1. Given a sorted integer array, find all pairs with a given sum k in it using only O(1) extra space.
    a. Example:
    arr = [1, 2, 2, 2, 4, 5, 5, 5, 7]
    k=6
    Output: (1, 5) and (2, 4)

Algorithm:

1. Set Up Pointers: Place one pointer at the start of the array (“left”) and another at the end (“right”)
2. Iterate Through the Array: Continue as long as the “left” pointer is to the left of the “right” pointer:
    - Calculate Sum: Add the elements at the “left” and “right” pointers to find their sum. There are several cases:
        - If the sum equals k, we find a valid pair and put it in the output list. After that, we adjust both pointers towards the center (left+1, right-1) for other potential paris.
        - If the sum is less than k, that indicates we need some larger value to add to the right. We would move the left pointer to the right by one position to try to increase the sum.
        - If the sum is more than k, that indicates we need to decrease the sum. We do it by moving the right pointer left by one position.
3. Terminate: stop moving the pointer when left and right meet, which indicates we’ve exhausted our search.

Proof:

- Initially, left is at the start (index 0) and right is at the end (index n-1) of the array.
- The algorithm maintains the invariant that all elements to the left of left and all elements to the right of right have been processed and that pairs that sum to k have been found.
- Finally, the algorithm ends when left and right are both exhausted and we’ve considered all possible paris.

Time and Space Complexity Analysis:

- Each pointer at worst exhausts all elements in the array, which takes n time. Therefore, we have O(N) time complexity.
- For the space, it's O(1) because the two pointers use a constant number of additional variables(the pointers themselves) and they are independent of the array size.

3. Let $G = (V, E)$ be a directed graph with nodes $v_1, \ldots, v_n$. We say that G is an *ordered graph* if it has the following properties.

(i) Each edge goes from a node with a lower index to a node with a higher index. That is, every directed edge has the form (v_i, v_j) with $i < j$.

(ii) Each node except v_n has at least one edge leaving it. That is, for every node v_i, $i = 1, 2, \ldots, n-1$, there is at least one edge of the form (v_i, v_j).

The length of a path is the number of edges in it. The goal in this question is to solve the following problem (see Figure 6.29 for an example).

Given an ordered graph G, find the length of the longest path that begins at v_1 and ends at v_n.

(a) Show that the following algorithm does not correctly solve this problem, by giving an example of an ordered graph on which it does not return the correct answer.

`Set` $w = v_1$
`Set` $L = 0$

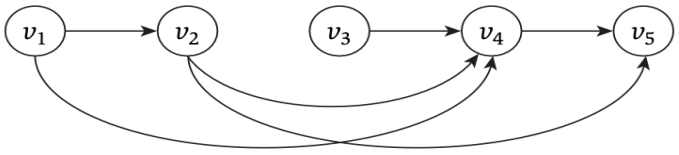


Figure 6.29 The correct answer for this ordered graph is 3: The longest path from v_1 to v_n uses the three edges (v_1, v_2),(v_2, v_4), and (v_4, v_5).

```
While there is an edge out of the node w
  Choose the edge (w, v_j)
     for which j is as small as possible
  Set w = v_j
  Increase L by 1
end while
Return L as the length of the longest path
```

In your example, say what the correct answer is and also what the algorithm above finds.

(b) Give an efficient algorithm that takes an ordered graph G and returns the *length* of the longest path that begins at v_1 and ends at v_n. (Again, the *length* of a path is the number of edges in the path.)

(a) The given algorithm does not work because for the example given: It will first visit v1. There are two edges out of v1(v2 and v3) and we choose the one with a smaller j which is v2. Then we move to the next node which is v2. There is only one edge out of it which is v5. According to the algorithm, we would move to v5, which leads to a false path from v1 -> v2 -> v5. The correct one should be v1-> v3 -> v4 -> v5.

(b) An efficient algorithm would be using dynamic programming.

- First initialize dp[i] which represents the length of the longest path from v1 to vi.
- Set dp[1] to 0, since the longest path from v1 to v1 has no edge. There is no negative edge from the condition given by the question.
- Then we continue to think about the rest of the nodes from v2 to vn. For each edge from v2 to vn, we look at every other node vj that connects to vi by one edge and is ahead(or less than) vi. If there is such vj, then:

  dp[i] = max (dp[i], dp[j]+1)
- We continue to move forward with the nodes until we reach vn since the formula above can be extended to any path. We return dp[n] at the end.

```
Initialize dp array with dp[1] = 0 and dp[i] = -∞ for i > 1.
For each node i from 2 to n:
   a. For each j from 1 to i-1:
      i. If there is an edge from v_j to v_i:
```

```
        - Update dp[i] = max(dp[i], dp[j] + 1)
Return dp[n]
```

Poof:
Assume that the algorithm correctly computes the longest paths for all vertices v_1 to v_k. We need to show that it correctly computes the longest path for vertex v_k+1 .

Consider vertex v_[k+1] . The algorithm examines all vertices v_j and updates dp[k+1] like below:

dp[k+1] = max(dp[k+1], dp[j] + 1)

This update considers all possible edges ending at v_[k+1] and adds 1 to the length of the longest path ending at each v_j . By the inductive hypothesis, dp[j] correctly represents the length of the longest path from v_1 to v_j . Therefore, adding 1 to dp[j] gives the correct length of the path from v_1 to v_[k+1] via (v_j).

Time Complexity:

- The outer loop from 2 to n takes O(n) times.
- For each iteration of the loop, we need to check every other previous node vj where j<i, and this at worst is 0(n-1).
- Therefore, with those two loops, our runtime is 0(n^2)

5. As some of you know well, and others of you may be interested to learn, a number of languages (including Chinese and Japanese) are written without spaces between the words. Consequently, software that works with text written in these languages must address the *word segmentation problem*—inferring likely boundaries between consecutive words in the

text. If English were written without spaces, the analogous problem would consist of taking a string like "meetateight" and deciding that the best segmentation is "meet at eight" (and not "me et at eight," or "meet ate ight," or any of a huge number of even less plausible alternatives). How could we automate this process?

A simple approach that is at least reasonably effective is to find a segmentation that simply maximizes the cumulative "quality" of its individual constituent words. Thus, suppose you are given a black box that, for any string of letters $x = x_1x_2 \cdots x_k$, will return a number *quality*(x). This number can be either positive or negative; larger numbers correspond to more plausible English words. (So *quality*("*me*") would be positive, while *quality*("*ght*") would be negative.)

Given a long string of letters $y = y_1y_2 \cdots y_n$, a segmentation of y is a partition of its letters into contiguous blocks of letters; each block corresponds to a word in the segmentation. The *total quality* of a segmentation is determined by adding up the qualities of each of its blocks. (So we'd get the right answer above provided that *quality*("*meet*") + *quality*("*at*") + *quality*("*eight*") was greater than the total quality of any other segmentation of the string.)

Give an efficient algorithm that takes a string y and computes a segmentation of maximum total quality. (You can treat a single call to the black box computing *quality*(x) as a single computational step.)

(*A final note, not necessary for solving the problem:* To achieve better performance, word segmentation software in practice works with a more complex formulation of the problem—for example, incorporating the notion that solutions should not only be reasonable at the word level, but also form coherent phrases and sentences. If we consider the example "theyouthevent," there are at least three valid ways to segment this into common English words, but one constitutes a much more coherent phrase than the other two. If we think of this in the terminology of formal languages, this broader problem is like searching for a segmentation that also can be parsed well according to a grammar for the underlying language. But even with these additional criteria and constraints, dynamic programming approaches lie at the heart of a number of successful segmentation systems.)

1. Frist initilize dp[0] = 0, we know an empty string has no word, thus the quality is 0.
2. We iteration through all position i from 1 to the length of the string(n), calculate dp[i] by considering all possible positions of j (0<= j <i) where the segment before could possibly end and j+1 would possibly start a new segment
3. For each j the quality of the substring from j+1 to i is [j+i : i], we can come up with the formula from calculating dp[i]:

    dp[i] = max(dp[i], dp[j]+quality(y[j+1:1]))

```python
def segment_string(y, quality):
    n = len(y)
    dp = [float('-inf')] * (n + 1)
    dp[0] = 0
    for i in range(1, n + 1):
        for j in range(i):
            segment = y[j:i]
            dp[i] = max(dp[i], dp[j] + quality(segment))
    return dp[n]
```

Proof:
By induction, the algorithm correctly computes the maximum quality score for segmenting any substring $y[0:i]$ for all i from 0 to n. The use of the dp array ensures that all possible segmentations are considered, and the use of the backtrack array allows for the optimal segmentation to be reconstructed. Therefore, the algorithm is correct and produces the optimal segmentation of the string.

Time Complexity:
There are two loops in our algorithm, the outer loop takes o(n) runtime and the inner one is also o(n) in the worse case. For the line dp[i] = max(dp[i], dp[j]+quality(y[j+1:1])) it is getting value, which takes constant o(1) time. Therefore, the total runtime should be o(n^2).

10. You're trying to run a large computing job in which you need to simulate a physical system for as many discrete *steps* as you can. The lab you're working in has two large supercomputers (which we'll call A and B) which are capable of processing this job. However, you're not one of the high-priority users of these supercomputers, so at any given point in time, you're only able to use as many spare cycles as these machines have available.

Here's the problem you face. Your job can only run on one of the machines in any given minute. Over each of the next n minutes, you have a "profile" of how much processing power is available on each machine. In minute i, you would be able to run $a_i > 0$ steps of the simulation if your job is on machine A, and $b_i > 0$ steps of the simulation if your job is on machine B. You also have the ability to move your job from one machine to the other; but doing this costs you a minute of time in which no processing is done on your job.

So, given a sequence of n minutes, a *plan* is specified by a choice of A, B, or "*move*" for each minute, with the property that choices A and

B cannot appear in consecutive minutes. For example, if your job is on machine A in minute i, and you want to switch to machine B, then your choice for minute $i+1$ must be *move*, and then your choice for minute $i+2$ can be B. The *value* of a plan is the total number of steps that you manage to execute over the n minutes: so it's the sum of a_i over all minutes in which the job is on A, plus the sum of b_i over all minutes in which the job is on B.

The problem. Given values $a_1, a_2, \ldots, a_n$ and $b_1, b_2, \ldots, b_n$, find a plan of maximum value. (Such a strategy will be called *optimal*.) Note that your plan can start with either of the machines A or B in minute 1.

Example. Suppose $n = 4$, and the values of a_i and b_i are given by the following table.

	Minute 1	Minute 2	Minute 3	Minute 4
A	10	1	1	10
B	5	1	20	20

Then the plan of maximum value would be to choose A for minute 1, then *move* for minute 2, and then B for minutes 3 and 4. The value of this plan would be $10 + 0 + 20 + 20 = 50$.

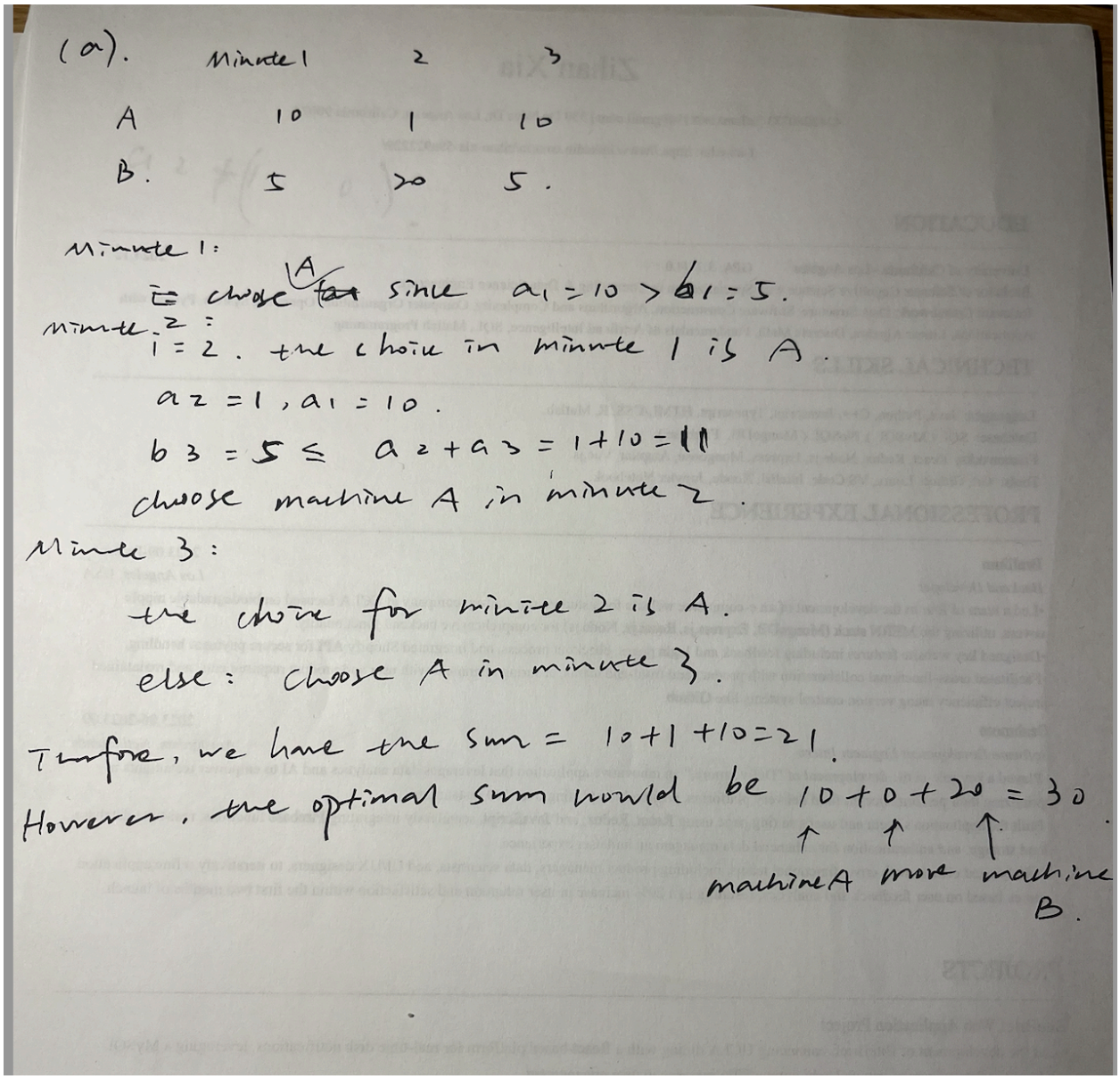
(a) Show that the following algorithm does not correctly solve this problem, by giving an instance on which it does not return the correct answer.

```
In minute 1, choose the machine achieving the larger of a_1, b_1
Set i = 2
While i ≤ n
   What was the choice in minute i − 1?
   If A:
      If b_{i+1} > a_i + a_{i+1} then
         Choose move in minute i and B in minute i + 1
         Proceed to iteration i + 2
      Else
         Choose A in minute i
         Proceed to iteration i + 1
      Endif
   If B: behave as above with roles of A and B reversed
EndWhile
```

In your example, say what the correct answer is and also what the algorithm above finds.

(b) Give an efficient algorithm that takes values for $a_1, a_2, \ldots, a_n$ and $b_1, b_2, \ldots, b_n$ and returns the *value* of an optimal plan.

(a)



(b)

Algorithm:

- **Inputs**: Two arrays a and b where:
  - a[i] is the number of steps that can be executed on machine A at minute i.
  - b[i] is the number of steps that can be executed on machine B at minute i.
- **DP Arrays**:
  - dpA[i]: The maximum number of steps that can be executed if the job is on machine A at minute i.

- dpB[i]: The maximum number of steps that can be executed if the job is on machine B at minute i.

1. **Base Case**:
    - For the first minute (minute 1):
        - dpA[0] = a[0] because the job starts on machine A at minute 1.
        - dpB[0] = b[0] because the job starts on machine B at minute 1.

Recurrence Relations

For each subsequent minute i (starting from minute 2):

2. **Stay on the Same Machine**:
    - If we stay on machine A, the total steps at minute i would be the steps at minute i-1 on machine A plus the steps at minute i on machine A:
        - dpA[i] = dpA[i-1] + a[i]
    - If we stay on machine B, the total steps at minute i would be the steps at minute i-1 on machine B plus the steps at minute i on machine B:
        - dpB[i] = dpB[i-1] + b[i]
3. **Switch Machines**:
    - If we switch from machine B to machine A, we need to skip a minute, hence considering i-2:
        - dpA[i] = dpB[i-2] + a[i]
    - If we switch from machine A to machine B, we need to skip a minute, hence considering i-2:
        - dpB[i] = dpA[i-2] + b[i]
4. **Combine choice and Take the max**:
    - We take the maximum value between staying and switching for both machines:
        - dpA[i] = max(dpA[i-1] + a[i], dpB[i-2] + a[i])
        - dpB[i] = max(dpB[i-1] + b[i], dpA[i-2] + b[i])

Proof:

1. **Base Case**: At minute 1, the algorithm initializes the correct values for dpA and dpB.
2. **Inductive Step**: For each subsequent minute, the algorithm considers both staying on the same machine and switching machines, ensuring that all possible transitions are taken into account.
3. Therefore, we compare the maximum values at each step and get the optimal result.

Runtime Analysis:

The iteration from 1 ninute to n takes O(n) time, and updating the array takes constant time. Therefore, the total runtime is O(n).

5. Imagine you have a piece of fabric that is n inches long. You also have a list of prices for fabric strips of various lengths, where each price corresponds to a length shorter than n. Your task is to determine the maximum revenue you can generate by cutting the fabric into smaller strips and selling them.
   Example:

Length	1	2	3	4	5	6	7	8
Price	1	5	8	9	10	17	17	20

   Total length is 8 and the price of different lengths are given as above. The maximum obtainable amount is 22 (by dividing into two of 2 and 6 ounces).

We are going to use dynamic programming for this problem.

There are two possible cases:

1. Not cutting the fabric.
   - For each iteration, we need to include the possibility that not cutting the fabric creates more revenue, which is the case max_revenue[i].
2. Cutting the fabric.
   - If this is the case, then we need to consider all possible ways to cut the fabric into two or more pieces.

Algorithm:

- First we create a table for stroing the max revenume for each length, we are going to renew it after each iteration from length i to n.
- Initialize max_revenue[0] to 0 since length 0 gives us no revenue,
- For length i from 1 to n:

  For j from 1 to i-1:

  Max_revenue[i] = max(max_revenue[i], price[j]+max_revenue[i-j])

  Thus we are iterating through all possible cutting position j from 1 to i-1 for each i, and updating the max between the current max revenue and the max of the cutting by adding the revenue of cutting j and the price of the remaining i-j.

Proof:

By Induction:

Base Case: R[0] = 0

Inductive steps: If for an arbitrary length k less than 1(0<= k < i), we assume R[k] is calculating the max revenue so far correctly, then R[i] should also calculate the max revenue correctly.

Runtime Analysis:
There are two nested loops. The outer loop iterates through the length i of the fabric from 1 to n, with an inner loop running i times (from 1 to i-1) for each length i. Initializing the max_revenue array takes O(n) times and accessing the value of max_revenue[n] takes O(1) times. Thus the overall runtime would be O(n^2).

6. Consider a row of n coins of values $v_1 \ldots v_n$, where n is even. We play a game against an opponent by alternating turns (you can both see all coins at all times). In each turn, a player selects either the first or last coin from the row, removes it from the row permanently, and receives the value of the coin.
   We define the guaranteed winning value for a strategy as the amount of money we can win using that strategy, no matter what actions the opponent takes. Determine the maximum guaranteed winning value we can achieve, for the player that starts the game.
   - Example 1: [5, 3, 7, 3]: we pick 5, the opponent must pick one of the 3s. lastly, we pick 7. This approach guarantees 12 no matter what strategy the opponent uses.
   - Example 2: [8, 15, 3, 2]: we can maximize the guaranteed value by picking 2 first. The opponent has to pick 8 or 3, and we pick 15 afterwards.

Algorithm:

1. Initialize a 2D array dp of size n x n where n is the number of coins
2. For each i from 0 to n-1
   Set dp[i][i] = value of coin at i // Base case: Only one coin to pick
3. For each length from 1 to n-1 // length is the size of the sub-problems
   For each i from 0 to n-length-1
      Set j = i + length
      Choose a = dp[i+2][j] if i+2 <= j else 0 // When you take coin i and opponent takes coin i+1
      Choose b = dp[i+1][j-1] if i+1 <= j-1 else 0 // When you take coin i, opponent takes coin j, or vice versa
      Choose c = dp[i][j-2] if i <= j-2 else 0 // When you take coin j and opponent takes coin j-1
      Set dp[i][j] = max(value[i] + min(a, b), value[j] + min(b, c))
        // max of choosing the first or last coin and minimizing the best response by opponent
4. The result is dp[0][n-1] // This gives the maximum value the first player can guarantee

Proof:

Base Case:

1. Single Coin Scenario: When the subarray consists of a single coin, say at position i (so $i = j$), the optimal choice is trivially to take that coin, since no other choices exist. Thus, $dp[i][i] = v[i]$, where $v[i]$ is the value of the coin at position i. This establishes the base case of our induction.

Inductive Step:

Assume that for all subarrays of length k or less (where $k < l$), the entries $dp[i][j]$ have been correctly computed, meaning they reflect the maximum guaranteed value the starting player can achieve when choosing optimally.

We need to show that $dp[i][j]$ for any subarray of length l (where $j = i + l$) is also correctly computed using the values from subarrays of length less than l.

2. Choice Analysis:
   - When the player picks the first coin $v[i]$, the remaining subarray is $v[i+1]$ to $v[j]$. The opponent, playing optimally, could:
   - Take $v[i+1]$, leaving the subarray $v[i+2]$ to $v[j]$ (value a).
   - Take $v[j]$, leaving the subarray $v[i+1]$ to $v[j-1]$ (value b).

The worst case for the first player after picking $v[i]$ is the minimum of these two values, hence $dp[i][j]$ must consider $v[i] + \min(dp[i+2][j], dp[i+1][j-1])$.

   - Similarly, if the player picks the last coin $v[j]$, the analyses involve subarrays $v[i]$ to $v[j-1]$ and the worst case involves a decision by the opponent among coins $v[i]$ and $v[j-1]$, considered in the calculation $v[j] + \min(dp[i+1][j-1], dp[i][j-2])$.

3. Optimal Substructure:
   - The recursive choice thus considers the maximum value between taking the first or last coin and securing the minimum of the optimal plays available to the opponent. This guarantees that $dp[i][j]$ for length l is optimally computed based on the correctness of smaller subproblems.

Runtime Analysis:

- The outer loop runs for the length of the subarrays, which iterates $n-1$ times.
- The inner loop runs for each possible starting index i for a given length, which can be up to n times in total but decreases as the length increases.

Therefore, the runtime is $O(n^2)$.