Material here is adapted from the books “The Algorithm Design Manual” of Steven Skiena and “How to solve it” of George Polya.

1.1 The Attack Plan

Many people, upon given a problem, simply stare at the paper without knowing what to do next. Many other people instead have a wild goose chase hoping that somehow they will stumble into a right path. Steven Skiena, a master of algorithm design, suggests the following list of questions to help you find the next step.\(^1\) When answering a question like “Can I do it this way?”, don’t just say “No”, but “No, because . . . ” By seeking an explicit, clear answer, you can check whether you have missed a possibility. In many cases, you can’t find a convincing explanation for something because your conclusion is wrong.

1. Do you really understand the problem?
   (a) What exactly does the input consist of?
   (b) What exactly are the desired results or output?
   (c) Can you construct an input example small enough to solve by hand? What happens when you try to solve it?

2. Will brute-force always solve your problem correctly by searching through all subsets or arrangements and picking the best one?
   (a) If so, why are you sure that this algorithm always gives the correct answer?
   (b) How fast is your brute-force solution?

3. Are there special cases of the problem that you know how to solve?
   (a) Can you solve the problem efficiently when you ignore some of the input parameters?
   (b) Does the problem become easier to solve when you set some of the input parameters to trivial values, such as 0 or 1?
   (c) Can you can simplify the problem to the point where you can solve it efficiently?
   (c) Why can’t this special-case algorithm be generalized to a wider class of inputs?

4. Which of the standard algorithm design paradigms are most relevant to your problems?
   (a) Is there a set of items that can be sorted by size or some key? Does this sorted order make it easier to find the answer?

\(^1\)Actually, for the purpose of our class, here I only give a simplified version of Skiena’s questions.
(b) Is there a way to split the problem in two smaller problems, perhaps by doing a binary search? How about partitioning the elements into big and small, or left and right? Does this suggest a divide-and-conquer algorithm?

(c) Do the input objects or desired solution have a natural left-to-right order, such as characters in a string, elements of a permutation, or leaves of a tree? Can you use dynamic programming to exploit this order?

(d) Are there certain operations being done repeatedly, such as searching, or finding the largest (or smallest) element? Can you use a data structure to speed up these queries? What about a hash table or a heap?

1.2 An Example: Sum of Two Elements

To illustrate how to carry out the attack plan above, let’s consider the problem below.

**Question:** You are given a number \( T \) and an array \( A \) of \( n \) numbers. Find two elements in the array \( A \) such that their sum is \( T \).

**The discovery dialog.** Below is a dialog between a teacher and a student. The teacher tries to help the student to discover a solution of the problem above by asking him questions from the attack plan.

1. **Teacher:** Could you come up with a brute-force algorithm? How fast is it?
   **Student:** I can go over every pair \((i, j)\) to check if \( A[i] + A[j] = T \). There are \( O(N^2) \) such pairs, where \( N \) is the size of \( A \). So my running time is \( O(N^2) \).

2. **Teacher:** Here you’re viewing the problem as searching two elements of the array simultaneously. Could you rewrite the brute-force algorithm so that it searches for a single item at a time? For example, when you need to deal with two variables, you can fix one variable and handle the other.
   **Student:** For each item \( A[i] \), we need to find if there is \( A[j] = T - A[i] \). It takes \( O(N) \) time to go over the entire array to find if there is a matching item \( T - A[i] \). Totally we need \( O(N^2) \) time.

3. **Teacher:** We aim to reduce this to \( O(N) \) time. We can first try to go to \( O(N \log(N)) \) time. Is there a set of items that can be sorted by size or some key?
   **Student:** If we can tolerate \( O(N \log(N)) \) time then the obvious approach is to sort the array. Then the brute-force algorithm above now only needs \( O(N \log(N)) \) time. In particular, for each item, we only need \( O(\log(N)) \) time, via binary search, to find if there is a matching one.

4. **Teacher:** Now let’s aim for \( O(N) \) time, so sorting is not an option anymore. Let’s go back to the brute-force solution. Are there certain operations being done repeatedly, such as searching, or finding the largest (or smallest) element? Can you use a data structure to speed up these queries? What about a hash table or a heap?
   **Student:** In this case, the searching part is done repeatedly, so maybe hashing can help. Let’s say we store the elements of the array into a hash table. This takes \( O(N) \) time. Now the brute-force algorithm can be improved to \( O(N) \) time, because for each element, it takes us \( O(1) \) time to find if there is a matching element.

**Reflection.** A crucial step in solving the problem above is to realize that we need to search for \( A[j] = T - A[i] \), instead of looking for a pair whose sum is \( T \). The former view is a conventional problem that we
can rely on data structure to improve the running time. In contrast, the latter view is unorthodox and we don’t know how to deal with it.

You may think that the intermediate solution of sorting is redundant, but it is not. Its role is to help us to realize that there’s a repeated operation (namely searching), so that we can come up with a hash table to reduce the running time.

1.3 General Advice

**Visualization.** Suppose that you now have a (possibly brute-force) algorithm for your problem, but you are not happy with its asymptotic performance. It’s important to identify the bottlenecks of your current approach so that you can focus on improving it. It usually helps if you can somehow visualize the execution of your algorithm on a small (worst-case) example; our brains are much better with pictures than with abstract formulas.

**Specialization.** To find attack ideas for a problem, it is often useful to look at some special cases. Many people however don’t know how to select good special cases; they instead look over countless of concrete instances, hoping they can realize something from that. However, this approach is often ineffective. In the next section, you’ll find an illustrative example for identifying good special cases.

**Reflection.** Once you obtain a solution for the given problem, you should re-consider and re-examine the result and the path that led to it. By doing so, you should consolidate your knowledge and develop an ability to solve problems. For example, even if your solution appears to be correct, errors are always possible, and thus you should verify the correctness of your algorithm. If there is a quick and intuitive way to check the correctness, it should not be overlooked.

**Verifying your solution.** A simple way to show that an algorithm is incorrect is to produce a counter-example. Here are some guidelines from Skiena for finding counter-examples:

- **Think small:** When algorithm fails, there is usually a very simple example on which they fail. Amateur algorists tend to draw a big messy instance and then stare at it helplessly. The pros look carefully at several small examples, because they are easier to verify and reason about.

- **Think exhaustively:** There are only a small number of possibilities for the smallest nontrivial value of $n$, where $n$ is the size of your problem.

- **Hunt for the weakness:** If a proposed algorithm is of the form “always take the biggest” (better known as the greedy algorithm), think about why that might prove to be the wrong thing to do.

- **Go for a tie:** A devious way to break a greedy heuristic is to provide instances where everything is the same size. Suddenly the heuristic has nothing to base its decision on, and perhaps has the freedom to return something suboptimal as the answer.

- **Seek extremes:** Bad counter-examples are mixtures of huge and tiny, left and right, few and many, near and far. It is usually easier to verify or reason about extreme examples than more muddled ones.
1.4 Hunting for Good Special Cases: An Example

**Question:** You are given a sorted array of distinct integers $A[1 : n]$. Give an $O(\log(n))$ algorithm to determine whether there exists an $i$ index such that $A[i] = i$.

**The Discovery Dialog.** Below is again another dialog between a teacher and a student for solving the problem above.

1. **Teacher:** What exactly does the input consist of?
   **Student:** A sorted array $A$. The elements are distinct, and they are integers.

2. **Teacher:** Is there a way to split the problem in two smaller problems, perhaps by doing a binary search? How about partitioning the elements into big and small, or left and right? Does this suggest a divide-and-conquer algorithm?
   **Student:** Yes, I think we should do some sort of binary search, given that we have only $O(\log(n))$ time, and the array is sorted.

3. **Teacher:** Any idea how to carry a binary-search variant here?
   **Student:** I think I’ll need to look at the middle element, and then decide whether I should look into the left subarray, or the right subarray. But I don’t know how to proceed further.

4. **Teacher:** Let’s examine a small example to learn the idea. Consider $A = [-3, -1, 2, 4, 9]$. What’s the middle element, and will you need to search left or right?
   **Student:** In this case the middle element $A[3] = 2$. The index we want is 4. We should go right.

5. **Teacher:** Let’s try to generalize this a bit. It seems that if we have $A[* , *, 2 , *, *]$ then upon checking the middle element $A[3] = 2$, we should always turn right? Can we find a counter-example such that this is false? In other words, can we set $A[1] = 1$ or $A[2] = 2$?
   **Student:** No, given that the elements are distinct and $A[3] = 2$, we can’t have $A[2] = 2$ anymore. In addition, if $A[1] = 1$ and $A[3] = 2$, there is no possible value for $A[2]$ because the array is strictly increasing and its elements are integers.

6. **Teacher:** So we can conclude that for an array of size 5, if the middle element is 2 then we must turn right. Let’s generalize this even further, when the middle element $A[mid] = X$. Identify the constraint on $X$ so that we can’t set $A[1] = 1$ or $A[2] = 2$. In other words, if we can pick $A[1] = 1$ or $A[2] = 2$ then what would imply on the values of $X$?

7. **Teacher:** Let’s now consider an array of size $n$. Let the middle element $A[mid] = X$.
   **Student:** So here we want to find the constraint on $X$ so that we can’t set $A[1] = 1$, or $A[2] = 2, \ldots$, or $A[mid - 1] = mid - 1$. But it looks rather complicated, because there are too many cases to consider.

8. **Teacher:** Could you reduce the number of cases?

9. **Teacher:** Identify the constraint on $X$ so that we can’t set $A[1] = 1$.

10. **Teacher**: Do you have a plan to attack the problem?

**Student**: Yes, I’ll compare \[A[mid]\] with mid. If \[A[mid] = mid\] then we’re done. If \[A[mid] < mid\] then we’ll recurse on the right. Otherwise we’ll recurse on the left.

**Reflection**. Let’s re-examine how we pick the special cases here. We start with a concrete array of size 5, which is small enough for a careful analysis, and big enough to be nontrivial. We then look at its generalized versions \[[*,*,2,*,*]\] and \[[*,*,X,*,*]\]. These are complex enough to reveal the key ideas of the algorithm, but still simple enough so that analysis is easy. Then, when we look at the general case, we encounter a complex condition, namely \[A[1] = 1\], or \[A[2] = 2\]. . . When you have a complex problem statement, you should try to simplify it by seeking a different interpretation, or by rephrasing the problem. In this case we realize that we only need to consider a single case \[A[1] = 1\], which makes our proof substantially simpler.

### 1.5 Yet Another Example: Missing Element

**Question**: Suppose that you are given a sorted array \(A[1 : n]\). The elements of \(A\) are distinct, and drawn from \(\{1, \ldots, m\}\), with \(m > n\). Given an \(O(\log(n))\) algorithm to find an integer \(x \in \{1, \ldots, m\}\) that is not present in \(A\).

**The Discovery Dialog**. Below is a dialog between a teacher and a student for solving the problem above.

1. **Teacher**: What exactly does the input consist of?

   **Student**: A sorted array \(A\). The elements are distinct, and they are integers within \(\{1, \ldots, m\}\), with \(m > n\).

2. **Teacher**: What is the output?

   **Student**: An integer \(x \in \{1, \ldots, m\}\) that is not present in \(A\).

3. **Teacher**: Does that \(x\) exist?

   **Student**: Yes, there are only \(n\) elements in \(A\), so there will be \(m - n\) missing elements in the range \(\{1, \ldots, m\}\).

4. **Teacher**: Can you come up with a brute-force algorithm?

   **Student**: Yes. I can create another array \(B[1 : m]\) whose elements are 0. I then make a linear scan over \(A\). For each \(A[i]\), I let \(v = A[i]\) and set \(B[v] = 1\). Finally, I make a linear scan over \(B\) to find an element \(B[j] = 0\) and output \(j\). My running time is \(O(m)\).

5. **Teacher**: Let’s improve it to \(O(n)\). In addition, don’t use any additional data structure. Look at a concrete example, say \(m = 5\) and \(A = [1, 2, 4, 5]\). The missing element is 3, but how would you find that?

   **Student**: I can make a linear scan over \(A\) to find some “jump” in consecutive elements. In particular, if there is an index \(i\) such that \(A[i + 1] \neq A[i] + 1\) then I’ll output \(A[i] + 1\).

6. **Teacher**: Could you somehow visualize the ideas in your algorithm?

   **Student**: Yes. See Figure 1.1.

7. **Teacher**: What if you have something like \(A = [1, 2, 3, 4]\) with \(m = 5\)? There’s no jump in this case.

   **Student**: In that case either 1 or \(m\) is missing. If \(A[1] \neq 1\) then \(x = 1\). Otherwise \(m\) is missing.
8. **Teacher**: Now let’s improve the running time to $O(\log(n))$. Is there a way to split the problem into two smaller problems, perhaps by doing a binary search? How about partitioning the elements into big and small, or left and right? Does this suggest a divide-and-conquer algorithm?

**Student**: Yes, I think we should do some sort of binary search, given that we have only $O(\log(n))$ time, and the array is sorted.

9. **Teacher**: Any idea how to carry a binary-search variant here?

**Student**: I think I’ll need to look at the middle element, and then decide whether I should look into the left subarray, or the right subarray. But I don’t know how to proceed further.

10. **Teacher**: Explicitly state your obstacle. You can’t hope to bypass it if you don’t know what it is.

**Student**: Normally for binary search variants, we search for something in the array. But here we search for something is not in the array.

11. **Teacher**: What about your linear-scan algorithm? What does it search?

**Student**: It looks for a “jump” in the consecutive elements. So for our binary search variant, we also need to search for such a jump.

12. **Teacher**: Look at a concrete example, say $m = 6$ and $A = [1, 2, 4, 5, 6]$. The middle element is $A[3] = 4$. The missing element is 3. Would you go left or right?

**Student**: We should go left, since the jump is on the left, between $A[2]$ and $A[3]$.

13. **Teacher**: Let’s generalize this further. Let’s say $m = 6$ and $A = [\ast, \ast, 4, \ast, \ast]$. It seems that we should search the left subarray. But what if we have something like $A[2, 3, 4, \ast, \ast]$? There’s no jump on the left subarray.

**Student**: In that case, the missing element is at the boundary. That is, if $A[1] \neq 1$ then 1 is the missing element.

14. **Teacher**: So for $A[1, \ast, 4, \ast, \ast]$, there is certainly a jump on the left subarray. Why is it so?


15. **Teacher**: Let’s now generalize the problem further. Consider $A[1, \ast, X, \ast, \ast]$. What should be the value of $A[3] = X$ so that we should search the left subarray?

**Student**: If there is no jump then $X$ is at most 3. So if $X \geq 4$ then we should search the left subarray, because there’s certainly a jump there.
16. **Teacher**: Generalize this to an array of \( n \) elements. The middle element \( A[\text{mid}] = X \), and \( A[1] = 1 \). Identify the constraint on \( X \) so that there is a jump in the left subarray.

**Student**: In this case if \( A[\text{mid}] \geq \text{mid} + 1 \) then we should search the left subarray, because there’s certainly a jump there.

17. **Teacher**: Do you have a plan to attack the problem?

**Student**: Yes, I’ll compare \( A[\text{mid}] \) with \( \text{mid} \). If \( A[\text{mid}] > \text{mid} \) then we should search the left subarray. Otherwise we’ll recurse on the right.

**Reflection**. When you are given a search problem, you should always ask if what you search even exists. Sometimes the answer is obvious, but if it is not then answering this question will help you understand the problem better. In this case, by realizing this gap between the range \( m \) and the number \( n \) of elements implicitly helps the student to realize later that (i) he should first look at the boundary \( A[1] \) and \( A[n] \), and (ii) if \( A[1] = 1 \) and \( A[n] = m \) then he should look for a jump between two adjacent elements.

In this problem, it’s important to state the issue of the problem explicitly (namely searching something that is *not* in the array) so that you can understand why you’re stuck, and find ways to bypass it. Additionally, it’s good to have a visualization of the linear algorithm so that you can easily realize that you need to search for a jump, and therefore solve the issue above.