Chapter 21
Greedy Algorithms and Divide-and-Conquer

21.1 Introduction to Algorithm Families

Recall that an **algorithm** is a well-defined set of instructions that can be followed to solve a problem. Algorithms typically take in some form of input and then perform steps on this input to transform it into some form of output. For instance, consider the assortment of sorting algorithms covered previously. Each of these algorithms performs a sequence of operations on an input container to produce an output container with its contents in sorted order. However, the sequence of operations performed is drastically different across different sorting algorithms, even if the end goal is the same. Some sorting algorithms, such as bubble sort and selection sort, utilize a sequence of greedy decisions to sort a container. Other sorting algorithms, such as quicksort and mergesort, divide the input container into smaller segments and perform operations on these smaller chunks before recombining them to obtain the sorted output.

It turns out that many algorithms can be categorized into different **algorithm families**, each of which share a similar approach or pattern toward solving a problem. Some algorithms, like Prim's and Kruskal's algorithms, solve a problem by making locally optimal decisions that lead to a globally optimal solution. We consider these algorithms as part of the *greedy* algorithm family. Other algorithms, such as quicksort and the Karatsuba multiplication algorithm, recursively split a problem into smaller subproblems until each subproblem is simple enough to be solved trivially, before combining the results together to obtain a solution to the original problem. We denote these algorithms as part of the *divide-and-conquer* algorithm family. By categorizing algorithms into different families, we are better able to identify and analyze methods that can be used to tackle problems that share similar characteristics. For instance, if a problem involves optimizing a value given a set of constraints, we can show that a greedy approach will always find an optimal solution as long as certain conditions are met. In addition, problems that are suited for certain algorithm families may not be suited for others; as you will see later, problems that involve overlapping subproblems can be solved using dynamic programming, but not divide-and-conquer.

As a preview of what's to come, the topic of algorithm families will be split into several chapters. In this chapter, we will focus on the algorithm families of *brute force*, *greedy*, and *divide-and-conquer*. In chapter 22, we will discuss the algorithm families of *backtracking* and *branch and bound*. Lastly, in chapter 23, we will discuss the algorithm family of *dynamic programming*. We will then apply all of these algorithm families in chapter 24 to solve the *knapsack problem*, a combinatorial optimization problem important to the field of computer science.

21.2 Brute Force

A **brute force** algorithm is one that solves a problem in the most simple, direct, or obvious way. Brute force algorithms are not typically distinguishable by structure or form, and two different brute force algorithms can be implemented completely differently. Regardless of how a brute force algorithm is implemented, however, the goal of such an algorithm is to try out every possible solution, often relying on sheer computing power to do so. Because brute force algorithms check every possible answer to a problem, they are guaranteed to get the right answer for the problem they are trying to solve.

For certain types of problems, brute force is the only way to go. For example, if you wanted to guess someone's password, where each guess does not give you any information other than whether you are right or wrong, then you have no choice but to try everything. However, in many cases, brute force does more work than necessary, and there are more efficient ways to solve a given problem, even if such solutions are not immediately straightforward or obvious.

To highlight an example of the brute force process, suppose you are a cashier with forty coins — one quarter, three dimes, six nickels, and thirty pennies — and you want to use these coins to return 30¢ of change using as few coins as possible. If you rely on brute force to solve this problem, you will have to check all possible subsets of coins, determine if they sum up to 30¢, and then return the subset that uses the fewest number of coins. A table depicting the subsets you would need to check is shown below (note that this table is abbreviated, since it only contains subsets that sum to 30¢ — in reality, you would need to check *all* possible subsets that can be created using the forty coins you have, even those that do not sum up to the target amount).

# Quarters	# Dimes	# Nickels	# Pennies	# Coins
0	0	0	30	30
0	0	1	25	26
0	0	2	20	22
0	0	3	15	18
0	0	4	10	14
0	0	5	5	10
0	0	6	0	6
0	1	0	20	21
0	1	1	15	17
0	1	2	10	13
0	1	3	5	9
0	1	4	0	5
0	2	0	10	12
0	2	1	5	8
0	2	2	0	4
0	3	0	0	3
1	0	0	5	6
1	0	1	0	2

The last row of the table — one quarter and one nickel — allows you to obtain 30¢ using the fewest number of coins, so this would be the solution to the problem. However, we had to check 2^{40} possible subsets of coins before we could come up with this answer! The 2^{40} was obtained using the fundamental counting principle: since there are forty coins total and two possibilities for each of these coins (either included or excluded from a subset), the total number of possible subsets is $2 \times 2 \times \ldots \times 2 = 2^{40}$.

In general, if there are n total coins, there are a total of 2^n possible subsets that could potentially be our solution. If we use the brute force approach to solve the coin change problem, we would then have to check the sum of all 2^n subsets to determine if it sums to the desired amount, and then choose the feasible subset that consists of the fewest number of coins. It takes $\Theta(n)$ time to count the number of coins in a subset and determine if it sums to the desired amount; since these operations are performed for all 2^n possible subsets, the time complexity of the brute force solution to the coin change problem is bounded above by $\Theta(n2^n)$. This is computationally expensive and is pretty much infeasible for large values of n! Luckily, there are ways to do better than $\Theta(n2^n)$ by using algorithm families that can be used to improve the efficiency of optimization problems. One such family is the *greedy approach*, which will be explored in the following section.

21.3 Greedy Algorithms

※ 21.3.1 *The Greedy Approach and Optimization Problems*

Greedy algorithms can be used to solve certain types of problems in a manner that is often asymptotically faster than brute force. Before we begin our discussion on greedy algorithms, we will first introduce a category of problems known as **optimization problems**. Optimization problems involve minimizing or maximizing an *objective function* given a set of *constraints*. We consider solutions that satisfy our constraints as *feasible solutions*, and the optimal solution is the best solution among the possible solutions in the feasible solution set. It is entirely valid for an optimization problem to have no constraints, and it is also valid for a problem to have more than one constraint.

For example, the coin change problem is an optimization problem, where the objective function is the number of coins returned, and the constraint is that the change must sum up to 30¢. Other examples of optimization problems are shown below:

1. Determine the time allocation to each of your final exams to maximize total points earned, given that you only have 24 hours left to study.
    - *Type of optimization problem:* maximization problem
    - *Objective function:* total points earned across all exams
    - *Constraint(s):* you only have 24 hours left to study

2. Find a path from Pierpont Commons to the François-Xavier Bagnoud (FXB) building that minimizes total distance.
   - *Type of optimization problem:* minimization problem
   - *Objective function:* distance of path from Pierpont to FXB
   - *Constraint(s):* none
3. Given a set of items (each with a weight and a value) and a knapsack of capacity C, determine which items you should take to maximize total value without exceeding the weight capacity of the knapsack.
   - *Type of optimization problem:* maximization problem
   - *Objective function:* total value of items taken
   - *Constraint(s):* total weight of all items taken cannot exceed C
4. You are currently on a road trip, and your car can run for M miles on a full tank of gas. There are N gas stations along your route, and you are given the distances of each gas station from your starting point. Identify which gas stations you should refuel at if you want to make the fewest number of stops for gas without ever running out.
   - *Type of optimization problem:* minimization problem
   - *Objective function:* number of stops you need to make for gas
   - *Constraint(s):* your car must run for M miles without ever running out of gas

When dealing with optimization problems, there are four algorithm families that are typically used:
- Brute Force
- Greedy
- Branch and Bound *(chapter 22)*
- Dynamic Programming *(chapter 23)*

In a greedy approach, we solve a problem by making a sequence of *locally optimal* choices: at each step of a greedy algorithm, we make the best choice we can at the current moment toward our goal, as long as that choice is feasible. Once a choice is made, we *never reconsider that decision*. Eventually, after making all of our decisions, we hope to end up with a globally optimal solution. However, as we will see later on, the greedy approach is *not* guaranteed to find an optimal solution for all optimization problems, unlike the other three algorithm families listed above!

For example, let's return to the coin change problem. If we wanted to minimize the total number of coins needed to return any change amount, the locally optimal decision would be to consider the coins in order of decreasing denomination and pick the highest-valued coin that is feasible at every step. In our example, we would keep on adding quarters to our change until we cannot add any more, at which point we then add dimes, then nickels, then pennies, until we obtain our desired amount.

Example 21.1 You want to make change for 30¢, and your current coin denominations are 25¢, 10¢, 5¢, and 1¢. What coins would you use to make change if you use the greedy approach to minimize the total number of coins you need?

Using the greedy approach, you will always select the highest-value coin that does not bring you over the desired change amount. In this case, since we want to make change for 30¢, the highest-value coin that does not go over is the quarter, so we add a quarter to our solution. We now have 5¢ left to return, so the highest-value coin that does not cause us to go over is the nickel. After adding a nickel to our solution, we now have our desired amount of change, so the solution of the greedy approach would be to return 30¢ in the form of one quarter and one nickel.

Example 21.2 You want to make change for 94¢, and your current coin denominations are 25¢, 10¢, 5¢, and 1¢. What coins would you use to make change if you use the greedy approach to minimize the total number of coins you need?

We first select as many quarters as possible without going over, so we would add three quarters to our solution, for a total of 75¢. We now have 19¢ left to return, so we select as many dimes as possible without going over. In this case, we add one dime to our solution, leaving us with 9¢ of change remaining. Continuing this process, we would then add one nickel to our solution (leaving us with 4¢ remaining), followed by four pennies. This gives us our desired change amount, so the solution of the greedy approach would be to return 94¢ in the form of three quarters, one dime, one nickel, and four pennies.

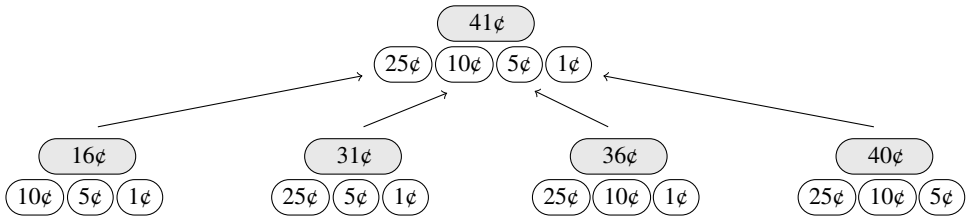
Example 21.3 You want to make change for 30¢, but you do *not* have any nickels! Thus, your current coin denominations are 25¢, 10¢, and 1¢. What coins would you use to make change if you use the greedy approach to minimize the total number of coins you need?

We first select as many quarters as possible without going over, so we would add one quarter to our solution. We now have 5¢ left to return, so the only coins we can add to our solution without going over are pennies (since we have no nickels), of which we need to add five. Thus, the solution of the greedy approach would be to return 30¢ in the form of one quarter and five pennies.

※ 21.3.2 *Proving the Correctness of a Greedy Approach*

When used correctly, the greedy approach can be used to solve an optimization problem very quickly: if we are given n coins in the coin change problem above, we can obtain a greedy solution in $\Theta(n\log(n))$ time if the coin denominations aren't already sorted, and $\Theta(n)$ time if they are. However, you must be careful: the greedy approach is *not* guaranteed to return an optimal solution for all optimization problems! In examples 21.1 and 21.2, the greedy approach did return the correct solution. However, in example 21.3, the greedy approach failed; we ended up returning six coins to make change for 30¢, when we only needed three (using three dimes). Just by changing up the coin denominations, we ended up negating the correctness of the greedy approach for the coin change problem. *This is the pitfall of the greedy approach: it can be used to solve problems efficiently, but you **must** prove the correctness of the greedy approach before you can apply it toward solving an optimization problem!*

How can we prove that a greedy approach works? Optimization problems that are solvable using a greedy approach exhibit two key traits: an *optimal substructure* and the *greedy-choice property*. A problem with an **optimal substructure** is one whose optimal solution can be constructed using the optimal solutions of its subproblems (a *subproblem* is a version of the original problem, typically with a smaller input size, that is solved on the way toward solving the original problem). The following illustrates the optimal substructure of the coin change problem, where the optimal solution of any change amount can be built using the optimal solutions of smaller change amounts (the example uses $41¢$, but this structure holds for any change amount desired). As an example, the optimal solution for $41¢$ can be constructed by adding a $10¢$ coin to the optimal solution for $31¢$. In the context of greedy algorithms, we can prove optimal substructure by showing that, if we are given an optimal solution to the subproblem that excludes the greedy choice, we can always combine this solution with the greedy choice to obtain an optimal solution to the original problem. *If a problem has an optimal substructure, we can always obtain an optimal solution if we perform a greedy choice, and then combine this choice with the optimal solution of the subproblem that remains.*



A problem satisfies the **greedy-choice property** if there exists at least one optimal solution that contains the first greedy choice. That is, if we make a greedy choice on a problem that satisfies the greedy-choice property for a given greedy algorithm, there is guaranteed to be at least one optimal solution that includes the choice we just made. This ensures that a greedy choice can be safely made without ever preventing us from finding an optimal solution.

These two characteristics give us a method for proving the efficacy of a greedy approach on an optimization problem. First, we want to frame the problem in a way such that only one subproblem remains after a greedy choice is made. Then, we want to show that the problem satisfies the greedy-choice property for our greedy algorithm — this ensures that the greedy choice is always safe to make. Lastly, we want to demonstrate that the problem has an optimal substructure, where an optimal solution for the entire problem can be obtained by combining the greedy choice with the optimal solution of the subproblem that remains.

If we can prove that a problem has an optimal substructure and satisfies the greedy-choice property for a specific greedy algorithm, then we have also successfully proven the correctness of the greedy algorithm on that problem. Why is this so? If a problem satisfies the greedy-choice property and has an optimal substructure, we can use *mathematical induction* to show that the greedy choice must always lead to an optimal solution. This process is shown below.

A Proof Using Mathematical Induction

Let $P(n)$ be the claim that a greedy algorithm always finds an optimal solution after n greedy choices (where n is the number of choices needed to obtain a solution) for problems that satisfy the greedy-choice property and have an optimal substructure.

Base Step: *We want to show that $P(n)$ is true for some initial value of n.*

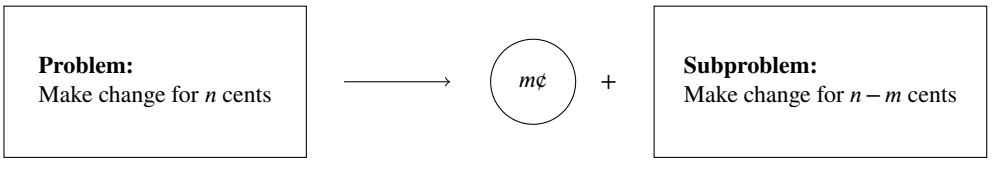
In this case, we can trivially show that $P(1)$ is true. This is because the problem satisfies the greedy-choice property, which means the first greedy choice must be a part of the optimal solution. Therefore, if a solution can be obtained after a single choice, then that solution must be the optimal solution.

Inductive Step: *We want to show that $\forall k \geq 1,\ P(k) \rightarrow P(k+1)$. That is, $P(k)$ being true implies that $P(k+1)$ is also true.*

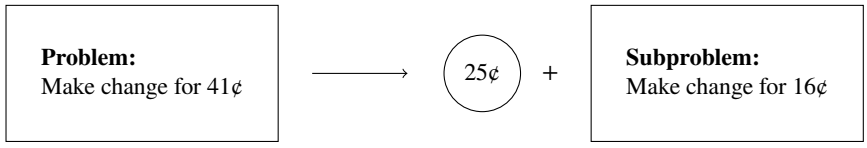
Suppose we are given input for which $k+1$ greedy choices are needed before a solution is obtained. Via the greedy-choice property, we know that the first greedy choice is part of some optimal solution. In addition, once we make a greedy choice, we are left with another subproblem. Because our original problem has an optimal substructure, we know that the optimal solution for this remaining subproblem can be combined with the greedy choice we just made to obtain an optimal solution for the entire problem. We just made a choice for the problem that required $k+1$ choices to obtain a solution, so there are k choices left to make in the remaining subproblem. Since $P(k)$ is true via the inductive hypothesis, the greedy algorithm must return an optimal solution for the remaining subproblem with k choices. Thus, we can combine this solution with the greedy choice to obtain an optimal solution for the original problem with $k+1$ choices, successfully proving that $P(k+1)$ is true.

Thus, we can conclude using mathematical induction that $P(n)$ is true for all $n \geq 1$. This implies that a greedy algorithm will always produce an optimal solution for problems with an optimal substructure that satisfy the greedy-choice property for that greedy algorithm.

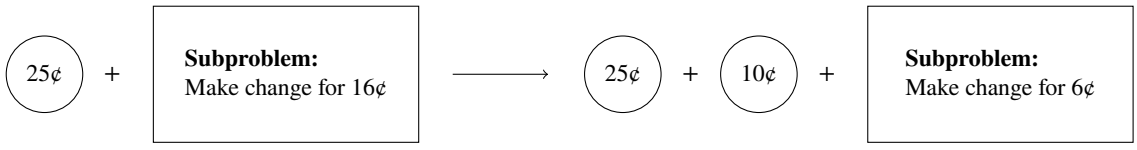
To demonstrate this procedure, let's consider the coin change problem for denominations of $25¢$, $10¢$, $5¢$, and $1¢$. To prove that greedily selecting the coin with the highest-denomination always leads to an optimal solution, we first want to frame the problem as one in which we make a greedy choice and then are left with one subproblem to solve. This is relatively straightforward for the coin change problem. If we want to make change for n cents, and the coin we greedily add to our solution has a value of m cents, we are left with the subproblem of making change for $n-m$ cents.



For example, suppose we wanted to make change for 41¢ using only quarters, dimes, nickels, and pennies. If we greedily select the feasible coin with the highest denomination at every choice, we would add a quarter to our solution first. This leaves us with 16¢ of change to fulfill.



Notice that this structure allows us to apply the greedy choice recursively to obtain a greedy solution, since the remaining subproblem has the same form as the original problem. In this case, after we make a greedy choice, we are essentially left with the same coin change problem as before, but for 16¢ rather than 41¢ (this process can be recursively applied until all choices are made and a potential solution is obtained).



Next, we want to prove that the greedy-choice property holds, and that the greedy choice is always safe to make. This is done by ensuring that the first greedy choice we make does not ever invalidate the optimal solution. One common strategy for proving that an optimal solution always contains the greedy choice is to use an *exchange argument*, which exhibits the following pattern:

1. Assume that an arbitrary optimal solution (which we will denote as O) does *not* include the first choice made by the greedy algorithm (which we will denote as g).
2. Modify the optimal solution O to create a new solution O' that includes g.
3. Show that O' is a valid solution that is no worse than O.

By using the exchange argument, we can show that any optimal solution that does *not* contain the first choice made by a greedy algorithm can be modified to create a solution that *does* include this first choice, and that this new solution is guaranteed to be no worse (and perhaps even better) than the optimal solution we had before. This would prove that there always exists an optimal solution that includes the first greedy choice, which implies that the greedy-choice property holds.

Example 21.4 Prove the greedy-choice property holds for the coin change problem for denominations 25¢, 10¢, 5¢, and 1¢.

To show this, we will use a proof by cases, where the change we want to make is either greater than 24¢, between 10¢ and 24¢, between 5¢ and 9¢, or less than 5¢. Note that an optimal solution must follow these rules:

- The number of pennies in an optimal solution must be less than five. Otherwise, we can replace these pennies with a single nickel to improve the solution.
- The number of nickels in an optimal solution must be less than two. Otherwise, we can replace these nickels with a single dime to improve the solution.
- The combined number of dimes and nickels in an optimal solution must be less than three. Otherwise,
  - If we have three dimes, we can improve the solution by replacing them with a quarter and a nickel.
  - If we have two dimes and one nickel, we can improve the solution by replacing them with a single quarter.
  - From our previous rule, we cannot have more than one nickel in our optimal solution.

If the amount of change we want to make is greater than 24¢, then the greedy choice would be to add a quarter to our solution. Assume that there exists an optimal solution O that does not include a quarter. If this were the case, we would have to make change for at least 25¢ using dimes, nickels, and pennies. This would require O to include at least three coins worth of dimes and nickels. However, from the rules above, we can modify this solution to get another solution O' that replaces any combination of two dimes and one nickel with a single quarter, and any combination of three dimes with one quarter and one nickel. This would imply that O' returns fewer coins than O, contradicting our initial assumption that O is optimal. Thus, if we have to make change for more than 24¢, there must exist an optimal solution that contains a quarter.

If the amount of change we want to make is between 10¢ and 24¢, then the greedy choice would be to add a dime to our solution. Assume that there exists an optimal solution O that does not include a dime. If this were the case, we would have to make change for at least 10¢ worth of change using nickels and pennies, which would require O to use at least two nickels. However, we can modify this solution to get another solution O' that replaces any combination of two nickels with a single dime. This would imply that O' returns fewer coins than O, contradicting our initial assumption that O is optimal. Thus, if we have to make change for 10-24¢, there must exist an optimal solution that contains a dime.

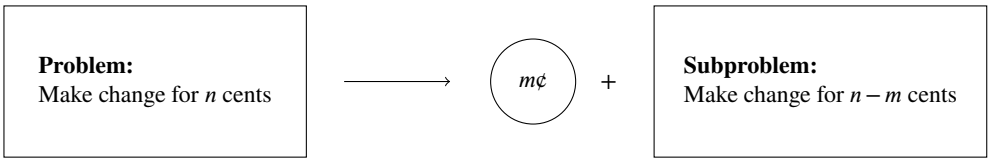
If the amount of change we want to make is between 5¢ and 9¢, then the greedy choice would be to add a nickel to our solution. Assume that there exists an optimal solution O that does not include a nickel. If this were the case, we would have to make change for at least 5¢ worth of change using only pennies, which would require O to use at least five pennies. However, we can modify this solution to get another solution O' that replaces any combination of five pennies with a single nickel. This would imply that O' returns fewer coins than O, contradicting our initial assumption that O is optimal. Thus, if we have to make change for 5-9¢, there must exist an optimal solution that contains a nickel.

If the amount of change we want to make is under 5¢ , then we only have one type of coin that can be added to our solution. This trivially implies the optimal solution must contain a penny. We have successfully completed our proof.

After showing that the problem satisfies the greedy-choice property, we want to demonstrate that it also exhibits an optimal substructure, where the greedy choice can be combined with the optimal solution of the remaining subproblem to obtain an optimal solution for the entire problem. Once we have identified an optimal substructure for the problem, we can then apply induction on the solution size to prove that the greedy choice will always find an optimal solution, as shown previously.

Example 21.5 Show that the greedy coin change approach exhibits optimal substructure for coin denominations of 25¢, 10¢, 5¢, and 1¢.

To show this, recall the illustration of the algorithm from before. Since our greedy algorithm makes a greedy choice and is then left with one subproblem to solve, we can prove optimal substructure by showing we can always combine the greedy choice of an m-cent coin with an optimal solution for $n-m$ cents to obtain an optimal solution for n cents.



We can do this using a proof by contradiction. Assume that the optimal solution for $n-m$ cents returns k coins (for some arbitrary natural number k), and that combining this optimal solution with an m-cent coin does not yield an optimal solution for n cents. This would mean that there exists a way to make change for n cents using fewer than $k+1$ coins. We know via the greedy-choice property that there must exist one optimal solution that includes the greedy choice of an m-cent coin. Therefore, we can take this optimal solution and remove the m-cent coin to return change for $n-m$ cents. Since the optimal solution for n cents uses fewer than $k+1$ coins, removing the m-cent coin would produce a solution for $n-m$ cents that uses fewer than k coins. This results in a contradiction, since the optimal solution for $n-m$ cents returns k coins. Thus, an optimal solution for n cents cannot use fewer than $k+1$ coins if the optimal solution of the remaining subproblem uses k coins. This implies that the greedy choice can always be combined with the optimal solution of the remaining subproblem to obtain an optimal solution for the entire problem, and our proof is now complete.

We will not go too deep into the weeds with proofs in this class, but it is good to understand how greedy algorithms are designed, and how their correctness can be proven. The greedy approach plays an important role in several of the algorithms that we have already discussed in this course, such as Prim's and Kruskal's algorithms, which uses the greedy property to efficiently construct a minimum spanning tree for weighted, connected, and undirected graphs.

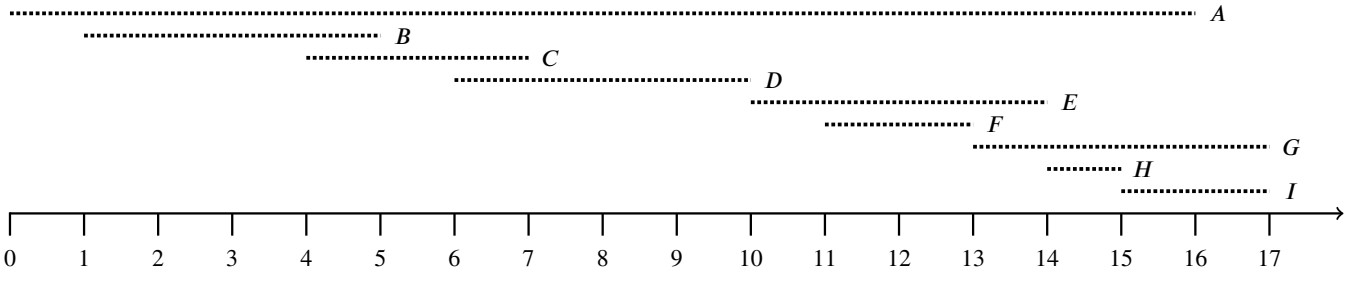
In the next few subsections, we will look at some additional examples of problems that can be solved using a greedy approach. You won't need to fully understand the proofs for why the greedy approach works in each of these examples; rather, the goal is to expose you to a few types of problems where a greedy algorithm can be successfully applied.

※ 21.3.3 *Activity Selection Problem*

In the **activity selection problem**, you are given a set of activities that share a common resource. Each of these activities has a specified starting and ending time, and only one activity can be scheduled on the shared resource at any time. Once an activity begins using the shared resource, it must be allowed to run to completion without any preemption. Your goal is to maximize the total number of activities that are scheduled.

The activity selection problem is an important problem in computer science, since its design can be applied to many different real-life applications, from scheduling a series of lectures that share the same lecture hall to assigning computational tasks to a shared processor. As an example, let's consider the following set of jobs, with the following start and end times:

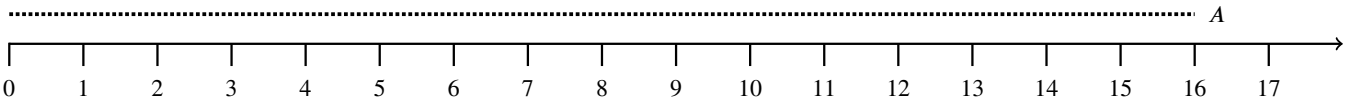
Job	Start Time	End Time
A	0	16
B	1	5
C	4	7
D	6	10
E	10	14
F	11	13
G	13	17
H	14	15
I	15	17



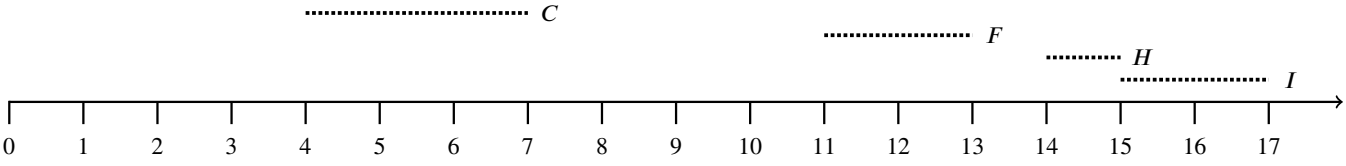
How would we go about approaching this problem? One way would be to check all possible subsets of activities, determine if each subset is feasible, and keep track of the feasible subset that maximizes the total number of activities scheduled. This brute force approach, however, would take exponential time. Is there a way to do better?

The key to notice here is that we have an optimization problem, where we want to maximize total activities scheduled under the constraint that only one activity can use the shared resource at a time. As a result, there may exist a greedy method that can be used to solve the problem. However, finding the right greedy approach (if there even is one) can be quite tricky — what metric should we use to determine what the locally optimal decision is at each step?

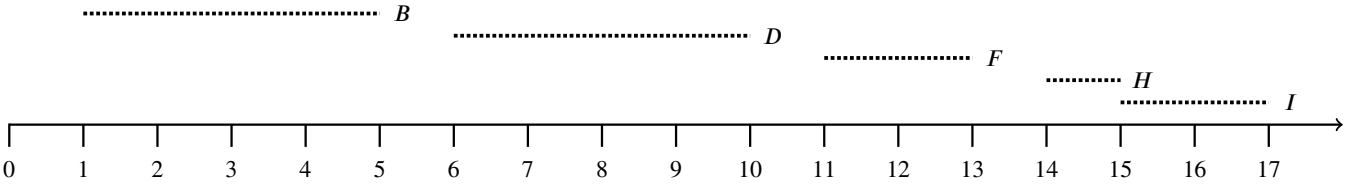
Instinctively, it might make sense to greedily select activities with the earliest start time. That is, the greedy algorithm would pick the available job that starts first, remove all jobs that conflict with the chosen job, and repeat until all jobs have been considered. However, this method does not work. If we used this approach to select jobs in the given example, we would end up scheduling job A and nothing else — this is clearly not an optimal solution!



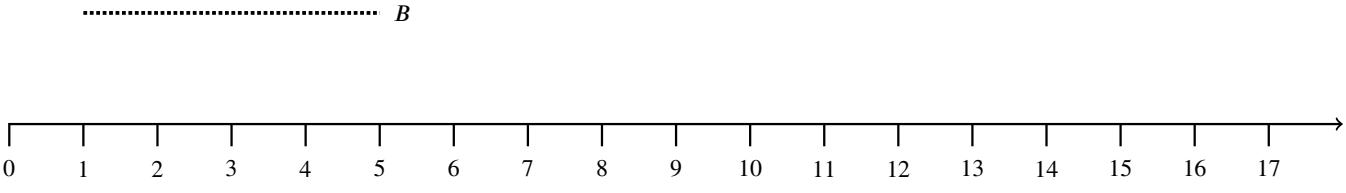
Another possibility would be to greedily select activities with the shortest duration (as long as they do not conflict with any activities already chosen). However, this approach also fails to discover an optimal solution in all cases. If we used this greedy method on our example, we would schedule the following jobs:



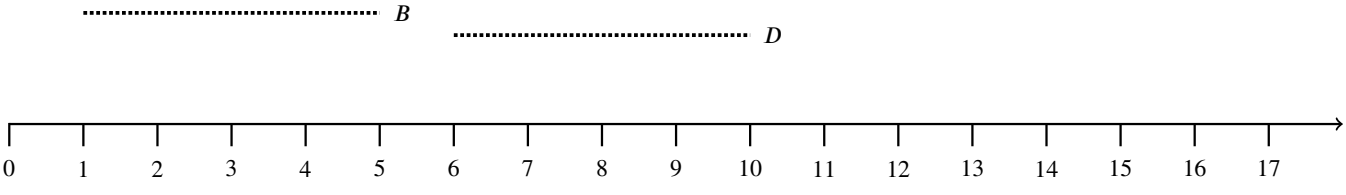
This is not optimal, since we can improve our solution by scheduling jobs B and D instead of job C:



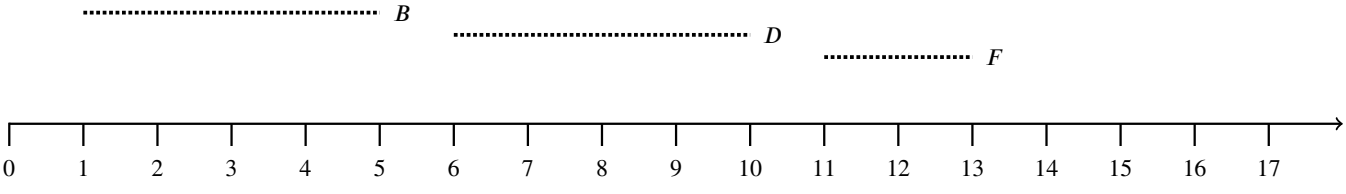
Fortunately, there does exist a greedy strategy that discovers the optimal solution in all cases, even if it is not the most obvious approach. In order to solve the activity selection problem, the correct strategy is to greedily select available activities with the *earliest finish time* first. This ensures that every activity we add to our schedule leaves open as much time as possible for any activities that may occur later down the line. In our example, job B has the earliest finish time, so we would add job B to our schedule first and discard activities that overlap (jobs A and C).



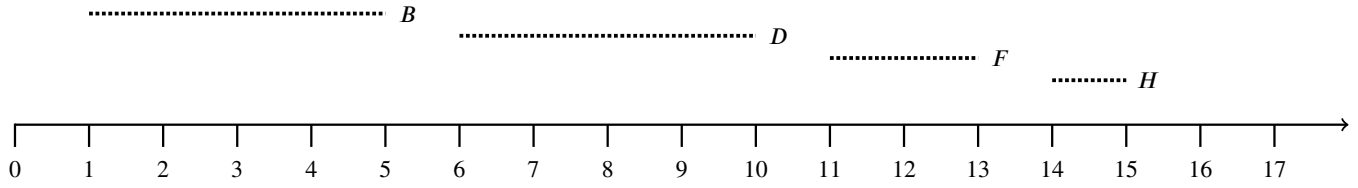
The next available job with the earliest finish time is job D, so we add job D to our schedule and discard any jobs that overlap (in this case, job D overlaps with jobs A and C, but these two jobs have already been discarded, so nothing more needs to be done).



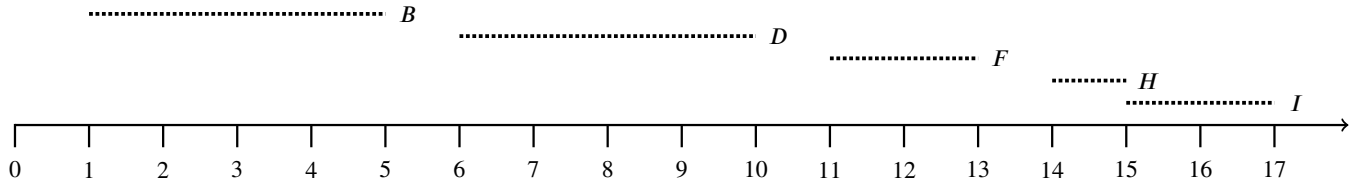
The next available job with the earliest finish time is job F, so we add job F to our schedule and discard job E, which overlaps.



The next available job with the earliest finish time is job H, so we add job H to our schedule and discard job G, which overlaps.



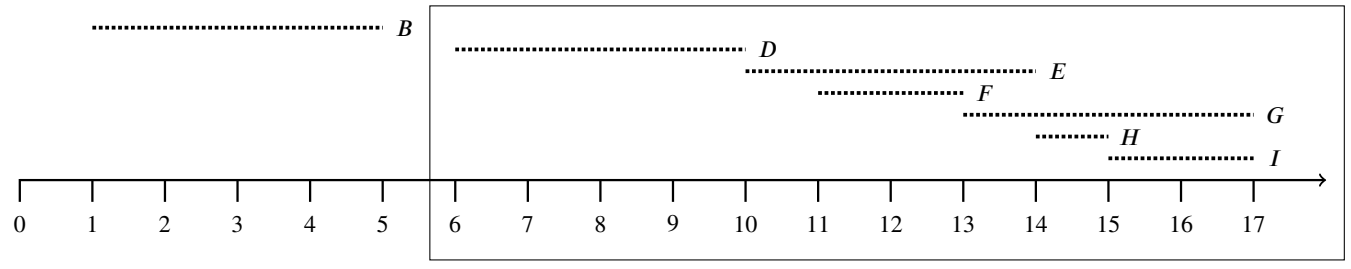
The next available job with the earliest finish time is job I, so we add job I to our schedule. All of the jobs have either been added or discarded, so we are now done. This is the final result, with an optimal solution of five scheduled jobs.



In our example, greedily selecting activities in order of earliest finish time got us the optimal solution. However, can we ensure that this greedy approach *always* returns the optimal solution, for any subset of jobs? Before we employ a greedy algorithm, we must make sure that it always returns an optimal solution, especially since the effectiveness of this approach for the activity selection problem is not immediately obvious. To prove that greedily selecting jobs by finish time works in all cases, we will follow the proof framework that was established earlier.

Frame the problem as one in which only one subproblem remains after a greedy choice is made.

This step is relatively straightforward for this problem. If our greedy algorithm selects some arbitrary job k that has the earliest finish time among the jobs still available, we are then left with the subproblem of maximizing the number of jobs we can schedule among the jobs that begin after job k ends. We do not need to worry about any other subproblems, since any other jobs must have already been considered (either added or discarded) if they begin before job k ends, since we know job k has the earliest finish time among the jobs still available. In our example, after making the greedy choice of scheduling job B, we are left with the subproblem of solving the activity selection problem for jobs D through I.



Show that the greedy-choice property is satisfied, such that at least one optimal solution to the original problem includes the first greedy choice made by the greedy algorithm.

Let a_k represent the activity with the earliest finish time, and thus the activity chosen by the first greedy choice. Assume that an optimal solution O does not include a_k. If this were the case, we can construct a new solution O' by removing the activity in O with the earliest finish time (which we will denote as a_f). Since no activities in O conflict with a_f, all of the remaining activities in O' must begin after a_f finishes. We know that activity a_k is the activity with the earliest finish time, so a_k must finish at the same time or before a_f finishes. This implies that all the remaining activities in O' must also begin after a_k finishes, and that a_k is compatible with all the activities currently in O'. Thus, it is safe to add a_k to O' to obtain an optimal solution that schedules the same number of jobs as O. Since O is optimal, O' must also be optimal. We have successfully proven that there must exist an optimal solution that contains the activity with the earliest finish time, and that the greedy-choice property holds.

Show that the problem exhibits an optimal substructure, where the optimal solution of the remaining subproblem can be combined with the greedy choice to obtain an optimal solution for the original problem.

Let S represent the original activity selection problem we are trying to solve, and let S' represent the subproblem that remains after a greedy choice is made. Assume that the optimal solution for S' schedules n jobs, for some arbitrary natural number n. We then claim that the problem does not exhibit an optimal substructure, and that the optimal solution for S' cannot be combined with the greedy choice a_k to obtain an optimal solution for S. This implies that the optimal solution for S must schedule more than $n+1$ jobs. We know via the greedy-choice property that there must exist an optimal solution O that includes the activity that finishes first. If we remove this activity from O, we would end up with a solution for S' that schedules more than n jobs, since O schedules more than $n+1$ jobs. This results in a contradiction! Thus, if the subproblem S' optimally schedules n jobs, the optimal solution for S must schedule exactly $n+1$ jobs… otherwise, we would be able to devise a solution for S' that schedules more than n jobs, contradicting our initial claim that S' optimally schedules n jobs. We have successfully proven that the problem exhibits an optimal substructure.

We have successfully shown that the greedy approach of selecting activities in order of increasing finish time satisfies the greedy-choice property, and that the activity selection problem exhibits optimal substructure. At this point, we can use mathematical induction to prove that this greedy approach will always discover the optimal solution in all cases (the inductive approach will not be explicitly shown since it is not specific to this problem; rather, the template provided on page 596 can be applied to all greedy algorithms once you prove that the greedy-choice property and optimal substructure are satisfied).

The activity scheduling problem can thus be solved using the following procedure:

- Sort the activities in order of increasing finish time.
- Greedily select the available activity with the earliest finish time and add it to the solution. Once an activity is scheduled, remove any conflicting activities from consideration.
- Repeat until all activities are either scheduled or discarded.

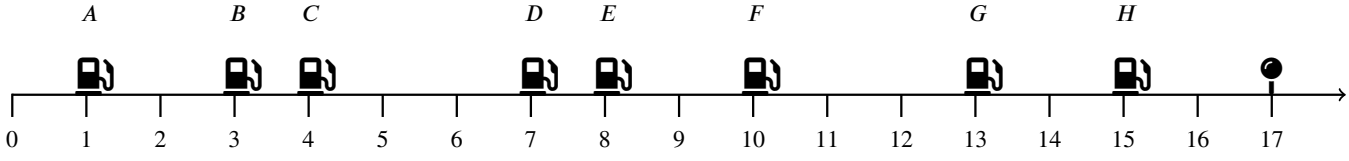
The time complexity of solving the activity selection problem is $\Theta(n \log(n))$, where the sorting process is the bottleneck of the entire algorithm. However, if the activities are already sorted in order of finish time, the time complexity would become $\Theta(n)$, since only a linear pass would then be needed to schedule all the activities. Regardless, this is exponentially better than the exponential time complexity of a brute force approach!

※ 21.3.4 *Breakpoint Selection Problem* (✻)

In this problem, you are on a planned route with n gas stations, which we will denote as b_0, b_1, …, b_{n-1}. Given that your car can only go at most m miles on a full tank, you want to identify a refueling schedule that minimizes the total number of stops you will need to make to travel between two locations without ever running out of gas (assuming you start with a full tank). An example is shown below:

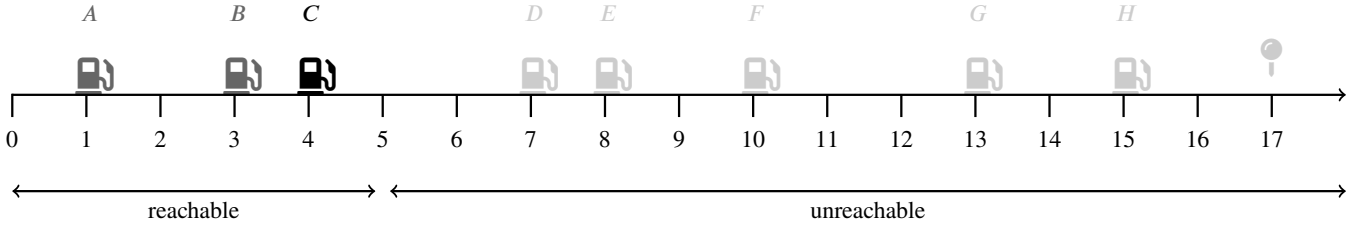
Capacity: 5 miles on a full tank
Destination: 17 miles

Gas Station	Distance (miles)
A	1
B	3
C	4
D	7
E	8
F	10
G	13
H	15

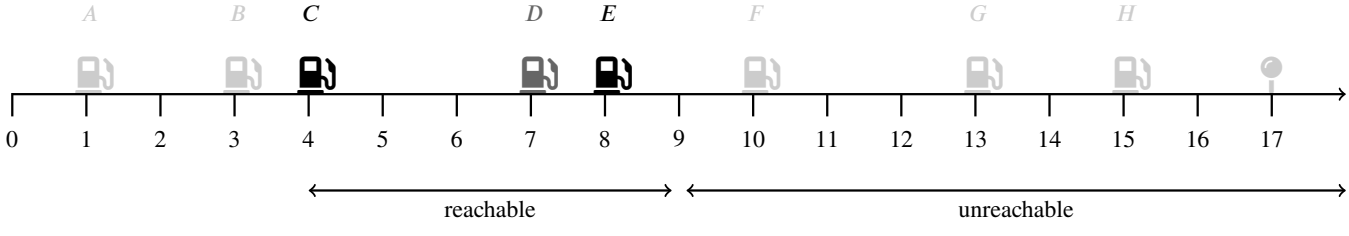


Since you want to minimize the number of stops you have to make, a reasonable greedy approach would be drive as far as you can before refueling. More formally, if you previously refueled at gas station b_i, and your car has a capacity m, the greedy choice would be to refuel at the gas station b_j with the greatest distance satisfying $\text{dist}(b_j) - \text{dist}(b_i) \leq m$.

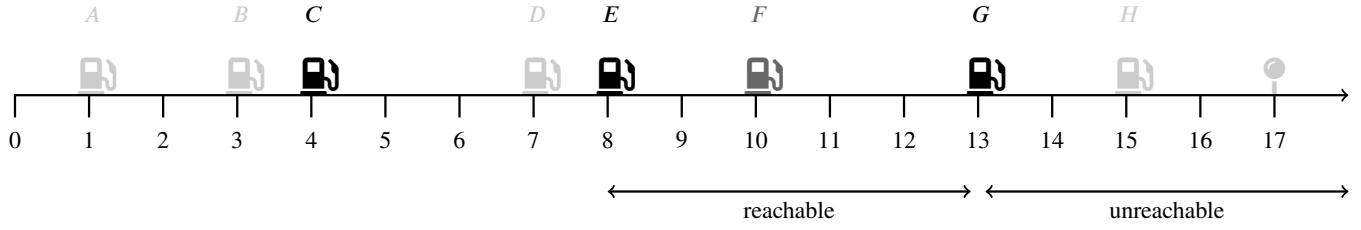
Let's consider the previous example. Since your car can drive up to 5 miles on a full tank, you can only reach gas stations *A*, *B*, and *C* before you run out of gas. Using the above greedy method, you would choose to refuel at gas station *C* since it is the reachable gas station that is farthest from your current position.



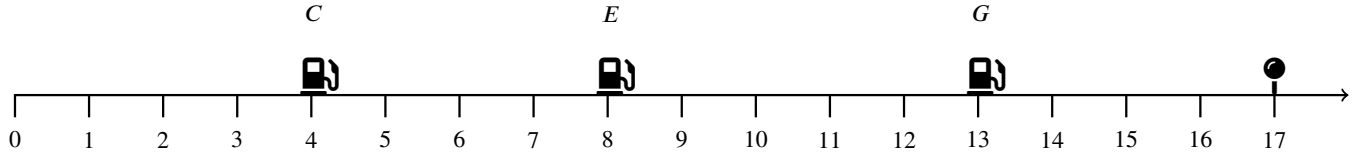
Your car would then have a full tank at position 4. At this point, you can reach all gas stations up to position 9, or stations *D* and *E*. Similar to before, the greedy approach would choose to refuel at gas station *E* since it is the reachable gas station that is farthest from your current position.



Your car would then have a full tank at position 8. Gas stations F and G are reachable, so the greedy choice refuels at station G.



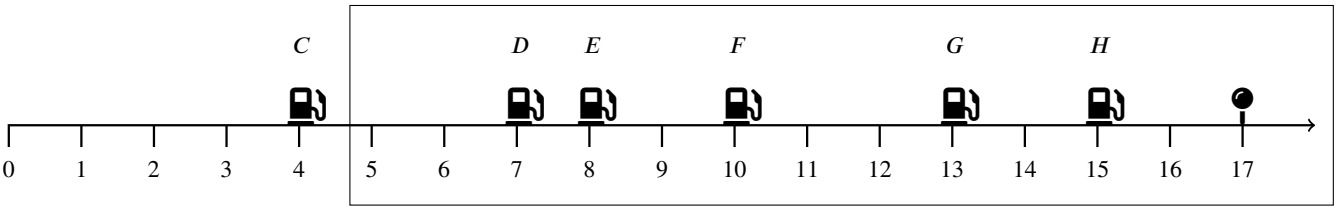
The destination is now reachable. Thus, the solution returned by the greedy approach would refuel at stations C, E, and G.



Is it possible to do better? You can try brute forcing all possible combinations of gas stations, but there exists no solution that involves fewer than three refueling stops. It turns out the greedy approach of choosing the farthest reachable gas station discovered an optimal solution for this example. However, one example is not enough to prove the efficacy of a greedy approach on all possible inputs! To prove that this greedy method works in all cases, we will apply the proof strategy covered earlier.

Frame the problem as one in which only one subproblem remains after a greedy choice is made.

If our greedy algorithm selects the farthest gas station that is reachable from the current position, we are left with the subproblem of minimizing the number of stops we need among the remaining gas stations along our trip. For example, after making the greedy choice of gas station C, we are left with the subproblem of solving the breakpoint selection problem for gas stations D through H with an initial starting position of 4.



Show that the greedy-choice property is satisfied, such that at least one optimal solution to the original problem includes the first greedy choice made by the greedy algorithm.

Let b_k represent the farthest gas station that is reachable from our current position, and thus the greedy choice. Assume that an optimal solution O does not include b_k. If this were the case, O must instead include some other gas station b_j as the first refueling stop. We know that the position of b_j must be before b_k, since b_k is the farthest reachable gas station. This implies that any gas station reachable from b_j must also be reachable from b_k. Thus, we can safely construct a new solution O' by replacing b_j with b_k, as the next stop in O is also reachable from b_k. O' involves the same number of refueling stops as O, and since O is optimal, O' must also be optimal. We have now successfully proven that there must exist an optimal solution that contains the farthest gas station that is reachable from our starting position, and that the greedy-choice property holds.

Show that the problem exhibits an optimal substructure, where the optimal solution of the remaining subproblem can be combined with the greedy choice to obtain an optimal solution for the original problem.

Let S represent the original breakpoint selection problem we are trying to solve, and let S' represent the subproblem that remains after a greedy choice is made. Assume that the optimal solution for S' stops at n gas stations, for some arbitrary natural number n. We then claim that the problem does not exhibit an optimal substructure, and that the optimal solution for S' cannot be combined with the greedy choice b_k to obtain an optimal solution for S. This implies that the optimal solution for S must visit fewer than $n+1$ gas stations. We know via the greedy-choice property that there must exist an optimal solution O that includes the farthest gas station that is reachable from our current position. If we remove this gas station from O, we would end up with a solution for S' that involves fewer than n stops, since O involves fewer than $n+1$ stops. This results in a contradiction! Thus, if the subproblem S' optimally visits n gas stations, the optimal solution for S must visit exactly $n+1$ gas stations… otherwise, we would be able to devise a solution for S' that makes fewer than n stops, contradicting our initial claim that S' optimally makes n stops. We have successfully proven that the problem exhibits an optimal substructure.

We have successfully shown that the greedy approach of selecting the farthest reachable gas station satisfies the greedy-choice property, and that the breakpoint selection problem exhibits an optimal substructure. Thus, we can conclude via mathematical induction that this greedy approach will always discover the optimal solution.

The breakpoint selection problem can therefore be solved using the following procedure:

- Sort the gas stations in order of distance away from your starting position.
- Greedily refuel at the farthest gas station that is reachable from your current position, given your car's capacity.
- Repeat until the destination is reached (or until you run out of reachable gas stations, which indicates that the destination is unreachable).

The time complexity of solving the breakpoint selection problem is $\Theta(n \log(n))$ if the gas stations are not already sorted in order of distance, and $\Theta(n)$ if they are. Similar to the activity selection problem, once we can show that a greedy approach works on all inputs for a problem, we can solve the problem using an algorithm that is often significantly better than brute force.

※ 21.3.5 *Huffman Coding* (✻)

As our last example, we will look at another practical application of greedy algorithms. This isn't a topic that will be covered in this class, so these notes won't go into too much detail here. However, this is knowledge that is good to know, especially as it relates to greedy algorithms and their usefulness in solving important problems we deal with on a daily basis.

Suppose you wanted to compress an ASCII text file to reduce its size on your hard drive, or to upload it to the internet faster. How would you do this? Recall that ASCII characters (`char`) are represented as one byte (or eight bits) in memory. For example, the string "hello world" would be represented like this in binary, where each character takes up eight bits ($8 \times 11 = 88$ bits total):

01101000 01100101 01101100 01101100 01101111 00100000 01110111 01101111 01110010 01101100 01100100

Here, the character 'h' has a value of 104 in ASCII (01101000 in binary), the letter 'e' has a value of 101 in ASCII (01100101 in binary), the letter 'l' has a value of 108 in binary (01101100 in binary), and so on. After receiving this sequence of bits, the computer then translates it into the string "hello world". However, this raises an interesting question: if each character must take up at least 8 bits of space in order to be uniquely identifiable, how is it possible to compress a text file even further? Wouldn't data compression be a physically impossible task for text files?

Well, obviously text compression must be possible, or else we wouldn't be able to compress text files without losing data! So how is it done? One potential way of compressing a text file is to assign a new *fixed-length code* based on which characters appear in the file. For instance, consider a file that only has the characters 'a', 'b', 'c', 'd', 'e', and 'f', each appearing the following number of times:

Character	a	b	c	d	e	f
Frequency	20	5	12	14	39	10

Since we know there are only six unique characters in this file, there is no need to use eight bits to represent each character. Instead, we can safely create a new mapping that can represent all possible characters using a fixed size of $\lceil \log_2(6) \rceil = 3$ bits. The following is an example of a potential mapping that can be used for these six characters.

Character	a	b	c	d	e	f
Code	000	001	010	011	100	101

How effective is this method of compression? Previously, without any compression involved, the file's contents would take up a total of 100 characters $\times$ 8 bits = 800 bits. After converting each character to a new three-bit code, the size of the file's contents drops to 100 characters $\times$ 3 bits = 300 bits.

That being said, the fixed-length approach obviously doesn't work in all cases. What if every possible ASCII character were present in the file we wanted to compress? In such a scenario, there would be no way to map each character to a new value that reduces the number of bits used per character. Since a fixed-length code isn't always ideal, a better way to compress the file would be to create a *variable-length code*, where each character in the file may be mapped to a variable number of bits. Here is one potential mapping that uses variable-length codes: some characters use up one bit, while others use up two.

Character	a	b	c	d	e	f
Code	0	1	00	01	10	11

Using this compression approach, the total size of the file's contents drops to

(20 'a' $\times$ 1 bit) + (5 'b' $\times$ 1 bit) + (12 'c' $\times$ 2 bits) + (14 'd' $\times$ 2 bits) + (39 'e' $\times$ 2 bits) + (10 'f' $\times$ 2 bits) = 175 bits

However, notice that the specific character assignments matter! If we instead changed the mappings around so that 'a' and 'e' were mapped to one bit, we end up dropping the file content size to

(20 'a' $\times$ 1 bit) + (5 'b' $\times$ 2 bits) + (12 'c' $\times$ 2 bits) + (14 'd' $\times$ 2 bits) + (39 'e' $\times$ 1 bit) + (10 'f' $\times$ 2 bits) = 141 bits

By assigning smaller values to more frequent characters, we ended up compressing our file even more.

In an ideal world, we could just map the most frequent characters to values that take up the fewest bits. However, this is not possible when working with variable-length codes. This is because, when a computer looks at the binary of a file, it can only see a *continuous* sequence of 0's and 1's. When our codes had a fixed-length, it was easy to determine where one letter ended and another began. However, we don't have this luxury with variable-length encoding; for example, if a computer sees the binary sequence "1010", the underlying character string could be "baba" (1 0 1 0), "bae" (1 0 10), "eba" (10 1 0), or "ee" (10 10).

Thus, to correctly develop an algorithm that uses variable-length codes, we also need to ensure that the characters in our encoding can be uniquely distinguishable within a sequence of 0's and 1's. To do so, we can only assign codes such that no code is a prefix of any other code (codes that satisfy this property are known as *prefix codes*) — otherwise, the character that begins an encoded file may be ambiguous.

Upon first glance, this problem seems challenging; we want to minimize the total size of a text file after compression, but we can only assign codes that do not share a common prefix with any other previously assigned code. In fact, this was very much an open problem in the early 20th century, one that puzzled even the greatest of computer scientists at the time. However, in 1951, MIT graduate student David Huffman solved this problem by inventing the concept of **Huffman coding**, a greedy approach for constructing a uniquely-identifiable, variable-length code system that guarantees optimal compression on all possible inputs.[1] The steps of this algorithm are as follows:

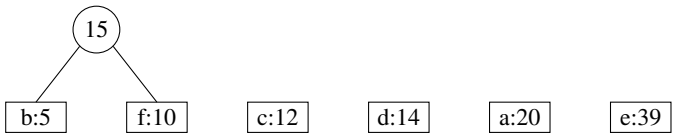
1. Sort the characters in the file in order of frequency.
2. Greedily select the two smallest frequency values and merge them together into a single node whose combined frequency is equal to the sum of its children.
3. Repeat step 2 until a single tree remains.
4. Assign variable-length codes to each character based on its position in the tree. If you need to travel down a left branch to get to the character, append a 0 to its prefix code; if you need to travel down a right branch, append a 1 to its prefix code.

To illustrate an example of how Huffman coding works, consider a file with the same character frequencies as before:

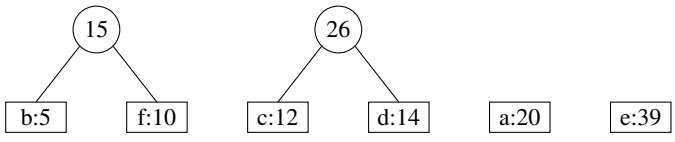
Character	a	b	c	d	e	f
Frequency	20	5	12	14	39	10

First, the characters are sorted in order of frequency:

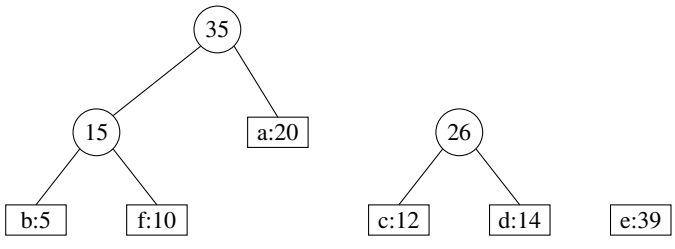
Then, the two smallest values are merged into a single node whose value is equal to the sum of its children. Here, ‘b’ and ‘f’ have the smallest frequencies of 5 and 10, so they are merged together into a node whose frequency value is $5 + 10 = 15$.



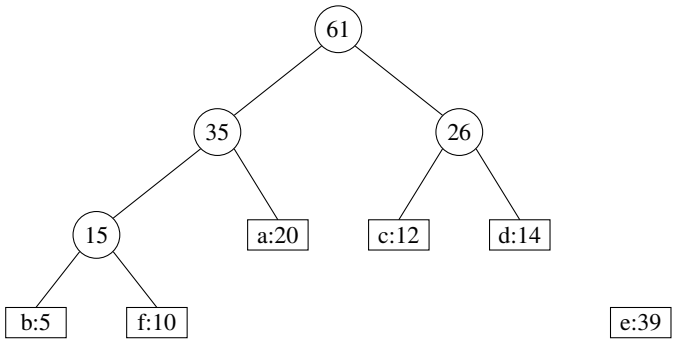
This merging process is repeated until a single tree remains. The characters ‘c’ and ‘d’ have the next smallest frequencies of 12 and 14, so they are merged into a single node whose frequency value is $12 + 14 = 26$.



The next two smallest frequency values are 15 for the "bf" node and 20 for ‘a’. Thus, these two components are merged into a single node whose frequency value is $15 + 20 = 35$.

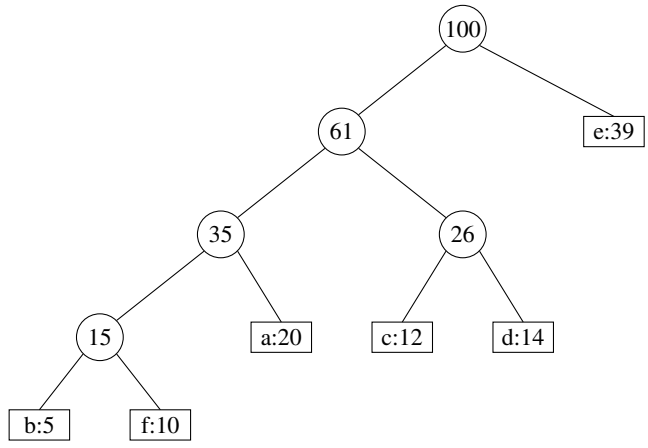


The next two smallest frequency values are 26 for the "cd" node and 35 for the "bfa" node. We merge these two components together to get a single node whose frequency value is $26 + 35 = 61$.

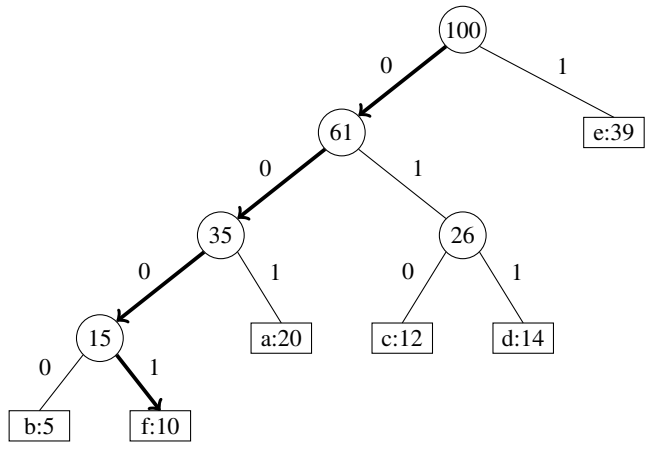


[1] For one of his classes, Huffman’s professor gave students the option to either take the final exam or write a term paper on finding the most optimal binary code. Huffman didn’t want to take the final exam, so he decided to write the paper instead. This led to the discovery of Huffman coding, which actually solved a problem that Huffman’s own professor had struggled with — unbeknownst to Huffman at the time, his professor had actually assigned an open problem that he was also trying to solve! ~~Surely something to think about if your EECS 376 professor ever challenges you to prove $P = NP$ instead of taking the final exam…~~

Now, the two smallest frequency values are 39 for ‘e’ and 61 for the remaining characters. We merge these two together to get a single node whose frequency value is 61 + 39 = 100. Our Huffman tree is now complete.



At this point, variable-length codes are assigned to each character based on their position in the tree. Whenever we travel down a left branch, we add a 0 to the character’s binary code. Whenever we travel down a right branch, we add a 1 to the character’s binary code. For example, the binary mapping of the character ‘f’ in this file would be 0001, since we travel down a left branch three times, and a right branch once.



The other binary mappings are shown in the table below. This is an optimal assignment of characters to bits that minimizes file size while also ensuring that characters are individually distinguishable within a sequence of bits (each bit indicates a direction to move down the tree, and we know that a new character begins once a character, or root node, is encountered).

Character	a	b	c	d	e	f
Code	001	0000	010	011	1	0001

Using the example from earlier, the size of the file contents using this variable-code system is:

$$(20\text{ ‘a’} \times 3\text{ bits}) + (5\text{ ‘b’} \times 4\text{ bits}) + (12\text{ ‘c’} \times 3\text{ bits}) + (14\text{ ‘d’} \times 3\text{ bits}) + (39\text{ ‘e’} \times 1\text{ bit}) + (10\text{ ‘f’} \times 4\text{ bits}) = 237\text{ bits}$$

Huffman coding guarantees a mapping that maximizes compression for variable-length coding, where the frequency of each symbol is known. Because of this, the algorithm forms the basis of many widely used compression formats, including JPEG, MP3, and gzip (which has file extension .gz, the compression format used to compress files before submission to the autograder).

※ 21.3.6 *Solving Problems Using a Greedy Approach*

In this section, we will look at a few interview-style questions that can be solved using a greedy approach. For these problems, proving that a greedy approach works allows you to implement a solution that is more performant than if you tried to implement the solution in any other way.

Example 21.6 Your company has two office locations in two different cities, which we will denote as offices A and B. The company wants to interview $2n$ candidates for a job opening, and they want to make sure an equal number of people are interviewed at each office.

You are given a vector `costs`, where `costs[i]` stores the cost of flying the i^{th} person to cities A and B (in the form `costs[i] = [cost_a, cost_b]`. Return the minimum cost required to fly every candidate to an office such that exactly n people end up in each office. The function header is shown below:

```
int32_t min_interview_cost(std::vector<std::vector<int32_t>>& costs);
```

Example: Given `costs = [[13, 42], [33, 65], [17, 29], [70, 68]]`, the optimal strategy is to send candidates 1 and 2 to office A and candidates 3 and 4 to office B, for a total cost of $13 + 33 + 29 + 68 = 143$. Thus, your should return 143 for this input.

The simplest solution would be to go through the list of costs and brute force your way to the optimal solution. However, that is certainly not an optimal strategy. Since this is an optimization problem, let's first try to see if a greedy approach can be used to derive a better solution.

There are several ways we can make a greedy choice. One way would be to greedily assign candidates in order of increasing cost. For instance, the lowest cost is 13, so we would first assign candidate 1 to office A. The next lowest cost among the remaining candidates is 17, so we then assign candidate 3 to office A. Since we now have two candidates at office A, the remaining two candidates must be assigned to office B. However, this approach clearly does not work: we end up with a cost $13 + 17 + 65 + 68 = 163$, which is not optimal. This is because the high expenses of assigning candidates 2 and 4 to office B far outweigh the value of cheaply assigning candidates 1 and 3 to office A.

Instead of looking purely at the absolute cheapest costs to assign each candidate to an office, a better method would be to consider the *total savings* (i.e., the difference in costs) of sending a candidate to one office rather than the another. By doing so, we can avoid situations like the one we ran into above. For instance, consider a candidate whose cost to office A is low, but whose cost to office B is not significantly higher. Even if sending this candidate to office A may be the better decision in isolation, it may be worthwhile to send them to office B if this allows us to reserve a position in office A for another candidate whose cost to office B is much higher.

Let's look at this strategy in action using the example provided. The following are the savings associated with each candidate (we are using a signed number here to highlight which office is better: if the number is negative, there is an advantage in sending that candidate to office A; if the number of positive, there is an advantage in sending that candidate to office B):

- *Candidate 1:* 13 - 42 = **-29**
- *Candidate 2:* 33 - 65 = **-32**
- *Candidate 3:* 17 - 29 = **-12**
- *Candidate 4:* 70 - 68 = **+2**

If we sort these numbers, we can see that the two smallest numbers (-32 and -29) belong to candidates 1 and 2. This means that sending these candidates to office A yields us the best advantage. On the other hand, the candidates with the larger two numbers (-12 and +2) belong to candidates 3 and 4. Using similar reasoning, it is best to send these two candidates to office B. The total cost of this arrangement ends up being $13 + 33 + 29 + 68$, which is our solution of 143.

If greedily selecting candidates by cost difference always yields an optimal solution, then the problem's implementation becomes trivial: we can just sort the candidates in order of cost difference and then assign the first half of candidates to office A and the second half of candidates to office B. However, is it safe to assume that this greedy approach always works? It turns out that it is, which we will show in the following proof using the exchange argument and a bit of math. Let us consider our candidates, sorted in order of cost difference d (defined as cost a - cost b):

$$[d_1, d_2, \ldots, d_n, d_{n+1}, \ldots, d_{2n}]$$

Let candidate X be any arbitrary candidate in the first half of the sorted array (which we greedily assigned to office A), and let candidate Y be any arbitrary candidate in the second half of the sorted array (which we greedily assigned to office B). By definition, we know that $d_X \leq d_Y$. Furthermore, suppose that the solution to the greedy approach is some value C.

Now, consider what happens if we send X to office B and Y to office A (essentially swapping the two candidates). The net cost we incur from sending X to office B instead of office A is $b_X - a_X$, where a_X is the cost of sending X to office A, and b_X is the cost of sending X to office B. Similar logic can be applied to determine the cost incurred from sending Y to office A instead of office B, which ends up being $a_Y - b_Y$. We will denote our new cost after swapping X and Y as C':

$$C' = C + \underbrace{(b_X - a_X)}_{\substack{\text{net cost from} \\ \text{sending } X \text{ to } B}} + \underbrace{(a_Y - b_Y)}_{\substack{\text{net cost from} \\ \text{sending } Y \text{ to } A}}$$

Notice that $(b_X - a_X)$ is equal to $-d_X$, and $(a_Y - b_Y)$ is equal to d_Y. Thus, we can rewrite our new cost as:

$$C' = C - d_X + d_Y$$

Since $d_X \leq d_Y$, the expression $C - d_X + d_Y$ cannot possibly be less than C! Thus, our new cost of C' cannot be optimal, and X and Y should not be swapped. Since X and Y are arbitrary candidates chosen from offices A and B (using our greedy solution), we have also proven that there is no way to improve our cost by switching *any* two candidates in the greedy solution. Therefore, our greedy approach must always be optimal!

An implementation of this greedy solution is shown below:

```
1  struct CostDiffCompare {
2    bool operator() (const std::vector<int32_t>& v1, const std::vector<int32_t>& v2) {
3      return v1[0] - v1[1] < v2[0] - v2[1];
4    } // operator()()
5  };
6
7  int32_t min_interview_cost(std::vector<std::vector<int32_t>>& costs) {
8    CostDiffCompare comp;
9    std::sort(costs.begin(), costs.end(), comp);
10
11   int32_t total_cost = 0;
12   for (size_t i = 0; i < costs.size() / 2; ++i) {
13     total_cost += costs[i][0] + costs[i + (costs.size() / 2)][1];
14   } // for i
15
16   return total_cost;
17 } // min_interview_cost()
```

The `CostDiffCompare` object is a custom comparator that can be used to sort the `costs` vector in order of increasing cost difference, as shown on line 9. The loop on line 12 then adds the costs to a running counter that is returned. Notice the optimization in the body of the loop: because we know that an equal number of candidates are assigned to each office, we can add the costs of the candidates in the first half *alongside* the costs of the candidates in the second half. This allows us to reduce the number of loop iterations from `costs.size()` to `costs.size() / 2`. We can also replace the comparator with a lambda expression, as shown:

```
1  int32_t min_interview_cost(std::vector<std::vector<int32_t>>& costs) {
2    std::sort(costs.begin(), costs.end(),
3      [](const std::vector<int32_t>& v1, const std::vector<int32_t>& v2) {
4        return v1[0] - v1[1] < v2[0] - v2[1];
5    });
6
7    int32_t total_cost = 0;
8    for (size_t i = 0; i < costs.size() / 2; ++i) {
9      total_cost += costs[i][0] + costs[i + (costs.size() / 2)][1];
10   } // for i
11
12   return total_cost;
13 } // min_interview_cost()
```

The time complexity of this solution is $\Theta(n \log(n))$, since the sorting step acts as the bottleneck of the entire algorithm. This complexity class is much better than any other solution we could have come up with, had we not pursued a greedy approach!

Example 21.7 You are playing a video game that requires you to go on side quests to gain experience before starting an important mission. Due to time constraints, you are only allowed to complete k side quests before beginning the mission. Each side quest requires a specific amount of experience (in the form of a positive integer), and you can only begin a side quest if you have the necessary experience beforehand. You can gain experience by completing side quests, each of which award a predetermined amount of experience. Because you want to be as prepared as possible for the important mission, you want to complete the side quests that allow you to gain the most experience.

Given a vector of n side quests, your initial experience, and the number of missions you must complete (i.e., k), implement a function that computes the maximum experience possible after completing k distinct side quests. The function header is shown below:

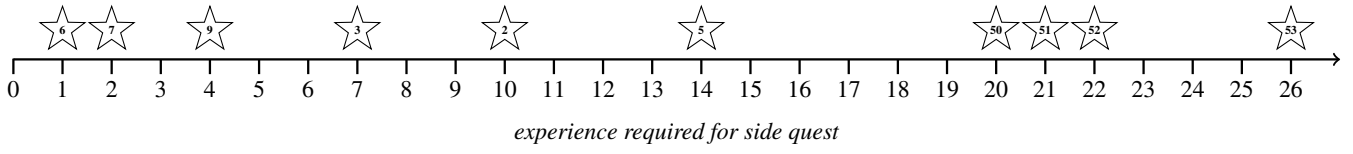
```
1  struct SideQuest {
2    int32_t exp_required;  // minimum experience required to start this side quest
3    int32_t exp_rewarded;  // experience gained by completing this side quest
4  };
5
6  int32_t max_experience(std::vector<SideQuest>& side_quests, int32_t init_exp, int32_t k);
```

Example: Given $k = 4$, an initial experience of 3, and the following ten side quests to choose from:

- *Side Quest A:* Experience Required: 1, Experience Rewarded: 6
- *Side Quest B:* Experience Required: 2, Experience Rewarded: 7
- *Side Quest C:* Experience Required: 4, Experience Rewarded: 9
- *Side Quest D:* Experience Required: 7, Experience Rewarded: 3
- *Side Quest E:* Experience Required: 10, Experience Rewarded: 2
- *Side Quest F:* Experience Required: 14, Experience Rewarded: 5
- *Side Quest G:* Experience Required: 20, Experience Rewarded: 50
- *Side Quest H:* Experience Required: 21, Experience Rewarded: 51
- *Side Quest I:* Experience Required: 22, Experience Rewarded: 52
- *Side Quest J:* Experience Required: 26, Experience Rewarded: 53

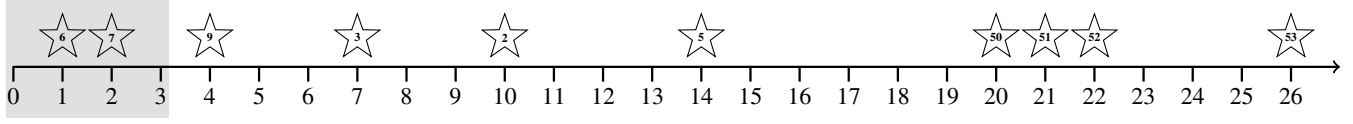
you would select side quests A, B, C, and I to maximize your experience to a value of 3 (initial) + 6 (side quest A) + 7 (side quest B) + 9 (side quest C) + 52 (side quest I) = 77.

This is another problem that can be solved using a greedy approach. In fact, you may have noticed that this problem is very similar to the gas station example we covered earlier: much like with the gas stations, greedily selecting the attainable side quest that rewards the most experience will get you to the optimal solution. For example, consider the side quests in the example, each rewarding a differing amount of experience.

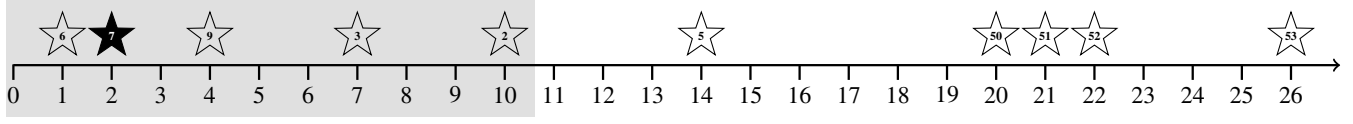


experience required for side quest

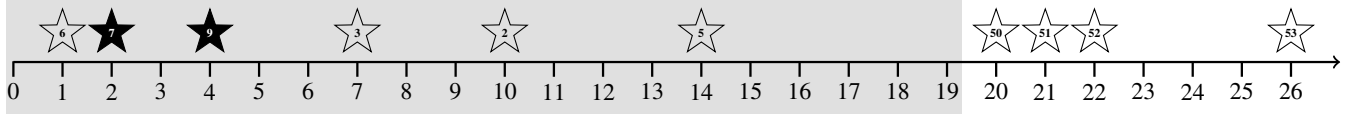
At the beginning, you start off with 3 experience. This gives you access to the first two side quests, as shown:



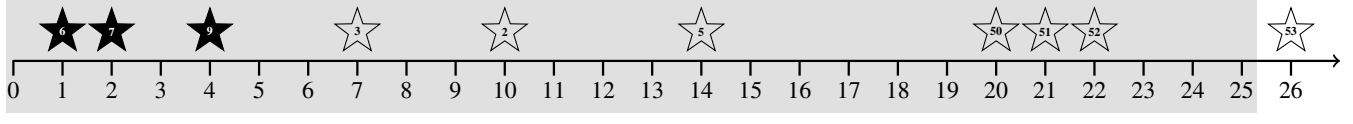
The greedy solution is to select the side quest that yields the highest experience among the quests that you are able to complete. In this case, you would choose the side quest that yields 7 experience, bringing your total experience up to $3 + 7 = 10$. This gives you access to several additional side quests up to an experience of 10.



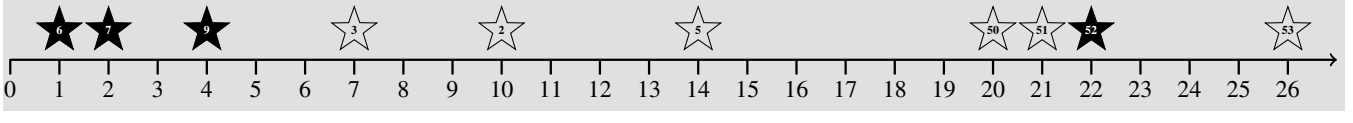
Among these side quests, the one that yields the most experience rewards 9, bringing you up to a total experience of $10 + 9 = 19$.



Among these side quests, the one that yields the most experience rewards 6, bringing you up to a total experience of $19 + 6 = 25$.



Among these side quests, the one that yields the most experience rewards 52, bringing you up to a total experience of $25 + 52 = 77$.



A total of $k = 4$ side quests have been completed, so we are done. The most experience you can gain is therefore 77.

The reason why this greedy approach works is similar to the reasoning behind the gas station problem covered earlier. Picking the side quest that gives you the most experience maximizes the choices you have for later side quests, giving you more opportunities to unlock side quests that may reward more experience in the future. As a result, choosing any side quest other than the one that yields the most experience can never lead to an outcome better than that of the greedy approach.

There are several ways to implement this problem, one of which is shown below. Here, we first sort the vector of side quests in order of experience required. Then, we insert the attainable projects into a max-priority queue based on the experience rewarded by each side quest. As long as this priority queue is not empty, we continuously pop off the side quest at the top, add its experience to our total, and push in any additional quests that are now attainable with our new experience. After k quests are completed, we return our total experience as the solution.

```
1  struct SideQuestCompare {
2    bool operator() (const SideQuest& q1, const SideQuest& q2) {
3      return q1.exp_required < q2.exp_required;
4    } // operator()()
5  };
6
7  int32_t max_experience(std::vector<SideQuest>& side_quests, int32_t init_exp, int32_t k) {
8    SideQuestCompare comp;
9    std::sort(side_quests.begin(), side_quests.end(), comp);
10
11   int32_t curr_exp = init_exp;
12   std::priority_queue<int32_t> quest_pq;
13   for (int32_t i = 0; k > 0; --k) {
14     while (i < side_quests.size() && curr_exp >= side_quests[i].exp_required) {
15       quest_pq.push(side_quests[i++].exp_rewarded);
16     } // while
17     if (!quest_pq.empty()) {
18       curr_exp += quest_pq.top();
19       quest_pq.pop();
20     } // if
21   } // for i
22
23   return curr_exp;
24 } // max_experience()
```

Both sorting the side quests (line 9) and pushing and popping the quests from the priority queue (lines 13-21) are bounded by a time complexity of $\Theta(n\log(n))$, where n is the number of side quests you are given in the `side_quests` vector. Since these steps serve as the bottleneck of the algorithm, the overall time complexity of this solution is also $\Theta(n\log(n))$.

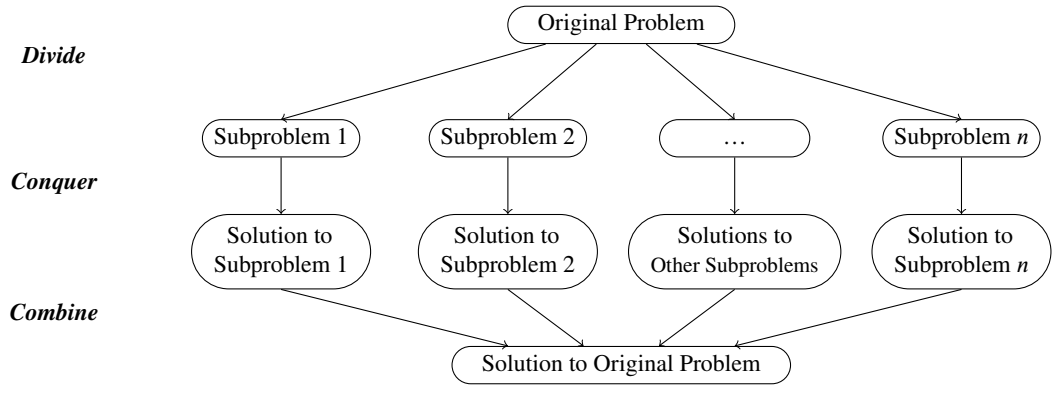
21.4 Divide-and-Conquer

※ 21.4.1 *Divide-and-Conquer and Independent Subproblems*

Suppose you are given a group assignment consisting of multiple questions that can be answered independently. If we assume that everyone in your group is equally available and competent, what is the fastest way to get this assignment done? A reasonable approach would be to split the assignment up among the group members, have each member complete their assigned questions, and combine all of your answers at the end. This is much faster than having someone complete all the questions on their own.

The same idea can be applied when developing algorithms for certain types of problems. As long as a given problem can be split up into different *independent* subproblems[2] (where the answer to one subproblem does not depend on the answers to other subproblems), we can divide the problem into smaller components and solve each component separately, and then combine the solutions of these components to obtain a solution to the original problem. This is the foundation of the **divide-and-conquer** algorithmic approach. Divide-and-conquer is often implemented recursively, with the following three steps applied at each level of the recursion:

- **Divide** a larger problem into smaller versions of the same problem that can be solved independently.
- **Conquer** the subproblems by solving them individually (either by using recursion, or by using a straightforward approach if the input size is small enough).
- **Combine** the solutions of the subproblems in a meaningful way to construct the solution for the original, larger problem.



[2] What if we have *dependent* subproblems, where the answer to one subproblem may depend on another? In that case, we would use *dynamic programming* instead, which will be covered in chapter 23.

The combining step is trivial for problems that only examine one smaller subproblem at every step, since there isn't anything to "combine" the solution of this subproblem with. An example of an algorithm that divides its input into only one subproblem is binary search, which was covered in chapter 15.

Remark: Problems such as binary search are often considered an "edge case" of the divide-and-conquer paradigm, since they only divide a problem into *one* smaller subproblem, and thus there is no combining work to do. To differentiate this special case from situations where there are two or more subproblems (and thus may involve combining work), some have proposed a new category of algorithms known as *decrease-and-conquer*, comprised of algorithms that reduce a problem to a *single* smaller subproblem with each recursive call. However, this algorithm family is not consistently applied, and many sources still use the divide-and-conquer paradigm to label algorithms that divide its input into one subproblem. For these notes, we will not introduce any additional algorithm families that aren't defined in the class or most established algorithm textbooks, such as CLRS. Therefore, we will follow the convention of defining algorithms that only examine one subproblem at each recursive depth, such as binary search, as part of the divide-and-conquer algorithm family.

The recursive nature of divide-and-conquer algorithms — where the original input size is split into several independent subproblems — makes them ideal candidates for the Master Theorem. If a divide-and-conquer algorithm

- separates a problem of input size n into a smaller subproblems whose input size is $1/b$ the size of the original problem ($a \geq 1$, $b > 1$)
- the work to divide/combine the input takes $f(n)$ time

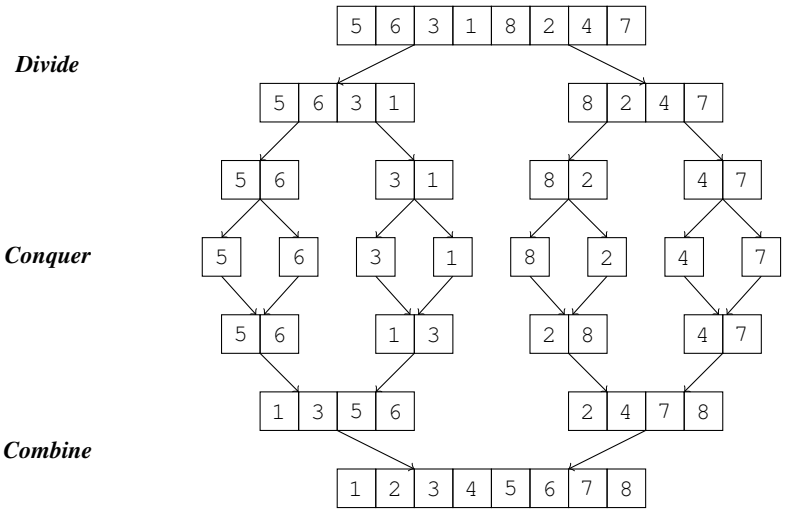
we can represent the algorithm's runtime as:

$$T(n) = aT(\tfrac{n}{b}) + f(n)$$

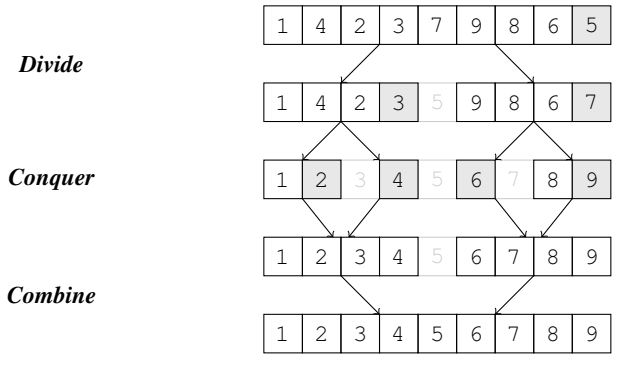
Then, we can apply the Master Theorem to obtain the time complexity of the algorithm:

$$T(n) = \begin{cases} \Theta(n^{\log_b(a)}), & \text{if } a > b^c \\ \Theta(n^c \log(n)), & \text{if } a = b^c \\ \Theta(n^c), & \text{if } a < b^c \end{cases}$$

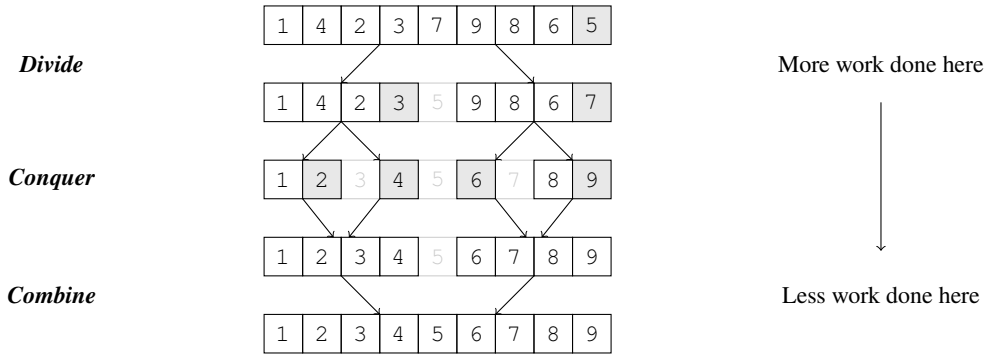
A common example of the divide-and-conquer paradigm is top-down recursive mergesort, since the dividing, conquering, and combining steps are well-established. First, mergesort continuously *divides* the input into smaller subarrays, until the input size of each subarray is small enough to be solved trivially. Then, the algorithm *conquers* the subproblems by sorting them individually, and then *combines* the results in sorted order using the `merge()` function. A diagram of this process is shown below:



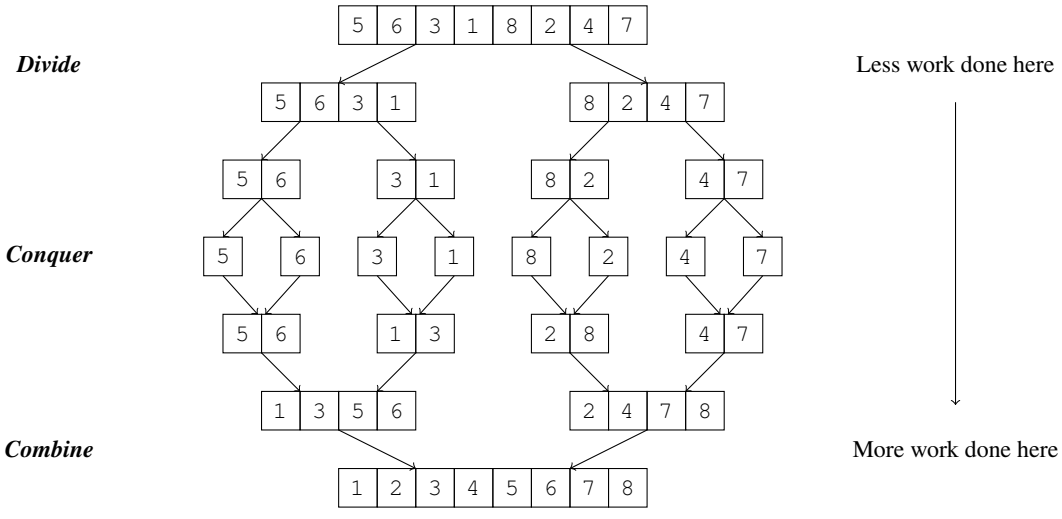
Quicksort is another algorithm that uses divide-and-conquer. At each step, quicksort partitions the input based on a pivot value, then *divides* the input into smaller subarrays based on the position of the pivot. The algorithm then *conquers* the subproblems by sorting the values to the left and right of the pivot, and then *combines* the results to obtain the sorted result for the original input.



Generally, both mergesort and quicksort are accepted as part of the divide-and-conquer algorithm family. However, there is a slight difference between the two with respect to where most of the work is done, even if both algorithms share the same structure. With quicksort, the implementation is top-heavy, with the work primarily done during the dividing step (i.e., partitioning with the pivot value). However, this allows the combining step to be completed trivially, as all the values end up in their correct sorted positions after each subproblem is conquered.



The opposite is true for mergesort. In the case of mergesort, the implementation is bottom-heavy, as the work is primarily done during the combining step (i.e., merging the individual subcomponents). This work allows the dividing step to be completed trivially, since no preprocessing work needs to be done before the input is split into several subproblems.



To differentiate between these two types, you may see the term **combine-and-conquer** in this class to describe algorithms that break a problem into smaller independent subproblems, but rely primarily on the combining step to obtain the solution of the original problem. Combine-and-conquer algorithms start with the smallest subdomain possible, then combine increasingly larger subdomains until the original, larger problem is solved. Under this definition, mergesort can be treated as a "combine-and-conquer" algorithm, since much of its work is spent combining together individual subproblems, rather than dividing the input (which is done trivially). On the contrary, algorithms that primarily rely on the dividing step to solve a problem, such as quicksort and binary search, retain the "divide-and-conquer" moniker under this definition.

Divide-and-conquer can be used to improve the time complexity of problems whose input can be divided into smaller, independent subproblems. A few examples are provided below.

※ 21.4.2 *Power Function*

Suppose you wanted to write a simple power function that can be used to take the power of any integer.

```
uint32_t power(uint32_t x, uint32_t n);  // returns x^n
```

A naïve solution would be to multiply `x` a total of `n - 1` times:

```
x^n = x * x * x * x * x * ... * x
```

However, this approach would take $\Theta(n)$ time, since $n-1$ multiplications have to be done. We can reduce the number of multiplications we need by recursively solving the subproblem of $x^{n/2}$, and then multiplying the result with itself to obtain the solution for x^n (note: if n is odd, solve for $x^{\lfloor n/2 \rfloor}$, multiply it with itself, and then multiply again by an additional factor of x). The recurrence relation is shown below:

$$x^n = \begin{cases} 1, & \text{if } n = 0 \\ x^{n/2} * x^{n/2}, & \text{if } n > 0, \text{even} \\ x * x^{\lfloor n/2 \rfloor} * x^{\lfloor n/2 \rfloor}, & \text{if } n > 0, \text{odd} \end{cases}$$

The divide-and-conquer implementation is shown below (for simplicity, we will ignore overflow):

```
1  uint32_t power(uint32_t x, uint32_t n) {
2    if (n == 0) {
3      return 1;
4    } // if
5    int result = power(x, n / 2);
6    result *= result;
7    if (n % 2 != 0) {  // if n is odd
8      result *= x;
9    } // if
10   return result;
11 } // power()
```

We can make this implementation tail recursive by adding an accumulator argument (covered in section 5.3) to perform the multiplication before the recursive call returns:

```
1  uint32_t power(uint32_t x, uint32_t n, uint32_t result = 1) {
2    if (n == 0) {
3      return result;
4    } // if
5    else if (n % 2 == 0) {  // even
6      return power(x * x, n / 2, result);
7    } // else if
8    else {  // odd
9      return power(x * x, n / 2, result * x);
10   } // else
11 } // power()
```

The divide-and-conquer solution to the power function can be expressed using the following recurrence relation:

$$T(n) = T(n/2) + \Theta(1)$$

Using the Master Theorem with $a = 1$, $b = 2$, and $c = 0$, we can conclude that the time complexity of this new solution is $\Theta(\log(n))$ with respect to the power value n. By splitting x^n into independent subproblems with powers less than n and using the solutions to these subproblems to obtain the solution for x^n, we were able to bring the time complexity of our implementation down from $\Theta(n)$ to $\Theta(\log(n))$.

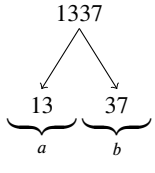
※ 21.4.3 *Integer Multiplication and the Karatsuba Algorithm (✻)*

Another prominent application of divide-and-conquer can be demonstrated using the **Karatsuba algorithm**, which is an optimized method that can be used to multiply two integers. To understand the usefulness of Karatsuba, let's first consider the standard integer multiplication strategy taught in grade school. Here, we first multiply the top number (1337) with the last digit in the final number (1). Then, we multiply the top number (1337) with the second-to-last digit of the bottom number (8), but with the result shifted one position to the left. Then, we multiply the top number with the third-to-last digit, then the fourth-to-last digit, etc. shifting the result one position to the left each time. Once the top number is multiplied for all the digits of the bottom number, we add the results (in this case, 1337 + 106960 + 267400 = 375697) to get the final answer.

$$\begin{array}{r} 1337 \\ \times \quad 281 \\ \hline 1337 \\ 10696 \\ +\ 2674 \\ \hline 375697 \end{array}$$

What is the time complexity of this algorithm, in terms of the number of digits in the bigger input number, n? Each multiplication would take $\Theta(n)$ time, and since we have to multiply the top digit once for every digit in the bottom number, computing the partial results takes $\Theta(n^2)$ time. Then, we have to add these n numbers, each of which have $\Theta(n)$ digits. Adding two $\Theta(n)$ digit numbers also takes $\Theta(n)$ time, so adding n of them takes a total of $\Theta(n^2)$ time. Since both of these steps occur sequentially, the time complexity of entire algorithm in terms of the number of digits n is $\Theta(n^2) + \Theta(n^2) = \Theta(n^2)$. Is there a way to do better than $\Theta(n^2)$?

The Karatsuba algorithm reduces this time complexity by using a divide-and-conquer approach. To understand Karatsuba, first notice what happens if we split a given number into two halves. We will denote the number in the left half as a and the number in the right half as b (for example, after splitting the number 1337 in half, we end up with $a = 13$ and $b = 37$).



Notice that there is a relationship between these two numbers and the original number:

$$\begin{aligned}1337 &= 13\times 100 + 37\\ &= 13\times 10^2 + 37\\ &= 13\times 10^{n/2} + 37\end{aligned}$$

In fact, we can split any n-digit number N into two halves a and b, such that $N = a\times 10^{n/2} + b$ (here, the number is in base-10, but you can replace 10 with any base you are working with). We can use this fact to simplify the multiplication of two numbers, which we will denote as N_1 and N_2. If we split N_1 into a and b, and N_2 into c and d, we get:

$$\begin{aligned}N_1 &= a\times 10^{n/2} + b\\ N_2 &= c\times 10^{n/2} + d\end{aligned}$$

We can then compute $N_1 \times N_2$ as:

$$\begin{aligned}N_1\times N_2 &= (a\times 10^{n/2}+b)\times(c\times 10^{n/2}+d)\\ &= (a\times c\times 10^n)+(a\times d\times 10^{n/2})+(b\times c\times 10^{n/2})+(b\times d)\\ &= (a\times c\times 10^n)+[(a\times d+b\times c)\times 10^{n/2}]+(b\times d)\end{aligned}$$

Instead of calculating $(a\times c)$, $(a\times d)$, and $(b\times c)$, and $(b\times d)$ directly, the Karatsuba algorithm only computes the following three values, which we will denote as m_1, m_2, and m_3:

$$\begin{aligned}m_1 &= (a+b)\times(c+d)\\ m_2 &= a\times c\\ m_3 &= b\times d\end{aligned}$$

This is because these three values are the only ones we need to know to be able to solve $N_1 \times N_2$, allowing us to minimize the total number of multiplications required. Observe that, when written out, m_1 is equal to:

$$\begin{aligned}m_1 &= (a+b)\times(c+d)\\ &= (a\times c)+(a\times d)+(b\times c)+(b\times d)\end{aligned}$$

If we subtract m_2 and m_3 from this result, we get:

$$\begin{aligned}m_1 - m_2 - m_3 &= (a\times c)+(a\times d)+(b\times c)+(b\times d)-(a\times c)-(b\times d)\\ &= (a\times d)+(b\times c)\end{aligned}$$

We can therefore substitute these values into our initial equation for $N_1 \times N_2$, as shown below. This completes the Karatsuba algorithm.

$$\begin{aligned}N_1\times N_2 &= (a\times c\times 10^n)+[(a\times d+b\times c)\times 10^{n/2}]+(b\times d)\\ &= (m_2\times 10^n)+[(m_1-m_2-m_3)\times 10^{n/2}]+m_3\end{aligned}$$

How efficient is Karatsuba? Instead of performing n multiplications (as with the grade school multiplication method), we only need to solve for m_1, m_2, and m_3, and substitute them into the equation for $N_1 \times N_2$. This leaves us with the following computations:

- Computing m_1 performs two additions of $n/2$-digit numbers, which takes $\Theta(n)$ time. The two $n/2$-digit numbers we get from the addition are then multiplied recursively (which contributes a term of $T(n/2)$ to our final recurrence relation).
- Computing m_2 and m_3 each requires a multiplication of two $n/2$-digit numbers, which is done by recursively applying Karatsuba twice with an input size of $n/2$. Each of these contribute a term of $T(n/2)$ to our final recurrence relation.
- After computing these three intermediary values, we combine the values by solving for $(m_2\times 10^n)+[(m_1-m_2-m_3)\times 10^{n/2}]+m_3$. This requires two additions and two subtractions, which take $\Theta(n)$ time. It also requires two multiplications with powers of ten, but these can also be done in $\Theta(n)$ time — this is because a number multiplied with 10^n simply appends n zeros to the end of the number (this can be done using a left shift, which was briefly introduced in chapter 17).

To summarize, the overall algorithm requires us to perform three recursive multiplications witn $n/2$-digit numbers, $\Theta(n)$ work to add two numbers when computing m_1, and $\Theta(n)$ work to solve for $(m_2\times 10^n)+[(m_1-m_2-m_3)\times 10^{n/2}]+m_3$ after computing m_1, m_2, and m_3. This results in the following recurrence relation:

$$T(n) = 3T(n/2)+\Theta(n)$$

Using the Master Theorem with $a = 3$, $b = 2$, and $c = 1$, we can conclude that the time complexity of the Karatsuba algorithm is $\Theta(n^{\log_b a}) = \Theta(n^{\log_2 3}) \approx \Theta(n^{1.585})$. By using divide-and-conquer, we have devised an integer multiplication algorithm whose time complexity is better than the $\Theta(n^2)$ time complexity produced by a standard grade school multiplication approach!

Just to illustrate an example of the Karatsuba algorithm using actual numbers, consider the multiplication of 1337×281, as covered earlier. With Karatsuba, we would break these two numbers into $a = 13$, $b = 37$, $c = 02$, and $d = 81$, where $n = 4$ (the number of digits in 1337). Then, we solve for the following three values:

$$m_1 = (a+b) \times (c+d) = (13+37) \times (02+81) = 50 \times 83 = 4150$$
$$m_2 = a \times c = 13 \times 2 = 26$$
$$m_3 = b \times d = 37 \times 81 = 2997$$

We can then use these three values to solve for 1337×281, as shown:

$$\begin{aligned} 1337 \times 281 &= (m_2 \times 10^n) + [(m_1 - m_2 - m_3) \times 10^{n/2}] + m_3 \\ &= (26 \times 10^4) + [(4150 - 26 - 2997) \times 10^2] + 2997 \\ &= (26 \times 10^4) + (1127 \times 10^2) + 2997 \\ &= 260000 + 112700 + 2997 \\ &= 375697 \end{aligned}$$

This gives us a solution of 375697, which is the same solution we got using the less efficient grade school method.

※ 21.4.4 *The Maximum Subarray Problem Using Divide-and-Conquer* (✻)

In this problem, you are given a one-dimensional array that contains both positive and negative integers, and you want to find the largest sum that can be obtained from a *contiguous* subarray of numbers in the array. For example, given the following array, the maximum subarray spans from index 4 to index 8 (inclusive), with a sum of 9 - 8 + 2 - 1 + 8 = 10.

5	1	−3	−4	**9**	**−8**	**2**	**−1**	**8**	−5	−6	2	3	−4	−1	7
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

The most obvious approach would be to use brute force and check every contiguous sequence, keeping track of the one with the largest sum. This can be done by looping through the array and considering all possible subsequences that begin at each index.

5	1	−3	−4	9	−8	2	−1	8	−5	−6	2	3	−4	−1	7
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sum: 5
Best so far: 5

5	**1**	−3	−4	9	−8	2	−1	8	−5	−6	2	3	−4	−1	7
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sum: 6
Best so far: 6

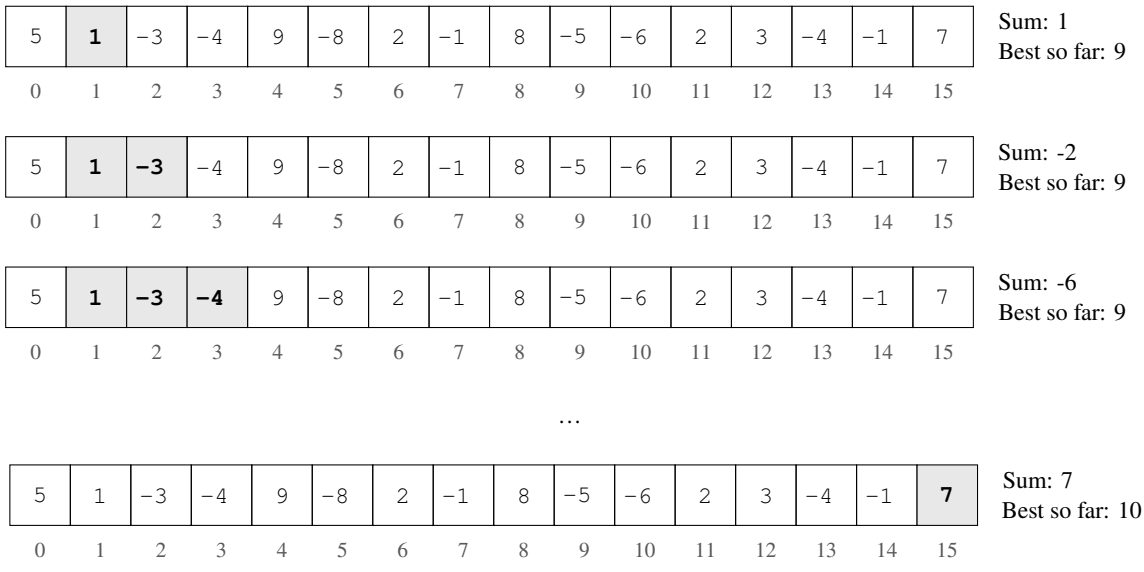
5	**1**	**−3**	−4	9	−8	2	−1	8	−5	−6	2	3	−4	−1	7
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sum: 3
Best so far: 6

…

5	**1**	**−3**	**−4**	**9**	**−8**	**2**	**−1**	**8**	**−5**	**−6**	**2**	**3**	**−4**	**−1**	**7**
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sum: 5
Best so far: 9

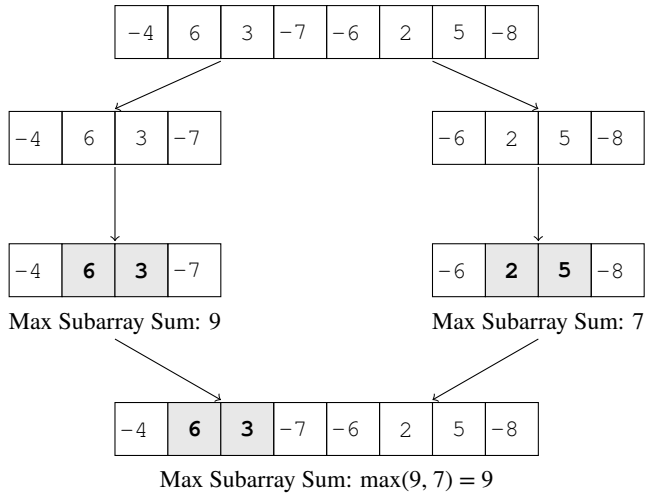


The code for the brute force approach is shown below:

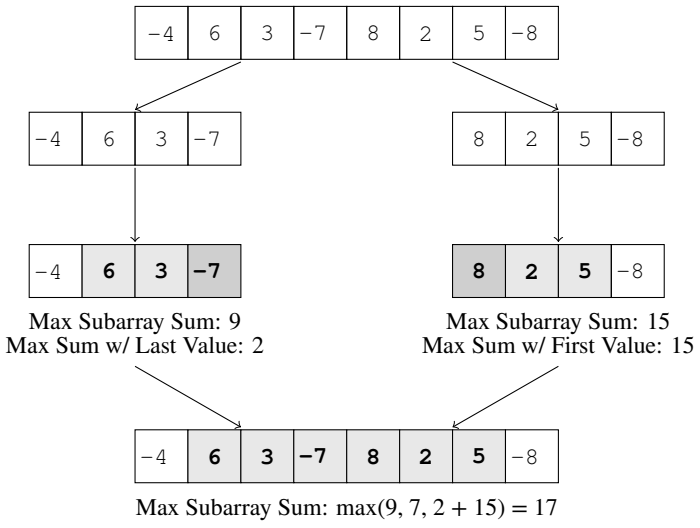
```
1   int32_t max_subarray_sum(const std::vector<int32_t>& nums) {
2     int32_t best = std::numeric_limits<int32_t>::min();
3     for (size_t i = 0; i < nums.size(); ++i) {
4       int32_t current_subarray_sum = 0;
5       for (size_t j = i; j < nums.size(); ++j) {
6         current_subarray_sum += nums[j];
7         best = std::max(current_subarray_sum, best);
8       } // for j
9     } // for i
10    return best;
11  } // max_subarray_sum()
```

The time complexity of this solution is $\Theta(n^2)$. Can we do better?

We can indeed do better if we use a divide-and-conquer approach. The key to notice here is that we can split the original input array into several subarrays whose individual solutions can be solved independently of other subarrays. That is, if we divide the initial array in half and recursively identify the maximum subset sum of these halves (which can each be solved independently, without knowing on the maximum subset sum of the other half), we can combine this information to obtain the maximum subset sum of the entire array. An example is shown below:



If you paid close attention, you may have noticed that this procedure does *not* fully work. This is because it fails to consider the case where the maximum subarray sum includes values from both subarrays. However, this issue can be resolved with a simple fix: instead of only finding the maximum subarray sum of the left and right halves, we will also find the maximum sum of *any subarray that crosses the boundary between the left and right subarrays*. This "crossing sum" can be calculated by finding the maximum contiguous sum in the left subarray that includes its rightmost element, as well as the maximum contiguous sum in the right subarray that includes its leftmost element, and then summing these two values together. An example is shown below:



Therefore, the maximum subarray problem can be solved using a divide-and-conquer approach as follows:

1. Divide the given input array into two halves.
2. Return the maximum of the following three values:
    - The maximum subarray sum of the left subarray.
    - The maximum subarray sum of the right subarray.
    - The maximum subarray sum of any subarray that includes elements from both sides.

An implementation of this solution is shown below:

```
1   int32_t max_subarray_sum(const std::vector<int32_t>& nums) {
2     return max_sum_helper(nums, 0, nums.size());
3   } // max_subarray_sum()
4
5   // inclusive left, exclusive right
6   int32_t max_sum_helper(const std::vector<int32_t>& nums, int32_t left, int32_t right) {
7     if (left == right - 1) {
8       return nums[left];
9     } // if
10    int32_t mid = left + (right - left) / 2;
11    int32_t current_sum = 0;
12    int32_t best_left_sum_with_mid = std::numeric_limits<int32_t>::min();
13    int32_t best_right_sum_with_mid = std::numeric_limits<int32_t>::min();
14    for (int32_t i = mid - 1; i >= left; --i) {
15      current_sum += nums[i];
16      best_left_sum_with_mid = std::max(best_left_sum_with_mid, current_sum);
17    } // for i
18    current_sum = 0;
19    for (int32_t j = mid; j < right; ++j) {
20      current_sum += nums[j];
21      best_right_sum_with_mid = std::max(best_right_sum_with_mid, current_sum);
22    } // for j
23    int32_t best_crossing_sum = best_left_sum_with_mid + best_right_sum_with_mid;
24    int32_t max_left_sum = max_sum_helper(nums, left, mid);
25    int32_t max_right_sum = max_sum_helper(nums, mid, right);
26    return std::max({max_left_sum, max_right_sum, best_crossing_sum});
27  } // max_sum_helper()
```

What is the time complexity of this divide-and-conquer approach? The process of finding the maximum sum that includes values from both subarrays (lines 14-23) requires a worst-case $\Theta(n)$ iteration of the subarray, starting from the boundary edge. Then, two recursive calls are completed (lines 24-25) with half the input size. All remaining work takes constant time. Putting this all together, the total work done can be expressed using the following recurrence relation:

$$T(n) = 2T(n/2) + \Theta(n)$$

Using the Master Theorem with $a = 2$, $b = 2$, and $c = 1$, we can conclude that the time complexity of this new solution is $\Theta(n^c \log(n)) = \Theta(n^1 \log(n)) = \Theta(n \log(n))$ with respect to the size of the original array n. This is an improvement over the $\Theta(n^2)$ time complexity of the previous brute force approach.

※ 21.4.5 *The Maximum Subarray Problem Using Kadane's Algorithm* (✻)

However, there is actually a way to do better than the $\Theta(n \log(n))$ time complexity of a divide-and-conquer approach. The optimal solution to the maximum subarray problem utilizes an algorithm known as **Kadane's algorithm**. Kadane's algorithm incrementally calculates the maximum subarray sum ending at each index of the original input array. That is, it calculates the maximum sum attainable from any subarray whose last element is at index 0, then the maximum sum of any subarray whose last element is at index 1, then the maximum sum of any subarray whose last element is at index 2, and so on. The largest of these values would then be the solution to the entire maximum subarray problem.

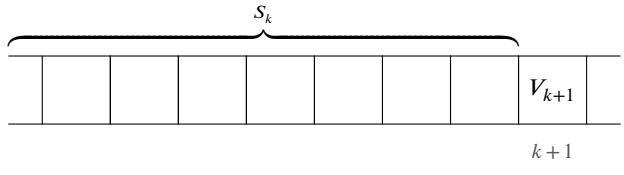
5	1	-3	-4	9	-8	2	-1	8	-5	-6	2	3	-4	-1	7
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

For example, the maximum subarray ending at index 6 is [9, -8, 2], since this subarray yields a larger sum than all other subarrays that end at index 6, as illustrated below:

- sum([5, 1, -3, -4, 9, -8, 2]) = 2
- sum([1, -3, -4, 9, -8, 2]) = -3
- sum([-3, -4, 9, -8, 2]) = -4
- sum([-4, 9, -8, 2]) = -1
- **sum([9, -8, 2]) = 3 → best!**
- sum([-8, 2]) = -6
- sum([2]) = 2

At this point, you may be wondering: isn't this just a rehash of the brute force approach, since you have to generate and compare all the subarrays that end at each index? Although the ideas are similar, Kadane's algorithm is not the same as the $\Theta(n^2)$ brute force approach, as it takes advantage of a key optimization: if you already know the maximum subarray sum of any subarray ending at index k, you can calculate the maximum subarray sum ending at index $k+1$ in *constant* time.

To understand why this works, consider the following example. Suppose we are given an array, and we want to find the maximum subarray sum of any subarray that ends at an arbitrary index $k+1$, where $k \geq 0$. Assume we know that the value at index $k+1$ is some number V_{k+1}, and that the maximum subarray sum that ends at index k is some number S_k.



We make the claim that the maximum subarray sum ending at index $k+1$, or S_{k+1}, must be the larger of:

- V_{k+1}
- $S_k + V_{k+1}$

Why must this be true? Assume that the maximum subarray sum ending at index $k+1$ is neither V_{k+1} nor $S_k + V_{k+1}$, but rather some other value T_{k+1}. We know that $T_{k+1} \neq V_{k+1}$, so T_{k+1} must be a combination of V_{k+1} and some other subarray sum ending at index k, which we will denote as T_k. However, we also know that $T_k \neq S_k$, or T_{k+1} would be equal to $S_k + V_{k+1}$, and we initially assumed that $T_{k+1} \neq S_k + V_{k+1}$. This means that T_k is a subarray sum ending at index k, but *not* the maximum subarray sum ending at index k. This implies that $T_{k+1} = T_k + V_{k+1} < S_k + V_{k+1}$. Since T_{k+1} is always less than $S_k + V_{k+1}$, it would be impossible for T_{k+1} to be the maximum subarray sum ending at index $k+1$ … a contradiction! Thus, if the maximum subarray sum ending at index $k+1$ is not V_{k+1}, it must be the sum of V_{k+1} and the maximum subarray sum ending at index k, or $S_k + V_{k+1}$.

Therefore, to calculate the maximum subarray sum ending at index $k+1$, we only need to compare the values of V_{k+1} and $S_k + V_{k+1}$. This can be done in constant time if we already know the value of S_k. Consider the same array as before:

5	1	−3	−4	9	−8	2	−1	8	−5	−6	2	3	−4	−1	7
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Using the example above, we would iterate over the array and calculate the following values using Kadane's algorithm:

- The maximum subarray sum ending at index 0 is 5 (the first value).
- The maximum subarray sum ending at index 1 is max(1, 5 + 1) = 6.
- The maximum subarray sum ending at index 2 is max(-3, 6 - 3) = 3.
- The maximum subarray sum ending at index 3 is max(-4, 3 - 4) = -1.
- The maximum subarray sum ending at index 4 is max(9, -1 + 9) = 9.
- The maximum subarray sum ending at index 5 is max(-8, 9 - 8) = 1.
- The maximum subarray sum ending at index 6 is max(2, 1 + 2) = 3.
- The maximum subarray sum ending at index 7 is max(-1, 3 - 1) = 2.
- **The maximum subarray sum ending at index 8 is max(8, 2 + 8) = 10 → best!**
- The maximum subarray sum ending at index 9 is max(-5, 10 - 5) = 5.
- The maximum subarray sum ending at index 10 is max(-6, 5 - 6) = -1.
- The maximum subarray sum ending at index 11 is max(2, -1 + 2) = 2.
- The maximum subarray sum ending at index 12 is max(3, 2 + 3) = 5.
- The maximum subarray sum ending at index 13 is max(-4, 5 - 4) = 1.
- The maximum subarray sum ending at index 15 is max(-1, 1 - 1) = 0.
- The maximum subarray sum ending at index 16 is max(7, 0 + 7) = 7.

The maximum subarray sum of the entire input is the largest of these values, or 10. The code is shown below:

```
1  int32_t max_subarray_sum(const std::vector<int32_t>& nums) {
2    int32_t max_current_index = nums[0];
3    int32_t best_so_far = max_current_index;
4    for (size_t i = 1; i < nums.size(); ++i) {
5      max_current_index = std::max(nums[i], max_current_index + nums[i]);
6      if (best_so_far < max_current_index) {
7        best_so_far = max_current_index;
8      } // if
9    } // for i
10   return best_so_far;
11 } // max_subarray_sum()
```

Since Kadane's algorithm performs a constant time operation for each index of the original input array, its runtime is $\Theta(n)$.

Our discussion of divide-and-conquer concludes here. However, this will not be the last you will see of this algorithm family in this class: we will revisit this paradigm in chapter 26 when we discuss the *closest pair of points problem*.

In summary, brute force, greedy, and divide-and-conquer are just a few algorithmic patterns that can be used to approach different problems. However, this is not all; we have merely scraped the surface of all the algorithm families that will be explored in this course! In the next chapter, we will examine two new algorithm families that at times have similar implementations, but are used to solve two different categories of problems: the algorithm families of *backtracking* and *branch and bound*.