Asian MathSci League, Inc (AMSLI)
Partner: National Olympiad in Informatics Philippines (NOI.PH)
Email address: amsliphil@yahoo.com / ask@noi.ph
Contact Nos: +632-9254526 +63906-1186067

Student Copy
AIEP Send-off Python 2 Session 7

DYNAMIC PROGRAMMING

In 1957, the mathematician Richard E. Bellman published the book *Dynamic Programming* and introduced an optimization method with the same name. The renowned computer scientist Donald Knuth would later recount an interesting anecdote about this work:

> A group of problems is collected under the heading "Exercises and Research Problems," with extremely trivial questions appearing in the midst of deep, unsolved problems. It is rumored that someone asked Dr. Bellman how to tell the exercises apart from the research problems, and he replied: "If you can solve it, it is an exercise; otherwise, it's a research problem."

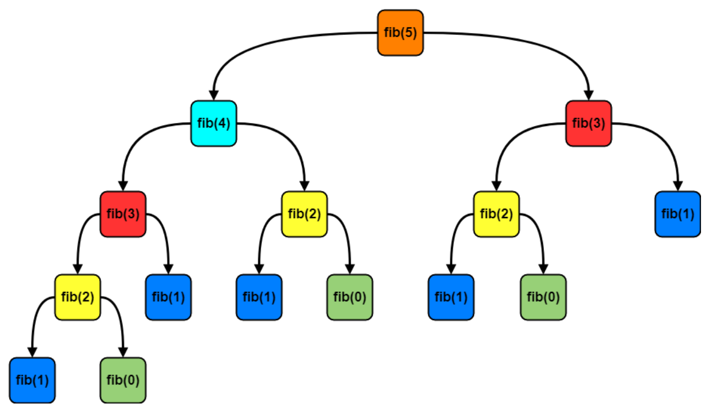
In Session 9 of the qualifying phase of your AIEP Python 2 training, you were introduced to dynamic programming (DP) and how it can be implemented via memoization (which is essentially "memorizing" the results of recursive calls). The goal of this handout is to take a more high-level approach to understanding DP. Implementation details are only glossed over; thus, it would be helpful to recall or review memoization before proceeding.

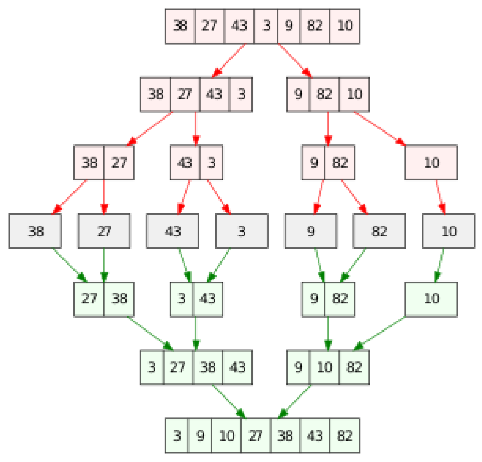
TO DP OR NOT TO DP?

In *Introduction to Algorithms* (hailed as the Bible of algorithms), Cormen, Leiserson, Rivest, and Stein[1] identify two hallmarks of a DP problem:

a) Overlapping subproblems. The task can be decomposed into smaller subtasks that are repeated or reused several times. The advantage offered by DP comes with "memorizing" (caching) the solutions to these repeated subtasks in order to avoid recomputation. However, if the subproblems are non-overlapping, caching does not provide any benefit, making divide and conquer the appropriate technique.

   The figure below shows representative tasks from both paradigms. In calculating the 5th term of the Fibonacci sequence (left/DP), there are repeated subtasks, such as computing the 3rd term (red) and the 2nd term (yellow). On the other hand, the subtasks in mergesort (right/divide-and-conquer) are independent of each other.





https://cdn.emre.me/2019-09-07-fibonacci-number.png
https://upload.wikimedia.org/wikipedia/commons/thumb/e/e6/Merge_sort_algorithm_diagram.svg/300px-Merge_sort_algorithm_diagram.svg.png

[1] Cormen, T.H., Leiserson, C.E., Rivest, R.L., & Stein, C. (2009). *Introduction to algorithms (3rd ed.).* Massachusetts Institute of Technology.

b) Optimal substructure. The optimal solutions to the subproblems are invoked in finding the optimal solution to the original problem. However, note that DP is not limited to optimization problems; in fact, it is widely used in counting and decision tasks, as we will see in our illustrative examples and exercises.

If there are optimal substructures and the greedy choice property also applies, the resulting program would be more efficient if a greedy algorithm is formulated instead. The greedy choice property is satisfied when the best solution to the original problem can be reached by always selecting what seems to be the best solution to the subproblem at the moment (though this can be difficult to prove).

While this conceptual understanding is central in recognizing when to employ DP vis-à-vis the other algorithmic paradigms, it may feel rather abstract and, hence, challenging to translate to an actual problem-solving framework. To this end, let us consider Halim and Halim's[2] strategy of viewing DP as an exercise in identifying states and transitions[3].

☝️ ONE-DIMENSIONAL DP

We open our discussion of states and transitions with a quintessential DP problem, which is a generalization of a question from the American Mathematics Competition 8.

Classic/Contest Climbing Stairs

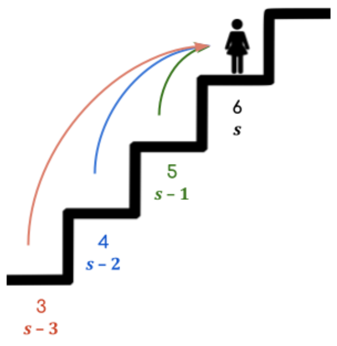
Adapted from the 2010 American Mathematics Competition 8 (AMC 8)

Every day at school, Jo climbs a flight of n stairs. Jo can take the stairs 1, 2, or 3 at a time. For example, Jo could climb 3, then 1, then 2. In how many ways can Jo climb the stairs? If $n = 6$, then there are 24 ways.

📝 **Discussion**

To get a feel for the problem, it may be helpful to run a small simulation. Suppose Jo would like to climb a flight of 6 stairs:

- To reach stair 6, she can come from stairs 3, 4, or 5.
- To reach stair 5, she can come from stairs 2, 3, or 4.
- To reach stair 4, she can come from stairs 1, 2, or 3.
- To reach stair 3, she can come from stairs 1 or 2, or she can also jump directly to stair 3.



Let us now try to think in terms of states and transitions. Here are some guide questions in identifying the state: What variables or parameters are changing in each subtask? What should we keep track of? In our simulation, notice how we have been keeping an eye on the stair where Jo is standing. The transitions relate the subtasks by describing how we move from one state (stair) to another. The problem specifies that Jo can climb 1, 2, or 3 stairs at a time. As seen in the figure above, this is equivalent to saying that, to reach stair s, Jo should "transition" from stairs $s-1$, $s-2$, or $s-3$.

[2] Halim, S., & Halim, F. (2010). *Competitive programming 3: The new lower bound of programming contests.* National University of Singapore.

[3] States and transitions are among the building blocks of *automata theory*, a branch of computer science that focuses on designing abstract machines and understanding the nature of computing itself. Some Bebras problems that you may have encountered during the first half of this AIEP send-off training program embed concepts from automata theory, such as *finite-state machines*.

Suppose that the function $f(s)$ denotes the number of ways to reach stair s. Since we have determined the states and transitions, a recurrence relation can now be established:

$$f(s) = f(s - 1) + f(s - 2) + f(s - 3).$$

We cap off our solution with the identification of the base cases:

- $f(1) = 1$: Jump directly to stair 1.
- $f(2) = 2$:
    - Jump directly to stair 2.
    - Climb stair 1 → stair 2.
- $f(3) = 4$:
    - Jump directly to stair 3.
    - Climb stair 1 → stair 2 → stair 3.
    - Climb stair 1 → stair 3.
    - Climb stair 2 → stair 3.

Using memoization, the core method can, thus, be implemented as follows:

```
memo = [-1 for _ in range(MAX_N + 1)]

def num_ways(stair):
    if memo[stair] == -1:
        if stair == 1:
            answer = 1
        elif stair == 2:
            answer = 2
        elif stair == 3:
            answer = 4
        else:
            answer = (num_ways(stair - 1) + num_ways(stair - 2) +
                num_ways(stair - 3))

        memo[stair] = answer

    return memo[stair]
```

This is an example of a one-dimensional DP problem since we only have to focus on a single parameter: the stair number. On another note, what do you think is the time complexity of our solution? Can you derive a general formula or strategy for obtaining the time complexity of a DP algorithm? *Hint: Consider the number of states or subtasks.*

Stepper, No Stepping!

If stepping on some stairs is prohibited, how would we modify our algorithm and our implementation to accommodate this restriction? For instance, if stair 4 is blocked, then Jo can reach stair 6 only if she comes from stairs 3 or 5, reducing the total number of ways to 10.

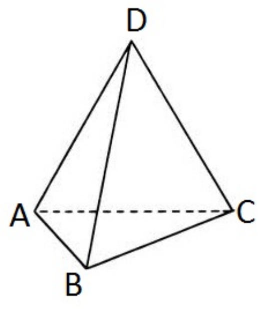
🙌 TWO-DIMENSIONAL DP

Moving forward, let us increase the number of parameters that comprise our state.

Contest Tetrahedron

Codeforces 166E

You are given a tetrahedron. Let's mark its vertices A, B, C, and D correspondingly.



An ant is standing in the vertex D of the tetrahedron. The ant is quite active and he wouldn't stay idle. At each moment of time he makes a step from one vertex to another one along some edge of the tetrahedron. The ant just can't stand on one place.

You do not have to do much to solve the problem: your task is to count the number of ways in which the ant can go from the initial vertex D to itself in exactly n steps. In other words, you are asked to find out the number of different cyclic paths with the length of n from vertex D to itself. As the number can be quite large, you should print it modulo 1000000007 $(10^9 + 7)$.

Input

The first line contains the only integer n $(1 \le n \le 10^7)$ — the required length of the cyclic path.

Output

Print the only integer — the required number of ways modulo 1000000007 $(10^9 + 7)$.

Sample Input	Sample Output
2	3
4	21

Note

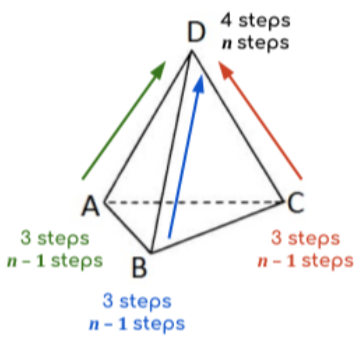
The required paths in the first sample are (1) $D \to A \to D$, (2) $D \to B \to D$, and (3) $D \to C \to D$.

📝 Discussion

Once again, we start with a small simulation. Suppose the length of the cyclic path is 4:

- $\square \to \square \to \square \to \square \to D$: To arrive at D in 4 steps, the ant must have landed at the previous vertex (which can be A, B, or C) in 3 steps.
- $\square \to \square \to \square \to A \to D$: Let that previous vertex be A. To arrive at A in 3 steps, it must have landed at the previous vertex (which can be B, C, or D) in 2 steps.
- $\square \to \square \to B \to A \to D$: Let that previous vertex be B. To arrive at B in 2 steps, it must have landed at the previous vertex (which can be A, C, or D) in 1 step.
- $\square \to C \to B \to A \to D$: Let that previous vertex be C. To satisfy the cyclic property, arriving at vertex C in 1 step forces the ant to have come from D since the initial and terminal vertices should always be D.

What is the state? In our simulation, we have been keeping track of two parameters: (1) the vertex where the ant is currently standing and (2) the number of steps it has taken to arrive there. The adjoining figure shows the transitions: to arrive at a vertex in exactly n steps, it is imperative for the ant to have landed at the previous vertex — which can be any of the other three vertices — in $n - 1$ steps.



Suppose that the function $f(v, n)$ denotes the number of ways to arrive at vertex v in exactly n steps. The recurrence relations are

$$f(D, n) = f(A, n-1) + f(B, n-1) + f(C, n-1)$$
$$f(A, n) = f(B, n-1) + f(C, n-1) + f(D, n-1)$$
$$f(B, n) = f(A, n-1) + f(C, n-1) + f(D, n-1)$$
$$f(C, n) = f(A, n-1) + f(B, n-1) + f(D, n-1)$$

Notice the uncanny similarity among the equations. By symmetry, vertices A, B, and C are actually equivalent. Taking A to be their representative, we can exploit this observation to reduce the number of function calls and halve the size of the `memo` table:

$$f(D, n) = 3f(A, n-1)$$
$$f(A, n) = 2f(A, n-1) + f(D, n-1)$$

As before, we cap off our solution with the identification of the base cases:

- $f(D, 1) = 0$: It is impossible to arrive at D in 1 step since D itself is the initial vertex.
- $f(A, 1) = 1$: The path is $D \rightarrow A$.

Using memoization, the core method can, thus, be implemented as follows:

```
memo = [[-1 for i in range(int(1e7 + 1))] for j in range(2)]
D = 0
A = 1
MOD = int(1e7 + 9)

def num_ways(vertex, n):
    if memo[vertex][n] == -1:
        if vertex == D and n == 1:
            answer = 0
        elif vertex == A and n == 1:
            answer = 1
        elif vertex == D:
            answer = 3 * num_ways(A, n - 1) % MOD
        else:
            answer = ((2 * num_ways(A, n - 1) % MOD + num_ways(D, n - 1) %
                MOD) % MOD)

        memo[vertex][n] = answer

    return memo[vertex][n]
```

💭 Competitive Programming Meets Contest Math

A probability problem similar to Tetrahedron appeared in the 1985 American Invitational Mathematics Examination. Try to generalize this question and solve it via DP.

Contest BAABAA...

Adapted from the 2008 American Invitational Mathematics Examination (AIME)

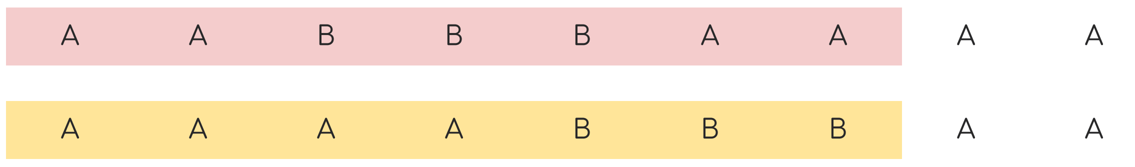
Consider strings that consist entirely of A's and B's and that have the property that every run of consecutive A's has even length, and every run of consecutive B's has odd length. Examples of such strings are AA, B, and $AABAA$, while $BBAB$ is not such a string. How many such strings have length n? If $n = 14$, then there are 172 such strings.

📝 Discussion

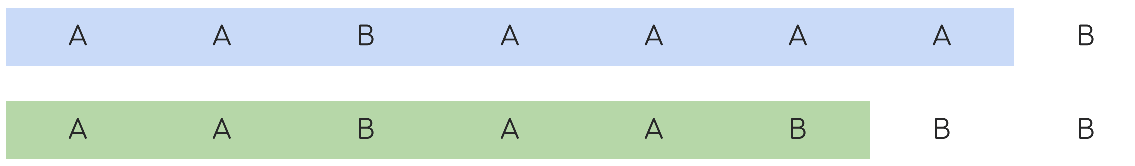
The gist of DP (and recursion) is reframing the original task into smaller versions of itself. This drives the heart of our thought process: our initial objective is to devise a strategy for generating a valid string[4] from another valid (albeit shorter) string, which we will call the "base string" for simplicity. The base string b of a valid string s is $s[0:x]$ [5] for some x such that b is the longest proper substring of s that also satisfies the problem conditions.

Since the rules for A's and B's are different, it may be helpful to consider two cases: strings that end with A and those that end with B.

- There are two ways to generate a valid string that ends with A: (1) having the base string end with an A then appending two A's or (2) having the base string end with a B then appending two A's. These are visualized below; the shaded substrings are the base strings.



- There are two ways to generate a valid string that ends with B: (1) having the base string end with an A then appending a B or (2) having the base string end with a B then appending 2 B's.



For the state, it is clear that we have to keep track of the last letter — but, aside from this, what is the parameter that changes at every subtask? What gets altered as we go from the original string to the base string? As the visualizations suggest, it is the string length.

[4] By "valid string," we are pertaining to a string that satisfies the conditions in the problem.
[5] We are using Python's string slice notation.

Reformulating our casework in terms of states and transitions:

- A valid string that ends with A and has length n is built on top of a valid string that ends with A and has length $n-2$ or on top of a valid string that ends with B and has length $n-2$.
- A valid string that ends with B and has length n is built on top of a valid string that ends with A and has length $n-1$ or on top of a valid string that ends with B and has length $n-2$.

Suppose that the function $f(e,\, n)$ denotes the number of valid strings that end with e and have length n. The recurrence relations are

$$f(A,\, n) = f(A,\, n-2) + f(B,\, n-2)$$
$$f(B,\, n) = f(A,\, n-1) + f(B,\, n-2)$$

Finally, we identify the base cases:

- $f(A,\, 1) = 0$: There should be an even number of A's.
- $f(A,\, 2) = 1$: The two-character string AA is valid.
- $f(B,\, 1) = 1$: The single-character string B is valid.
- $f(B,\, 2) = 0$. There should be an odd number of B's.

Using memoization, the core method can, thus, be implemented as follows:

```
memo = [[-1 for i in range(MAX_N + 1)] for j in range(2)]
A = 0
B = 1

def num_strings(end_letter, n):
    if memo[end_letter][n] == -1:
        if ((end_letter == A and n == 1) or
            (end_letter == B and n == 2)):
            answer = 0
        elif ((end_letter == A and n == 2) or
            (end_letter == B and n == 1)):
            answer = 1
        elif end_letter == A:
            answer = num_strings(A, n - 2) + num_strings(B, n - 2)
        else:
            answer = num_strings(A, n - 1) + num_strings(B, n - 2)

        memo[end_letter][n] = answer

    return memo[end_letter][n]
```

A note on terminology: The state is the *combination* of the parameters, as opposed to the individual parameters themselves. Hence, in this problem, the state is the combination of the last letter and the string length.

Classic Levenshtein Distance

Named after the Russian mathematician Vladimir Levenshtein, the Levenshtein distance is the minimum number of single-character edits (insertions, deletions, or substitutions) to convert one string to another. To illustrate:

- The Levenshtein distance between `DOG` and `DOGS` is 1:
  - `DOG` → `DOGS` (insertion)
- The Levenshtein distance between `CLARISSA` and `CARISSA` is 1:
  - `CLARISSA` → `CARISSA` (deletion)
- The Levenshtein distance between `AIEP` and `AMEP` is 1:
  - `AIEP` → `AMEP` (substitution)
- The Levenshtein distance between `KAGOME` and `KIKYO` is 5:
  - `KAGOME` → `KIGOME` (substitution)
  - `KIGOME` → `KIKOME` (substitution)
  - `KIKOME` → `KIKYOME` (insertion)
  - `KIKYOME` → `KIKYOE` (deletion)
  - `KIKYOE` → `KIKYO` (deletion)

📝 **Discussion**
Fortunately, the problem is explicit with the transitions: inserting, deleting, or substituting a single character. Unfortunately, a lot of things are changing when a word undergoes conversion, and it can be rather difficult to pinpoint which parameters are relevant to the state. To reiterate, the core of DP (and recursion) is to express the original task using smaller versions of itself — that is, to express the Levenshtein distance between two strings in terms of distances between their substrings. The actual sequence of operations involved in the conversion is abstracted into a black box[6].

To be consistent with our style of analysis so far, we start editing from the last character, going to the left. For our simulation, suppose our two strings are `KAGOME` and `KIKYO`:

- Insertion: For `KAGOME` to become `KIKYO`, the cheapest insertion is to append an `O`. As implied by the visualization below, the subtask boils down to solving for the Levenshtein distance between `KAGOME` and `KIKY`.

K	A	G	O	M	E	O
		K	I	K	Y	O

- Deletion: For `KAGOME` to become `KIKYO`, we delete the terminal `E`. As implied by the visualization below, the subtask boils down to solving for the Levenshtein distance between `KAGOM` and `KIKYO`.

K	A	G	O	M	E
K	I	K	Y	O	

[6] Nevertheless, the sequence of operations can be recovered via *backtracking* although this is outside the scope of our handout. For some examples, you may refer to these lecture notes.

- Substitution: For `KAGOME` to become `KIKYO`, the cheapest substitution is to replace the terminal `E` with an `O`. As implied by the visualization below, the subtask boils down to solving for the Levenshtein distance between `KAGOM` and `KIKY`.

K	A	G	O	M	O
	K	I	K	Y	O

What then is the state? The visualizations above suggest that we need to keep track of the substrings being compared, with special attention to the indices of their last characters relative to the original string. Given strings s_1 and s_2 with lengths n_1 and n_2, respectively, the transitions can be formalized as follows (add 1 to count the operation itself):

- Insertion: Add 1 to the distance between $s_1[0 : n_1]$ and $s_2[0 : n_2 - 1]$ [7].
- Deletion: Add 1 to the distance between $s_1[0 : n_1 - 1]$ and $s_2[0 : n_2]$.
- Substitution: Add 1 to the distance between $s_1[0 : n_1 - 1]$ and $s_2[0 : n_2 - 1]$.

To avoid the cost of having entire substrings as the parameters, let $d(n_1, n_2)$ denote the Levenshtein distance between $s_1[0 : n_1]$ and $s_2[0 : n_2]$. The recurrence relation is

$$d(n_1, n_2) = 1 + \min\left(d(n_1, n_2 - 1),\ d(n_1 - 1, n_2),\ d(n_1 - 1, n_2 - 1)\right).$$

We cap off our solution by listing the special and base cases:

- $d(0, n_2) = n_2$: If s_1 is empty, then all the characters in s_2 have to be inserted.
- $d(n_1, 0) = n_1$: If s_2 is empty, then all the characters in s_1 have to be deleted.
- If $s_1[n_1 - 1] = s_2[n_2 - 1]$, then $d(n_1, n_2) = d(n_1 - 1, n_2 - 1)$. If their last characters are already identical, then we just move on to the preceding characters.

Using memoization, the core method can, thus, be implemented as follows:

```python
memo = [[-1 for i in range(MAX_LEN_2 + 1)] for j in range(MAX_LEN_1 + 1)]

def levenshtein(n1, n2):
    if memo[n1][n2] == -1:
        if n1 == 0:
            answer = n2
        elif n2 == 0:
            answer = n1
        elif str1[n1 - 1] == str2[n2 - 1]:
            answer = levenshtein(n1 - 1, n2 - 1)
        else:
            answer = 1 + min(levenshtein(n1, n2 - 1),
                levenshtein(n1 - 1, n2), levenshtein(n1 - 1, n2 - 1))

        memo[n1][n2] = answer

    return memo[n1][n2]
```

[7] We are using Python's string slice notation.

💭 More Than Just a Distance

Aside from being used in some spell checkers, Levenshtein distance is also invoked in bioinformatics; one of its applications is in setting thresholds for approximate DNA and RNA string matching. Dynamic programming itself has been instrumental in developing efficient algorithms in key bioinformatics research areas, such as short read mapping and RNA structure prediction.

🎒 KNAPSACK-STYLE DP

Let us now explore two variations of another DP classic: the knapsack problem.

Classic 0-1 Knapsack Problem

Suppose that we have n items, each with a positive value v_i and positive weight w_i. Find the maximum total value of all the items that can fit inside a knapsack, with the restriction that their total weight should not exceed W. For instance, consider these items:

Item	1	2	3	4
Values	10	40	30	50
Weights	5	4	6	3

If the knapsack can carry only up to a total weight of 10, the maximum total value is 90, which is achieved by including items 2 and 4.

📝 **Discussion**

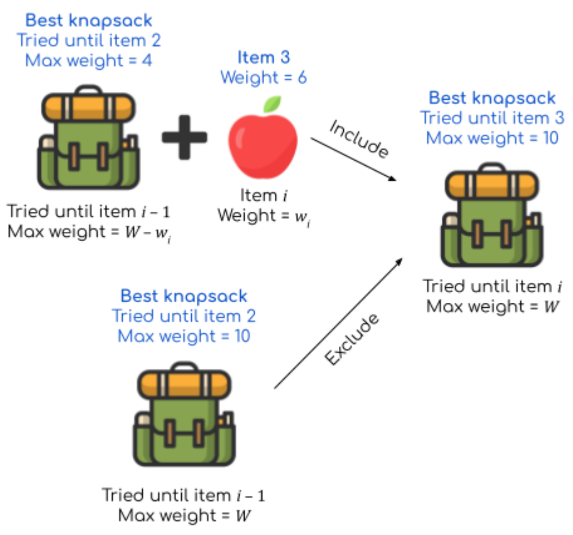
As tempting as it may seem, a greedy algorithm will not work[8]. Again, the crux of DP is rethinking the original task in terms of smaller versions of itself. In this regard, we ground our thought process on devising a strategy to determine the best knapsack[9] based on other best knapsacks (albeit with smaller weight capacities). This already tells us one of the parameters that we have to keep track: the knapsack's weight capacity. It might seem that we also have to know the items inside it, but it is too gnarly to deal with lists!

Instead of asking, "What items are inside our knapsack?" a more astute question would be "Have we *tried* stashing items 1 and 2?" The natural follow-up would then be "Have we *tried* stashing items 1, 2, and 3?" The emphasis on the word *tried* is crucial. *Trying* to stash items 1, 2, and 3 does not necessarily translate to including all three items in the knapsack. For all we know, the value is maximized by adding just one or perhaps two — the beauty of recursion is that it abstracts away these nitty-gritty details.

Since the order of adding items is immaterial to the answer, we can condense our question to "Have we tried stashing until item 3?" with the assumption that this covers all the items numbered from 1 up to 3.

[8] A greedy algorithm does work, however, for the *fractional* knapsack problem.
[9] By "best knapsack," we are pertaining to the knapsack that maximizes the total value.

In brief, for the state, we are keeping track of (1) the last item that we tried to include[10] and (2) the knapsack's weight capacity. The action *tried* also captures the transitions: the item can either be included or excluded. The figure on the right provides a visualization, alongside the generalization of our analysis.



Suppose that the function $v(i, W)$ denotes the maximum total value after trying to include until item i without exceeding the weight capacity W. More formally, trying to include until item i is equivalent to getting a subset of {item 1, item 2, ..., item i}. Note that:

- Item i is included: Its value is added, and its weight reduces the weight capacity.
- Item i is excluded: No value is added, and its weight does not affect the capacity.

The recurrence relation is formed by selecting the better decision between these two:

$$v(i, W) = \max\left(v_i + v(i-1, W - w_i),\ \ v(i-1, W)\right).$$

Finally, we identify the special and base cases:

- $v(0, W) = 0$: The value is 0 if no item is included in the knapsack.
- If $w_i > W$, then $v(i, W) = v(i-1, W)$: If the weight of item i is greater than the weight capacity, we are forced to exclude it since all the weights are positive.

Using memoization, the core method can be implemented as follows (for consistency with our designation of the first item as item 1, `weights[0]` and `values[0]` should be dummy values only):

```python
memo = [[-1 for i in range(MAX_W + 1)] for j in range(MAX_N + 1)]

def max_value(item_num, weight_capacity):
    if memo[item_num][weight_capacity] == -1:
        if item_num == 0:
            answer = 0
        elif weights[item_num] > weight_capacity:
            answer = max_value(item_num - 1, weight_capacity)
        else:
            answer = max(values[item_num] + max_value(item_num - 1,
                   weight_capacity - weights[item_num]),
                   max_value(item_num - 1, weight_capacity))

        memo[item_num][weight_capacity] = answer

    return memo[item_num][weight_capacity]
```

[10] This is implemented by tracking its (one-based) index.

Classic Subset Sum Problem

Given a set[11] of n positive integers and a target sum S, determine if it has a subset such that the sum of its elements is equal to S. For instance, if the set is $\{2, 3, 5, 7, 10\}$ and the target sum S is 12, then the answer is affirmative since the sum of 5 and 7 is 12. Note that an empty sum is allowed, i.e., the sum of the elements "inside" a null set is 0.

📝 Discussion

Following the insight that we have gained from our solution to the 0-1 knapsack problem, we have to keep track of two parameters for the state: (1) the last element that we tried to include in our summation[12] and (2) the target sum. For the transitions, the element can either be included or excluded.

Suppose that the function $f(i, S)$ returns True if the target sum S can be achieved after trying to include until element i; otherwise, it returns False. More formally, trying to include until element i is equivalent to getting a subset of $\{e_1, e_2, ..., e_i\}$. The recurrence relation can, thus, be established by taking the disjunction (logical or)[13] of the two possible transitions:

$$f(i, S) = f(i-1, S-e_i) \lor f(i-1, S).$$

Finally, we identify the special and base cases:

- $f(i, 0) = \text{True}$: The empty sum, which is allowed, is equal to 0.
- If $S \neq 0$, then $f(0, S) = \text{False}$: Unless we are talking about the empty sum, the target sum can never be reached if no element is included.
- If $e_i > S$, then $f(i, S) = f(i-1, S)$. If element i is greater than the target sum, we are forced to exclude it since the set consists only of positive integers.

This implementation of the core method is analogous to the one for the 0-1 knapsack[14]:

```python
memo = [[-1 for i in range(MAX_S + 1)] for j in range(MAX_N + 1)]

def is_sum_possible(element_num, target_sum):
    if memo[element_num][target_sum] == -1:
        if target_sum == 0:
            answer = True
        elif element_num == 0:
            answer = False
        elif elements[element_num] > target_sum:
            answer = is_sum_possible(element_num - 1, target_sum)
        else:
            answer = (is_sum_possible(element_num - 1, target_sum -
                elements[element_num]) or
                is_sum_possible(element_num - 1, target_sum))

        memo[element_num][target_sum] = answer

    return memo[element_num][target_sum]
```

[11] Technically, it is a *multiset* since multiple instances of the same element are allowed.
[12] This is implemented by tracking its (one-based) index.
[13] In discrete mathematics, the symbol for this operation is $\lor$.
[14] Since we designated the first element as element 1, `elements[0]` should be a dummy value only.

💭 DP and NP

You may have heard of the million-dollar P vs. NP or the notorious traveling salesman! One thing that the traveling salesman problem shares with the 0-1 knapsack and subset sum problems is that they all belong to a class of enigmas described as NP-complete. This feature from MIT News provides some insights on why this matters.

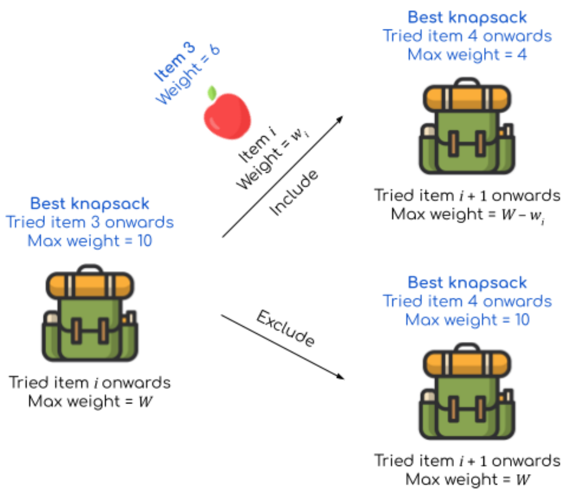
Even though NP-complete problems are juggernauts, all hope is not lost. After all, our DP solutions to the 0-1 knapsack and subset sum problems perform reasonably well, provided that the input size is not astronomical. Their runtime is formally described as pseudopolynomial — interestingly, all the known pseudopolynomial-time algorithms for NP-hard problems employ DP. For a more detailed discussion, you may refer to these lecture notes from the University of California, Berkeley.

👈👉 FORWARD VS. BACKWARD RECURSION

So far, we have consistently been employing backward recursion; our analysis begins with the last state, and we attempt to reframe it in terms of previous states. Consequently, the base cases pertain to the initial states. By symmetry, it is also possible to think "forward" by starting with the first state and expressing it in terms of succeeding states.

To illustrate, let us revisit our solution to the 0-1 knapsack, the heart of which is trying to include *until* item i. In a forward recursive approach, we try to include *starting* from item i, which translates to getting a subset of {item i, item $i + 1$, ..., item n}.

In this regard, the function $v(i, W)$ becomes the maximum total value after trying to include item i onwards. As seen in the adjoining figure, the logic of including and excluding this item remains unchanged, but the transition is now from i to $i + 1$. Ergo, the recurrence relation is



$$v(i, W) = \max\left(v_i + v(i + 1, W - w_i),\ \ v(i + 1, W)\right).$$

There are also slight changes to the special and base cases:
- If $i > n$, then $v(i, W) = 0$. There are only n items.
- If $w_i > W$, then $v(i, W) = v(i + 1, W)$: If the weight of item i is greater than the weight capacity, we are forced to exclude it since all the weights are positive.

In our backward recursive (original) solution, the answer is $v(n, W)$, where n is the number of items. However, in this forward recursive solution, the answer is $v(1, W)$, which denotes the maximum total value after trying to include item 1 onwards.

The implementation is left as an exercise for the reader.

Which approach is better? As cliche as it may sound, it depends — and it helps to be well versed in both. Most formal computer science literature employ backward recursion, probably in deference to mathematical conventions. But, if we informally survey coding platforms and contest editorials, it seems that there is a growing preference for forward recursion, as it allows for a more straightforward translation of transitions to recurrence relations. This may be advantageous for more "playful" DP tasks, like our finale problem:

Contest Nikola

2008 Croatian Regional Competition in Informatics

Against his own will, Nikola has become the main character in a game. The game is played on a row of N squares, numbered 1 to N. Nikola is initially in square 1 and can jump to other squares. Nikola's first jump must be to square 2. Each subsequent jump must satisfy two constraints:

- If the jump is in the forward direction, it must be one square longer than the preceding jump.
- If the jump is in the backward direction, it must be exactly the same length as the preceding jump.

For example, after the first jump (when on square 2), Nikola can jump back to square 1 or forwards to square 4. For instance, if $N = 6$ (see first example case below), one possible way to reach N would be by using the squares 1 → 2 → 1 → 3 → 6.

Every time he enters a square, Nikola must pay an entry fee. Nikola's goal is to get from square 1 to square N as cheaply as possible. Write a program that determines the smallest total cost for Nikola to get to square N.

Input

The first line contains the integer N, $2 \leq N \leq 1000$, the number of squares. Each of the following N lines contains the entry fee for one square, a positive integer less than 500. The entry fees will be given in order for squares 1 to N.

Output

Output the smallest total cost for Nikola to get to square N.

Sample Input	Sample Output
6 1 2 3 4 5 6	12
8 2 3 4 3 1 6 1 4	14

📝 Discussion

This feels similar to our opening problem (Climbing Stairs), but with some creative twists. We begin by tracing the given example:

- Nikola jumps forward from square 1 to square 2 (1 step).
- He jumps backward from square 2 to square 1 (1 step).
- He jumps forward from square 1 to square 3 (1 + 1 = 2 steps).
- He jumps forward from square 3 to square 6 (2 + 1 = 3 steps).

Our small simulation suggests that, for the state, we have to keep track of (1) the square where Nikola is standing and (2) the number of steps involved in the jump to that square. Fortunately, the problem statement already provides the transitions. As promised, we will express them with forward recursion in mind. If Nikola is currently standing at square s and he jumped to that square in p steps:

- A forward jump will take him to square $s + p + 1$, reaching it in $p + 1$ steps.
- A backward jump will take him to square $s - p$, reaching it in p steps.

Let $f(s, p)$ denote the smallest total cost for Nikola to get from square s to square N if the jump to square s involved p steps. Let e_s be the entry fee for square s. The recurrence relation is obtained by taking the cheaper between the two possible transitions:

$$f(s, p) = e_s + \min\left(f(s + p + 1, p + 1), f(s - p, p)\right).$$

We end this solution by enumerating the special and base cases:

- If $s < 1$ or $s > N$, then $f(s, p) = \infty$. The squares' numbers range from 1 to N only. Setting the return value to an extremely large number is a common DP technique to effectively eliminate illegal positions from the selection of smallest costs.
- $f(N, p) = e_N$. The cost of going from square N to square N is its entry fee.

Since the first jump is a one-step forward jump from square 1 to square 2, the answer is $f(2, 1)$. The implementation via memoization is as follows:

```python
memo = [[-1 for i in range(1001)] for i in range(1001)]
POS_INF = int(1e9)

def nikola(square, num_steps):
    if square < 1 or square > N:
        return POS_INF

    if square == N:
        return entry_fees[N]

    if memo[square][num_steps] == -1:
        answer = entry_fees[square] + min(nikola(square + num_steps + 1,
                 num_steps + 1), nikola(square - num_steps, num_steps))

        memo[square][num_steps] = answer

    return memo[square][num_steps]
```

Here are some notes regarding our implementation:

- Similar to our previous implementations, `entry_fees[0]` should be a dummy value only since we designated the first square as square 1.

- It may be tempting to follow the "template" in our previous implementations and place the highlighted code segments inside the

```
if memo[square][num_steps] == -1
```

  block. However, there are sneaky cases for which this will result in a runtime error. Could you identify these cases? What are some additional caveats that must be taken into consideration when implementing forward recursion?

When implementing DP, negative values for the parameters being tracked are usually red flags, especially if the problem constraints define them as illegal. However, since Python permits negative indexing — for better or for worse — our (buggy) code might already be running into the territory of negative parameter values under our noses.

Be mindful of this, especially if you are coming from a C++ or Java background.

💭 Top-Down vs. Bottom-Up

DP solutions can be implemented in two ways: either via a top-down or a bottom-up approach. In the interest of pedagogy, we have consistently used memoization, which is a top-down approach — both the thought process and the implementation start with the original problem, and the constituent subproblems are solved (or retrieved from the cache) along the way, as needed.

However, since memoization employs recursion, the risk of running into a stack overflow should not be discounted; in fact, our solution to Tetrahedron exceeds Python's default maximum recursion depth when $n \geq 970$. Moreover, although the overhead from function calls is generally negligible in competitive programming, critical systems in the real world are less forgiving. As operations scale, the accumulation of these "negligible" inefficiencies may ultimately lead to a "death by a thousand cuts."

A bottom-up approach avoids this by first computing the solutions to the subproblems and systematically combining them until the original problem is solved — albeit with the proviso that the programmers are now responsible for figuring out the order in which the subproblems have to be solved and combined to ensure correctness and optimality. As such, a bottom-up DP solution utilizes iteration (loops) instead of recursion.

1. To learn more about the differences between these approaches, you may refer to this discussion on Stack Overflow.
2. What are their respective advantages and disadvantages?
3. Translate the solutions in this handout to employ tabulation (bottom-up) instead of memoization (top-down).

👩‍💻 EXERCISES

Try solving these challenges using tabulation (bottom-up) and memoization (top-down).

Straightforward Memoization

1. sqrt log sin (University of Valladolid [UVa] Online Judge 11703)
   The expected value of $x_{1000000}$ is 648345.

   Using memoization will exceed Python's default maximum recursion depth. For this exercise — and only for the purposes of this exercise — you are allowed to change this default value to push the envelope. In actual contests and real-world tasks, you should utilize bottom-up tabulation for this type of problem.

Binomial Coefficient

The mathematical solution to Exercise 3 is via stars-and-bars (or balls-and-urns). Even if you are already familiar with this technique, a good exercise would be to reimagine the problem in terms of states and transitions.

2. Combination. Find the number of ways in which r objects can be chosen from among n objects. For example, there are 15 ways to choose 4 objects from among 6 objects.

3. How do you add? (UVa Online Judge 10943)

DP and Strings

Exercise 5 is related to Catalan numbers, named after the French-Belgian mathematician Eugène Charles Catalan. Even if you are already aware of this connection, a good exercise would be to derive the recurrence relation from scratch.

4. Vitas Words (Generalization of an item from the 21st Philippine Mathematical Olympiad Area Stage). A *Vitas* word is a string of letters that satisfies the following conditions:

   - It consists of only the letters B, L, R.
   - It begins with a B and ends in an L.
   - No two consecutive letters are the same.

   How many Vitas words are there with n letters? For instance, there are 341 Vitas words with 11 letters.

5. Dyck Words. Named after the German mathematician Walther von Dyck, a *Dyck word* is a balanced string of left and right parentheses. For example, (((()))), ((()())), and ()()()() are Dyck words; in total, there are 14 Dyck words of length 8. How many Dyck words of length n are there?

DP and Subsequences

The optimal solution to Exercise 6 uses Kadane's algorithm, a DP and greedy algorithm introduced by Carnegie Mellon University professor Jay Kadane. It can also be solved via a divide-and-conquer approach (although this would probably lead to a timeout verdict).

6. The jackpot (UVa Online Judge 10684)
7. Longest Increasing Subsequence (Classic/Leetcode 300)

```
-- return 0; --
```

Prepared by ***Mark Edward M. Gonzales***