CHAPTER

16 Dynamic Programming

The important fact to observe is that we have attempted to solve a maximization problem involving a particular value of x and a particular value of N by first solving the general problem involving an arbitrary value of x and an arbitrary value of N.

— "Dynamic Programming,"
R. E. BELLMAN, 1957

DP is a general technique for solving optimization, search, and counting problems that can be decomposed into subproblems. You should consider using DP whenever you have to make choices to arrive at the solution, specifically, when the solution relates to subproblems.

Like divide-and-conquer, DP solves the problem by combining the solutions of multiple smaller problems, but what makes DP different is that the same subproblem may reoccur. Therefore, a key to making DP efficient is caching the results of intermediate computations. Problems whose solutions use DP are a popular choice for hard interview questions.

To illustrate the idea underlying DP, consider the problem of computing Fibonacci numbers. The first two Fibonacci numbers are 0 and 1. Successive numbers are the sums of the two previous numbers. The first few Fibonacci numbers are $0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$. The Fibonacci numbers arise in many diverse applications—biology, data structure analysis, and parallel computing are some examples.

Mathematically, the nth Fibonacci number $F(n)$ is given by the equation $F(n) = F(n-1) + F(n-2)$, with $F(0) = 0$ and $F(1) = 1$. A function to compute $F(n)$ that recursively invokes itself has a time complexity that is exponential in n. This is because the recursive function computes some $F(i)$s repeatedly. Figure 16.1 on the next page graphically illustrates how the function is repeatedly called with the same arguments.

Caching intermediate results makes the time complexity for computing the nth Fibonacci number linear in n, albeit at the expense of $O(n)$ storage.

```
def fibonacci(n, cache={}):
    if n <= 1:
        return n
    elif n not in cache:
        cache[n] = fibonacci(n - 1) + fibonacci(n - 2)
    return cache[n]
```

Minimizing cache space is a recurring theme in DP. Now we show a program that computes $F(n)$ in $O(n)$ time and $O(1)$ space. Conceptually, in contrast to the above program, this program iteratively fills in the cache in a bottom-up fashion, which allows it to reuse cache storage to reduce the space complexity of the cache.

```
def fibonacci(n):
```

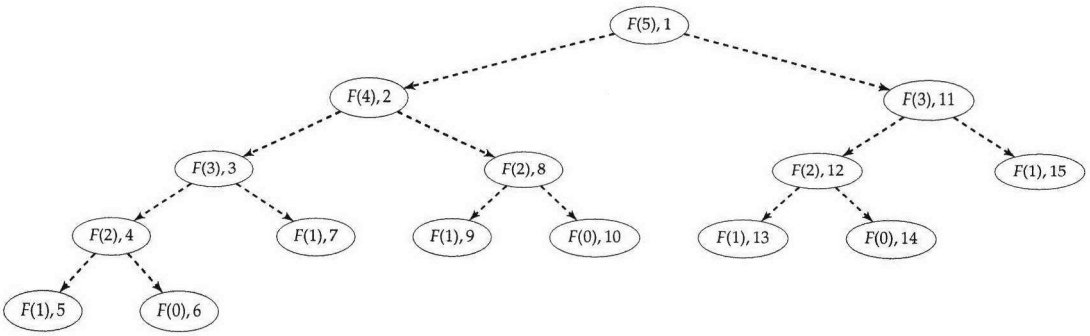


Figure 16.1: Tree of recursive calls when naively computing the 5th Fibonacci number, $F(5)$. Each node is a call: $F(x)$ indicates a call with argument x, and the italicized numbers on the right are the sequence in which calls take place. The children of a node are the subcalls made by that call. Note how there are 2 calls to $F(3)$, and 3 calls to each of $F(2)$, $F(1)$, and $F(0)$.

```
if n <= 1:
    return n

f_minus_2, f_minus_1 = 0, 1
for _ in range(1, n):
    f = f_minus_2 + f_minus_1
    f_minus_2, f_minus_1 = f_minus_1, f
return f_minus_1
```

The key to solving a DP problem efficiently is finding a way to break the problem into subproblems such that

- the original problem can be solved relatively easily once solutions to the subproblems are available, and
- these subproblem solutions are cached.

Usually, but not always, the subproblems are easy to identify.

Here is a more sophisticated application of DP. Consider the following problem: find the maximum sum over all subarrays of a given array of integer. As a concrete example, the maximum subarray for the array in Figure 16.2 starts at index 0 and ends at index 3.

904	40	523	12	-335	-385	-124	481	-31
$A[0]$	$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$	$A[6]$	$A[7]$	$A[8]$

Figure 16.2: An array with a maximum subarray sum of 1479.

The brute-force algorithm, which computes each subarray sum, has $O(n^3)$ time complexity—there are $\frac{n(n+1)}{2}$ subarrays, and each subarray sum takes $O(n)$ time to compute. The brute-force algorithm can be improved to $O(n^2)$, at the cost of $O(n)$ additional storage, by first computing $S[k] = \sum A[0,k]$ for all k. The sum for $A[i,j]$ is then $S[j] - S[i-1]$, where $S[-1]$ is taken to be 0.

Here is a natural divide-and-conquer algorithm. Take $m = \lfloor \frac{n}{2} \rfloor$ to be the middle index of A. Solve the problem for the subarrays $L = A[0,m]$ and $R = A[m+1, n-1]$. In addition to the answers for each, we also return the maximum subarray sum l for a subarray ending at the last entry in L, and the maximum subarray sum r for a subarray starting at the first entry of R. The maximum subarray

sum for A is the maximum of $l + r$, the answer for L, and the answer for R. The time complexity analysis is similar to that for quicksort, and the time complexity is $O(n \log n)$. Because of off-by-one errors, it takes some time to get the program just right.

Now we will solve this problem by using DP. A natural thought is to assume we have the solution for the subarray $A[0, n-2]$. However, even if we knew the largest sum subarray for subarray $A[0, n-2]$, it does not help us solve the problem for $A[0, n-1]$. Instead we need to know the subarray amongst all subarrays $A[0, i]$, $i < n-1$, with the smallest subarray sum. The desired value is $S[n-1]$ minus this subarray's sum. We compute this value by iterating through the array. For each index j, the maximum subarray sum ending at j is equal to $S[j] - \min_{k \leq j} S[k]$. During the iteration, we track the minimum $S[k]$ we have seen so far and compute the maximum subarray sum for each index. The time spent per index is constant, leading to an $O(n)$ time and $O(1)$ space solution. It is legal for all array entries to be negative, or the array to be empty. The algorithm handles these input cases correctly.

```
def find_maximum_subarray(A):
    min_sum = max_sum = 0
    for running_sum in itertools.accumulate(A):
        min_sum = min(min_sum, running_sum)
        max_sum = max(max_sum, running_sum - min_sum)
    return max_sum
```

Dynamic programming boot camp

The programs presented in the introduction for computing the Fibonacci numbers and the maximum subarray sum are good illustrations of DP.

16.1 Count the number of score combinations

In an American football game, a play can lead to 2 points (safety), 3 points (field goal), or 7 points (touchdown, assuming the extra point). Many different combinations of 2, 3, and 7 point plays can make up a final score. For example, four combinations of plays yield a score of 12:

- 6 safeties ($2 \times 6 = 12$),
- 3 safeties and 2 field goals ($2 \times 3 + 3 \times 2 = 12$),
- 1 safety, 1 field goal and 1 touchdown ($2 \times 1 + 3 \times 1 + 7 \times 1 = 12$), and
- 4 field goals ($3 \times 4 = 12$).

Write a program that takes a final score and scores for individual plays, and returns the number of combinations of plays that result in the final score.

Hint: Count the number of combinations in which there are 0 w_0 plays, then 1 w_0 plays, etc.

Solution: We can gain some intuition by considering small scores. For example, a 9 point score can be achieved in the following ways:

- scoring 7 points, followed by a 2 point play,
- scoring 6 points, followed by a 3 point play, and
- scoring 2 points, followed by a 7 point play.

Generalizing, an s point score can be achieved by an $s-2$ point score, followed by a 2 point play, an $s-3$ point score, followed by a 3 point play, or an $s-7$ point score, followed by a 7 point play. This gives us a mechanism for recursively enumerating all possible scoring sequences which lead to a given score. Note that different sequences may lead to the same score combination, e.g., a 2 point

Consider using DP whenever you have to **make choices** to arrive at the solution. Specifically, DP is applicable when you can construct a solution to the given instance from solutions to subinstances of smaller problems of the same kind.

In addition to optimization problems, DP is also **applicable** to **counting and decision problems**—any problem where you can express a solution recursively in terms of the same computation on smaller instances.

Although conceptually DP involves recursion, often for efficiency the cache is **built "bottom-up"**, i.e., iteratively. .

When DP is implemented recursively the cache is typically a dynamic data structure such as a hash table or a BST; when it is implemented iteratively the cache is usually a one- or multi-dimensional array.

To save space, **cache space** may be **recycled** once it is known that a set of entries will not be looked up again.

Sometimes, **recursion may out-perform a bottom-up DP** solution, e.g., when the solution is found early or subproblems can be **pruned** through bounding.
A common **mistake** in solving DP problems is trying to think of the recursive case by splitting the problem into **two equal halves**, *a la* quicksort, i.e., solve the subproblems for subarrays $A[0, \lfloor \frac{n}{2} \rfloor]$ and $A[\lfloor \frac{n}{2} \rfloor + 1, n]$ and combine the results. However, in most cases, these two subproblems are not sufficient to solve the original problem.

DP is based on **combining solutions** to subproblems to **yield a solution** to the original problem. However, for some problems DP will not work. For example, if you need to compute the longest path from City 1 to City 2 without repeating an intermediate city, and this longest path passes through City 3, then the subpaths from City 1 to City 3 and City 3 to City 2 may not be individually longest paths without repeated cities.

Table 16.1: Top Tips for Dynamic Programming

play followed by a 7 point play and a 7 point play followed by a 2 point play both lead to a final score of 9. A brute-force approach might be to enumerate these sequences, and count the distinct combinations within these sequences, e.g., by sorting each sequence and inserting into a hash table.

The time complexity is very high, since there may be a very large number of scoring sequences. Since all we care about are the combinations, a better approach is to focus on the number of combinations for each possible number of plays of a single type.

For example, if the final score is 12 and we are only allowed 2 point plays, there is exactly one way to get 12. Now suppose we are allowed both 2 and 3 point plays. Since all we care about are combinations, assume the 2 point plays come before the 3 point plays. We could have zero 2 point plays, for which there is one combination of 3 point plays, one 2 point play, for which there no combination of 3 point plays (since 3 does not evenly divide $12 - 2$, two 2 point plays, for which there no combination of 3 point plays (since 3 does not evenly $12 - 2 \times 2$), three 2 point plays, for which there is one combination of 3 point plays (since 3 evenly divides $12 - 2 \times 3$), etc. To count combinations when we add 7 point plays to the mix, we add the number of combinations of 2 and 3 that lead to 12 and to 5—these are the only scores from which 7 point plays can lead to 12.

Naively implemented, for the general case, i.e., individual play scores are $W[0], W[1], \ldots, W[n-1]$, and s the final score, the approach outlined above has exponential complexity because it repeat-

edly solves the same problems. We can use DP to reduce its complexity. Let the 2D array $A[i][j]$ store the number of score combinations that result in a total of j, using individual plays of scores $W[0], W[1], \ldots, W[i-1]$. For example, $A[1][12]$ is the number of ways in which we can achieve a total of 12 points, using 2 and/or 3 point plays. Then $A[i+1][j]$ is simply $A[i][j]$ (no $W[i+1]$ point plays used to get to j), plus $A[i][j - W[i+1]]$ (one $W[i+1]$ point play), plus $A[i][j - 2W[i+1]]$ (two $W[i+1]$ point plays), etc.

The algorithm directly based on the above discussion consists of three nested loops. The first loop is over the total range of scores, and the second loop is over scores for individual plays. For a given i and j, the third loop iterates over at most $j/W[i] + 1$ values. (For example, if $j = 10$, $i = 1$, and $W[i] = 3$, then the third loop examines $A[0][10], A[0][7], A[0][4], A[0][1]$.) Therefore number of iterations for the third loop is bounded by s. Hence, the overall time complexity is $O(sns) = O(s^2n)$ (first loop is to n, second is to s, third is bounded by s).

Looking more carefully at the computation for the row $A[i+1]$, it becomes apparent that it is not as efficient as it could be. As an example, suppose we are working with 2 and 3 point plays. Suppose we are done with 2 point plays. Let $A[0]$ be the row holding the result for just 2 point plays, i.e., $A[0][j]$ is the number of combinations of 2 point plays that result in a final score of j. The number of score combinations to get a final score of 12 when we include 3 point plays in addition to 2 point plays is $A[0][0] + A[0][3] + A[0][6] + A[0][9] + A[0][12]$. The number of score combinations to get a final score of 15 when we include 3 point plays in addition to 2 point plays is $A[0][0] + A[0][3] + A[0][6] + A[0][9] + A[0][12] + A[0][15]$. Clearly this repeats computation—$A[0][0] + A[0][3] + A[0][6] + A[0][9] + A[0][12]$ was computed when considering the final score of 12.

Note that $A[1][15] = A[0][15] + A[1][12]$. Therefore a better way to fill in $A[1]$ is as follows: $A[1][0] = A[0][0], A[1][1] = A[0][1], A[1][2] = A[0][2], A[1][3] = A[0][3] + A[1][0], A[1][4] = A[0][4] + A[1][1], A[1][5] = A[0][5] + A[1][2], \ldots$. Observe that $A[1][i]$ takes $O(1)$ time to compute—it's just $A[0][i] + A[1][i-3]$. See Figure 16.3 for an example table.

	0	1	2	3	4	5	6	7	8	9	10	11	12
2	1	0	1	0	1	0	1	0	1	0	1	0	1
2,3	1	0	1	1	1	1	2	1	2	2	2	2	3
2,3,7	1	0	1	1	1	1	2	2	2	3	3	3	4

Figure 16.3: DP table for 2, 3, 7 point plays (rows) and final scores from 0 to 12 (columns). As an example, for 2, 3, 7 point plays and a total of 9, the entry is the sum of the entry for 2, 3 point plays and a total of 9 (no 7 point plays) and the entry for 2, 3, 7 point plays and a total of 2 (one additional 7 point play).

The code below implements the generalization of this approach.

```
def num_combinations_for_final_score(final_score, individual_play_scores):
    # One way to reach 0.
    num_combinations_for_score = [[1] + [0] * final_score
                                  for _ in individual_play_scores]
    for i in range(len(individual_play_scores)):
        for j in range(1, final_score + 1):
            without_this_play = (num_combinations_for_score[i - 1][j]
                                 if i >= 1 else 0)
            with_this_play = (
```

```
            num_combinations_for_score[i][j - individual_play_scores[i]]
            if j >= individual_play_scores[i] else 0)
        num_combinations_for_score[i][j] = (
            without_this_play + with_this_play)
return num_combinations_for_score[-1][-1]
```

The time complexity is $O(sn)$ (two loops, one to s, the other to n) and the space complexity is $O(sn)$ (the size of the $2D$ array).

Variant: Solve the same problem using $O(s)$ space.

Variant: Write a program that takes a final score and scores for individual plays, and returns the number of sequences of plays that result in the final score. For example, 18 sequences of plays yield a score of 12. Some examples are $\langle 2,2,2,3,3\rangle$, $\langle 2,3,2,2,3\rangle$, $\langle 2,3,7\rangle$, $\langle 7,3,2\rangle$.

Variant: Suppose the final score is given in the form (s,s'), i.e., Team 1 scored s points and Team 2 scored s' points. How would you compute the number of distinct scoring sequences which result in this score? For example, if the final score is $(6,3)$ then Team 1 scores 3, Team 2 scores 3, Team 1 scores 3 is a scoring sequence which results in this score.

Variant: Suppose the final score is (s,s'). How would you compute the maximum number of times the team that lead could have changed? For example, if $s = 10$ and $s' = 6$, the lead could have changed 4 times: Team 1 scores 2, then Team 2 scores 3 (lead change), then Team 1 scores 2 (lead change), then Team 2 scores 3 (lead change), then Team 1 scores 3 (lead change) followed by 3.

Variant: You are climbing stairs. You can advance 1 to k steps at a time. Your destination is exactly n steps up. Write a program which takes as inputs n and k and returns the number of ways in which you can get to your destination. For example, if $n = 4$ and $k = 2$, there are five ways in which to get to the destination:

- four single stair advances,
- two single stair advances followed by a double stair advance,
- a single stair advance followed by a double stair advance followed by a single stair advance,
- a double stair advance followed by two single stairs advances, and
- two double stair advances.

16.2 Compute the Levenshtein distance

Spell checkers make suggestions for misspelled words. Given a misspelled string, a spell checker should return words in the dictionary which are close to the misspelled string.

In 1965, Vladimir Levenshtein defined the distance between two words as the minimum number of "edits" it would take to transform the misspelled word into a correct word, where a single edit is the *insertion*, *deletion*, or *substitution* of a single character. For example, the Levenshtein distance between "Saturday" and "Sundays" is 4—delete the first 'a' and 't', substitute 'r' by 'n' and insert the trailing 's'.

Write a program that takes two strings and computes the minimum number of edits needed to transform the first string into the second string.

Hint: Consider the same problem for prefixes of the two strings.

Solution: A brute-force approach would be to enumerate all strings that are distance $1,2,3,\ldots$ from the first string, stopping when we reach the second string. The number of strings may grow

enormously, e.g., if the first string is n 0s and the second is n 1s we will visit all of the 2^n possible bit strings from the first string before we reach the second string.

A better approach is to "prune" the search. For example, if the last character of the first string equals the last character of the second string, we can ignore this character. If they are different, we can focus on the initial portion of each string and perform a final edit step. (As we will soon see, this final edit step may be an insertion, deletion, or substitution.)

Let a and b be the length of strings A and B, respectively. Let $E(A[0, a-1], B[0, b-1])$ be the Levenshtein distance between the strings A and B. (Note that $A[0, a-1]$ is just A, but we prefer to write A using the subarray notation for exposition; we do the same for B.)

We now make some observations:

- If the last character of A equals the last character of B, then $E(A[0, a-1], B[0, b-1]) = E(A[0, a-2], B[0, b-2])$.
- If the last character of A is not equal to the last character of B then

$$E(A[0, a-1], B[0, b-1]) = 1 + \min \begin{pmatrix} E(A[0, a-2], B[0, b-2]), \\ E(A[0, a-1], B[0, b-2]), \\ E(A[0, a-2], B[0, b-1]) \end{pmatrix}$$

  The three terms correspond to transforming A to B by the following three ways:
  - Transforming $A[0, a-1]$ to $B[0, b-1]$ by transforming $A[0, a-2]$ to $B[0, b-2]$ and then substituting A's last character with B's last character.
  - Transforming $A[0, a-1]$ to $B[0, b-1]$ by transforming $A[0, a-1]$ to $B[0, b-2]$ and then adding B's last character at the end.
  - Transforming $A[0, a-1]$ to $B[0, b-1]$ by transforming $A[0, a-2]$ to $B[0, b-1]$ and then deleting A's last character.

These observations are quite intuitive. Their rigorous proof, which we do not give, is based on reordering steps in an arbitrary optimum solution for the original problem.

DP is a great way to solve this recurrence relation: cache intermediate results on the way to computing $E(A[0, a-1], B[0, b-1])$.

We illustrate the approach in Figure 16.4 on the facing page. This shows the E values for "Carthorse" and "Orchestra". Uppercase and lowercase characters are treated as being different. The Levenshtein distance for these two strings is 8.

```
def levenshtein_distance(A, B):
    def compute_distance_between_prefixes(A_idx, B_idx):
        if A_idx < 0:
            # A is empty so add all of B's characters.
            return B_idx + 1
        elif B_idx < 0:
            # B is empty so delete all of A's characters.
            return A_idx + 1
        if distance_between_prefixes[A_idx][B_idx] == -1:
            if A[A_idx] == B[B_idx]:
                distance_between_prefixes[A_idx][B_idx] = (
                    compute_distance_between_prefixes(A_idx - 1, B_idx - 1))
            else:
                substitute_last = compute_distance_between_prefixes(
                    A_idx - 1, B_idx - 1)
                add_last = compute_distance_between_prefixes(A_idx - 1, B_idx)
                delete_last = compute_distance_between_prefixes(
```

```
                A_idx, B_idx - 1)
            distance_between_prefixes[A_idx][B_idx] = (
                1 + min(substitute_last, add_last, delete_last))
        return distance_between_prefixes[A_idx][B_idx]

    distance_between_prefixes = [[-1] * len(B) for _ in A]
    return compute_distance_between_prefixes(len(A) - 1, len(B) - 1)
```

The value $E(A[0, a-1], B[0, b-1])$ takes time $O(1)$ to compute once $E(A[0, k], B[0, l])$ is known for all $k < a$ and $l < b$. This implies $O(ab)$ time complexity for the algorithm. Our implementation uses $O(ab)$ space.

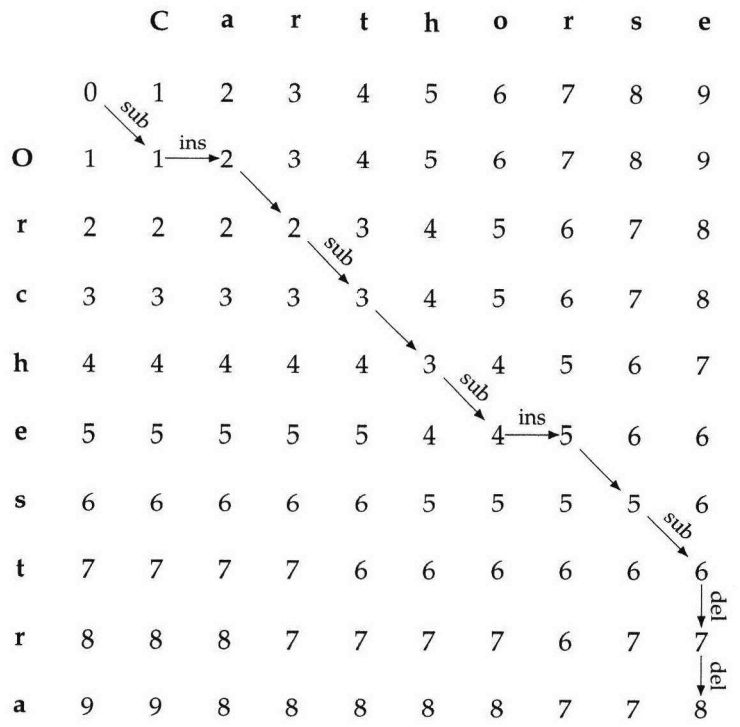


Figure 16.4: The E table for "Carthorse" and "Orchestra".

Variant: Compute the Levenshtein distance using $O(\min(a, b))$ space and $O(ab)$ time.

Variant: Given A and B as above, compute a longest sequence of characters that is a subsequence of A and of B. For example, the longest subsequence which is present in both strings in Figure 16.4 is $\langle r, h, s \rangle$.

Variant: Given a string A, compute the minimum number of characters you need to delete from A to make the resulting string a palindrome.

Variant: Given a string A and a regular expression r, what is the string in the language of the regular expression r that is closest to A? The distance between strings is the Levenshtein distance specified above.

Variant: Define a string t to be an interleaving of strings s_1 and s_2 if there is a way to interleave the characters of s_1 and s_2, keeping the left-to-right order of each, to obtain t. For example, if s_1

is "gtaa" and s_2 is "atc", then "gattaca" and "gtataac" can be formed by interleaving s_1 and s_2 but "gatacta" cannot. Design an algorithm that takes as input strings s_1, s_2 and t, and determines if t is an interleaving of s_1 and s_2.

16.3 Count the number of ways to traverse a 2D array

In this problem you are to count the number of ways of starting at the top-left corner of a 2D array and getting to the bottom-right corner. All moves must either go right or down. For example, we show three ways in a 5×5 2D array in Figure 16.5. (As we will see, there are a total of 70 possible ways for this example.)

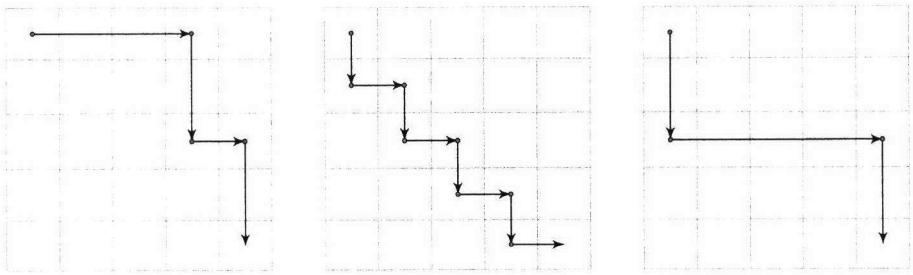


Figure 16.5: Paths through a 2D array.

Write a program that counts how many ways you can go from the top-left to the bottom-right in a 2D array.

Hint: If $i > 0$ and $j > 0$, you can get to (i, j) from $(i-1, j)$ or $(i, j-1)$.

Solution: A brute-force approach is to enumerate all possible paths. This can be done using recursion. However, there is a combinatorial explosion in the number of paths, which leads to huge time complexity.

The problem statement asks for the number of paths, so we focus our attention on that. A key observation is that because paths must advance down or right, the number of ways to get to the bottom-right entry is the number of ways to get to the entry immediately above it, plus the number of ways to get to the entry immediately to its left. Let's treat the origin $(0, 0)$ as the top-left entry. Generalizing, the number of ways to get to (i, j) is the number of ways to get to $(i-1, j)$ plus the number of ways to get to $(i, j-1)$. (If $i = 0$ or $j = 0$, there is only one way to get to (i, j) from the origin.) This is the basis for a recursive algorithm to count the number of paths. Implemented naively, the algorithm has exponential time complexity—it repeatedly recurses on the same locations. The solution is to cache results. For example, the number of ways to get to (i, j) for the configuration in Figure 16.5 is cached in a matrix as shown in Figure 16.6 on the facing page.

```
def number_of_ways(n, m):
    def compute_number_of_ways_to_xy(x, y):
        if x == y == 0:
            return 1

        if number_of_ways[x][y] == 0:
            ways_top = 0 if x == 0 else compute_number_of_ways_to_xy(x - 1, y)
            ways_left = 0 if y == 0 else compute_number_of_ways_to_xy(x, y - 1)
            number_of_ways[x][y] = ways_top + ways_left
```

```
    return number_of_ways[x][y]

number_of_ways = [[0] * m for _ in range(n)]
return compute_number_of_ways_to_xy(n - 1, m - 1)
```

The time complexity is $O(nm)$, and the space complexity is $O(nm)$, where n is the number of rows and m is the number of columns.

1	1	1	1	1
1	2	3	4	5
1	3	6	10	15
1	4	10	20	35
1	5	15	35	70

Figure 16.6: The number of ways to get from $(0,0)$ to (i,j) for $0 \leq i,j \leq 4$.

A more analytical way of solving this problem is to use the fact that each path from $(0,0)$ to $(n-1,m-1)$ is a sequence of $m-1$ horizontal steps and $n-1$ vertical steps. There are $\binom{n+m-2}{n-1} = \binom{n+m-2}{m-1} = \frac{(n+m-2)!}{(n-1)!(m-1)!}$ such paths.

Variant: Solve the same problem using $O(\min(n,m))$ space.

Variant: Solve the same problem in the presence of obstacles, specified by a Boolean 2D array, where the presence of a true value represents an obstacle.

Variant: A fisherman is in a rectangular sea. The value of the fish at point (i,j) in the sea is specified by an $n \times m$ 2D array A. Write a program that computes the maximum value of fish a fisherman can catch on a path from the upper leftmost point to the lower rightmost point. The fisherman can only move down or right, as illustrated in Figure 16.7.

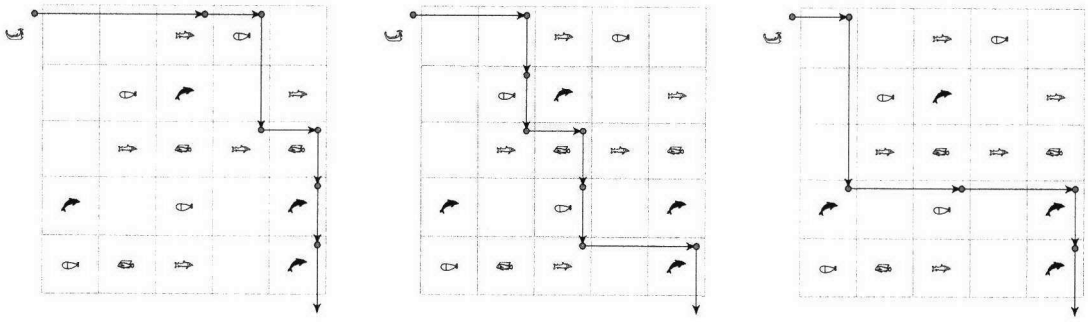


Figure 16.7: Alternate paths for a fisherman. Different types of fish have different values, which are known to the fisherman.

Variant: Solve the same problem when the fisherman can begin and end at any point. He must still move down or right. (Note that the value at (i,j) may be negative.)

Variant: A decimal number is a sequence of digits, i.e., a sequence over $\{0,1,2,\ldots,9\}$. The sequence has to be of length 1 or more, and the first element in the sequence cannot be 0. Call a decimal

number D *monotone* if $D[i] \leq D[i+1], 0 \leq i < |D|$. Write a program which takes as input a positive integer k and computes the number of decimal numbers of length k that are monotone.

Variant: Call a decimal number D, as defined above, *strictly monotone* if $D[i] < D[i+1], 0 \leq i < |D|$. Write a program which takes as input a positive integer k and computes the number of decimal numbers of length k that are strictly monotone.

16.4 Compute the binomial coefficients

The symbol $\binom{n}{k}$ is the short form for the expression $\frac{n(n-1)\cdots(n-k+1)}{k(k-1)\cdots(3)(2)(1)}$. It is the number of ways to choose a k-element subset from an n-element set. It is not obvious that the expression defining $\binom{n}{k}$ always yields an integer. Furthermore, direct computation of $\binom{n}{k}$ from this expression quickly results in the numerator or denominator overflowing if integer types are used, even if the final result fits in a 32-bit integer. If floats are used, the expression may not yield a 32-bit integer.

Design an efficient algorithm for computing $\binom{n}{k}$ which has the property that it never overflows if the final result fits in the integer word size.

Hint: Write an equation.

Solution: A brute-force approach would be to compute $n(n-1)\cdots(n-k+1)$, then $k(k-1)\cdots(3)(2)(1)$, and finally divide the former by the latter. As pointed out in the problem introduction, this can lead to overflows, even when the final result fits in the integer word size.

It is tempting to proceed by pairing terms in the numerator and denominator that have common factors and cancel them out. This approach is unsatisfactory because of the need to factor numbers, which itself is challenging.

A better approach is to avoid multiplications and divisions entirely. Fundamentally, the binomial coefficient counts the number of subsets of size k in a set of size n. We could enumerate k-sized subsets of $\{0, 1, 2, \ldots, n-1\}$ sets using recursion, as in Solution 15.5 on Page 226. The idea is as follows. Consider the nth element in the initial set. A subset of size k will either contain this element, or not contain it. This is the basis for a recursive enumeration—find all subsets of size $k-1$ amongst the first $n-1$ elements and add the nth element into these sets, and then find all subsets of size k amongst the first $n-1$ elements. The union of these two subsets is all subsets of size k.

However, since all we care about is the *number* of such subsets, we can do much better complexity-wise. The recursive enumeration also implies that the binomial coefficient must satisfy the following formula:

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

This identity yields a straightforward recursion for $\binom{n}{k}$. The base cases are $\binom{r}{r}$ and $\binom{r}{0}$, both of which are 1. The individual results from the subcalls are 32-bit integers and if $\binom{n}{k}$ can be represented by a 32-bit integer, they can too, so it is not possible for intermediate results to overflow.

For example, $\binom{5}{2} = \binom{4}{2}+\binom{4}{1}$. Expanding $\binom{4}{2}$, we get $\binom{4}{2} = \binom{3}{2}+\binom{3}{1}$. Expanding $\binom{3}{2}$, we get $\binom{3}{2} = \binom{2}{2}+\binom{2}{1}$. Note that $\binom{2}{2}$ is a base case, returning 1. Continuing this way, we get $\binom{4}{2}$, which is 6 and $\binom{4}{1}$, which is 4, so $\binom{5}{2} = 6 + 4 = 10$.

Naively implemented, the above recursion will have repeated subcalls with identical arguments and exponential time complexity. This can be avoided by caching intermediate results.

```
def compute_binomial_coefficient(n, k):
```

```
def compute_x_choose_y(x, y):
    if y in (0, x):
        return 1

    if x_choose_y[x][y] == 0:
        without_y = compute_x_choose_y(x - 1, y)
        with_y = compute_x_choose_y(x - 1, y - 1)
        x_choose_y[x][y] = without_y + with_y
    return x_choose_y[x][y]

x_choose_y = [[0] * (k + 1) for _ in range(n + 1)]
return compute_x_choose_y(n, k)
```

The number of subproblems is $O(nk)$ and once $\binom{n-1}{k}$ and $\binom{n-1}{k-1}$ are known, $\binom{n}{k}$ can be computed in $O(1)$ time, yielding an $O(nk)$ time complexity. The space complexity is also $O(nk)$; it can easily be reduced to $O(k)$.

16.5 Search for a sequence in a 2D array

Suppose you are given a 2D array of integers (the "grid"), and a 1D array of integers (the "pattern"). We say the pattern occurs in the grid if it is possible to start from some entry in the grid and traverse adjacent entries in the order specified by the pattern till all entries in the pattern have been visited. The entries adjacent to an entry are the ones directly above, below, to the left, and to the right, assuming they exist. For example, the entries adjacent to $(3,4)$ are $(3,3)$, $(3,5)$, $(2,4)$ and $(4,4)$. It is acceptable to visit an entry in the grid more than once.

As an example, if the grid is

$$\begin{bmatrix} 1 & 2 & 3 \\ 3 & 4 & 5 \\ 5 & 6 & 7 \end{bmatrix}$$

and the pattern is $\langle 1,3,4,6 \rangle$, then the pattern occurs in the grid—consider the entries $\langle (0,0),(1,0),(1,1),(2,1) \rangle$. However, $\langle 1,2,3,4 \rangle$ does not occur in the grid.

Write a program that takes as arguments a 2D array and a 1D array, and checks whether the 1D array occurs in the 2D array.

Hint: Start with length 1 prefixes of the 1D array, then move on to length $2,3,\ldots$ prefixes.

Solution: A brute-force approach might be to enumerate all 1D subarrays of the 2D subarray. This has very high time complexity, since there are many possible subarrays. Its inefficiency stems from not using the target 1D subarray to guide the search.

Let the 2D array be A, and the 1D array be S. Here is a guided way to search for matches. Let's say we have a suffix of S to match, and a starting point to match from. If the suffix is empty, we are done. Otherwise, for the suffix to occur from the starting point, the first entry of the suffix must equal the entry at the starting point, and the remainder of the suffix must occur starting at a point adjacent to the starting point.

For example, when searching for $\langle 1,3,4,6 \rangle$ starting at $(0,0)$, since we match 1 with $A[0][0]$, we would continue searching for $\langle 3,4,6 \rangle$ from $(0,1)$ (which fails immediately, since $A[0][1] \neq 3$) and from $(1,0)$. Since $A[0][1] = 3$, we would continue searching for $\langle 4,6 \rangle$ from $(0,1)$'s neighbors, i.e., from $(0,0)$ (which fails immediately), then from $(1,1)$ (which eventually leads to success).

In the program below, we cache intermediate results to avoid repeated calls to the recursion with identical arguments.

```
def is_pattern_contained_in_grid(grid, S):
    def is_pattern_suffix_contained_starting_at_xy(x, y, offset):
        if len(S) == offset:
            # Nothing left to complete.
            return True

        # Check if (x, y) lies outside the grid.
        if (0 <= x < len(grid) and 0 <= y < len(grid[x])
                and grid[x][y] == S[offset]
                and (x, y, offset) not in previous_attempts and any(
                    is_pattern_suffix_contained_starting_at_xy(
                        x + a, y + b, offset + 1)
                    for a, b in ((-1, 0), (1, 0), (0, -1), (0, 1)))):
            return True
        previous_attempts.add((x, y, offset))
        return False

    # Each entry in previous_attempts is a point in the grid and suffix of
    # pattern (identified by its offset). Presence in previous_attempts
    # indicates the suffix is not contained in the grid starting from that
    # point.
    previous_attempts = set()
    return any(
        is_pattern_suffix_contained_starting_at_xy(i, j, 0)
        for i in range(len(grid)) for j in range(len(grid[i])))
```

The complexity is $O(nm|S|)$, where n and m are the dimensions of A—we do a constant amount of work within each call to the match function, except for the recursive calls, and the number of calls is not more than the number of entries in the $2D$ array.

Variant: Solve the same problem when you cannot visit an entry in A more than once.

Variant: Enumerate all solutions when you cannot visit an entry in A more than once.

16.6 The knapsack problem

A thief breaks into a clock store. Each clock has a weight and a value, which are known to the thief. His knapsack cannot hold more than a specified combined weight. His intention is to take clocks whose total value is maximum subject to the knapsack's weight constraint.

His problem is illustrated in Figure 16.8 on the next page. If the knapsack can hold at most 130 ounces, he cannot take all the clocks. If he greedily chooses clocks, in decreasing order of value-to-weight ratio, he will choose P, H, O, B, I, and L in that order for a total value of \$669. However, $\{H, J, O\}$ is the optimum selection, yielding a total value of \$695.

Write a program for the knapsack problem that selects a subset of items that has maximum value and satisfies the weight constraint. All items have integer weights and values. Return the value of the subset.

Hint: Greedy approaches are doomed.

Solution: Greedy strategies such as picking the most valuable clock, or picking the clock with maximum value-to-weight ratio, do not always give the optimum solution.

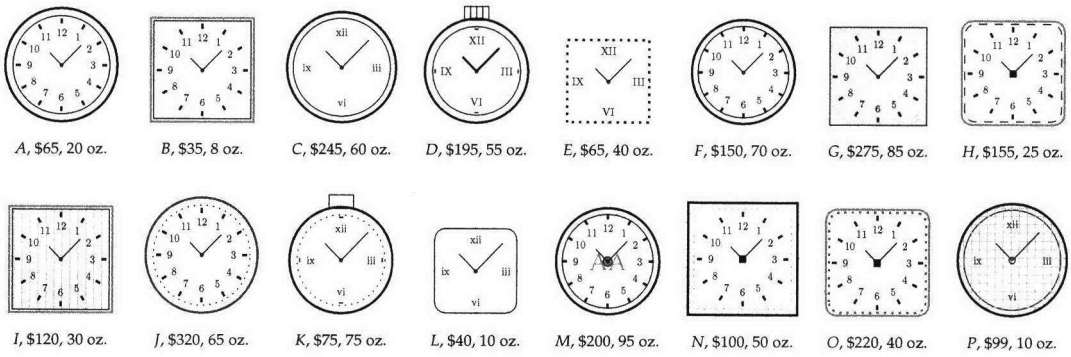


Figure 16.8: A clock store.

We can always get the optimum solution by considering all subsets, e.g., using Solution 15.4 on Page 225. This has exponential time complexity in the number of clocks. However, brute-force enumeration is wasteful because it ignores the weight constraint. For example, no subset that includes Clocks F and G can satisfy the weight constraint.

The better approach is to simultaneously consider the weight constraint. For example, what is the optimum solution if a given clock is chosen, and what is the optimum solution if that clock is not chosen? Each of these can be solved recursively with the implied weight constraint. For the given example, if we choose the Clock A, we need to find the maximum value of clocks from Clocks B–P with a capacity of 130 – 20 and add \$65 to that value. If we do not choose Clock A, we need to find the maximum value of clocks from Clocks B–P with a capacity of 130. The larger of these two values is the optimum solution.

More formally, let the clocks be numbered from 0 to $n-1$, with the weight and the value of the ith clock denoted by w_i and v_i. Denote by $V[i][w]$ the optimum solution when we are restricted to Clocks $0, 1, 2, \ldots, i-1$ and can carry w weight. Then $V[i][w]$ satisfies the following recurrence:

$$V[i][w] = \begin{cases} \max\left(V[i-1][w], V[i-1][w-w_i] + v_i\right), & \text{if } w_i \le w; \\ V[i-1][w], & \text{otherwise.} \end{cases}$$

We take $i = 0$ or $w = 0$ as bases cases—for these, $V[i][w] = 0$.

We demonstrate the above algorithm in Figure 16.9 on the next page. Suppose there are four items,  whose value and weight are given in Figure 16.9(a) on the following page. Then the corresponding V table is given in Figure 16.9(b) on the next page. As an example, the extreme lower right entry is the maximum of the entry above (70) and 30 plus the entry for  with capacity 5 – 2 (50), i.e., 80.

```
Item = collections.namedtuple('Item', ('weight', 'value'))


def optimum_subject_to_capacity(items, capacity):
    # Returns the optimum value when we choose from items[:k + 1] and have a
    # capacity of available_capacity.
    def optimum_subject_to_item_and_capacity(k, available_capacity):
        if k < 0:
            # No items can be chosen.
            return 0

        if V[k][available_capacity] == -1:
```

Item	Value	Weight
	$60	5 oz.
	$50	3 oz.
	$70	4 oz.
	$30	2 oz.

(a) Value-weight table.

	0	1	2	3	4	5
	0	0	0	0	0	60
	0	0	0	50	50	60
	0	0	0	50	70	70
	0	0	30	50	70	80

(b) Knapsack table for the items in (a). The columns correspond to capacities from 0 to 5.

Figure 16.9: A small example of the knapsack problem. The knapsackcapacity is 5 oz.

```
        without_curr_item = optimum_subject_to_item_and_capacity(
            k - 1, available_capacity)
        with_curr_item = (0 if available_capacity < items[k].weight else (
            items[k].value + optimum_subject_to_item_and_capacity(
                k - 1, available_capacity - items[k].weight)))
        V[k][available_capacity] = max(without_curr_item, with_curr_item)
    return V[k][available_capacity]

# V[i][j] holds the optimum value when we choose from items[:i + 1] and have
# a capacity of j.
V = [[-1] * (capacity + 1) for _ in items]
return optimum_subject_to_item_and_capacity(len(items) - 1, capacity)
```

The algorithm computes $V[n-1][w]$ in $O(nw)$ time, and uses $O(nw)$ space.

Variant: Solve the same problem using $O(w)$ space.

Variant: Solve the same problem using $O(C)$ space, where C is the number of weights between 0 and w that can be achieved. For example, if the weights are 100, 200, 200, 500, and $w = 853$, then $C = 9$, corresponding to the weights $0, 100, 200, 300, 400, 500, 600, 700, 800$.

Variant: Solve the fractional knapsack problem. In this formulation, the thief can take a fractional part of an item, e.g., by breaking it. Assume the value of a fraction of an item is that fraction times the value of the item.

Variant: In the "divide-the-spoils-fairly" problem, two thieves who have successfully completed a burglary want to know how to divide the stolen items into two groups such that the difference between the value of these two groups is minimized. For example, they may have stolen the clocks in Figure 16.8 on the preceding page, and would like to divide the clocks between them so that the difference of the dollar value of the two sets is minimized. For this instance, an optimum split is $\{A, G, J, M, O, P\}$ to one thief and the remaining to the other thief. The first set has value \$1179, and the second has value \$1180. An equal split is impossible, since the sum of the values of all the clocks is odd. Write a program to solve the divide-the-spoils-fairly problem.

Variant: Solve the divide-the-spoils-fairly problem with the additional constraint that the thieves have the same number of items.

Variant: The US President is elected by the members of the Electoral College. The number of electors per state and Washington, D.C., are given in Table 16.2 on the next page. All electors from each state

as well as Washington, D.C., cast their vote for the same candidate. Write a program to determine if a tie is possible in a presidential election with two candidates.

Table 16.2: Electoral college votes.

State	Electors	State	Electors	State	Electors
Alabama	9	Louisiana	8	Ohio	18
Alaska	3	Maine	4	Oklahoma	7
Arizona	11	Maryland	10	Oregon	7
Arkansas	6	Massachusetts	11	Pennsylvania	20
California	55	Michigan	16	Rhode Island	4
Colorado	9	Minnesota	10	South Carolina	9
Connecticut	7	Mississippi	6	South Dakota	3
Delaware	3	Missouri	10	Tennessee	11
Florida	29	Montana	3	Texas	38
Georgia	16	Nebraska	5	Utah	6
Hawaii	4	Nevada	6	Vermont	3
Idaho	4	New Hampshire	4	Virginia	13
Illinois	20	New Jersey	14	Washington	12
Indiana	11	New Mexico	5	West Virginia	5
Iowa	6	New York	29	Wisconsin	10
Kansas	6	North Carolina	15	Wyoming	3
Kentucky	8	North Dakota	3	Washington, D.C.	3

16.7 The BEDBATHANDBEYOND.COM problem

Suppose you are designing a search engine. In addition to getting keywords from a page's content, you would like to get keywords from Uniform Resource Locators (URLs). For example, bedbathandbeyond.com yields the keywords "bed, bath, beyond, bat, hand": the first two coming from the decomposition "bed bath beyond" and the latter two coming from the decomposition "bed bat hand beyond".

Given a dictionary, i.e., a set of strings, and a name, design an efficient algorithm that checks whether the name is the concatenation of a sequence of dictionary words. If such a concatenation exists, return it. A dictionary word may appear more than once in the sequence. For example, "a", "man", "a", "plan", "a", "canal" is a valid sequence for "amanaplanacanal".

Hint: Solve the generalized problem, i.e., determine for each prefix of the name whether it is the concatenation of dictionary words.

Solution: The natural approach is to use recursion, i.e., find dictionary words that begin the name, and solve the problem recursively on the remainder of the name. Implemented naively, this approach has very high time complexity on some input cases, e.g., if the name is N repetitions of "AB" followed by "C" and the dictionary words are "A", "B", and "AB" the time complexity is exponential in N. For example, for the string "ABABC", then we recurse on the substring "ABC" twice (once from the sequence "A", "B" and once from "AB").

The solution is straightforward—cache intermediate results. The cache keys are prefixes of the string. The corresponding value is a Boolean denoting whether the prefix can be decomposed into a sequence of valid words.

It's easy to determine if a string is a valid word—we simply store the dictionary in a hash table. A prefix of the given string can be decomposed into a sequence of dictionary words exactly if it is a dictionary word, or there exists a shorter prefix which can be decomposed into a sequence of dictionary words and the difference of the shorter prefix and the current prefix is a dictionary word.

For example, for "amanaplanacanal":

(1.) the prefix "a" has a valid decomposition (since "a" is a dictionary word),

(2.) the prefix "am" has a valid decomposition (since "am" is a dictionary word),

(3.) the prefix "ama" has a valid decomposition (since "a" has a valid decomposition, and "am" is a dictionary word), and

(4.) "aman" has a valid decomposition (since "am" has a valid decomposition and "an" is a dictionary word).

Skipping ahead,

5. "amanapl" does not have a valid decomposition (since none of "l", "pl", apl", etc. are dictionary words), and
6. "amanapla" does not have a valid decomposition (since the only dictionary word ending the string is "a" and "amanapl" does not have a valid decomposition).

The algorithm tells us if we can break a given string into dictionary words, but does not yield the words themselves. We can obtain the words with a little more book-keeping. Specifically, if a prefix has a valid decomposition, we record the length of the last dictionary word in the decomposition.

```
def decompose_into_dictionary_words(domain, dictionary):
    # When the algorithm finishes, last_length[i] != -1 indicates domain[:i +
    # 1] has a valid decomposition, and the length of the last string in the
    # decomposition is last_length[i].
    last_length = [-1] * len(domain)
    for i in range(len(domain)):
        # If domain[:i + 1] is a dictionary word, set last_length[i] to the
        # length of that word.
        if domain[:i + 1] in dictionary:
            last_length[i] = i + 1

        # If last_length[i] = -1 look for j < i such that domain[: j + 1] has a
        # valid decomposition and domain[j + 1:i + 1] is a dictionary word. If
        # so, record the length of that word in last_length[i].
        if last_length[i] == -1:
            for j in range(i):
                if last_length[j] != -1 and domain[j + 1:i + 1] in dictionary:
                    last_length[i] = i - j
                    break

    decompositions = []
    if last_length[-1] != -1:
        # domain can be assembled by dictionary words.
        idx = len(domain) - 1
        while idx >= 0:
            decompositions.append(domain[idx + 1 - last_length[idx]:idx + 1])
            idx -= last_length[idx]
        decompositions = decompositions[::-1]
    return decompositions
```

Let n be the length of the input string s. For each $k < n$ we check for each $j < k$ whether the substring $s[j+1, k]$ is a dictionary word, and each such check requires $O(k-j)$ time. This implies the time complexity is $O(n^3)$. We can improve the time complexity as follows. Let W be the length of the longest dictionary word. We can restrict j to range from $k-W$ to $k-1$ without losing any decompositions, so the time complexity improves to $O(n^2W)$.

If we want all possible decompositions, we can store all possible values of j that gives us a correct break with each position. Note that the number of possible decompositions can be exponential here. This is illustrated by the string "itsitsitsits...".

Variant: Palindromic decompositions were described in Problem 15.7 on Page 228. Observe every string s has at least one palindromic decomposition, which is the trivial one consisting of the individual characters. For example, if s is "0204451881" then "0", "2", "0", "4", "4", "5", "1", "8", "8", "1" is such a trivial decomposition. The minimum decomposition of s is "020", "44", "5", "1881". How would you compute a palindromic decomposition of a string s that uses a minimum number of substrings?

16.8 Find the minimum weight path in a triangle

A sequence of integer arrays in which the nth array consists of n entries naturally corresponds to a triangle of numbers. See Figure 16.10 for an example.

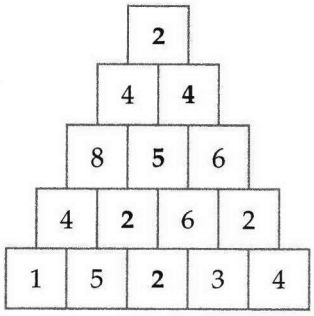


Figure 16.10: A number triangle.

Define a path in the triangle to be a sequence of entries in the triangle in which adjacent entries in the sequence correspond to entries that are adjacent in the triangle. The path must start at the top, descend the triangle continuously, and end with an entry on the bottom row. The weight of a path is the sum of the entries.

Write a program that takes as input a triangle of numbers and returns the weight of a minimum weight path. For example, the minimum weight path for the number triangle in Figure 16.10 is shown in bold face, and its weight is 15.

Hint: What property does the prefix of a minimum weight path have?

Solution: A brute-force approach is to enumerate all possible paths. Although these paths are quite easy to enumerate, there are 2^{n-1} such paths in a number triangle with n rows.

A far better way is to consider entries in the ith row. For any such entry, if you look at the minimum weight path ending at it, the part of the path that ends at the previous row must also be a minimum weight path. This gives us a DP solution. We iteratively compute the minimum weight of a path ending at each entry in Row i. Since after we complete processing Row i, we do not need the results for Row $i-1$ to process Row $i+1$, we can reuse storage.

```
def minimum_path_weight(triangle):
    min_weight_to_curr_row = [0]
    for row in triangle:
```

```
        min_weight_to_curr_row = [
            row[j] +
            min(min_weight_to_curr_row[max(j - 1, 0)],
                min_weight_to_curr_row[min(j, len(min_weight_to_curr_row) - 1)])
            for j in range(len(row))
        ]
    return min(min_weight_to_curr_row)
```

The time spent per element is $O(1)$ and there are $1 + 2 + \cdots + n = n(n+1)/2$ elements, implying an $O(n^2)$ time complexity. The space complexity is $O(n)$.

16.9 Pick up coins for maximum gain

In the pick-up-coins game, an even number of coins are placed in a line, as in Figure 16.11. Two players take turns at choosing one coin each—they can only choose from the two coins at the ends of the line. The game ends when all the coins have been picked up. The player whose coins have the higher total value wins. A player cannot pass his turn.

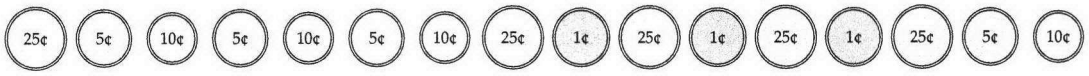


Figure 16.11: A row of coins.

Design an efficient algorithm for computing the maximum total value for the starting player in the pick-up-coins game.

Hint: Relate the best play for the first player to the best play for the second player.

Solution: First of all, note that greedily selecting the maximum of the two end coins does not yield the best solution. If the coins are $5, 25, 10, 1$, if the first player is greedy and chooses 5, the second player can pick up the 25. Now the first player will choose the 10, and the second player gets the 1, so the first player has a total of 15 and the second player has a total of 26. A better move for the first player is picking up 1. Now the second player is forced to expose the 25, so the first player will achieve 26.

The drawback of greedy selection is that it does not consider the opportunities created for the second player. Intuitively, the first player wants to balance selecting high coins with minimizing the coins available to the second player.

The second player is assumed to play the best move he possibly can. Therefore, the second player will choose the coin that maximizes *his* revenue. Call the sum of the coins selected by a player his revenue. Let $R(a, b)$ be the maximum revenue a player can get when it is his turn to play, and the coins remaining on the table are at indices a to b, inclusive. Let C be an array representing the line of coins, i.e., $C[i]$ is the value of the ith coin. If the first player selects the coin at a, since the second player plays optimally, the first player will end up with a total revenue of $C[a] + S(a+1, b) - R(a+1, b)$, where $S(a, b)$ is the sum of the coins from positions a to b, inclusive. If he selects the coin at b, he will end up with a total revenue of $C[b] + S(a, b-1) - R(a, b-1)$. Since the first player wants to maximize revenue, he chooses the greater of the two, i.e., $R(a, b) = \max(C[a]+S(a+1, b)-R(a+1, b), C[b]+S(a, b-1)-R(a, b-1))$. This recursion for R can be solved easily using DP.

Now we present a slightly different recurrence for R. Since the second player seeks to maximize his revenue, and the total revenue is a constant, it is equivalent for the the second player to move

so as to minimize the first player's revenue. Therefore, $R(a, b)$ satisfies the following equations:

$$R(a,b) = \begin{cases} \max\left(\begin{array}{l} C[a] + \min\left(\begin{array}{l} R(a+2,b), \\ R(a+1,b-1) \end{array} \right), \\ C[b] + \min\left(\begin{array}{l} R(a+1,b-1), \\ R(a,b-2) \end{array} \right) \end{array} \right), & \text{if } a \leq b; \\ 0, & \text{otherwise.} \end{cases}$$

In essence, the strategy is to minimize the maximum revenue the opponent can gain. The benefit of this "min-max" recurrence for $R(a, b)$, compared to our first formulation, is that it does not require computing $S(a + 1, b)$ and $S(a, b - 1)$.

For the coins $\langle 10, 25, 5, 1, 10, 5 \rangle$, the optimum revenue for the first player is 31. Some of subproblems encountered include computing the optimum revenue for $\langle 10, 25 \rangle$ (which is 25), $\langle 5, 1 \rangle$ (which is 5), and $\langle 5, 1, 10, 5 \rangle$ (which is 15).

For the coins in Figure 16.11 on the facing page, the maximum revenue for the first player is 140¢, i.e., whatever strategy the second player uses, the first player can guarantee a revenue of at least 140¢. In contrast, if both players always pick the more valuable of the two coins available to them, the first player will get only 120¢.

In the program below, we solve for R using DP.

```
def maximum_revenue(coins):
    def compute_maximum_revenue_for_range(a, b):
        if a > b:
            # No coins left.
            return 0

        if maximum_revenue_for_range[a][b] == 0:
            max_revenue_a = coins[a] + min(
                compute_maximum_revenue_for_range(a + 2, b),
                compute_maximum_revenue_for_range(a + 1, b - 1))
            max_revenue_b = coins[b] + min(
                compute_maximum_revenue_for_range(a + 1, b - 1),
                compute_maximum_revenue_for_range(a, b - 2))
            maximum_revenue_for_range[a][b] = max(max_revenue_a, max_revenue_b)
        return maximum_revenue_for_range[a][b]

    maximum_revenue_for_range = [[0] * len(coins) for _ in coins]
    return compute_maximum_revenue_for_range(0, len(coins) - 1)
```

There are $O(n^2)$ possible arguments for $R(a, b)$, where n is the number of coins, and the time spent to compute R from previously computed values is $O(1)$. Hence, R can be computed in $O(n^2)$ time.

16.10 Count the number of moves to climb stairs

You are climbing stairs. You can advance 1 to k steps at a time. Your destination is exactly n steps up.

Write a program which takes as inputs n and k and returns the number of ways in which you can get to your destination. For example, if $n = 4$ and $k = 2$, there are five ways in which to get to the destination:

- four single stair advances,

- two single stair advances followed by a double stair advance,
- a single stair advance followed by a double stair advance followed by a single stair advance,
- a double stair advance followed by two single stairs advances, and
- two double stair advances.

Hint: How many ways are there in which you can take the last step?

Solution: A brute-force enumerative solution does not make sense, since there are an exponential number of possibilities to consider.

Since the first advance can be one step, two steps,. . . , k steps, and all of these lead to different ways to get to the top, we can write the following equation for the number of steps $F(n,k)$:

$$F(n,k) = \sum_{i=1}^{k} F(n-i,k)$$

For the working example, $F(4,2) = F(4-2,2) + F(4-1,2)$. Recursing, $F(4-2,2) = F(4-2-2,2) + F(4-2-1,2)$. Both $F(0,2)$ and $F(1,2)$ are base-cases, with a value of 1, so $F(4-2,2) = 2$. Continuing with $F(4-1,2)$, $F(4-1,2) = F(4-1-2,2) + F(4-1-1,2)$. The first term is a base-case, with a value of 1. The second term has already been computed—its value is 2. Therefore, $F(4-1,2) = 3$, and $F(4,2) = 3+2$.

In the program below, we cache values of $F(i,k)$, $0 \leq i \leq n$ to improve time complexity.

```
def number_of_ways_to_top(top, maximum_step):
    def compute_number_of_ways_to_h(h):
        if h <= 1:
            return 1

        if number_of_ways_to_h[h] == 0:
            number_of_ways_to_h[h] = sum(
                compute_number_of_ways_to_h(h - i)
                for i in range(1, min(maximum_step, h) + 1))
        return number_of_ways_to_h[h]

    number_of_ways_to_h = [0] * (top + 1)
    return compute_number_of_ways_to_h(top)
```

We take $O(k)$ time to fill in each entry, so the total time complexity is $O(kn)$. The space complexity is $O(n)$.

16.11 The pretty printing problem

Consider the problem of laying out text using a fixed width font. Each line can hold no more than a fixed number of characters. Words on a line are to be separated by exactly one blank. Therefore, we may be left with whitespace at the end of a line (since the next word will not fit in the remaining space). This whitespace is visually unappealing.

Define the *messiness* of the end-of-line whitespace as follows. The messiness of a single line ending with b blank characters is b^2. The total messiness of a sequence of lines is the sum of the messinesses of all the lines. A sequence of words can be split across lines in different ways with different messiness, as illustrated in Figure 16.12 on the next page.

```
I have inserted a large number of␣␣␣
new examples from the papers for the
Mathematical Tripos during the last␣
twenty years, which should be useful
to Cambridge students.␣␣␣␣␣␣␣␣␣␣␣␣␣
```

(a) Messiness = $3^2 + 0^2 + 1^2 + 0^2 + 14^2 = 206$.

```
I have inserted a large number␣␣␣␣␣␣
of new examples from the papers␣␣␣␣␣
for the Mathematical Tripos during␣␣
the last twenty years, which should␣
be useful to Cambridge students.␣␣␣␣
```

(b) Messiness = $6^2 + 5^2 + 2^2 + 1^2 + 4^2 = 82$.

Figure 16.12: Two layouts for the same sequence of words; the line length L is 36.

Given text, i.e., a string of words separated by single blanks, decompose the text into lines such that no word is split across lines and the messiness of the decomposition is minimized. Each line can hold no more than a specified number of characters.

Hint: Focus on the last word and the last line.

Solution: A greedy approach is to fit as many words as possible in each line. However, some experimentation shows this is suboptimal. See Figure 16.13 for an example. In essence, the greedy algorithm does not spread words uniformly across the lines, which is the key requirement of the problem. Adding a new word may necessitate moving earlier words.

	0	1	2	3	4
Line 1	a		b		c
Line 2	d				

(a) Greedy placement: Line 1 has a messiness of 0^2, and Line 2 has a messiness of 4^2.

	0	1	2	3	4
Line 1	a		b		
Line 2	c		d		

(b) Optimum placement: Line 1 has a messiness of 2^2, and Line 2 has a messiness of 2^2.

Figure 16.13: Assuming the text is "a b c d" and the line length is 5, placing words greedily results in a total messiness of 16, as seen in (a), whereas the optimum placement has a total messiness of 8, as seen in (b).

Suppose we want to find the optimum placement for the ith word. As we have just seen, we cannot trivially extend the optimum placement for the first $i-1$ words to an optimum placement for the ith word. Put another way, the placement that results by removing the ith word from an optimum placement for the first i words is not always an optimum placement for the first $i-1$ words. However, what we can say is that if in the optimum placement for the ith word the last line consists of words $j, j+1, \ldots, i$, then in this placement, the first $j-1$ words must be placed optimally.

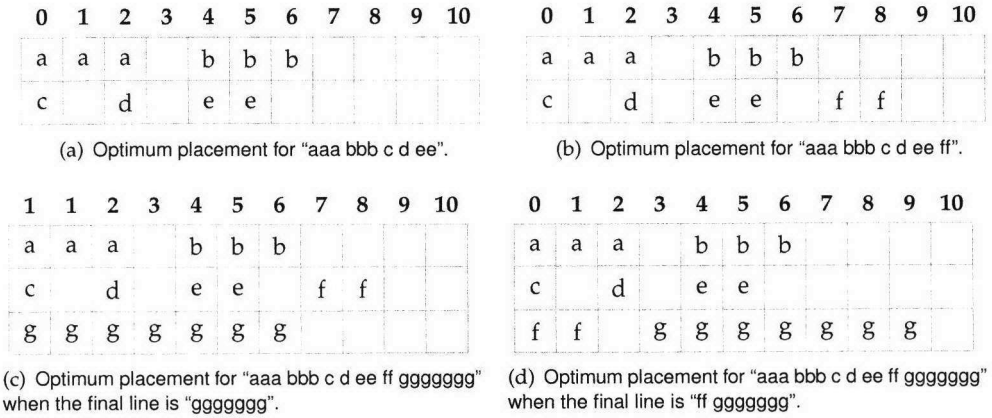
In an optimum placement of the first i words, the last line consists of *some* subset of words ending in the ith word. Furthermore, since the first i words are assumed to be optimally placed, the placement of words on the lines prior to the last one must be optimum. Therefore, we can write a recursive formula for the minimum messiness, $M(i)$, when placing the first i words. Specifically, $M(i)$, equals $\min_{j \le i} f(j,i) + M(j-1)$, where $f(j,i)$ is the messiness of a single line consisting of words j to i inclusive.

We give an example of this approach in Figure 16.14 on the next page. The optimum placement of "aaa bbb c d ee" is shown in Figure 16.14(a) on the following page. The optimum placement of "aaa bbb c d ee ff" is shown in Figure 16.14(b) on the next page.

To determine the optimum placement for "aaa bbb c d ee ff ggggggg", we consider two cases—the final line is "ff ggggggg", and the final line is "ggggggg". (No more cases are possible, since we

cannot fit "ee ff ggggggg" on a single line.)

If the final line is "ff ggggggg", then the "aaa bb c d ee" must be placed as in Figure 16.14(a). If the final line is "ggggggg", then "aaa bbb c d ee ff" must be placed as in Figure 16.14(b). These two cases are shown in Figure 16.14(c) and Figure 16.14(d), respectively. Comparing the two, we see Figure 16.14(c) has the lower messiness and is therefore optimum.



(a) Optimum placement for "aaa bbb c d ee".

(b) Optimum placement for "aaa bbb c d ee ff".

(c) Optimum placement for "aaa bbb c d ee ff ggggggg" when the final line is "ggggggg".

(d) Optimum placement for "aaa bbb c d ee ff ggggggg" when the final line is "ff ggggggg".

Figure 16.14: Solving the pretty printing problem for text "aaa bb c d ee ff ggggggg" and line length 11.

The recursive computation has exponential complexity, because it visits identical subproblems repeatedly. The solution is to cache the values for M.

```
def minimum_messiness(words, line_length):
    num_remaining_blanks = line_length - len(words[0])
    # min_messiness[i] is the minimum messiness when placing words[0:i + 1].
    min_messiness = ([num_remaining_blanks**2] + [float('inf')] *
                     (len(words) - 1))
    for i in range(1, len(words)):
        num_remaining_blanks = line_length - len(words[i])
        min_messiness[i] = min_messiness[i - 1] + num_remaining_blanks**2
        # Try adding words[i - 1], words[i - 2], ...
        for j in reversed(range(i)):
            num_remaining_blanks -= len(words[j]) + 1
            if num_remaining_blanks < 0:
                # Not enough space to add more words.
                break
            first_j_messiness = 0 if j - 1 < 0 else min_messiness[j - 1]
            current_line_messiness = num_remaining_blanks**2
            min_messiness[i] = min(min_messiness[i],
                                   first_j_messiness + current_line_messiness)
    return min_messiness[-1]
```

Let L be the line length. Then there can certainly be no more than L words on a line, so the amount of time spent processing each word is $O(L)$. Therefore, if there are n words, the time complexity is $O(nL)$. The space complexity is $O(n)$ for the cache.

Variant: Solve the same problem when the messiness is the sum of the messinesses of all but the last line.

Variant: Suppose the messiness of a line ending with b blank characters is defined to be b. Can you solve the messiness minimization problem in $O(n)$ time and $O(1)$ space?

16.12 Find the longest nondecreasing subsequence

The problem of finding the longest nondecreasing subsequence in a sequence of integers has implications to many disciplines, including string matching and analyzing card games. As a concrete instance, the length of a longest nondecreasing subsequence for the array in Figure 16.15 is 4. There are multiple longest nondecreasing subsequences, e.g., $\langle 0, 4, 10, 14\rangle$ and $\langle 0, 2, 6, 9\rangle$. Note that elements of non-decreasing subsequence are not required to immediately follow each other in the original sequence.

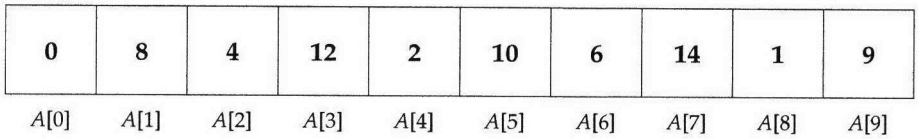


Figure 16.15: An array whose longest nondecreasing subsequences are of length 4.

Write a program that takes as input an array of numbers and returns the length of a longest nondecreasing subsequence in the array.

Hint: Express the longest nondecreasing subsequence ending at an entry in terms of the longest nondecreasing subsequence appearing in the subarray consisting of preceding elements.

Solution: A brute-force approach would be to enumerate all possible subsequences, testing each one for being nondecreasing. Since there are 2^n subsequences of an array of length n, the time complexity would be huge. Some heuristic pruning can be applied, but the program grows very cumbersome.

If we have processed the initial set of entries of the input array, intuitively this should help us when processing the next entry. For the given example, if we know the lengths of the longest nondecreasing subsequences that end at Indices $0, 1, \ldots, 5$ and we want to know the longest nondecreasing subsequence that ends at Index 6, we simply look for the longest subsequence ending at an entry in $A[0, 5]$ whose value is less than or equal to $A[6]$. For Index 6 in the example in Figure 16.15, there are two such longest subsequences, namely the ones ending at Index 2 and Index 4, both of which yield nondecreasing subsequences of length 3 ending at $A[6]$.

Generalizing, define $L[i]$ to be the length of the longest nondecreasing subsequence of A that ends at and includes Index i. As an example, for the array in Figure 16.15, $L[6] = 3$. The longest nondecreasing subsequence that ends at Index i is of length 1 (if $A[i]$ is smaller than all preceding entries) or has some element, say at Index j, as its penultimate entry, in which case the subsequence restricted to $A[0, j]$ must be the longest subsequence ending at Index j. Based on this, $L[i]$ is either 1 (if $A[i]$ is less than all previous entries), or $1 + \max\{L[j] | j < i \text{ and } A[j] \leq A[i]\}$.

We can use this relationship to compute L, recursively or iteratively. If we want the sequence as well, for each i, in addition to storing the length of the nondecreasing sequence ending at i, we store the index of the last element of the subsequence that we extended to get the value assigned to $L[i]$.

Applying this algorithm to the example in Figure 16.15, we compute L as follows:

(1.) $L[0] = 1$ (since there are no entries before Entry 0)
(2.) $L[1] = 1 + \max(L[0]) = 2$ (since $A[0] \leq A[1]$)
(3.) $L[2] = 1 + \max(L[0]) = 2$ (since $A[0] \leq A[2]$ and $A[1] > A[2]$)
(4.) $L[3] = 1 + \max(L[0], L[1], L[2]) = 3$ (since $A[0], A[1]$ and $A[2] \leq A[3]$)
(5.) $L[4] = 1 + \max(L[0]) = 2$ (since $A[0] \leq A[4], A[1], A[2]$ and $A[3] > A[4]$)

(6.) $L[5] = 1 + \max(L[0], L[1], L[2], L[4]) = 3$ (since $A[0], A[1], A[2], A[4] \leq A[5]$ and $A[3] > A[5]$)
(7.) $L[6] = 1 + \max(L[0], L[2], L[4]) = 3$ (since $A[0]$, $A[2]$, $A[4] \leq A[6]$ and $A[1]$, $A[3]$, $A[5] > A[6]$)
(8.) $L[7] = 1 + \max(L[0], L[1], L[2], L[3], L[4], L[5], L[6]) = 4$ (since $A[0]$,$A[1]$,$A[2]$, $A[3]$, $A[4]$, $A[5]$, $A[6] \leq A[7]$)
(9.) $L[8] = 1 + \max(L[0]) = 2$ (since $A[0] \leq A[8]$ and $A[1]$,$A[2]$, $A[3]$, $A[4]$, $A[5]$, $A[6]$, $A[7] > A[8]$)
(10.) $L[9] = 1 + \max(L[0], L[1], L[2], L[4], L[6], L[8]) = 4$ (since $A[0]$, $A[1]$, $A[2]$, $A[4]$, $A[6]$, $A[8] \leq A[9]$ and $A[3]$, $A[5]$, $A[7] > A[9]$)

Therefore the maximum length of a longest nondecreasing subsequence is 4. There are multiple longest nondecreasing sequences of length 4, e.g., $\langle 0, 8, 12, 14\rangle$, $\langle 0, 2, 6, 9\rangle$, $\langle 0, 4, 6, 9\rangle$.

Here is an iterative implementation of the algorithm.

```
def longest_nondecreasing_subsequence_length(A):
    # max_length[i] holds the length of the longest nondecreasing subsequence
    # of A[:i + 1].
    max_length = [1] * len(A)
    for i in range(1, len(A)):
        max_length[i] = max(1 + max(
            [max_length[j] for j in range(i)
             if A[i] >= A[j]], default=0), max_length[i])
    return max(max_length)
```

The time complexity is $O(n^2)$ (each $L[i]$ takes $O(n)$ time to compute), and the space complexity is $O(n)$ (to store L).

Variant: Write a program that takes as input an array of numbers and returns a longest nondecreasing subsequence in the array.

Variant: Define a sequence of numbers $\langle a_0, a_1, \ldots, a_{n-1}\rangle$ to be *alternating* if $a_i < a_{i+1}$ for even i and $a_i > a_{i+1}$ for odd i. Given an array of numbers A of length n, find a longest subsequence $\langle i_0, \ldots, i_{k-1}\rangle$ such that $\langle A[i_0], A[i_1], \ldots, A[i_{k-1}]\rangle$ is alternating.

Variant: Define a sequence of numbers $\langle a_0, a_1, \ldots, a_{n-1}\rangle$ to be *weakly alternating* if no three consecutive terms in the sequence are increasing or decreasing. Given an array of numbers A of length n, find a longest subsequence $\langle i_0, \ldots, i_{k-1}\rangle$ such that $\langle A[i_0], A[i_1], \ldots, A[i_{k-1}]\rangle$ is weakly alternating.

Variant: Define a sequence of numbers $\langle a_0, a_1, \ldots, a_{n-1}\rangle$ to be *convex* if $a_i < \frac{a_{i-1}+a_{i+1}}{2}$, for $1 \leq i \leq n-2$. Given an array of numbers A of length n, find a longest subsequence $\langle i_0, \ldots, i_{k-1}\rangle$ such that $\langle A[i_0], A[i_1], \ldots, A[i_{k-1}]\rangle$ is convex.

Variant: Define a sequence of numbers $\langle a_0, a_1, \ldots, a_{n-1}\rangle$ to be *bitonic* if there exists k such that $a_i < a_{i+1}$, for $0 \leq i < k$ and $a_i > a_{i+1}$, for $k \leq i < n-1$. Given an array of numbers A of length n, find a longest subsequence $\langle i_0, \ldots, i_{k-1}\rangle$ such that $\langle A[i_0], A[i_1], \ldots, A[i_{k-1}]\rangle$ is bitonic.

Variant: Define a sequence of points in the plane to be *ascending* if each point is above and to the right of the previous point. How would you find a maximum ascending subset of a set of points in the plane?

Variant: Compute the longest nondecreasing subsequence in $O(n \log n)$ time.