CHAPTER

17 Greedy Algorithms and Invariants

The intended function of a program, or part of a program, can be specified by making general assertions about the values which the relevant variables will take after *execution of the program.*

— *"An Axiomatic Basis for Computer Programming,"*
C. A. R. HOARE, 1969

Greedy algorithms

A greedy algorithm is an algorithm that computes a solution in steps; at each step the algorithm makes a decision that is locally optimum, and it never changes that decision.

A greedy algorithm does not necessarily produce the optimum solution. As example, consider making change for 48 pence in the old British currency where the coins came in 30, 24, 12, 6, 3, and 1 pence denominations. Suppose our goal is to make change using the smallest number of coins. The natural greedy algorithm iteratively chooses the largest denomination coin that is less than or equal to the amount of change that remains to be made. If we try this for 48 pence, we get three coins—30 + 12 + 6. However, the optimum answer would be two coins—24 + 24.

In its most general form, the coin changing problem is NP-hard, but for some coinages, the greedy algorithm is optimum—e.g., if the denominations are of the form $\{1, r, r^2, r^3\}$. (An *ad hoc* argument can be applied to show that the greedy algorithm is also optimum for US coinage.) The general problem can be solved in pseudo-polynomial time using DP in a manner similar to Problem 16.6 on Page 246.

Sometimes, there are multiple greedy algorithms for a given problem, and only some of them are optimum. For example, consider $2n$ cities on a line, half of which are white, and the other half are black. Suppose we need to pair white with black cities in a one-to-one fashion so that the total length of the road sections required to connect paired cities is minimized. Multiple pairs of cities may share a single section of road, e.g., if we have the pairing $(0, 4)$ and $(1, 2)$ then the section of road between Cities 0 and 4 can be used by Cities 1 and 2.

The most straightforward greedy algorithm for this problem is to scan through the white cities, and, for each white city, pair it with the closest unpaired black city. This algorithm leads to suboptimum results. Consider the case where white cities are at 0 and at 3 and black cities are at 2 and at 5. If the white city at 3 is processed first, it will be paired with the black city at 2, forcing the cities at 0 and 5 to pair up, leading to a road length of 5. In contrast, the pairing of cities at 0 and 2, and 3 and 5 leads to a road length of 4.

A slightly more sophisticated greedy algorithm does lead to optimum results: iterate through all the cities in left-to-right order, pairing each city with the nearest unpaired city of opposite color. Note that the pairing for the first city must be optimum, since if it were to be paired with any other city, we could always change its pairing to be with the nearest black city without adding any road. This observation can be used in an inductive proof of overall optimality.

Greedy algorithms boot camp

For US currency, wherein coins take values 1, 5, 10, 25, 50, 100 cents, the greedy algorithm for making change results in the minimum number of coins. Here is an implementation of this algorithm. Note that once it selects the number of coins of a particular value, it never changes that selection; this is the hallmark of a greedy algorithm.

```
def change_making(cents):
    COINS = [100, 50, 25, 10, 5, 1]
    num_coins = 0
    for coin in COINS:
        num_coins += cents // coin
        cents %= coin
    return num_coins
```

We perform 6 iterations, and each iteration does a constant amount of computation, so the time complexity is $O(1)$.

A greedy algorithm is often the right choice for an **optimization problem** where there's a natural set of **choices to select from**.

It's often easier to conceptualize a greedy algorithm recursively, and then **implement** it using iteration for higher performance.

Even if the greedy approach does not yield an optimum solution, it can give insights into the optimum algorithm, or serve as a heuristic.

Sometimes the correct greedy algorithm is **not obvious**.

Table 17.1: Top Tips for Greedy Algorithms

17.1 Compute an optimum assignment of tasks

We consider the problem of assigning tasks to workers. Each worker must be assigned exactly two tasks. Each task takes a fixed amount of time. Tasks are independent, i.e., there are no constraints of the form "Task 4 cannot start before Task 3 is completed." Any task can be assigned to any worker.

We want to assign tasks to workers so as to minimize how long it takes before all tasks are completed. For example, if there are 6 tasks whose durations are $5, 2, 1, 6, 4, 4$ hours, then an optimum assignment is to give the first two tasks (i.e., the tasks with duration 5 and 2) to one worker, the next two (1 and 6) to another worker, and the last two tasks (4 and 4) to the last worker. For this assignment, all tasks will finish after $\max(5 + 2, 1 + 6, 4 + 4) = 8$ hours.

Design an algorithm that takes as input a set of tasks and returns an optimum assignment.

Hint: What additional task should be assigned to the worker who is assigned the longest task?

Solution: Simply enumerating all possible sets of pairs of tasks is not feasible—there are too many of them. (The precise number assignments is $\binom{n}{2}\binom{n-2}{2}\binom{n-4}{2}\cdots\binom{4}{2}\binom{2}{2} = n!/2^{n/2}$, where n is the number of tasks.)

Instead we should look more carefully at the structure of the problem. Extremal values are important—the task that takes the longest needs the most help. In particular, it makes sense to pair the task with longest duration with the task of shortest duration. This intuitive observation can be understood by looking at any assignment in which a longest task is not paired with a shortest task.

By swapping the task that the longest task is currently paired with with the shortest task, we get an assignment which is at least as good.

Note that we are not claiming that the time taken for the optimum assignment is the sum of the maximum and minimum task durations. Indeed this may not even be the case, e.g., if the two longest duration tasks are close to each other in duration and the shortest duration task takes much less time than the second shortest task. As a concrete example, if the task durations are $1, 8, 9, 10$, the optimum delay is $8 + 9 = 17$, not $1 + 10$.

In summary, we sort the set of task durations, and pair the shortest, second shortest, third shortest, etc. tasks with the longest, second longest, third longest, etc. tasks. For example, if the durations are $5, 2, 1, 6, 4, 4$, then on sorting we get $1, 2, 4, 4, 5, 6$, and the pairings are $(1, 6)$, $(2, 5)$, and $(4, 4)$.

```
PairedTasks = collections.namedtuple('PairedTasks', ('task_1', 'task_2'))


def optimum_task_assignment(task_durations):
    task_durations.sort()
    return [
        PairedTasks(task_durations[i], task_durations[-i])
        for i in range(len(task_durations) // 2)
    ]
```

The time complexity is dominated by the time to sort, i.e., $O(n \log n)$.

17.2 Schedule to minimize waiting time

A database has to respond to a set of client SQL queries. The service time required for each query is known in advance. For this application, the queries must be processed by the database one at a time, but can be done in any order. The time a query waits before its turn comes is called its waiting time.

Given service times for a set of queries, compute a schedule for processing the queries that minimizes the total waiting time. Return the minimum waiting time. For example, if the service times are $\langle 2, 5, 1, 3 \rangle$, if we schedule in the given order, the total waiting time is $0 + (2) + (2 + 5) + (2 + 5 + 1) = 17$. If however, we schedule queries in order of decreasing service times, the total waiting time is $0 + (5) + (5 + 3) + (5 + 3 + 2) = 23$. As we will see, for this example, the minimum waiting time is 10.

Hint: Focus on extreme values.

Solution: We can solve this problem by enumerating all schedules and picking the best one. The complexity is very high—$O(n!)$ to be precise, where n is the number of queries.

Intuitively, it makes sense to serve the short queries first. The justification is as follows. Since the service time of each query adds to the waiting time of all queries remaining to be processed, if we put a slow query before a fast one, by swapping the two we improve the waiting time for all queries between the slow and fast query, and do not change the waiting time for the other queries. We do increase the waiting time of the slow query, but this cancels out with the decrease in the waiting time of the fast query. Hence, we should sort the queries by their service time and then process them in the order of nondecreasing service time.

For the given example, the best schedule processes queries in increasing order of service times. It has a total waiting time of $0 + (1) + (1 + 2) + (1 + 2 + 3) = 10$. Note that scheduling queries with longer service times, which we gave as an earlier example, is the worst approach.

```
def minimum_total_waiting_time(service_times):
    # Sort the service times in increasing order.
    service_times.sort()
    total_waiting_time = 0
    for i, service_time in enumerate(service_times):
        num_remaining_queries = len(service_times) - (i + 1)
        total_waiting_time += service_time * num_remaining_queries
    return total_waiting_time
```

The time complexity is dominated by the time to sort, i.e., $O(n \log n)$.

17.3 The interval covering problem

Consider a foreman responsible for a number of tasks on the factory floor. Each task starts at a fixed time and ends at a fixed time. The foreman wants to visit the floor to check on the tasks. Your job is to help him minimize the number of visits he makes. In each visit, he can check on all the tasks taking place at the time of the visit. A visit takes place at a fixed time, and he can only check on tasks taking place at exactly that time. For example, if there are tasks at times $[0, 3]$, $[2, 6]$, $[3, 4]$, $[6, 9]$, then visit times $0, 2, 3, 6$ cover all tasks. A smaller set of visit times that also cover all tasks is $3, 6$. In the abstract, you are to solve the following problem.

You are given a set of closed intervals. Design an efficient algorithm for finding a minimum sized set of numbers that covers all the intervals.

Hint: Think about extremal points.

Solution: Note that we can restrict our attention to numbers which are endpoints without losing optimality. A brute-force approach might be to enumerate every possible subset of endpoints, checking for each one if it covers all intervals, and if so, determining if it has a smaller size than any previous such subset. Since a set of size k has 2^k subsets, the time complexity is very high.

We could simply return the left end point of each interval, which is fast to compute but, as the example in the problem statement showed, may not be the minimum number of visit times. Similarly, always greedily picking an endpoint which covers the most intervals may also lead to suboptimum results, e.g., consider the six intervals $[1, 2]$, $[2, 3]$, $[3, 4]$, $[2, 3]$, $[3, 4]$, $[4, 5]$. The point 3 appears in four intervals, more than any other point. However, if we choose 3, we do not cover $[1, 2]$ and $[4, 5]$, so we need two additional points to cover all six intervals. If we pick 2 and 4, each individually covers just three intervals, but combined they cover all six.

It is a good idea to focus on extreme cases. In particular, consider the interval that ends first, i.e., the interval whose right endpoint is minimum. To cover it, we must pick a number that appears in it. Furthermore, we should pick its right endpoint, since any other intervals covered by a number in the interval will continue to be covered if we pick the right endpoint. (Otherwise the interval we began with cannot be the interval that ends first.) Once we choose that endpoint, we can remove all other covered intervals, and continue with the remaining set.

The above observation leads to the following algorithm. Sort all the intervals, comparing on right endpoints. Select the first interval's right endpoint. Iterate through the intervals, looking for the first one not covered by this right endpoint. As soon as such an interval is found, select its right endpoint and continue the iteration.

For the given example, $[1, 2]$, $[2, 3]$, $[3, 4]$, $[2, 3]$, $[3, 4]$, $[4, 5]$, after sorting on right endpoints we get $[1, 2]$, $[2, 3]$, $[2, 3]$, $[3, 4]$, $[3, 4]$, $[4, 5]$. The leftmost right endpoint is 2, which covers the first three intervals. Therefore, we select 2, and as we iterate, we see it covers $[1, 2]$, $[2, 3]$, $[2, 3]$. When we get

to [3,4], we select its right endpoint, i.e., 4. This covers [3,4], [3,4], [4,5]. There are no remaining intervals, so {2,4} is a minimum set of points covering all intervals.

```
Interval = collections.namedtuple('Interval', ('left', 'right'))


def find_minimum_visits(intervals):
    # Sort intervals based on the right endpoints.
    intervals.sort(key=operator.attrgetter('right'))
    last_visit_time, num_visits = float('-inf'), 0
    for interval in intervals:
        if interval.left > last_visit_time:
            # The current right endpoint, last_visit_time, will not cover any
            # more intervals.
            last_visit_time = interval.right
            num_visits += 1
    return num_visits
```

Since we spend $O(1)$ time per index, the time complexity after the initial sort is $O(n)$, where n is the number of intervals. Therefore, the time taken is dominated by the initial sort, i.e., $O(n \log n)$.

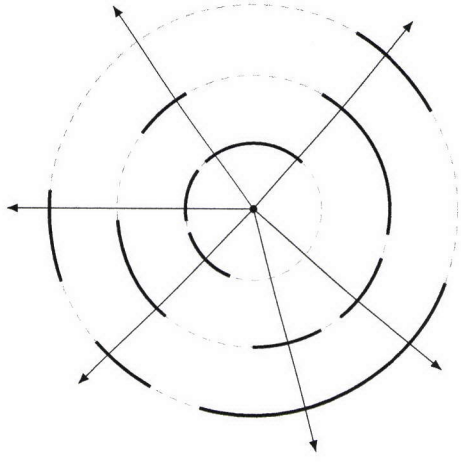


Figure 17.1: An instance of the minimum ray covering problem, with 12 partially overlapping arcs. Arcs have been drawn at different distances for illustration. For this instance, six cameras are sufficient, corresponding to the six rays.

Variant: You are responsible for the security of a castle. The castle has a circular perimeter. A total of n robots patrol the perimeter—each robot is responsible for a closed connected subset of the perimeter, i.e., an arc. (The arcs for different robots may overlap.) You want to monitor the robots by installing cameras at the center of the castle that look out to the perimeter. Each camera can look along a ray. To save cost, you would like to minimize the number of cameras. See Figure 17.1 for an example.

Variant: There are a number of points in the plane that you want to observe. You are located at the point (0, 0). You can rotate about this point, and your field-of-view is a fixed angle. Which direction should you face to maximize the number of visible points?

Invariants

A common approach to designing an efficient algorithm is to use invariants. Briefly, an invariant is a condition that is true during execution of a program. This condition may be on the values of the

variables of the program, or on the control logic. A well-chosen invariant can be used to rule out potential solutions that are suboptimal or dominated by other solutions.

For example, binary search, maintains the invariant that the space of candidate solutions contains all possible solutions as the algorithm executes.

Sorting algorithms nicely illustrate algorithm design using invariants. Intuitively, selection sort is based on finding the smallest element, the next smallest element, etc. and moving them to their right place. More precisely, we work with successively larger subarrays beginning at index 0, and preserve the invariant that these subarrays are sorted, their elements are less than or equal to the remaining elements, and the entire array remains a permutation of the original array.

As a more sophisticated example, consider Solution 14.7 on Page 208, specifically the $O(k)$ algorithm for generating the first k numbers of the form $a + b\sqrt{2}$. The key idea there is to process these numbers in sorted order. The queues in that code maintain multiple invariants: queues are sorted, duplicates are never present, and the separation between elements is bounded.

Invariants boot camp

Suppose you were asked to write a program that takes as input a sorted array and a given target value and determines if there are two entries in the array that add up to that value. For example, if the array is $\langle -2, 1, 2, 4, 7, 11 \rangle$, then there are entries adding to 6, to 0, and to 13, but not to 10.

The brute-force algorithm for this problem consists of a pair of nested for loops. Its complexity is $O(n^2)$, where n is the length of the input array. A faster approach is to add each element of the array to a hash table, and test for each element e if $K - e$, where K is the target value, is present in the hash table. While reducing time complexity to $O(n)$, this approach requires $O(n)$ additional storage for the hash.

The most efficient approach uses invariants: maintain a subarray that is guaranteed to hold a solution, if it exists. This subarray is initialized to the entire array, and iteratively shrunk from one side or the other. The shrinking makes use of the sortedness of the array. Specifically, if the sum of the leftmost and the rightmost elements is less than the target, then the leftmost element can never be combined with some element to obtain the target. A similar observation holds for the rightmost element.

```
def has_two_sum(A, t):
    i, j = 0, len(A) - 1

    while i <= j:
        if A[i] + A[j] == t:
            return True
        elif A[i] + A[j] < t:
            i += 1
        else:  # A[i] + A[j] > t.
            j -= 1
    return False
```

The time complexity is $O(n)$, where n is the length of the array. The space complexity is $O(1)$, since the subarray can be represented by two variables.

17.4 The 3-sum problem

Design an algorithm that takes as input an array and a number, and determines if there are three entries in the array (not necessarily distinct) which add up to the specified number. For example, if the array is $\langle 11, 2, 5, 7, 3 \rangle$ then there are three entries in the array which add up to 21 (3, 7, 11 and

Identifying the right invariant is an art. The key strategy to determine whether to use an invariant when designing an algorithm is to work on **small examples** to hypothesize the invariant. Often, the invariant is a subset of the set of input space, e.g,. a subarray.

Table 17.2: Top Tips for Invariants

5, 5, 11). (Note that we can use 5 twice, since the problem statement said we can use the same entry more than once.) However, no three entries add up to 22.

Hint: How would you check if a given array entry can be added to two more entries to get the specified number?

Solution: The brute-force algorithm is to consider all possible triples, e.g., by three nested for-loops iterating over all entries. The time complexity is $O(n^3)$, where n is the length of the array, and the space complexity is $O(1)$.

Let A be the input array and t the specified number. We can improve the time complexity to $O(n^2)$ by storing the array entries in a hash table first. Then we iterate over pairs of entries, and for each $A[i] + A[j]$ we look for $t - (A[i] + A[j])$ in the hash table. The space complexity now is $O(n)$.

We can avoid the additional space complexity by first sorting the input. Specifically, sort A and for each $A[i]$, search for indices j and k such that $A[j] + A[k] = t - A[i]$. We can do each such search in $O(n \log n)$ time by iterating over $A[j]$ values and doing binary search for $A[k]$.

We can improve the time complexity to $O(n)$ by starting with $A[0] + A[n-1]$. If this equals $t - A[i]$, we're done. Otherwise, if $A[0] + A[n-1] < t - A[i]$, we move to $A[1] + A[n-1]$—there is no chance of $A[0]$ pairing with any other entry to get $t - A[i]$ (since $A[n-1]$ is the largest value in A). Similarly, if $A[0] + A[n-1] > t - A[i]$, we move to $A[0] + A[n-2]$. This approach eliminates an entry in each iteration, and spends $O(1)$ time in each iteration, yielding an $O(n)$ time bound to find $A[j]$ and $A[k]$ such that $A[j] + A[k] = t - A[i]$, if such entries exist. The invariant is that if two elements which sum to the desired value exist, they must lie within the subarray currently under consideration.

For the given example, after sorting the array is $\langle 2, 3, 5, 7, 11 \rangle$. For entry $A[0] = 2$, to see if there are $A[j]$ and $A[k]$ such that $A[0]+A[j]+A[k] = 21$, we search for two entries that add up to $21-2 = 19$.

The code for this approach is shown below.

```
def has_three_sum(A, t):
    A.sort()
    # Finds if the sum of two numbers in A equals to t - a.
    return any(has_two_sum(A, t - a) for a in A)
```

The additional space needed is $O(1)$, and the time complexity is the sum of the time taken to sort, $O(n \log n)$, and then to run the $O(n)$ algorithm to find a pair in a sorted array that sums to a specified value, which is $O(n^2)$ overall.

Variant: Solve the same problem when the three elements must be distinct. For example, if $A = \langle 5, 2, 3, 4, 3 \rangle$ and $t = 9$, then $A[2] + A[2] + A[2]$ is not acceptable, $A[2] + A[2] + A[4]$ is not acceptable, but $A[1] + A[2] + A[3]$ and $A[1] + A[3] + A[4]$ are acceptable.

Variant: Solve the same problem when k, the number of elements to sum, is an additional input.

Variant: Write a program that takes as input an array of integers A and an integer T, and returns a 3-tuple $(A[p], A[q], A[r])$ where p, q, r are all distinct, minimizing $|T - (A[p] + A[q] + A[r])|$, and

$A[p] \leq A[r] \leq A[s]$.

Variant: Write a program that takes as input an array of integers A and an integer T, and returns the number of 3-tuples (p, q, r) such that $A[p] + A[q] + A[r] \leq T$ and $A[p] \leq A[q] \leq A[r]$.

17.5 Find the majority element

Several applications require identification of elements in a sequence which occur more than a specified fraction of the total number of elements in the sequence. For example, we may want to identify the users using excessive network bandwidth or IP addresses originating the most Hypertext Transfer Protocol (HTTP) requests. Here we consider a simplified version of this problem.

You are reading a sequence of strings. You know *a priori* that more than half the strings are repetitions of a single string (the "majority element") but the positions where the majority element occurs are unknown. Write a program that makes a single pass over the sequence and identifies the majority element. For example, if the input is $\langle b, a, c, a, a, b, a, a, c, a \rangle$, then a is the majority element (it appears in 6 out of the 10 places).

Hint: Take advantage of the existence of a majority element to perform elimination.

Solution: The brute-force approach is to use a hash table to record the repetition count for each distinct element. The time complexity is $O(n)$, where n is the number of elements in the input, but the space complexity is also $O(n)$.

Randomized sampling can be used to identify a majority element with high probability using less storage, but is not exact.

The intuition for a better algorithm is as follows. We can group entries into two subgroups—those containing the majority element, and those that do not hold the majority element. Since the first subgroup is given to be larger in size than the second, if we see two entries that are different, at most one can be the majority element. By discarding both, the difference in size of the first subgroup and second subgroup remains the same, so the majority of the remaining entries remains unchanged.

The algorithm then is as follows. We have a candidate for the majority element, and track its count. It is initialized to the first entry. We iterate through remaining entries. Each time we see an entry equal to the candidate, we increment the count. If the entry is different, we decrement the count. If the count becomes zero, we set the next entry to be the candidate.

Here is a mathematical justification of the approach. Let's say the majority element occurred m times out of n entries. By the definition of majority element, $\frac{m}{n} > \frac{1}{2}$. At most one of the two distinct entries that are discarded can be the majority element. Hence, after discarding them, the ratio of the number of remaining majority elements to the total number of remaining elements is either $\frac{m}{(n-2)}$ (neither discarded element was the majority element) or $\frac{(m-1)}{(n-2)}$ (one discarded element was the majority element). It is simple to verify that if $\frac{m}{n} > \frac{1}{2}$, then both $\frac{m}{(n-2)} > \frac{1}{2}$ and $\frac{(m-1)}{(n-2)} > \frac{1}{2}$.

For the given example, $\langle b, a, c, a, a, b, a, a, c, a \rangle$, we initialize the candidate to b. The next element, a is different from the candidate, so the candidate's count goes to 0. Therefore, we pick the next element c to be the candidate, and its count is 1. The next element, a, is different so the count goes back to 0. The next element is a, which is the new candidate. The subsequent b decrements the count to 0. Therefore the next element, a, is the new candidate, and it has a nonzero count till the end.

```
def majority_search(input_stream):
    candidate, candidate_count = None, 0
```

```
for it in input_stream:
    if candidate_count == 0:
        candidate, candidate_count = it, candidate_count + 1
    elif candidate == it:
        candidate_count += 1
    else:
        candidate_count -= 1
return candidate
```

Since we spend $O(1)$ time per entry, the time complexity is $O(n)$. The additional space complexity is $O(1)$.

The code above assumes a majority word exists in the sequence. If no word has a strict majority, it still returns a word from the stream, albeit without any meaningful guarantees on how common that word is. We could check with a second pass whether the returned word was a majority. Similar ideas can be used to identify words that appear more than n/k times in the sequence, as discussed in Problem 24.33 on Page 399.

17.6 The gasup problem

In the gasup problem, a number of cities are arranged on a circular road. You need to visit all the cities and come back to the starting city. A certain amount of gas is available at each city. The amount of gas summed up over all cities is equal to the amount of gas required to go around the road once. Your gas tank has unlimited capacity. Call a city *ample* if you can begin at that city with an empty tank, refill at it, then travel through all the remaining cities, refilling at each, and return to the ample city, without running out of gas at any point. See Figure 17.2 for an example.

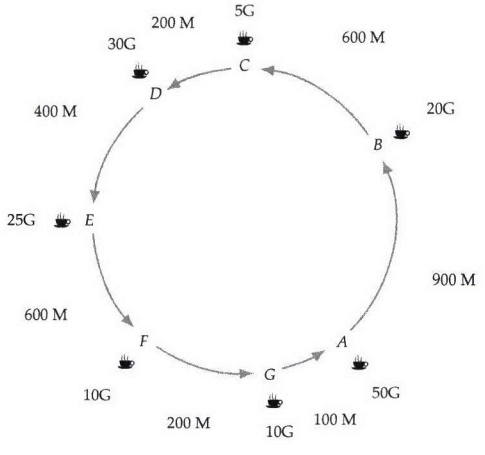


Figure 17.2: The length of the circular road is 3000 miles, and your vehicle gets 20 miles per gallon. The distance noted at each city is how far it is from the next city. For this configuration, we can begin with no gas at City D, and complete the circuit successfully, so D is an ample city.

Given an instance of the gasup problem, how would you efficiently compute an ample city? You can assume that there exists an ample city.

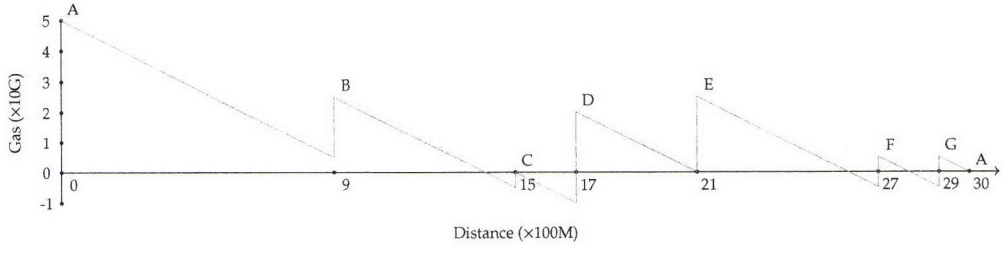
Hint: Think about starting with more than enough gas to complete the circuit without gassing up. Track the amount of gas as you perform the circuit, gassing up at each city.

Solution: The brute-force approach is to simulate the traversal from each city. This approach has time $O(n^2)$ time complexity, where n is the number of cities.

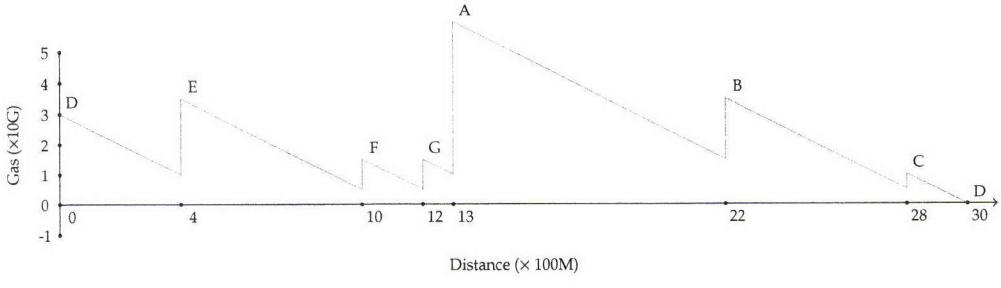
Greedy approaches, e.g., finding the city with the most gas, the city closest to the next city, or the city with best distance-to-gas ratio, do not work. For the given example, A has the most gas, but you cannot get to C starting from A. The city G is closest to the next city (A), and has the lowest distance-to-gas ratio (100/10) but it cannot get to D.

We can gain insight by looking at the graph of the amount of gas as we perform the traversal. See Figure 17.3 for an example. The amount of gas in the tank could become negative, but we ignore the physical impossibility of that for now. These graphs are the same up to a translation about the Y-axis and a cyclic shift about the X-axis.

In particular, consider a city where the amount of gas in the tank is minimum when we enter that city. Observe that it does not depend where we begin from—because graphs are the same up to translation and shifting, a city that is minimum for one graph will be a minimum city for all graphs. Let z be a city where the amount of gas in the tank before we refuel at that city is minimum. Now suppose we pick z as the starting point, with the gas present at z. Since we never have less gas than we started with at z, and when we return to z we have 0 gas (since it's given that the total amount of gas is just enough to complete the traversal) it means we can complete the journey without running out of gas. Note that the reasoning given above demonstrates that there always exists an ample city.



(a) Gas vs. distance, starting at A.



(b) Gas vs. distance, starting at D.

Figure 17.3: Gas as a function of distance for different starting cities for the configuration in Figure 17.2 on the preceding page.

The computation to determine z can be easily performed with a single pass over all the cities simulating the changes to amount of gas as we advance.

```
MPG = 20


# gallons[i] is the amount of gas in city i, and distances[i] is the
# distance city i to the next city.
def find_ample_city(gallons, distances):
    remaining_gallons = 0
```

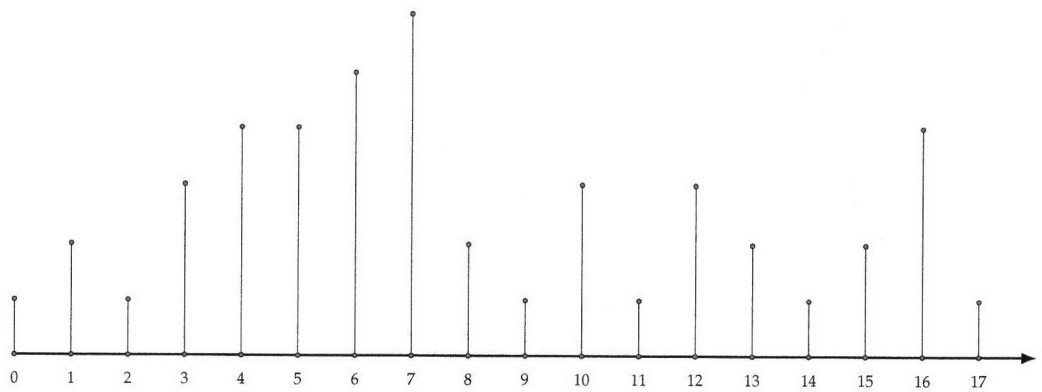
```
    CityAndRemainingGas = collections.namedtuple('CityAndRemainingGas',
                                                 ('city', 'remaining_gallons'))
    city_remaining_gallons_pair = CityAndRemainingGas(0, 0)
    num_cities = len(gallons)
    for i in range(1, num_cities):
        remaining_gallons += gallons[i - 1] - distances[i - 1] // MPG
        if remaining_gallons < city_remaining_gallons_pair.remaining_gallons:
            city_remaining_gallons_pair = CityAndRemainingGas(
                i, remaining_gallons)
    return city_remaining_gallons_pair.city
```

The time complexity is $O(n)$, and the space complexity is $O(1)$.

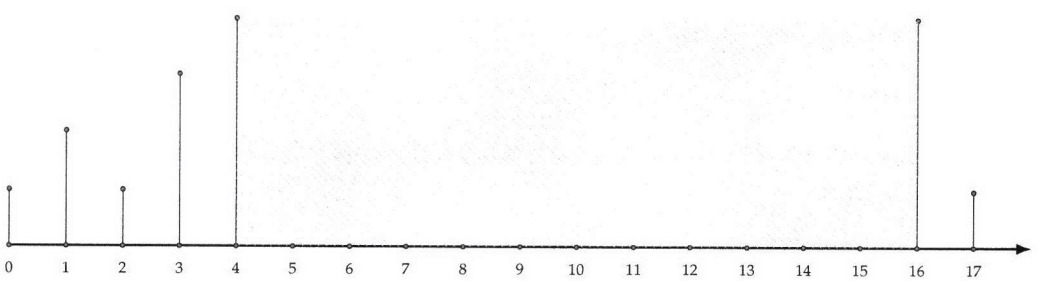
Variant: Solve the same problem when you cannot assume that there exists an ample city.

17.7 Compute the maximum water trapped by a pair of vertical lines

An array of integers naturally defines a set of lines parallel to the Y-axis, starting from $x = 0$ as illustrated in Figure 17.4(a). The goal of this problem is to find the pair of lines that together with the X-axis "trap" the most water. See Figure 17.4(b) for an example.



(a) A graphical depiction of the array $\langle 1, 2, 1, 3, 4, 4, 5, 6, 2, 1, 3, 1, 3, 2, 1, 2, 4, 1 \rangle$.



(b) The shaded area between 4 and 16 is the maximum water that can be trapped by the array in (a).

Figure 17.4: Example of maximum water trapped by a pair of vertical lines.

Write a program which takes as input an integer array and returns the pair of entries that trap the maximum amount of water.

Hint: Start with 0 and $n - 1$ and work your way in.

Solution: Let A be the array, and n its length. There is a straightforward $O(n^2)$ brute-force solution—for each pair of indices $(i, j), i < j$, the water trapped by the corresponding lines is $(j - i) \times \min(A[i], A[j])$, which can easily be computed in $O(1)$ time. The maximum of all these is the desired quantity.

In an attempt to achieve a better time complexity, we can try divide-and-conquer. We find the maximum water that can be trapped by the left half of A, the right half of A, and across the center of A. Finding the maximum water trapped by an element on the left half and the right half entails considering combinations of the $n/2$ entries from left half and $n/2$ entries on the right half. Therefore, the time complexity $T(n)$ of this divide-and-conquer approach satisfies $T(n) = 2T(n/2) + O(n^2/4)$ which solves to $T(n) = O(n^2)$. This is no better than the brute-force solution, and considerably trickier to code.

A good starting point is to consider the widest pair, i.e., 0 and $n-1$. We record the corresponding amount of trapped water, i.e., $((n-1) - 0) \times \min(A[0], A[n-1])$. Suppose $A[0] > A[n-1]$. Then for any $k > 0$, the water trapped between k and $n-1$ is less than the water trapped between 0 and $n-1$, so going forward, we only need to focus on the maximum water that can be trapped between 0 and $n-2$. The converse is true if $A[0] \leq A[n-1]$—we need never consider 0 again.

We use this approach iteratively to continuously reduce the subarray that must be explored, while recording the most water trapped so far. In essence, we are exploring the best way in which to trade-off width for height.

For the given example, we begin with $(0, 17)$, which has a capacity of $1 \times 17 = 17$. Since the height of the left line is less than or equal to the height of the right line, we advance to $(1, 17)$. The capacity is $1 \times 16 = 16$. Since $2 > 1$, we advance to $(1, 16)$. The capacity is $2 \times 15 = 30$. Since $2 < 4$, we advance to $(2, 16)$. The capacity is $1 \times 14 = 14$. Since $1 < 4$, we advance to $(3, 16)$. The capacity is $3 \times 13 = 39$. Since $3 < 4$, we advance to $(4, 16)$. The capacity is $4 \times 12 = 48$. Future iterations, which we do not show, do not surpass 48, which is the result.

```
def get_max_trapped_water(heights):
    i, j, max_water = 0, len(heights) - 1, 0
    while i < j:
        width = j - i
        max_water = max(max_water, width * min(heights[i], heights[j]))
        if heights[i] > heights[j]:
            j -= 1
        else:  # heights[i] <= heights[j].
            i += 1
    return max_water
```

We iteratively eliminate one line or two lines at a time, and we spend $O(1)$ time per iteration, so the time complexity is $O(n)$.

17.8 Compute the largest rectangle under the skyline

You are given a sequence of adjacent buildings. Each has unit width and an integer height. These buildings form the skyline of a city. An architect wants to know the area of a largest rectangle contained in this skyline. See Figure 17.5 on the next page for an example.

Let A be an array representing the heights of adjacent buildings of unit width. Design an algorithm to compute the area of the largest rectangle contained in this skyline.

Hint: How would you efficiently find the largest rectangle which includes the ith building, and has height $A[i]$?

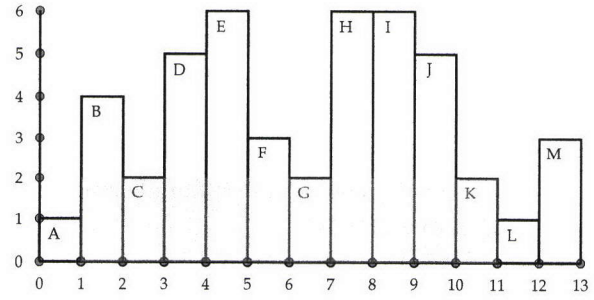


Figure 17.5: A collection of unit-width buildings, and the largest contained rectangle. The text label identifying the building is just below and to the right of its upper left-hand corner. The shaded area is the largest rectangle under the skyline. Its area is $2 \times (11 - 1)$. Note that the tallest rectangle is from 7 to 9, and the widest rectangle is from 0 to 1, but neither of these are the largest rectangle under the skyline.

Solution: A brute-force approach is to take each (i, j) pair, find the minimum of subarray $A[i, j]$, and multiply that by $j - i + 1$. This has time complexity $O(n^3)$, where n is the length of A. This can be improved to $O(n^2)$ by iterating over i and then $j \geq i$ and tracking the minimum height of buildings from i to j, inclusive. However, there is no reasonable way to further refine this algorithm to get the time complexity below $O(n^2)$.

Another brute-force solution is to consider for each i the furthest left and right we can go without dropping below $A[i]$ in height. In essence, we want to know the largest rectangle that is "supported" by Building i, which can be viewed as acting like a "pillar" of that rectangle. For the given example, the largest rectangle supported by G extends from 1 to 11, and the largest rectangle supported by F extends from 3 to 6.

We can easily determine the largest rectangle supported by Building i with a single forward and a single backward iteration from i. Since i ranges from 0 to $n - 1$, the time complexity of this approach is $O(n^2)$.

This brute-force solution can be refined to get a much better time complexity. Specifically, suppose we iterate over buildings from left to right. When we process Building i, we do not know how far to the right the largest rectangle it supports goes. However, we do know that the largest rectangles supported by earlier buildings whose height is greater than $A[i]$ cannot extend past i, since Building i "blocks" these rectangles. For example, when we get to F in Figure 17.5, we know that the largest rectangles supported by Buildings B, D, and E (whose heights are greater than F's height) cannot extend past 5.

Looking more carefully at the example, we see that there's no need to consider B when examining F, since C has already blocked the largest rectangle supported by B. In addition, there's no reason to consider C: since G's height is the same as C's height, and C has not been blocked yet, G and C will have the same largest supported rectangle. Generalizing, as we advance through the buildings, all we really need is to keep track of is buildings that have not been blocked yet. Additionally, we can replace existing buildings whose height equals that of the current building with the current building. Call these buildings the set of active pillars.

Initially there are no buildings in the active pillar set. As we iterate from 0 to 12, the active pillar sets are $\{A\}$, $\{A, B\}$, $\{A, C\}$, $\{A, C, D\}$, $\{A, C, D, E\}$, $\{A, C, F\}$, $\{A, G\}$, $\{A, G, H\}$, $\{A, G, I\}$, $\{A, G, J\}$, $\{A, K\}$, $\{L\}$, and $\{L, M\}$.

Whenever a building is removed from the active pillar set, we know exactly how far to the right the largest rectangle that it supports goes to. For example, when we reach C we know B's supported rectangle ends at 2, and when we reach F, we know that D and E's largest supported rectangles end

at 5.

When we remove a blocked building from the active pillar set, to find how far to the left its largest supported rectangle extends we simply look for the closest active pillar that has a lower height. For example, when we reach F, the active pillars are $\{A, C, D, E\}$. We remove E and D from the set, since both are taller than F. The largest rectangle supported by E has height 6 and begins after D, i.e., at 4; its area is $6 \times (5 - 4) = 6$. The largest rectangle supported by D has height 5 and begins after C, i.e., at 3; its area is $5 \times (5 - 3) = 10$.

There are a number of data structures that can be used to store the active pillars and the right data structure is key to making the above algorithm efficient. When we process a new building we need to find the buildings in the active pillar set that are blocked. Because insertion and deletion into the active pillar set take place in last-in first-out order, a stack is a reasonable choice for maintaining the set. Specifically, the rightmost building in the active pillar set appears at the top. Using a stack also makes it easy to find how far to the left the largest rectangle that's supported by a building in the active pillar set extends—we simply look at the building below it in the stack. For example, when we process F, the stack is A, C, D, E, with E at the top. Comparing F's height with E, we see E is blocked so the largest rectangle under E ends at where F begins, i.e., at 5. Since the next building in the stack is D, we know that the largest rectangle under E begins where D ends, i.e., at 4.

The algorithm described above is almost complete. The missing piece is what to do when we get to the end of the iteration. The stack will not be empty when we reach the end—at the very least it will contain the last building. We deal with these elements just as we did before, the only difference being that the largest rectangle supported by each building in the stack ends at n, where n is the number of elements in the array. For the given example, the stack contains L and M when we get to the end, and the largest supported rectangles for both of these end at 13.

```
def calculate_largest_rectangle(heights):
    pillar_indices, max_rectangle_area = [], 0
    # By appending [0] to heights, we can uniformly handle the computation for
    # rectangle area here.
    for i, h in enumerate(heights + [0]):
        while pillar_indices and heights[pillar_indices[-1]] >= h:
            height = heights[pillar_indices.pop()]
            width = i if not pillar_indices else i - pillar_indices[-1] - 1
            max_rectangle_area = max(max_rectangle_area, height * width)
        pillar_indices.append(i)
    return max_rectangle_area
```

The time complexity is $O(n)$. When advancing through buildings, the time spent for building is proportional to the number of pushes and pops performed when processing it. Although for some buildings, we may perform multiple pops, in total we perform at most n pushes and at most n pops. This is because in the advancing phase, an entry i is added at most once to the stack and cannot be popped more than once. The time complexity of processing remaining stack elements after the advancing is complete is also $O(n)$ since there are at most n elements in the stack, and the time to process each one is $O(1)$. Thus, the overall time complexity is $O(n)$. The space complexity is $O(n)$, which is the largest the stack can grow to, e.g., if buildings appear in ascending order.

Variant: Find the largest square under the skyline.