j] = \min_{1 \leq t \leq i} \{L[t-1, j-1] + \sum_{l=t}^{i} |A[l] - A\left[\lfloor (t+i)/2 \rfloor\right]|\}$
  - return($L[n, k]$)
```

Running time: The time to compute the $L[i, j]$ is $O(i)$. So, the overall running time of the algorithm is $O(n^2 k)$.

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: We can solve this problem by *reducing* this problem to the 0/1 Knapsack problem that was discussed in the tutorial. Recall that in the 0/1 Knapsack problem, you were given n integers $\{y_1, ..., y_n\}$ (interpreted as weights) and a sack with capacity W and the goal was to design an algorithm that maximizes the weight of items picked subject to the total weight being $\leq W$. Let `Knapsack(`$(y_1, ..., y_n)$`, `W`)` denote the algorithm that returns the maximum value achievable. Here is the pseudocode for the algorithm for the given problem.

```
MinDiff($(x_1, ..., x_n)$)
    - $v \leftarrow \sum_{i=1}^{n} x_i$
    - $v_1 \leftarrow$ Knapsack($(x_1, ...., x_n), \lfloor v/2 \rfloor$)
    - return($v - 2 \cdot v_1$)
```

Proof of correctness (*you were not asked to give this*): Let v and v_1 be as defined in the algorithm. Suppose for the sake of contradiction, there is a partition I_1, I_2 with respect to which $|S_1 - S_2|$ is smaller than $((v - v_1) - v_1)$. In that case, consider the smaller partition of I_1 and I_2. This partition will have total weight $\leq \lfloor v/2 \rfloor$ and but larger than v_1. This contradicts with the optimality of the Knapsack algorithm.

Running time: The running time of the Knapsack algorithm is $O(n \cdot W)$ (where W is the weight of the sack). So, the running time of our algorithm is $O(n \cdot (nK)) = O(n^2 K)$ (since $v \leq nK$).