CIS 4930 – Problem Solving **Spring 2026**

Scribe 1: How to Design (Basic) Algorithms

Instructor: Viet Tung Hoang

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?
   (d) Are you trying to solve a numerical problem? A graph algorithm problem? A geometric problem? A string problem? A set problem? Which formulation seems easiest?
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.

Exercise 1: A sequence of $n > 0$ integers is called a *jolly jumper* if the absolute values of the difference between successive elements take on all the values 1 through $n - 1$. For instance,

1 4 2 3

is a jolly jumper, because the absolute differences are $3, 2$, and 1 respectively. The definition implies that any sequence of a single integer is a jolly jumper. Design an algorithm to determine or not a given sequence is a jolly jumper.

Exercise 2: Give an efficient algorithm for computing the mode (the value that occurs most frequently) of a sequence of N integers.

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 $A[3]$ is 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?

   Student: If $A[2] = 2$ then of course $X \geq 3$. Likewise, if $A[1] = 1$ then $A[2] \geq 2$, and thus $X \geq 3$. So if $X < 3$ then we won't be able to set $A[1] = 1$ or $A[2] = 2$.

7. *Teacher*: Let's now consider an array of size n. Let the middle element $A[\text{mid}] = X$.

   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 $A[\text{mid} - 1] = \text{mid} - 1$. But it looks rather complicated, because there are too many cases to consider.

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

   Student: I think it's enough to consider the case $A[1] = 1$, because if we can set, say $A[3] = 3$ then we can also set $A[1] = 1$ and $A[2] = 2$ without affecting the choice of X.

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

   Student: In this case if $A[1] = 1$ then $A[2] \geq 2$, $A[3] \geq 3$, and so on, until $A[\text{mid}] \geq \text{mid}$. So if $X < \text{mid}$ then we won't be able to set $A[1] = 1$.

10. *Teacher*: Do you have a plan to attack the problem? Let's say you have to write the code for $\textsc{Search}(A, \text{low}, \text{high})$.

    Student: Yes, for the base case $\text{high} \leq \text{low} + 1$, I'll use a brute-force search. For the recursive case, I'll compare $A[\text{mid}]$ with $\text{mid} = \lfloor(\text{low} + \text{high})/2\rfloor$. If $A[\text{mid}] = \text{mid}$ then we're done. If $A[\text{mid}] < \text{mid}$ 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 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.

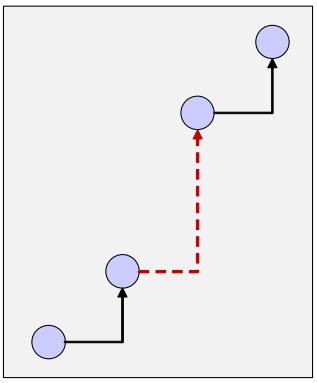

Figure 1.1: Illustration of the linear algorithm for identifying the missing element. The algorithm makes a linear scan and looks for a jump (drawn in dashed lines) between two adjacent elements.

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 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.

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 $A = [*, *, 4, *, *]$. It seems that we should search the left subarray. But what if we have something like $A[2, 3, 4, *, *]$? 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 $A[1, *, 4, *, *]$, there is certainly a jump on the left subarray. Why is it so?

    Student: Because from $A[1]$ to $A[3]$, there are only two steps $A[1] \to A[2] \to A[3]$. If at each step there is no jump then $A[3]$ is at most $A[1] + 2 = 3$. Since $A[3] = 4$, there must be some jump.

15. *Teacher*: Let's now generalize the problem further. Consider $A[1, *, X, *, *]$. 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? Let's say you have to write the code for $\textsc{Search}(A, \text{low}, \text{high})$.

    Student: Yes, I'll compare $A[\text{mid}]$ with $\text{mid} = \lfloor(\text{low} + \text{high})/2\rfloor$. If $A[\text{mid}] > \text{mid}$ then we should search the left subarray. Otherwise we'll recurse on the right.

18. *Teacher*: Test your algorithm for $A = [1, 2, 3, 4, 6, 7, 8, 9, 11]$, and $\text{low} = 5$ and $\text{high} = 9$.

    Student: Here $\text{mid} = 7$, so $A[\text{mid}] = 8 > \text{mid}$. So we'll recurse on the left-subarray $A[5 : 7] = [6, 7, 8]$. But this subarray has no jump, although the initial array $A[5 : 9]$ does contain a jump between $A[8]$ and $A[9]$. So there's something wrong, but I don't know why.

19. *Teacher*: Implicitly in the checking $A[\text{mid}] > \text{mid}$, you are assuming that $A[\text{low}] \leq \text{low}$, so that there must be a jump on the left subarrray. However here $A[\text{low}] = 6 > \text{low} = 5$, violating your assumption.

    I have to admit that the counter-example above is a cheat. Your algorithm actually *works* if you run it on $A[1 : 9]$. You will narrow down the search as follows: $A[1 : 9] \to A[1 : 5] \to A[3 : 5] \to A[4 : 5]$, and you'll find the jump from $A[4]$ to $A[5]$.

    In order to show that your algorithm does work, you should explicitly show that it has the invariant $A[\text{low}] \leq \text{low}$. This means that you'll never encounter the counter-example above, because it violates the invariant. You need to show that your algorithm maintains the invariant when you narrow down the search.

    Student: If $A[\text{mid}] > \text{mid}$ then I'll search the left subarray $A[\text{low} : \text{high}]$ and the invariant continues to hold. If $A[\text{mid}] \leq \text{mid}$ then I'll search the right subarray $A[\text{mid} : \text{high}]$, and the invariant still holds for this subarray.

20. *Teacher*: An alternative approach is to refine your algorithm so that it works for any $A[\text{low} : \text{high}]$, without assuming that $A[\text{low}] \leq \text{low}$.

    Student: I'll compare $A[\text{mid}]$ with $A[\text{low}] + (\text{mid} - \text{low})$, instead of mid. If $A[\text{mid}] > A[\text{low}] + (\text{mid} - \text{low})$, there must be a jump since there are $\text{mid} - \text{low}$ steps, but the distance is $A[\text{mid}] - A[\text{low}] > \text{mid} - \text{low}$.

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.

Exercise : You are given an (unsorted) array $A[1 : n]$ of distinct elements. A local minimum of the array is an element that are smaller than the adjacent elements. (Still, boundary elements $A[1]$ and $A[n]$ only need to be smaller than the only adjacent element.) Find a local minimum using $O(\log(n))$ time; if there are many local minimum, you only need to return one.

1.6 More On Binary Search

1.6.1 The Bisection Method

Suppose that we we need to find a root for the equation $f(x) = 0$ in the interval $[a, b]$. Suppose that f is increasing in this interval, with $f(a) < 0$ and $f(b) > 0$. In this case, there is only a root for the equation $f(x) = 0$; see Figure 1.2 for an illustration. To find this root, we will perform a binary search in the interval $[a, b]$. We start with comparing $f(\text{mid})$ with 0, where $\text{mid} = (a + b)/2$. If $f(\text{mid}) = 0$ then we are done. If $f(\text{mid}) > 0$ then we need to search the left interval $[a, \text{mid}]$. Otherwise we need to search the right interval $[\text{mid}, b]$.

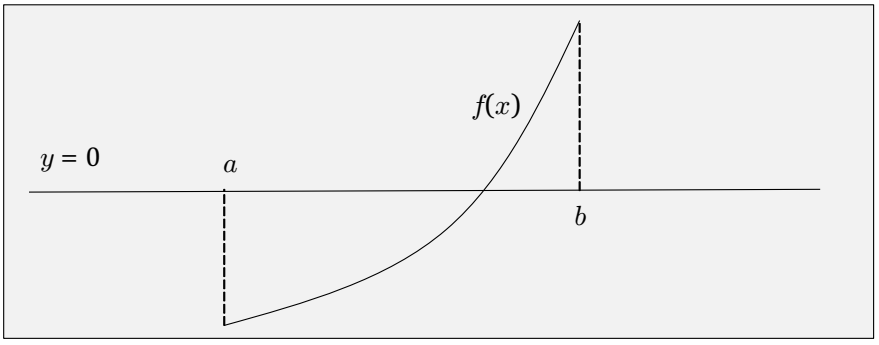

Figure 1.2: An increasing function $f(x)$ in the interval $[a, b]$, with $f(a) < 0$ and $f(b) > 0$.

The base case for the binary search above is when the search interval is smaller than the error level, say 0.01 if you want two-digit precision. In that case, we can output the midpoint in the search range.

Exercise: A narrow street is lined with tall buildings. An x-foot long ladder is rested at the base of the building on the right side of the street and leans on the building on the left side. A y-foot long ladder is rested at the base of the building on the left side of the street and leans on the building on the right side. The point where the two ladders cross is exactly c feet from the ground. See Figure 1.3 for an illustration. How wide is the street?

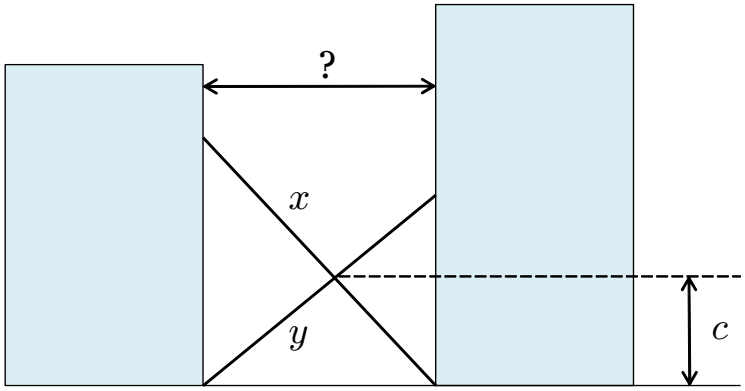

Figure 1.3: Illustration for the ladder problem

1.6.2 General Binary Search

For a general binary search, you are given a set of N items, and need to find one particular item among them. The algorithm would need to narrow down the search range within a fraction of the set, say N/c items where $c > 1$ is a constant, and then continue recursively. (See Figure 1.4 for an illustration.) In practice it's often the case that $c = 2$.

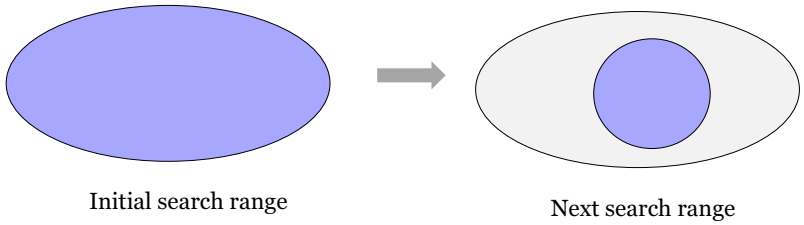

Figure 1.4: Illustration of the general binary search.

An Example: Coin Weighing. You are given $N \geq 3$ balls, all equal in weight except for one that is either heavier or lighter. You are also given a two-pan balance to use. Your task is to design a strategy to determine which is the odd ball and whether it is heavier or lighter than other balls using $O(\log(N))$ uses of the balance.

The discovery dialog. Below is a dialog between a teacher and a student for solving this problem.

1. *Teacher*: Can you construct an input example small enough to solve by hand? What happens when you try to solve it?

   Student: I can solve $N = 3$ easily. I'll put one ball on the left pan, another on the right. If balanced, the odd ball is the remaining one. If not, the remaining ball is an ordinary ball. I can then compare it with, say the ball on the left pan.

2. *Teacher*: Now, let's consider the general problem. What will you do for the *first* balance?

   Student: It looks like I need to put some balls on the left pan, and some other balls on the right one. But how many? I have no idea.

3. *Teacher*: Here's an an approach from Rene Descartes for problem solving:

   - Reduce your problem to a problem of algebra.
   - Then, reduce the problem of algebra to the solution of a single equation.

   Descartes wanted to use this approach to find a universal method to solve all mathematical problems, but his project ultimately failed. Yet his advice can be useful in many cases. Let's try to apply his advice for your case. First, is there some unknown you're looking for?

   Student: Yes, I'm looking for the number of balls I should put on each pan in the first weighing.

4. *Teacher*: Let's call this number B, meaning that you put B balls on the left pan, and other B balls on the right pan. We'll find an equation for B later, but let's now proceed with a general B. Assume that the scale is balanced, what would you do next?

   Student: It means that the odd ball is within the remaining $N - 2B$ balls. So in the next balance, I should pick balls within this pile, and ....

5. *Teacher*: Resist the temptation to jump into the details. It'd be very difficult to do that with a general parameter B. Let's reread the problem statement more carefully. Here you're supposed to solve the problem within $O(\log(N))$ time. What does it remind you?

   Student: It sounds like we need to perform a general binary search here. This algorithm has two steps: (1) narrowing down the search range, and (2) recursively calling the algorithm. I already successfully narrow down the search range from N balls to $N - 2B$ balls. It means now I should call the recursion to magically take care of those $N - 2B$ balls.

6. *Teacher*: What if the first weighing is *not* balanced? For concreteness, consider $N = 12$ and $B = 3$.

   Student: In that case it means that the remaining 6 balls are ordinary balls. I can weigh three of them against the balls on the left pan. In any case, I can locate the odd ball within three balls, and recursively run the algorithm.

7. *Teacher*: Describe your algorithm for general N and B when the first weighing is not balanced.

   Student: This means that the remaining $N - 2B$ balls are ordinary balls. I can weigh the B balls on the left against B ordinary balls on the remaining pile. In any case, I can locate the odd ball within B balls, and recursively run the algorithm.

8. *Teacher*: Any requirement on the number B?

   Student: I need $B \leq N - 2B$ so that I can pick B balls from the remaining pile of $N - 2B$ balls. This means that $B \leq N/3$.

9. *Teacher*: Any intuition on what's the best choice of B?

   Student: It looks like the bigger B is, the faster the algorithm is, because after the first weighing, we need to recursively solve the problem of size up to $\max\{B, N - 2B\} = N - 2B$.

10. *Teacher*: Back to Descartes' advice. Do you now have an equation for B?

    Student: I have a constraint that $1 \leq B \leq N/3$. I also need to find the biggest integer B within that range. So $B = \lfloor N/3 \rfloor$.

11. *Teacher*: What's the asymptotic cost?

    Student: This is like binary search but better because after $O(1)$ operations, you can narrow down the odd ball to pile of size $N/3$. So the cost is of order $\log_3(N)$. Thus the running time is $\Theta(\log(N))$.

Exercise: You are given two sorted arrays, each of size N. Find the median of all elements within $O(\log(N))$ time.

1.6.3 Optimization via Searching

While binary search is typically used in searching problems, it can be a powerful tool for optimization problems. To see how to do this, we begin with an example.

An illustrative example: Book copying. You have m books (numbered $1, 2, \ldots, m$) that may have different number of pages $(p_1, \ldots, p_m)$, and you want to make one copy of each of them. You task is to divide these books among k scribes, $k \leq m$. Each book can be assigned to a single scribe only, and every scribe must get a *continuous* sequence of books. That means, there exists an increasing succession of numbers $0 = b_0 < b_1 < \cdots < b_{k-1} \leq b_k = m$ such that the i-th scribe gets a sequence of books with numbers between $b_{i-1} + 1$ and b_i. The time needed to make a copy of all the books is determined by the scribe who was assigned the most work. Therefore, our goal is to minimize the maximum number of pages assigned to a single scribe. Your task is to find the optimal assignment.

Searching the solution. To illustrate the ideas of the solution, let's consider the concrete case $k = 3$ and $k = 5$, and $p_1 = 300, p_2 = 200, p_3 = 400, p_4 = 400$, and $p_5 = 500$.

When we have a hard problem, we should always seek easier related problems as stepping stones. Our problem consists of two subproblems:

(1) Find the minimum value X of the heaviest workload of the scribes.

(2) Given a number L, find a way to assign the load so that everybody reads at most L pages.

A way to relax the original problem is to drop the first task. To understand the subproblem (2), first consider $L = 1000$. The obvious solution is a greedy strategy: assign as much load to the first scribe as possible, as long as it is at most 1000 pages, because it'll reduce the load for the other scribes. So we'll assign the first three books to the first scribe. Again, for the remaining two books, we'll employ the same greedy strategy, assigning as much load to the second scribe as possible, as long as it's at most 1000 pages. So we'll assign the last two books to the second scribe, and there's no book for the third one. (Additionally, this special case also informs us that $X \leq 1000$, since there's a feasible assignment with maximum load 1000.)

Next, consider $L = 700$. Again, we should assign as much load to the first scribe as possible, as long as it is at most 700 pages. So we'll assign the first two books to the first scribe. Again, for the remaining three books, we'll assign as much load to the second scribe as possible, as long as it's at most 700 pages. This means that we'll assign the third book to the second scribe. But even in that case, the load for the third scribe still exceeds 700 pages. This suggests that it's impossible to find an assignment for $L = 700$. (Additional, this special case also informs us that $X > 700$.)

The two examples above gives us a greedy algorithm for solving the subproblem (2). The total running time is $O(n + k)$. As $k \leq n$, it is simply $O(n)$. More importantly, by running (2) on a given choice L, we can tell if X is smaller than L, meaning that we can binary search for X in the range $[1, 2, \ldots, P]$, where $P = p_1 + \cdots + p_m$. This costs us at most $O(\log_2(P))$ iterations, and in each iteration, we have to use $O(n)$ time for solving (2) with the corresponding L. Once we find X, we can solve the original problem by running (2) on $L = X$.

Reflection. In the example above, we are given an *optimization* problem: find an assignment to minimize the heaviest workload. We instead consider an easier variant: given a number L, check if it's possible to assign such that the heaviest load is at most L. This variant is known as the *decision* version of the original one. The optimization problem can be solved by combining the decision version with a binary search. Below are some exercises to practice this technique.

Exercise 1: One day, the residents of Main Street got together and decided that they would install wireless internet on their street, with coverage for every house. Now they need your help to decide where they should place the wireless access points. They would like to have as strong a signal as possible in every house, but they have only a limited budget for purchasing n access points. They would like to place the available access points so that the maximum distance between any house and the access point closest to it is as small as possible.

Main Street is a perfectly straight road. You are given a (sorted) array $A[1:m]$ in which $A[i]$ indicates the distance from the i-th house in the Main Street to the beginning of the street. Design an efficient algorithm to minimize the maximum distance between any house and the access point nearest to it.

For example, suppose that there are $m = 5$ houses with $A[1] = 20$, $A[2] = 100$, $A[3] = 200$, $A[4] = 500$, and $A[5] = 900$. Suppose that there are $n = 2$ access points. Let d_i denote the distance from house i to its nearest access point, and $D = \max\{d_1, \ldots, d_m\}$. Each placement gives rises to a different value D. For example, if we place the access points at locations 150 and 600 then $D = \max\{130, 50, 50, 100, 300\} = 300$. This is however not the best placement. For example, if we place the access points at locations 250 and 650 then $D = \max\{230, 150, 50, 250, 250\} = 250$. Your job is to find the placement that minimizes D.

Exercise 2: You are given n jewels, each with its cost c_i and weight w_i. Let $k \leq n$ be an integers. Find exactly k jewels with the maximum ratio $(\sum c_i)/(\sum w_i)$.

For example, suppose there are $n = 4$ jewels, with $(c_1, w_1) = (10, 5), (c_2, w_2) = (20, 4), (c_3, w_3) = (15, 2)$ and $(c_4, w_4) = (50, 5)$. There are many ways to pick $k = 2$ jewels out of those four. For example, if we pick the first and the fourth jewels then the corresponding cost-weight ratio is $(10 + 50)/(5 + 5) = 6$. If we instead pick the second and the fourth jewels then the cost-weight ratio is bigger, $(20 + 50)/(4 + 5) \approx 7.77$. Your job is to pick k jewels to maximize the cost-weight ratio.