Asian MathSci League, Inc (AMSLI)
Website: amsliphil.com
Email address: amsliphil@yahoo.com

Student Copy

AIEP Sendoff HS Session 8

INTRODUCTION TO THE DESIGN OF ALGORITHMS

"Programming is the art of algorithm design and the craft of debugging errant code."

~ Ellen Ullman

In session 6, we introduced the notion of algorithm efficiency and how this is described using the big O notation. Admittedly, it is a bit difficult to wrap our heads around the gap between linear and linearithmic rates of growth — just how big of an improvement does it give us? This can be answered via a posteriori analysis, running an actual experiment.

The table below shows data from an experiment comparing the machine execution time of six sorting algorithms (written in C, one of the fastest high-level programming languages). The time complexities in parentheses pertain to the average case:

Number of Values Sorted	Average Machine Execution Time (seconds)					
	Bubble $O(n^2)$	Selection $O(n^2)$	Insertion $O(n^2)$	Shell $O(n^{1.25})$	Merge $O(n \log n)$	Quick $O(n \log n)$
2^{13}	0.339062	0.092188	0.060937	0.004687	0.001563	0.001563
2^{14}	1.387500	0.348438	0.229687	0.007813	0.001563	0.003125
...						
2^{19}	1561.425000	351.946875	239.129687	0.231250	0.142187	0.114063
2^{20}	6369.376562	1461.418750	982.882813	0.482812	0.265625	0.248438

The complexity of Shellsort is conjectured based on experiments; its theoretical value remains an open problem.

At relatively small input sizes, the advantage of linearithmic-time over quadratic-time algorithms is negligible since the differences in machine execution time are well under a second. However, as the number of values to be sorted becomes larger, the difference is magnified; at 2^{20} values, bubble sort clocked half an hour whereas quicksort registered only a quarter of a second. This motivates our topic: the design of (good) algorithms.

Prepared by Mark Edward M. Gonzales

WARM-UP: Primality Testing and $6k \pm 1$

We begin our discussion with a warm-up: How do we determine if a number is prime? We can do trial division, testing all candidate factors up to the square root of the number. It is easy to see that this is slow; what is not easy to see is how we can improve it.

Let us list down the first few prime numbers greater than 3: 5, 7, 11, 13, 17, 19, 23, 29, 31. Notice that they are all either one less than or one more than a multiple of 6. In fact, this is not a mere coincidence. Suppose x is a positive integer greater than 3:

a) If $x \equiv 0 \pmod 6$, $x \equiv 2 \pmod 6$, or $x \equiv 4 \pmod 6$, then it is even, so it is composite.
b) If $x \equiv 3 \pmod 6$, then it is divisible by 3, so it is composite.
c) Ergo, primes greater than 3 are certainly either $x \equiv 5 \pmod 6$ or $x \equiv 1 \pmod 6$.

Combined with the fact that every number can be decomposed via prime factorization (Fundamental Theorem of Arithmetic), this leads to a crucial optimization. Ideally, instead of testing all positive integers up to the square root of the number, we only need to test if the number is divisible by a prime number less than or equal to its square root.

However, using prime numbers to test primality tends towards circular reasoning. Thus, we can "loosen" our procedure; rather than testing all positive integers, we are only interested in those of the form $6k \pm 1$. While not all numbers of the form $6k \pm 1$ are prime, all primes greater than 3 are of the form $6k \pm 1$.

```
def is_prime_optimized(num):
    if num <= 3:
        return num > 1
    if num % 2 == 0 or num % 3 == 0:
        return False
    candidate_factor = 5
    while candidate_factor * candidate_factor <= num:
        if num % candidate_factor == 0 or num % (candidate_factor + 2) == 0:
            return False
        candidate_factor = candidate_factor + 6
    return True
```

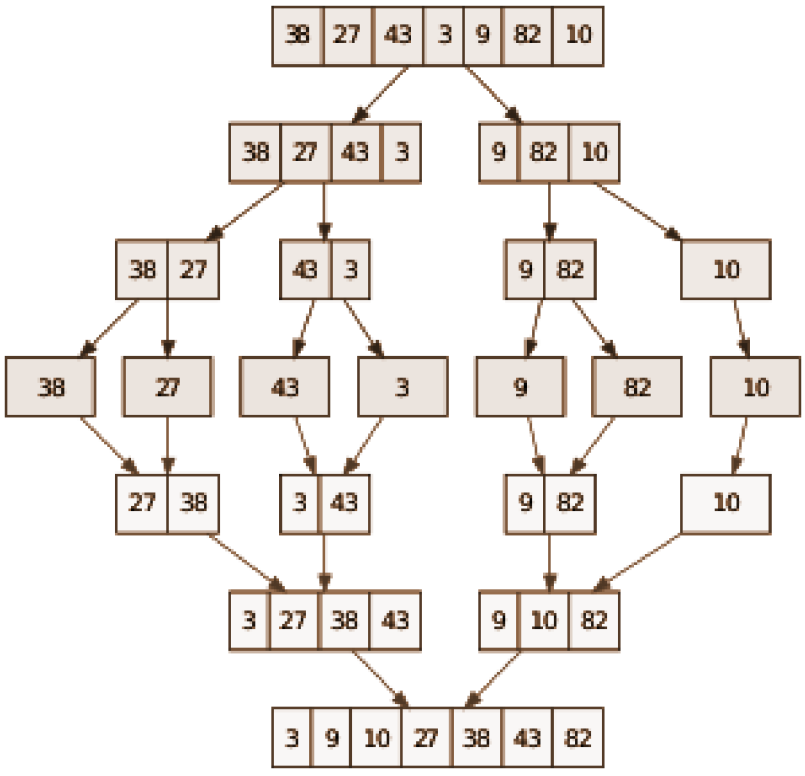
Is this optimization the best that we can do? Certainly not. Primality testing is a crucial topic in computer science, serving as the cornerstone of modern computer security. More sophisticated algorithms, such as Fermat's, Miller-Rabin, and AKS, have been devised by borrowing both age-old and novel results in number theory.

DIVIDE AND CONQUER

The divide-and-conquer algorithmic paradigm, which fittingly borrows its name from the age-old tactic employed by colonizers, consists of two steps:

1. **Divide**. The problem is recursively broken down into non-overlapping subproblems.
2. **Conquer**. The solutions to these subproblems are then "fused" together in order to solve the original problem.

Mergesort



www.teachyourselfpython.com/challenges.php?a=01_Solve_and_Learn&t=7-Sorting_Searching_Algorithms&s=01c_Merge_Sort
[Slightly recolored]

An excellent example of a divide-and-conquer algorithm is mergesort. Introduced by the Hungarian-American polymath and computer science pioneer John von Neumann in the 1940s, the divide-and-conquer idea is as follows:

1. Divide. Split the list of *n* data values into two sublists, then into four sublists, and so on, until *n* single-element sublists are produced. The trace of this step is shaded in a darker color in the figure above.

2. Conquer. The *n* single-element sublists are combined (in the reverse of the manner in which the original list was divided) in a sorted fashion until they form a single *n*-element list. This is the sorted list that we seek. The trace of this step is shaded in a lighter color in the figure above.

```
def merge(array, temp_array, left_start, left_end, right_start, right_end):
    temp_index = left_start
    left_start_copy = left_start

    while left_start <= left_end and right_start <= right_end:
        if array[left_start] <= array[right_start]:
            temp_array[temp_index] = array[left_start]
            left_start = left_start + 1
            temp_index = temp_index + 1
        else:
            temp_array[temp_index] = array[right_start]
            right_start = right_start + 1
            temp_index = temp_index + 1

    while left_start <= left_end:
        temp_array[temp_index] = array[left_start]
        left_start = left_start + 1
        temp_index = temp_index + 1
```

```
    while right_start <= right_end:
        temp_array[temp_index] = array[right_start]
        right_start = right_start + 1
        temp_index = temp_index + 1

    while left_start_copy <= right_end:
        array[left_start_copy] = temp_array[left_start_copy]
        left_start_copy = left_start_copy + 1

def mergesort_helper(array, temp_array, left, right):
    if left < right:
        middle = (left + right) // 2
        mergesort_helper(array, temp_array, left, middle)
        mergesort_helper(array, temp_array, middle + 1, right)
        merge(array, temp_array, left, middle, middle + 1, right)

def mergesort(array):
    temp_array = [None] * len(array)
    mergesort_helper(array, temp_array, 0, len(array) - 1)
```

Its average-case performance is $O(n \log n)$, and linearithmic time is actually the best that general-purpose, comparison-based sorting algorithms can achieve on average. Besides its efficiency, one of its advantages is its characteristic as a **stable sorting algorithm**, that is, repeated values appear in the same order both in the input and in the output.

> For example, we have the following students: Elaina, Emilia, and Merlin. We want to arrange them in ascending order based on their grades — but it so happened that all three of them have the same grade.
>
> Non-stable algorithms will not necessarily preserve the order in which they appear; thus, the output might be Emilia, Merlin, then Elaina. On the contrary, stable algorithms, like mergrsort, will preserve this order; ergo, the output will still be Elaina, Emilia, then Merlin.

For this reason, mergesort is a widely used under-the-hood algorithm in programming languages. The built-in `sort()` function of Python is Timsort, a hybrid of mergsort and insertion sort (cf. Session 6, Exercise #7). Java SE 7 also borrows the idea of Timsort in its own implementation of `Collections.sort`.

Largest Continuous Sum

Recall the largest continuous sum problem from Session 6. In the handout, we showed the cubic-, quadratic-, and linear-time solutions. As promised, we are now going to take a look at the linearithmic-time solution, which follows this idea:

1. Divide. In mergesort, we divided the problem by recursively "chopping" it into left and right sublists. However, the largest continuous sum problem is slightly more complicated than that. Notice that the desired segment is not isolated on either the left (green) or right (blue) portion; it can span across the center (white font).

-1	2	-3	4	5	6	7	-8

   Therefore, there are three subproblems: getting the maximum in the left half, the maximum in the right half, and the maximum that spans across the center.

   To get the maximum spanning across the center, we have to keep track of the maximum sum in the left half "touching" the center and the maximum sum in the right half "touching" the center; in our Python implementation, we we will call these `max_sum_left_edge` and `max_sum_right_edge`, respectively.

2. Conquer. Take the maximum among the three subproblems.

```
def largest_cont_sum_helper(array, left, right):
    if left == right:
        return max(0, array[left])
    middle = (left + right) // 2
    max_sum_left = largest_cont_sum_helper(array, left, middle)
    max_sum_right = largest_cont_sum_helper(array, middle + 1, right)
```

```
    sum_left_edge = 0
    sum_right_edge = 0
    max_sum_left_edge = 0
    max_sum_right_edge = 0

    for i in range(middle, left - 1, -1):
        sum_left_edge = sum_left_edge + array[i]
        max_sum_left_edge = max(max_sum_left_edge, sum_left_edge)
    for i in range(middle + 1, right + 1):
        sum_right_edge = sum_right_edge + array[i]
        max_sum_right_edge = max(max_sum_right_edge, sum_right_edge)

    return max(max(max_sum_left, max_sum_right), max_sum_left_edge \
                                                 + max_sum_right_edge)

def largest_cont_sum(array):
    return largest_cont_sum_helper(array, 0, len(array) - 1)
```

As a final note, notice how we implemented our divide-and-conquer algorithms. There is a "helper" routine that has more parameters (besides the list, the indices are also passed); it is also the one that we call recursively. Then, we have a "wrapper" routine with the list as its only parameter; its job is to call the "helper." Be extra careful in specifying base cases; otherwise, the recursion will not terminate, and we will get nothing but a stack overflow!

GREEDY ALGORITHM

A greedy algorithm solution relies on the optimal solution at the given moment (properly termed as the locally optimal choice). From this cursory description, we can surmise that it is applicable if and only if the problem has the following properties:

1. **Optimal substructure**. Having this means that the best solution to the entire problem can be built by exploiting the best solutions to its subproblems.

2. **Greedy choice property**. This is the attribute that makes this paradigm absolutely "greedy." We are only concerned with what is best at that given moment and that moment alone. This makes a greedy algorithm "short-sighted" — moreover, it never turns back. Once that "best at that given moment" has already been found, it is set in stone and will not be changed no matter how far we progress with the solution.

Fractional Knapsack Problem

Let us start our examples of greedy algorithms with an all-too-familiar scenario. Suppose Kino is a student with four assignments due tomorrow. However, she only has 60 minutes left until the deadline. The number of marks per item and the number of minutes it takes to finish each question are shown in the table below:

Item	1	2	3	4
Marks	7	15	12	22
Number of minutes	10	25	20	40

Kino wants to maximize the number of marks that she can obtain within the given time constraint. Partial points are awarded; for instance, solving "half" of problem 3 gives 6 points and takes only 10 minutes. This is the premise of the fractional knapsack problem (the adjective "fractional" comes from the fact that partial points are awarded).

With some mathematical insight, we can deduce that getting the ratio of the marks to the number of minutes is key to accomplishing the task. In particular,

1. Arrange the items in a descending fashion based on the ratio of the marks to the number of minutes.
2. Select the items greedily until all the available time is exhausted. If necessary, only a fraction of an item is included.

Since step 1 has an $O(n \log n)$ time complexity (achieved through the use of an efficient comparison-based sorting algorithm, like mergesort) and step 2 has a time complexity of $O(n)$ (since we iterate through the entries), the greedy algorithm solution is linearithmic.

```
def fractional_knapsack(marks, num_minutes, limit_minutes):
    marks_minutes_ratio = []
    for i in range(len(marks)):
        marks_minutes_ratio.append(marks[i] / num_minutes[i])
        marks_minutes_ratio_enum = list(enumerate(marks_minutes_ratio))
    marks_minutes_ratio_enum.sort(key = lambda x : x[1], reverse = True)

    total_minutes = 0
    max_marks = 0
    for i in range(len(marks_minutes_ratio_enum)):
        item_index = marks_minutes_ratio_enum[i][0]
        temp_total_minutes = total_minutes + num_minutes[item_index]
        if temp_total_minutes <= limit_minutes:
            total_minutes = temp_total_minutes
            max_marks = max_marks + marks[item_index]
        else:
            partial = (limit_minutes - total_minutes) / num_minutes[item_index]
            max_marks = max_marks + partial * marks[item_index]
            break

    return max_marks
```

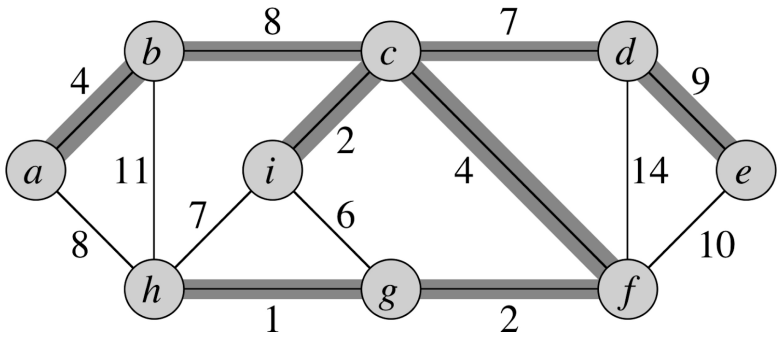
Prim's Algorithm

Created by the Czech mathematician Vojltěch Jarník in 1930 and rediscovered in the late 1950s by Robert Prim and Edsger Dikjstra, Prim's algorithm is used to find the **minimum spanning tree (MST)** of a weighted undirected graph. The MST is a tree that connects all the vertices of the graph in such a way that the total weight of the edges is minimized. In computer science, this is useful when designing networks.

The idea of Prim's algorithm is as follows:

1. Start at an arbitrarily chosen vertex of the graph. This is the first node of our MST.
2. Construct the tree one edge at a time by selecting the minimum-weight edge with respect to all the nodes already on the tree. It is this step that makes it greedy.

Asian MathSci League, Inc (AMSLI)
Website: amsliphil.com
Email address: amsliphil@yahoo.com



https://homes.luddy.indiana.edu/achauhan/Teaching/B403/LectureNotes/09-mst.html

Let us trace how the MST of the graph shown is obtained using Prim's algorithm:

1. Start at vertex *a*.
2. From *a*, go to *b* since 4 is the minimum cost considering *a*.
3. From *b*, go to *c* since 8 is the minimum cost considering *a* and *b*. (It is also correct to go from *a* to *h* since the cost is also 8; the length of our minimum spanning tree remains unaffected. When these cases occur, we implement tie-breaking schemes.).
4. From *c*, go to *i* since 2 is the minimum cost considering *a*, *b*, and *c*.
5. From *c*, go to *f* since 4 is the minimum cost considering *a*, *b*, *c*, and *i*.
6. From *f*, go to *g* since 2 is the minimum cost considering *a*, *b*, *c*, *i*, and *f*.
7. From *g*, go to *h* since 1 is the minimum cost considering *a*, *b*, *c*, *i*, *f*, and *g*.
8. From *c*, go to *d* since 7 is the minimum cost considering *a*, *b*, *c*, *i*, *f*, *g*, and *h*. (Note that going from *h* to *i* is invalid even if the cost is also 7 since doing so would lead to a cycle, violating the acyclic nature of trees.)
9. From *d*, go to *e* since 9 is the minimum cost considering *a*, *b*, *c*, *i*, *f*, *g*, *d*, and *h*. (Note that going from *h* to *a* i invalid even if the cost is just 8 since doing so would lead to a cycle, violating the acyclic nature of trees.)
10. Since all the vertices of the graph have been connected, we have our MST.

Below is a rudimentary working implementation of Prim's algorithm:

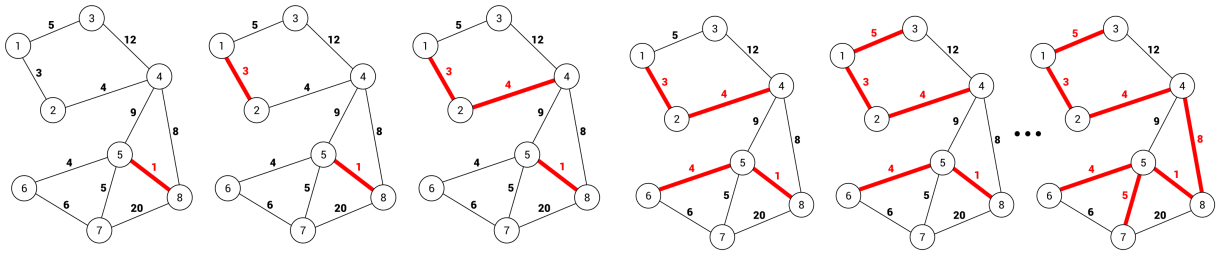
```
def prims_mst_length(adjacency_matrix, start_vertex):
    num_vertices = len(adjacency_matrix)
    total_edges = num_vertices - 1
    curr_num_edges = 0
    mst_length = 0
```

```
in_mst = [False] * num_vertices
in_mst[start_vertex] = True

while curr_num_edges < total_edges:
    minimum_weight = float('inf')
    for vertex in range(num_vertices):
       if in_mst[vertex]:
           for other_vertex in range(num_vertices):
              if not in_mst[other_vertex] \
              and adjacency_matrix[vertex][other_vertex] != 0:
                  if adjacency_matrix[vertex][other_vertex] < minimum_weight:
                      minimum_weight = adjacency_matrix[vertex][other_vertex]
                      to_vertex = other_vertex
    in_mst[to_vertex] = True
    curr_num_edges = curr_num_edges + 1
    mst_length = mst_length + minimum_weight

return mst_length
```

The `in_mst` list in the code above also serves to avoid cycles (instead of using a cycle detection algorithm). As a sidenote, this implementation — which accepts the adjacency matrix representation as parameter — is not really all that efficient. A better way to write Prim's involves the use of fairly advanced data structures, such as a Fibonacci heap.



https://www.oreilly.com/library/view/c-data-structures/9781788833738/a2714e77-af8c-494a-a485-df5521ced3f0.xhtml [Edited]

There are also other algorithms to find the MST. The most famous alternative is Kruskal's algorithm, another greedy algorithm (shown in the image above). However, unlike Prim's, which always maintains a single tree, Kruskal's creates the MST from forests.

DYNAMIC PROGRAMMING

Dynamic programming (DP) is oddly similar to the two previously discussed algorithmic paradigms. Among the three, this is probably the hardest to "unearth." Note that a DP solution can be formulated if and only if the problem has the following properties:

1. **Optimal substructure**. Again, having this means that the best solution to the entire problem can be built by exploiting the best solutions to its subproblems.

2. **Overlapping subproblems**. This distinguishes it from divide and conquer. Since the subproblems are overlapping, this means that the solutions are actually "reused," presenting an opportunity for optimization through two approaches:
    a. **Top-Down Approach**. We start with the main problem then solve it by finding the solutions to its subproblems.
    b. **Bottom-Up Approach**. We start with the subproblems then work our way up to solve the main problem.

In hindsight, we realize that we have already encountered DP problems back in session 6: finding the largest continuous sum and largest continuous product (Session 6, Exercise 4); *it may be a good idea to revisit these problems after this discussion*. Admittedly, DP feels like a "smart" brute-force solution or a "refined" recursion, as seen in these examples:

Fibonacci Sequence

This is one of the first programming challenges given to those learning loops (iterative constructs). Incidentally, this is also one of the representative examples given to illustrate the difference between top-down and bottom-up DP.

Before this, however, let us take a look at the naive recursive approach — easy to write, but at the cost of extreme resource wastage:

```
def fibonacci_bad_recursive(n):
    if n == 1 or n == 2:
       return 1
    return fibonacci_bad_recursive(n - 1) + fibonacci_bad_recursive(n - 2)
```

1. Top-down solution

Suppose we want to calculate the 6th term in the Fibonacci sequence. The naive recursive approach, which has a time complexity of $O(2^n)$, has to calculate the values of all the cells in the table shown **(ignore the annotations and color for now)**. To be more specific, in order to get the value of the cell/s in a row, it first has to get the values of the cell/s in the row below it — in a true recursive fashion.

<table>
<tr><td colspan="8">Term #6</td></tr>
<tr><td colspan="5">Term #5</td><td colspan="3">Term #4
(memoized)</td></tr>
<tr><td colspan="3">Term #4</td><td colspan="2">Term #3
(memoized)</td><td colspan="2">Term #3
(memoized)</td><td>Term #2
(memoized)</td></tr>
<tr><td colspan="2">Term #3</td><td>Term #2
(memoized)</td><td>Term #2
(memoized)</td><td>Term #1
(memoized)</td><td>Term #2
(memoized)</td><td>Term #1
(memoized)</td><td></td></tr>
<tr><td>Term #2</td><td>Term #1</td><td colspan="6"></td></tr>
</table>

Studying the table exposes redundancies (for example, the 2nd term is used to get the 3rd and 4th terms), and this is to be expected since the Fibonacci sequence consists of overlapping subproblems.

Instead of recomputing these values over and over, we can "memorize" them by storing them in a data structure that ideally supports constant lookup time, like a dictionary. Interestingly, the proper term for this "memorization technique" is memoization (there is really no "r").

Study the memoization solution below. It is considered a top-down approach since we start with the n^{th} term and descend to the preceding terms. The pink cells in the table shown correspond to the memoized (memorized) values. Observe how this curbs time complexity to $O(n)$.

```
def fibonacci_memoized(n):
    memoization_table = {}
    memoization_table[1] = 1
    memoization_table[2] = 1
```

```
    if n in memoization_table.keys():
        return memoization_table[n]
    else:
        nth_term = fibonacci_memoized(n - 1) + fibonacci_memoized(n - 2)
        memoization_table[n] = nth_term
        return nth_term
```

2. Bottom-up solution

The top-down solution boasts of an $O(n)$ time complexity, but it suffers from taking up $O(n)$ space as well from its usage of a dictionary. In reality, the iterative solution taught to us in our introductory programming classes is the bottom-up approach:

```
def fibonacci_bottom_up(n):
    t1 = 1
    t2 = 1
    if n == 1:
        return t1
    if n == 2:
        return t2
    for i in range(3, n + 1):
        t2 = t1 + t2
        t1 = t2 - t1
    return t2
```

Notice how the solution "climbs" all the way up to the n^{th} term, making it a bottom-up approach. Furthermore, only a fixed set of atomic variables is used, keeping the space complexity at $O(1)$.

Is this DP solution the fastest way to find a term in the Fibonacci sequence? No. In reality, there is an $O(\log n)$ solution mentioned by Donald E. Knuth in his seminal multi-volume work *The Art of Computer Programming* that makes use of an elegant identity from linear algebra. There is even an $O(1)$ solution using Binet's formula.

Longest Common Subsequence

The largest common subsequence of two strings is the longest sequence of characters that appear in both of them; the characters do not need to occupy consecutive positions. For example, the longest common subsequence of "RESOLVE" and "SOLUTE" is "SOLE". In computer science, this problem is crucial to version control systems that keep track of changes made to a file.

Its nature inspires us to consider two cases, taking full advantage of its recursive flavor:

1. The last letters of the two strings are the same.

   The longest common subsequence (LCS) is the LCS of the first string without its last letter and of the second string without its last letter, with the common last letter appended. For instance, the LCS of "RESOLVE" and "SOLUTE" is the LCS of "RESOLV" and "SOLUT", plus the letter "E".

2. The last letters of the two strings are different.

   This one requires a little more thought. Suppose we are comparing the words "RESOLVE" and "SOLVENT". Since their last letters are different, a common subsequence can be extracted by taking the LCS of "RESOLV" (its last letter is removed) and "SOLVENT". Another common subsequence can be extracted by taking the LCS of "RESOLVE" and "SOLVEN" (its last letter is removed).

   The LCS is the longer of these two common subsequences. In general, if the last letters of the two strings are different, the LCS is the longer between *(i)* the LCS of the first string without its last letter and the second string *(ii)* the LCS of the first string and the second string without its last letter.

Always remember to specify the base case. In this task, it occurs when at least one of the strings is empty. Since we are after a DP solution, our goal is to reuse the values obtained at every subproblem to avoid expensive recomputations. Once again, we store them in a data structure; this time, we are going to use a table — implemented as a two-dimensional array in C-style languages or as a list of lists in Python.

For simplicity, our worked example and implementation will focus only on the length of the LCS. Consider the table found on the next page:

	Ø	S	O	L	U	T	E
Ø	0	0	0	0	0	0	0
R	0	0	0	0	0	0	0
E	0	0	0	0	0	0	1
S	0	1	1	1	1	1	1
O	0	1	2	2	2	2	2
L	0	1	2	3	2	2	2
V	0	1	2	3	3	3	3
E	0	1	2	3	3	3	4

The first row and first columns merely handle the base case (the symbol Ø stands for an empty character) so they are not all that interesting. Let us focus on the second row:

a. LCS of R and S follows case 2. Thus, we want the longer between the LCS of Ø and S, and the LCS of R and Ø. Both have lengths 0; thus, we enter 0 for that cell.
b. LCS of R and SO follows case 2. Thus, we want the longer between the LCS of Ø and SO, and the LCS of R and S (shaded in blue). Both have lengths 0; thus, we enter 0 for that cell.
c. LCS of R and SOL follows case 2. Thus, we want the longer between the LCS of Ø and SOL, and the LCS of R and SO (shaded in red). Both have lengths 0; thus, we enter 0 for that cell.
d. All the other entries that follow case 2 are supplied in the same fashion. Observe that, in terms of the table, we are taking the larger between the cell above it and the cell to the left of it.

Case 1 is a little bit more elusive. We consider two instances of it:

a. Focus on the yellow-shaded cells, and consider the LCS of RE and SOLUTE. The length of the LCS of R and SOLUT is 0; thus, the length of the LCS of RE and SOLUTE is 1 (since their common last letter is appended).

b. Focus on the gray-shaded cells, and consider the LCS of RESO and SO. The length of the LCS of RES and S is 1; thus, the length of the LCS of RESO and SO is 2 (since their common last letter is appended).
c. All the other entries that follow case 1 are supplied in the same fashion. Observe that, in terms of the table, we are just adding 1 to the value of the cell located to its northwest.

We implement this bottom-up algorithm in Python:

```
def lcs_length(str1, str2):
    table = [[0] * (len(str2) + 1) for i in range(len(str1) + 1)]
    for i in range(1, len(str1) + 1):
        for j in range(1, len(str2) + 1):
            if str1[i - 1] == str2[j - 1]:
                table[i][j] = table[i - 1][j - 1] + 1
            else:
                table[i][j] = max(table[i - 1][j], table[i][j - 1])

    return table[len(str1)][len(str2)]
```

The DP solution reduces the time complexity to $O(mn)$, where *m* and *n* are the lengths of the strings, at the cost of $O(mn)$ space complexity as well. Compare this to the $O(m \cdot 2^n)$ time complexity that results from a naive brute-force approach of going through all the 2^n subsequences of one string and checking if the other string also has them.

NOT SO FAST!

1. The design and analysis of algorithms open up the nexus between mathematics and computer science. Our discussion of some well-known algorithms was limited to presenting the motivation, a walkthrough, and the implementation in Python. We neither proved their correctness nor analyzed their efficiency rigorously.

2. One way to prove the correctness of an algorithm, especially those that are based on recursive paradigms, is through mathematical induction; for iterative solutions, we study loop invariants. Efficiency is analyzed by solving recurrence relations and borrowing theorems from number theory.

   Although outside the scope of our discussion, studying these proofs reveals a treasure trove of results vis-à-vis unsolved problems that are of both theoretical and practical significance.

EXERCISES

Caveat! Always specify the base case when using recursions.

Divide and Conquer

1. Quicksort. Like mergesort, quicksort is another sorting algorithm that features an average-case performance of $O(n \log n)$. In practice, quicksort runs faster; however, unlike mergesort, which maintains its linearithmic performance even at the worst case, it is possible for quicksort to degrade to quadratic.

   Your task is to Implement quicksort using Lomuto's partitioning scheme. The description of the algorithm is as follows:

   - Choose the last element of the array as the "pivot."
   - Place all the values that are less than the pivot to its left, and place all the values that are greater than the pivot to its right. This results in the pivot being in its final position.
   - Recursively repeat the process for both the left and right subarrays (with reference to the pivot) until all the values are sorted.

Greedy Algorithm

2. Egyptian Fraction. Write a function that accepts the numerator and denominator of a fraction and finds a representation of this fraction as an Egyptian fraction: the sum of a set of fractions with 1 as their numerator (called "unit fractions"). Return a list of these unit fractions arranged in descending order.

3. Greedy No More. Recall our discussion of the fractional knapsack problem and consider this slight modification: suppose that partial points are not awarded. In other words, getting a "fraction" of an item is not allowed. This is the premise of the 0-1 knapsack problem.

   Can the 0-1 knapsack problem be solved using the same greedy algorithm that we used to solve the fractional version?

   If yes, give an explanation; try to be as rigorous as possible. If no, give a counterexample and state which among the three algorithmic paradigms that we discussed gives an efficient solution.

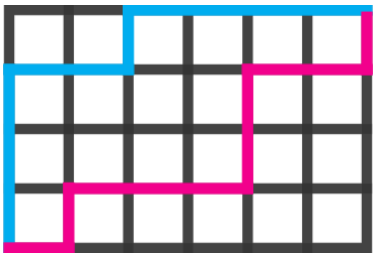
Dynamic Programming

4. Tribonacci Numbers. For the Tribonacci sequence, every term after the third term is computed as the sum of the three terms immediately before it. Suppose that the first, second, and third terms are 0, 0, and 1, respectively.

   Write two functions to obtain the n^{th} Tribonacci number: one function should use a top-down approach (memoization) while the other should use a bottom-up approach.

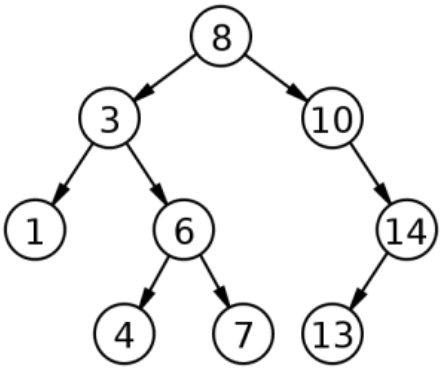
5. Number of Paths. Given an $m \times n$ rectangular grid, write an $O(mn)$ function that counts the number of paths from the point at its southwest corner to the point at its northeast corner. Assume that there are only two valid directions: right and up.

   Challenge: Try to come up with another (non-brute-force) solution that does not employ dynamic programming. One such method involves the concept of "combinations" from combinatorics.



https://brilliant.org/wiki/rectangular-grid-walk-no-restriction/

The blue and pink traces in the figure above show two such possible paths for a 6×4 grid.



https://humanwhocodes.com/blog/2009/06/16/computer-science-in-javascript-binary-search-tree-part-2/

6. **Fairly Odd Parents.** Given a binary tree, write a function that returns the sum of the keys in the nodes whose parents have keys that are odd numbers. For example, in the tree above, the return value is 1 + 6 = 7. Note that the root does not have any parent, so its key is automatically excluded from the sum.

   Your function should accept the root, which is of the class `BinaryTreeNode`, as the parameter. The `BinaryTreeNode` class follows the usual definition:

```
class BinaryTreeNode:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key
```

-- return 0; --