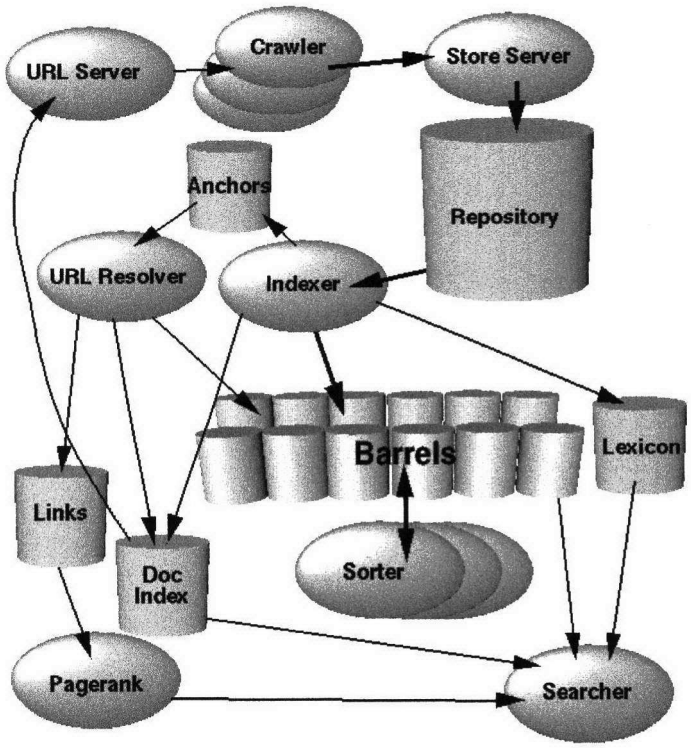
CHAPTER

11 Searching



— *"The Anatomy of A Large-Scale Hypertextual Web Search Engine,"*
S. M. BRIN AND L. PAGE, 1998

Search algorithms can be classified in a number of ways. Is the underlying collection static or dynamic, i.e., inserts and deletes are interleaved with searching? Is it worth spending the computational cost to preprocess the data so as to speed up subsequent queries? Are there statistical properties of the data that can be exploited? Should we operate directly on the data or transform it?

In this chapter, our focus is on static data stored in sorted order in an array. Data structures appropriate for dynamic updates are the subject of Chapters 10, 12, and 14.

The first collection of problems in this chapter are related to binary search. The second collection pertains to general search.

Binary search

Given an arbitrary collection of n keys, the only way to determine if a search key is present is by examining each element. This has $O(n)$ time complexity. Fundamentally, binary search is a natural elimination-based strategy for searching a sorted array. The idea is to eliminate half the keys from consideration by keeping the keys in sorted order. If the search key is not equal to the middle element of the array, one of the two sets of keys to the left and to the right of the middle element can be eliminated from further consideration.

Questions based on binary search are ideal from the interviewers perspective: it is a basic technique that every reasonable candidate is supposed to know and it can be implemented in a few lines of code. On the other hand, binary search is much trickier to implement correctly than it appears—you should implement it as well as write corner case tests to ensure you understand it properly.

Many published implementations are incorrect in subtle and not-so-subtle ways—a study reported that it is correctly implemented in only five out of twenty textbooks. Jon Bentley, in his book "*Programming Pearls*" reported that he assigned binary search in a course for professional programmers and found that 90% failed to code it correctly despite having ample time. (Bentley's students would have been gratified to know that his own published implementation of binary search, in a column titled "Writing Correct Programs", contained a bug that remained undetected for over twenty years.)

Binary search can be written in many ways—recursive, iterative, different idioms for conditionals, etc. Here is an iterative implementation adapted from Bentley's book, which includes his bug.

```
def bsearch(t, A):
    L, U = 0, len(A) - 1
    while L <= U:
        M = (L + U) // 2
        if A[M] < t:
            L = M + 1
        elif A[M] == t:
            return M
        else:
            U = M - 1
    return -1
```

The error is in the assignment `M = (L + U) / 2` in Line 4, which can potentially lead to overflow. This overflow can be avoided by using `M = L + (U - L) / 2`.

The time complexity of binary search is given by $T(n) = T(n/2) + c$, where c is a constant. This solves to $T(n) = O(\log n)$, which is far superior to the $O(n)$ approach needed when the keys are unsorted. A disadvantage of binary search is that it requires a sorted array and sorting an array takes $O(n \log n)$ time. However, if there are many searches to perform, the time taken to sort is not an issue.

Many variants of searching a sorted array require a little more thinking and create opportunities for missing corner cases.

Searching boot camp

When objects are comparable, they can be sorted and searched for using library search functions. Typically, the language knows how to compare built-in types, e.g., integers, strings, library classes for date, URLs, SQL timestamps, etc. However, user-defined types used in sorted collections must explicitly implement comparison, and ensure this comparison has basic properties such as transitivity. (If the comparison is implemented incorrectly, you may find a lookup into a sorted collection fails, even when the item is present.)

Suppose we are given as input an array of students, sorted by descending GPA, with ties broken on name. In the program below, we show how to use the library binary search routine to perform fast searches in this array. In particular, we pass binary search a custom comparator which compares students on GPA (higher GPA comes first), with ties broken on name.

```
Student = collections.namedtuple('Student', ('name', 'grade_point_average'))


def comp_gpa(student):
    return (-student.grade_point_average, student.name)


def search_student(students, target, comp_gpa):
    i = bisect.bisect_left([comp_gpa(s) for s in students], comp_gpa(target))
    return 0 <= i < len(students) and students[i] == target
```

Assuming the i-th element in the sequence can be accessed in $O(1)$ time, the time complexity of the program is $O(\log n)$.

Binary search is an effective search tool. It is applicable to more than just searching in **sorted arrays**, e.g., it can be used to search an **interval of real numbers or integers**. If your solution uses sorting, and the computation performed after sorting is faster than sorting, e.g., $O(n)$ or $O(\log n)$, **look for solutions that do not perform a complete sort**. Consider **time/space tradeoffs**, such as making multiple passes through the data.

Table 11.1: Top Tips for Searching

Know your searching libraries

The `bisect` module provides binary search functions for sorted list. Specifically, assuming a is a sorted list.

- To find the first element that is not less than a targeted value, use `bisect.bisect_left(a,x)`. This call returns the index of the first entry that is greater than or equal to the targeted value. If all elements in the list are less than `x`, the returned value is `len(a)`.
- To find the first element that is greater than a targeted value, use `bisect.bisect_right(a,x)`. This call returns the index of the first entry that is greater than the targeted value. If all elements in the list are less than or equal to `x`, the returned value is `len(a)`.

In an interview, if it is allowed, use the above functions, instead of implementing your own binary search.

11.1 Search a sorted array for first occurrence of k

Binary search commonly asks for the index of *any* element of a sorted array that is equal to a specified element. The following problem has a slight twist on this.

-14	-10	2	108	108	243	285	285	285	401
$A[0]$	$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$	$A[6]$	$A[7]$	$A[8]$	$A[9]$

Figure 11.1: A sorted array with repeated elements.

Write a method that takes a sorted array and a key and returns the index of the *first* occurrence of that key in the array. Return -1 if the key does not appear in the array. For example, when applied to the array in Figure 11.1 your algorithm should return 3 if the given key is 108; if it is 285, your algorithm should return 6.

Hint: What happens when every entry equals k? Don't stop when you first see k.

Solution: A naive approach is to use binary search to find the index of any element equal to the key, k. (If k is not present, we simply return -1.) After finding such an element, we traverse backwards from it to find the first occurrence of that element. The binary search takes time $O(\log n)$, where n is the number of entries in the array. Traversing backwards takes $O(n)$ time in the worst-case—consider the case where entries are equal to k.

The fundamental idea of binary search is to maintain a set of candidate solutions. For the current problem, if we see the element at index i equals k, although we do not know whether i is the first element equal to k, we do know that no subsequent elements can be the first one. Therefore we remove all elements with index $i + 1$ or more from the candidates.

Let's apply the above logic to the given example, with $k = 108$. We start with all indices as candidates, i.e., with $[0, 9]$. The midpoint index, 4 contains k. Therefore we can now update the candidate set to $[0, 3]$, and record 4 as an occurrence of k. The next midpoint is 1, and this index contains -10. We update the candidate set to $[2, 3]$. The value at the midpoint 2 is 2, so we update the candidate set to $[3, 3]$. Since the value at this midpoint is 108, we update the first seen occurrence of k to 3. Now the interval is $[3, 2]$, which is empty, terminating the search—the result is 3.

```
def search_first_of_k(A, k):
    left, right, result = 0, len(A) - 1, -1
    # A[left:right + 1] is the candidate set.
    while left <= right:
        mid = (left + right) // 2
        if A[mid] > k:
            right = mid - 1
        elif A[mid] == k:
            result = mid
            right = mid - 1  # Nothing to the right of mid can be solution.
        else:  # A[mid] < k.
            left = mid + 1
    return result
```

The complexity bound is still $O(\log n)$—this is because each iteration reduces the size of the candidate set by half.

Variant: Design an efficient algorithm that takes a sorted array and a key, and finds the index of the *first* occurrence of an element greater than that key. For example, when applied to the array in Figure 11.1 on the preceding page your algorithm should return 9 if the key is 285; if it is -13, your algorithm should return 1.

Variant: Let A be an unsorted array of n integers, with $A[0] \geq A[1]$ and $A[n-2] \leq A[n-1]$. Call an index i a *local minimum* if $A[i]$ is less than or equal to its neighbors. How would you efficiently find a local minimum, if one exists?

Variant: Write a program which takes a sorted array A of integers, and an integer k, and returns the interval enclosing k, i.e., the pair of integers L and U such that L is the first occurrence of k in A and U is the last occurrence of k in A. If k does not appear in A, return $[-1,-1]$. For example if $A = \langle 1,2,2,4,4,4,7,11,11,13 \rangle$ and $k = 11$, you should return $[7,8]$.

Variant: Write a program which tests if p is a prefix of a string in an array of sorted strings.

11.2 Search a sorted array for entry equal to its index

Design an efficient algorithm that takes a sorted array of distinct integers, and returns an index i such that the element at index i equals i. For example, when the input is $\langle -2,0,2,3,6,7,9 \rangle$ your algorithm should return 2 or 3.

Hint: Reduce this problem to ordinary binary search.

Solution: A brute-force approach is to iterate through the array, testing whether the ith entry equals i. The time complexity is $O(n)$, where n is the length of the array.

The brute-force approach does not take advantage of the fact that the array (call it A) is sorted and consists of distinct elements. In particular, note that the difference between an entry and its index increases by at least 1 as we iterate through A. Observe that if $A[j] > j$, then no entry after j can satisfy the given criterion. This is because each element in the array is at least 1 greater than the previous element. For the same reason, if $A[j] < j$, no entry before j can satisfy the given criterion.

The above observations can be directly used to create a binary search type algorithm for finding an i such that $A[i] = i$. A slightly simpler approach is to search the secondary array B whose ith entry is $A[i] - i$ for 0, which is just ordinary binary search. We do not need to actually create the secondary array, we can simply use $A[i] - i$ wherever $B[i]$ is referenced.

For the given example, the secondary array B is $\langle -2,-1,0,0,2,2,3 \rangle$. Binary search for 0 returns the desired result, i.e., either of index 2 or 3.

```
def search_entry_equal_to_its_index(A):
    left, right = 0, len(A) - 1
    while left <= right:
        mid = (left + right) // 2
        difference = A[mid] - mid
        # A[mid] == mid if and only if difference == 0.
        if difference == 0:
            return mid
        elif difference > 0:
```

```
            right = mid - 1
        else:  # difference < 0.
            left = mid + 1
    return -1
```

The time complexity is the same as that for binary search , i.e., $O(\log n)$, where n is the length of A.

Variant: Solve the same problem when A is sorted but may contain duplicates.

11.3 Search a cyclically sorted array

An array is said to be cyclically sorted if it is possible to cyclically shift its entries so that it becomes sorted. For example, the array in Figure 11.2 is cyclically sorted—a cyclic left shift by 4 leads to a sorted array.

378	478	550	631	103	203	220	234	279	368
$A[0]$	$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$	$A[6]$	$A[7]$	$A[8]$	$A[9]$

Figure 11.2: A cyclically sorted array.

Design an $O(\log n)$ algorithm for finding the position of the smallest element in a cyclically sorted array. Assume all elements are distinct. For example, for the array in Figure 11.2, your algorithm should return 4.

Hint: Use the divide and conquer principle.

Solution: A brute-force approach is to iterate through the array, comparing the running minimum with the current entry. The time complexity is $O(n)$, where n is the length of the array.

The brute-force approach does not take advantage of the special properties of the array, A. For example, for any m, if $A[m] > A[n-1]$, then the minimum value must be an index in the range $[m+1, n-1]$. Conversely, if $A[m] < A[n-1]$, then no index in the range $[m+1, n-1]$ can be the index of the minimum value. (The minimum value may be at $A[m]$.) Note that it is not possible for $A[m] = A[n-1]$, since it is given that all elements are distinct. These two observations are the basis for a binary search algorithm, described below.

```
def search_smallest(A):
    left, right = 0, len(A) - 1
    while left < right:
        mid = (left + right) // 2
        if A[mid] > A[right]:
            # Minimum must be in A[mid + 1:right + 1].
            left = mid + 1
        else:  # A[mid] < A[right].
            # Minimum cannot be in A[mid + 1:right + 1] so it must be in A[left:mid + 1].
            right = mid
    # Loop ends when left == right.
    return left
```

The time complexity is the same as that of binary search, namely $O(\log n)$.

Note that this problem cannot, in general, be solved in less than linear time when elements may be repeated. For example, if A consists of $n-1$ 1s and a single 0, that 0 cannot be detected in the worst-case without inspecting every element.

Variant: A sequence is strictly ascending if each element is greater than its predecessor. Suppose it is known that an array A consists of a strictly ascending sequence followed by a strictly descending sequence. Design an algorithm for finding the maximum element in A.

Variant: Design an $O(\log n)$ algorithm for finding the position of an element k in a cyclically sorted array of distinct elements.

11.4 Compute the integer square root

Write a program which takes a nonnegative integer and returns the largest integer whose square is less than or equal to the given integer. For example, if the input is 16, return 4; if the input is 300, return 17, since $17^2 = 289 < 300$ and $18^2 = 324 > 300$.

Hint: Look out for a corner-case.

Solution: A brute-force approach is to square each number from 1 to the key, k, stopping as soon as we exceed k. The time complexity is $O(k)$. For a 32 bit integer, this algorithm may take over one billion iterations.

Looking more carefully at the problem, it should be clear that it is wasteful to take unit-sized increments. For example, if $x^2 < k$, then no number smaller than x can be the result, and if $x^2 > k$, then no number greater than or equal to x can be the result.

This ability to eliminate large sets of possibilities is suggestive of binary search. Specifically, we can maintain an interval consisting of values whose squares are unclassified with respect to k, i.e., might be less than or greater than k.

We initialize the interval to $[0, k]$. We compare the square of $m = \lfloor (l + r)/2 \rfloor$ with k, and use the elimination rule to update the interval. If $m^2 \leq k$, we know all integers less than or equal to m have a square less than or equal to k. Therefore, we update the interval to $[m + 1, r]$. If $m^2 > k$, we know all numbers greater than or equal to m have a square greater than k, so we update the candidate interval to $[l, m - 1]$. The algorithm terminates when the interval is empty, in which case every number less than l has a square less than or equal to k and l's square is greater than k, so the result is $l - 1$.

For example, if $k = 21$, we initialize the interval to $[0, 21]$. The midpoint $m = \lfloor (0 + 21)/2 \rfloor = 10$; since $10^2 > 21$, we update the interval to $[0, 9]$. Now $m = \lfloor (0 + 9)/2 \rfloor = 4$; since $4^2 < 21$, we update the interval to $[5, 9]$. Now $m = \lfloor (5 + 8)/2 \rfloor = 7$; since $7^2 > 21$, we update the interval to $[5, 6]$. Now $m = \lfloor (5 + 6)/2 \rfloor = 5$; since $5^2 > 21$, we update the interval to $[5, 4]$. Now the right endpoint is less than the left endpoint, i.e., the interval is empty, so the result is $5 - 1 = 4$, which is the value returned.

For $k = 25$, the sequence of intervals is $[0, 25], [0, 11], [6, 11], [6, 7], [6, 5]$. The returned value is $6 - 1 = 5$.

```
def square_root(k):
    left, right = 0, k
    # Candidate interval [left, right] where everything before left has square
    # <= k, everything after right has square > k.
    while left <= right:
```

```
        mid = (left + right) // 2
        mid_squared = mid * mid
        if mid_squared <= k:
            left = mid + 1
        else:
            right = mid - 1
    return left - 1
```

The time complexity is that of binary search over the interval $[0, k]$, i.e., $O(\log k)$.

11.5 Compute the real square root

Square root computations can be implemented using sophisticated numerical techniques involving iterative methods and logarithms. However, if you were asked to implement a square root function, you would not be expected to know these techniques.

Implement a function which takes as input a floating point value and returns its square root.

Hint: Iteratively compute a sequence of intervals, each contained in the previous interval, that contain the result.

Solution: Let x be the input. One approach is to find an integer n such that $n^2 \leq x$ and $(n+1)^2 > x$, using, for example, the approach in Solution 11.4 on the preceding page. We can then search within $[n, n+1]$ to find the square root of x to any specified tolerance.

We can avoid decomposing the computation into an integer computation followed by a floating point computation by directly performing binary search. The reason is that if a number is too big to be the square root of x, then any number bigger than that number can be eliminated. Similarly, if a number is too small to be the square root of x, then any number smaller than that number can be eliminated.

Trivial choices for the initial lower bound and upper bound are 0 and the largest floating point number that is representable. The problem with this is that it does not play well with finite precision arithmetic—the first midpoint itself will overflow on squaring.

We cannot start with $[0, x]$ because the square root may be larger than x, e.g., $\sqrt{1/4} = 1/2$. However, if $x \geq 1.0$, we can tighten the lower and upper bounds to 1.0 and x, respectively, since if $1.0 \leq x$ then $x \leq x^2$. On the other hand, if $x < 1.0$, we can use x and 1.0 as the lower and upper bounds respectively, since then the square root of x is greater than x but less than 1.0. Note that the floating point square root problem differs in a fundamental way from the integer square root (Problem 11.4 on the facing page). In that problem, the initial interval containing the solution is always $[0, x]$.

```
def square_root(x):
    # Decides the search range according to x's value relative to 1.0.
    left, right = (x, 1.0) if x < 1.0 else (1.0, x)

    # Keeps searching as long as left != right.
    while not math.isclose(left, right):
        mid = 0.5 * (left + right)
        mid_squared = mid * mid
        if mid_squared > x:
            right = mid
```

```
        else:
            left = mid
    return left
```

The time complexity is $O(\log \frac{x}{s})$, where s is the tolerance.

Variant: Given two positive floating point numbers x and y, how would you compute $\frac{x}{y}$ to within a specified tolerance ϵ if the division operator cannot be used? You cannot use any library functions, such as *log* and *exp*; addition and multiplication are acceptable.

Generalized search

Now we consider a number of search problems that do not use the binary search principle. For example, they focus on tradeoffs between RAM and computation time, avoid wasted comparisons when searching for the minimum and maximum simultaneously, use randomization to perform elimination efficiently, use bit-level manipulations to identify missing elements, etc.

11.6 Search in a 2D sorted array

Call a 2*D* array sorted if its rows and its columns are nondecreasing. See Figure 11.3 for an example of a 2*D* sorted array.

	C0	*C1*	*C2*	*C3*	*C4*
R0	**-1**	**2**	**4**	**4**	**6**
R1	**1**	**5**	**5**	**9**	**21**
R2	**3**	**6**	**6**	**9**	**22**
R3	**3**	**6**	**8**	**10**	**24**
R4	**6**	**8**	**9**	**12**	**25**
R5	**8**	**10**	**12**	**13**	**40**

Figure 11.3: A 2*D* sorted array.

Design an algorithm that takes a 2D sorted array and a number and checks whether that number appears in the array. For example, if the input is the 2D sorted array in Figure 11.3, and the number is 7, your algorithm should return false; if the number is 8, your algorithm should return true.

Hint: Can you eliminate a row or a column per comparison?

Solution: Let the 2D array be A and the input number be x. We can perform binary search on each row independently, which has a time complexity $O(m \log n)$, where m is the number of rows and n is the number of columns. (If searching on columns, the time complexity is $O(n \log m)$.)

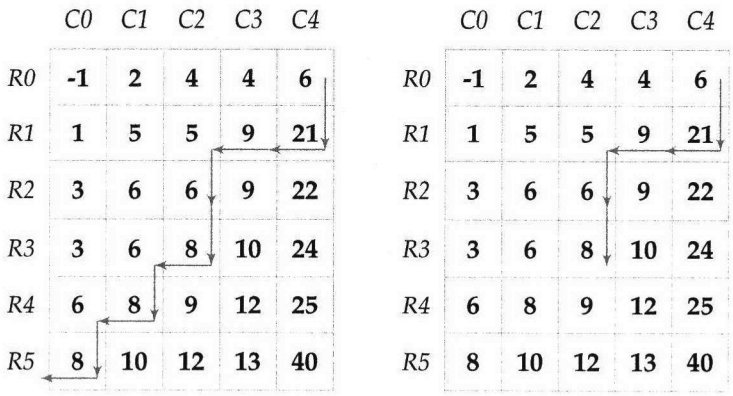
Note that the above approach fails to take advantage of the fact that both rows and columns are sorted—it treats separate rows independently of each other. For example, if $x < A[0][0]$ then no row or column can contain x—the sortedness property guarantees that $A[0][0]$ is the smallest element in A.

However, if $x > A[0][0]$, we cannot eliminate the first row or the first column of A. Searching along both rows and columns will lead to a $O(mn)$ solution, which is far worse than the previous solution. The same problem arises if $x < A[m-1][n-1]$.

A good rule of design is to look at extremal cases. We have already seen that there is nothing to be gained by comparing with $A[0][0]$ and $A[m-1][n-1]$. However, there are some more extremal cases. For example, suppose we compare x with $A[0][n-1]$. If $x = A[0][n-1]$, we have found the desired value. Otherwise:

- $x > A[0][n-1]$, in which case x is greater than all elements in Row 0.
- $x < A[0][n-1]$, in which case x is less than all elements in Column $n-1$.

In either case, we have a 2D array with one fewer row or column to search. The other extremal case, namely comparing with $A[m-1][0]$ yields a very similar algorithm.



(a) Search for 7 (not present) (b) Search for 8 (present)

Figure 11.4: Sample searches in a 2D array.

In Figure 11.4(a), we show how the algorithm proceeds when the input number is 7. It compares the top-right entry, $A[0][4] = 6$ with 7. Since $7 > 6$, we know 7 cannot be present in Row 0. Now we compare with $A[1][4] = 21$. Since $7 < 21$, we know 7 cannot be present in Column 4. Now we compare with $A[1][3] = 9$. Since $7 < 9$, we know 7 cannot be present in Column 3. Now we compare with $A[1][2] = 5$. Since $7 > 5$, we know 7 cannot be present in Row 1. Now we compare with $A[2][2] = 6$. Since $7 > 6$, we know 7 cannot be present in Row 2. Now we compare with $A[3][2] = 8$. Since $7 < 8$, we know 7 cannot be present in Column 2. Now we compare with $A[3][1] = 6$. Since $7 > 6$, we know 7 cannot be present in Row 3. Now we compare with $A[4][1] = 8$. Since $7 < 8$, we know 7 cannot be present in Column 1. Now we compare with $A[4][0] = 6$. Since $7 > 6$, we know 7 cannot be present in Row 4. Now we compare with $A[5][0] = 8$. Since $7 < 8$, we know 7 cannot be present in Column 0. There are no remaining entries, so we return false.

In Figure 11.4(b), we show how the algorithm proceeds when the input number is 8. We eliminate Row 0, then Column 4, then Column 3, then Row 1, then Row 2. When we compare with $A[3][2]$ we have a match so we return true.

```
def matrix_search(A, x):
    row, col = 0, len(A[0]) - 1  # Start from the top-right corner.
    # Keeps searching while there are unclassified rows and columns.
    while row < len(A) and col >= 0:
        if A[row][col] == x:
```

```
            return True
        elif A[row][col] < x:
            row += 1  # Eliminate this row.
        else:  # A[row][col] > x.
            col -= 1  # Eliminate this column.
    return False
```

In each iteration, we remove a row or a column, which means we inspect at most $m+n-1$ elements, yielding an $O(m+n)$ time complexity.

11.7 Find the min and max simultaneously

Given an array of comparable objects, you can find either the *min* or the *max* of the elements in the array with $n-1$ comparisons, where n is the length of the array.

Comparing elements may be expensive, e.g., a comparison may involve a number of nested calls or the elements being compared may be long strings. Therefore, it is natural to ask if both the min and the max can be computed with less than the $2(n-1)$ comparisons required to compute the min and the max independently.

Design an algorithm to find the min and max elements in an array. For example, if $A = \langle 3, 2, 5, 1, 2, 4 \rangle$, you should return 1 for the min and 5 for the max.

Hint: Use the fact that $a < b$ and $b < c$ implies $a < c$ to reduce the number of compares used by the brute-force approach.

Solution: The brute-force approach is to compute the min and the max independently, i.e., with $2(n-1)$ comparisons. We can reduce the number of comparisons by 1 by first computing the min and then skipping the comparison with it when computing the max.

One way to think of this problem is that we are searching for the strongest and weakest players in a group of players, assuming players are totally ordered. There is no point in looking at any player who won a game when we want to find the weakest player. The better approach is to play $n/2$ matches between disjoint pairs of players. The strongest player will come from the $n/2$ winners and the weakest player will come from the $n/2$ losers.

Following the above analogy, we partition the array into min candidates and max candidates by comparing successive pairs—this will give us $n/2$ candidates for min and $n/2$ candidates for max at the cost of $n/2$ comparisons. It takes $n/2-1$ comparisons to find the min from the min candidates and $n/2-1$ comparisons to find the max from the max candidates, yielding a total of $3n/2-2$ comparisons.

Naively implemented, the above algorithm need $O(n)$ storage. However, we can implement it in streaming fashion, by maintaining candidate min and max as we process successive pairs. Note that this entails three comparisons for each pair.

For the given example, we begin by comparing 3 and 2. Since $3 > 2$, we set min to 2 and max to 3. Next we compare 5 and 1. Since $5 > 1$, we compare 5 with the current max, namely 3, and update max to 5. We compare 1 with the current min, namely 2, and update min to 1. Then we compare 2 and 4. Since $4 > 2$, we compare 4 with the current max, namely 5. Since $4 < 5$, we do not update max. We compare 2 with the current min, namely 1 Since $2 > 1$, we do not update min.

```
MinMax = collections.namedtuple('MinMax', ('smallest', 'largest'))
```

```
def find_min_max(A):
    def min_max(a, b):
        return MinMax(a, b) if a < b else MinMax(b, a)

    if len(A) <= 1:
        return MinMax(A[0], A[0])

    global_min_max = min_max(A[0], A[1])
    # Process two elements at a time.
    for i in range(2, len(A) - 1, 2):
        local_min_max = min_max(A[i], A[i + 1])
        global_min_max = MinMax(
            min(global_min_max.smallest, local_min_max.smallest),
            max(global_min_max.largest, local_min_max.largest))
    # If there is odd number of elements in the array, we still need to
    # compare the last element with the existing answer.
    if len(A) % 2:
        global_min_max = MinMax(
            min(global_min_max.smallest, A[-1]),
            max(global_min_max.largest, A[-1]))
    return global_min_max
```

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

Variant: What is the least number of comparisons required to find the min and the max in the worst-case?

11.8 Find the kth largest element

Many algorithms require as a subroutine the computation of the kth largest element of an array. The first largest element is simply the largest element. The nth largest element is the smallest element, where n is the length of the array.

For example, if the input array $A = \langle 3, 2, 1, 5, 4 \rangle$, then $A[3]$ is the first largest element in A, $A[0]$ is the third largest element in A, and $A[2]$ is the fifth largest element in A.

Design an algorithm for computing the kth largest element in an array.

Hint: Use divide and conquer in conjunction with randomization.

Solution: The brute-force approach is to sort the input array A in descending order and return the element at index $k - 1$. The time complexity is $O(n \log n)$, where n is the length of A.

Sorting is wasteful, since it does more than what is required. For example, if we want the first largest element, we can compute that with a single iteration, which is $O(n)$.

For general k, we can store a candidate set of k elements in a min-heap, in a fashion analogous to Solution 10.4 on Page 138, which will yield a $O(n \log k)$ time complexity and $O(k)$ space complexity. This approach is faster than sorting but is not in-place. Additionally, it does more than what's required—it computes the k largest elements in sorted order, but all that's asked for is the kth largest element.

Conceptually, to focus on the kth largest element in-place without completely sorting the array we can select an element at random (the "pivot"), and partition the remaining entries into those

greater than the pivot and those less than the pivot. (Since the problem states all elements are distinct, there cannot be any other elements equal to the pivot.) If there are exactly $k-1$ elements greater than the pivot, the pivot must be the kth largest element. If there are more than $k-1$ elements greater than the pivot, we can discard elements less than or equal to the pivot—the k-largest element must be greater than the pivot. If there are less than $k-1$ elements greater than the pivot, we can discard elements greater than or equal to the pivot.

Intuitively, this is a good approach because on average we reduce by half the number of entries to be considered.

Implemented naively, this approach requires $O(n)$ additional memory. However, we can avoid the additional storage by using the array itself to record the partitioning.

```
# The numbering starts from one, i.e., if A = [3, 1, -1, 2]
# find_kth_largest(1, A) returns 3, find_kth_largest(2, A) returns 2,
# find_kth_largest(3, A) returns 1, and find_kth_largest(4, A) returns -1.
def find_kth_largest(k, A):
    def find_kth(comp):
        # Partition A[left:right + 1] around pivot_idx, returns the new index of
        # the pivot, new_pivot_idx, after partition. After partitioning,
        # A[left:new_pivot_idx] contains elements that are greater than the
        # pivot, and A[new_pivot_idx + 1:right + 1] contains elements that are
        # less than the pivot.
        #
        # Note: "less than" is defined by the comp object.
        #
        # Returns the new index of the pivot element after partition.
        def partition_around_pivot(left, right, pivot_idx):
            pivot_value = A[pivot_idx]
            new_pivot_idx = left
            A[pivot_idx], A[right] = A[right], A[pivot_idx]
            for i in range(left, right):
                if comp(A[i], pivot_value):
                    A[i], A[new_pivot_idx] = A[new_pivot_idx], A[i]
                    new_pivot_idx += 1
            A[right], A[new_pivot_idx] = A[new_pivot_idx], A[right]
            return new_pivot_idx

        left, right = 0, len(A) - 1
        while left <= right:
            # Generates a random integer in [left, right].
            pivot_idx = random.randint(left, right)
            new_pivot_idx = partition_around_pivot(left, right, pivot_idx)
            if new_pivot_idx == k - 1:
                return A[new_pivot_idx]
            elif new_pivot_idx > k - 1:
                right = new_pivot_idx - 1
            else:  # new_pivot_idx < k - 1.
                left = new_pivot_idx + 1

    return find_kth(operator.gt)
```

Since we expect to reduce the number of elements to process by roughly half, the average time complexity $T(n)$ satisfies $T(n) = O(n) + T(n/2)$. This solves to $T(n) = O(n)$. The space complexity is $O(1)$. The worst-case time complexity is $O(n^2)$, which occurs when the randomly selected pivot is

the smallest or largest element in the current subarray. The probability of the worst-case reduces exponentially with the length of the input array, and the worst-case is a nonissue in practice. For this reason, the randomize selection algorithm is sometimes said to have almost certain $O(n)$ time complexity.

Variant: Design an algorithm for finding the median of an array.

Variant: Design an algorithm for finding the kth largest element of A in the presence of duplicates. The kth largest element is defined to be $A[k-1]$ after A has been sorted in a stable manner, i.e., if $A[i] = A[j]$ and $i < j$ then $A[i]$ must appear before $A[j]$ after stable sorting.

Variant: A number of apartment buildings are coming up on a new street. The postal service wants to place a single mailbox on the street. Their objective is to minimize the total distance that residents have to walk to collect their mail each day. (Different buildings may have different numbers of residents.)

Devise an algorithm that computes where to place the mailbox so as to minimize the total distance, that residents travel to get to the mailbox. Assume the input is specified as an array of building objects, where each building object has a field indicating the number of residents in that building, and a field indicating the building's distance from the start of the street.

11.9 Find the missing IP address

The storage capacity of hard drives dwarfs that of RAM. This can lead to interesting space-time trade-offs.

Suppose you were given a file containing roughly one billion IP addresses, each of which is a 32-bit quantity. How would you programmatically find an IP address that is not in the file? Assume you have unlimited drive space but only a few megabytes of RAM at your disposal.

Hint: Can you be sure there is an address which is not in the file?

Solution: Since the file can be treated as consisting of 32-bit integers, we can sort the input file and then iterate through it, searching for a gap between values. The time complexity is $O(n \log n)$, where n is number of entries. Furthermore, to keep the RAM usage low, the sort will have to use disk as storage, which in practice is very slow.

Note that we cannot just compute the largest entry and add one to it, since if the largest entry is 255.255.255.255 (the highest possible IP address), adding one to it leads to overflow. The same holds for the smallest entry. (In practice this would be a good heuristic.)

We could add all the IP addresses in the file to a hash table, and then enumerate IP addresses, starting with 0.0.0.0, until we find one not in the hash table. This requires a minimum of 4 gigabytes of RAM to store the data.[1]

We can reduce the storage requirement by an order of magnitude by using a bit array representation for the set of all possible IP addresses. Specifically, we allocate an array of 2^{32} bits, initialized to 0, and write a 1 at each index that corresponds to an IP address in the file. Then we iterate through the bit array, looking for an entry set to 0. There are $2^{32} \approx 4 \times 10^9$ possible IP addresses, so

[1] A hash table has additional memory overhead, e.g., the table itself, as well as the next-fields in the collision chains, which amounts to roughly 6 gigabytes on a 32-bit machine.

not all IP addresses appear in the file. The storage is $2^{32}/8$ bytes, is half a gigabyte. This is still well in excess of the storage limit.

Since the input is in a file, we can make multiple passes through it. We can use this to narrow the search down to subsets of the space of all IP addresses as follows. We make a pass through the file to count the number of IP addresses present whose leading bit is a 1, and the number of IP addresses whose leading bit is a 0. At least one IP address must exist which is not present in the file, so at least one of these two counts is below 2^{31}. For example, suppose we have determined using counting that there must be an IP address which begins with 0 and is absent from the file. We can focus our attention on IP addresses in the file that begin with 0, and continue the process of elimination based on the second bit. This entails 32 passes, and uses only two integer-valued count variables as storage.

Since we have more storage, we can count on groups of bits. Specifically, we can count the number of IP addresses in the file that begin with $0, 1, 2, \ldots, 2^{16} - 1$ using an array of 2^{16} integers that can be represented with 32 bits. For every IP address in the file, we take its 16 MSBs to index into this array and increment the count of that number. Since the file contains fewer than 2^{32} numbers, there must be one entry in the array that is less than 2^{16}. This tells us that there is at least one IP address which has those upper bits and is not in the file. In the second pass, we can focus only on the addresses whose leading 16 bits match the one we have found, and use a bit array of size 2^{16} to identify a missing address.

```
def find_missing_element(stream):
    NUM_BUCKET = 1 << 16
    counter = [0] * NUM_BUCKET
    stream, stream_copy = itertools.tee(stream)
    for x in stream:
        upper_part_x = x >> 16
        counter[upper_part_x] += 1

    # Look for a bucket that contains less than (1 << 16) elements.
    BUCKET_CAPACITY = 1 << 16
    candidate_bucket = next(i for i, c in enumerate(counter)
                            if c < BUCKET_CAPACITY)

    # Finds all IP addresses in the stream whose first 16 bits are equal to
    # candidate_bucket.
    candidates = [0] * BUCKET_CAPACITY
    stream = stream_copy
    for x in stream_copy:
        upper_part_x = x >> 16
        if candidate_bucket == upper_part_x:
            # Records the presence of 16 LSB of x.
            lower_part_x = ((1 << 16) - 1) & x
            candidates[lower_part_x] = 1

    # At least one of the LSB combinations is absent, find it.
    for i, v in enumerate(candidates):
        if v == 0:
            return (candidate_bucket << 16) | i
```

The storage requirement is dominated by the count array, i.e., 2^{16} integer entries.

If an array contains $n-1$ integers, each between 0 and $n-1$, inclusive, and all numbers in the array are distinct, then it must be the case that exactly one number between 0 and $n-1$ is absent.

We can determine the missing number in $O(n)$ time and $O(1)$ space by computing the sum of the elements in the array. Since the sum of all the numbers from 0 to $n-1$, inclusive, is $\frac{(n-1)n}{2}$, we can subtract the sum of the numbers in the array from $\frac{(n-1)n}{2}$ to get the missing number.

For example, if the array is $\langle 5,3,0,1,2\rangle$, then $n = 6$. We subtract $(5+3+0+1+2) = 11$ from $\frac{5(6)}{2} = 15$, and the result, 4, is the missing number.

Similarly, if the array contains $n+1$ integers, each between 0 and $n-1$, inclusive, with exactly one element appearing twice, the duplicated integer will be equal to the sum of the elements of the array minus $\frac{(n-1)n}{2}$.

Alternatively, for the first problem, we can compute the missing number by computing the XOR of all the integers from 0 to $n-1$, inclusive, and XORing that with the XOR of all the elements in the array. Every element in the array, except for the missing element, cancels out with an integer from the first set. Therefore, the resulting XOR equals the missing element. The same approach works for the problem of finding the duplicated element. For example, the array $\langle 5,3,0,1,2\rangle$ represented in binary is $\langle (101)_2, (011)_2, (000)_2, (001)_2, (010)_2\rangle$. The XOR of these entries is $(101)_2$. The XOR of all numbers from 0 to 5, inclusive, is $(001)_2$. The XOR of $(101)_2$ and $(001)_2$ is $(100)_2 = 4$, which is the missing number.

We now turn to a related, though harder, problem.

You are given an array of n integers, each between 0 and $n-1$, inclusive. Exactly one element appears twice, implying that exactly one number between 0 and $n-1$ is missing from the array. How would you compute the duplicate and missing numbers?

Hint: Consider performing multiple passes through the array.

Solution: A brute-force approach is to use a hash table to store the entries in the array. The number added twice is the duplicate. After having built the hash table, we can test for the missing element by iterating through the numbers from 0 to $n-1$, inclusive, stopping when a number is not present in the hash table. The time complexity and space complexity are $O(n)$. We can improve the space complexity to $O(1)$ by sorting the array, subsequent to which finding duplicate and missing values is trivial. However, the time complexity increases to $O(n \log n)$.

We can improve on the space complexity by focusing on a collective property of the numbers in the array, rather than the individual numbers. For example, let t be the element appearing twice, and m be the missing number. The sum of the numbers from 0 to $n-1$, inclusive, is $\frac{(n-1)n}{2}$, so the sum of the elements in the array is exactly $\frac{(n-1)n}{2} + t - m$. This gives us an equation in t and m, but we need one more independent equation to solve for them.

We could use an equation for the product of the elements in the array, or for the sum of the squares of the elements in the array. This is not a good idea in practice because it results in very large integers.

The introduction to this problem showed how to find a missing number from an array of $n-2$ distinct numbers between 0 and $n-1$ using XOR. Applying the same idea to the current problem, i.e., computing the XOR of all the numbers from 0 to $n-1$, inclusive, and the entries in the array, yields $m \oplus t$. This does not seem very helpful at first glance, since we want m and t. However, since

$m \neq t$, there must be some bit in $m \oplus t$ that is set to 1, i.e., m and t differ in that bit. For example, the XOR of $(01101)_2$ and $(11100)_2$ is $(10001)_2$. The 1s in the XOR are exactly the bits where $(01101)_2$ and $(11100)_2$ differ.

This fact allows us to focus on a subset of numbers from 0 to $n-1$ where we can guarantee exactly one of m and t is present. Suppose we know m and t differ in the kth bit. We compute the XOR of the numbers from 0 to $n-1$ in which the kth bit is 1, and the entries in the array in which the kth bit is 1. Let this XOR be h—by the logic described in the problem statement, h must be one of m or t. We can make another pass through A to determine if h is the duplicate or the missing element.

For example, for the array $\langle 5,3,0,3,1,2\rangle$, the duplicate entry t is 3 and the missing entry m is 4. Represented in binary the array is $\langle (101)_2, (011)_2, (000)_2, (011)_2, (001)_2, (010)_2\rangle$. The XOR of these entries is $(110)_2$. The XOR of the numbers from 0 to 5, inclusive, is $(001)_2$. The XOR of $(110)_2$ and $(001)_2$ is $(111)_2$. This tells we can focus our attention on entries where the least significant bit is 1. We compute the XOR of all numbers between 0 and 5 in which this bit is 1, i.e., $(001)_2, (011)_2$, and $(101)_2$, and all entries in the array in which this bit is 1, i.e., $(101)_2, (011)_2, (011)_2$, and $(001)_2$. The XOR of these seven values is $(011)_2$. This implies that $(011)_2 = 3$ is either the missing or the duplicate entry. Another pass through the array shows that it is the duplicate entry. We can then find the missing entry by forming the XOR of $(011)_2$ with all entries in the array, and XORing that result with the XOR of all numbers from 0 to 5, which yields $(100)_2$, i.e., 4.

```
DuplicateAndMissing = collections.namedtuple('DuplicateAndMissing',
                                             ('duplicate', 'missing'))


def find_duplicate_missing(A):
    # Compute the XOR of all numbers from 0 to |A| - 1 and all entries in A.
    miss_XOR_dup = functools.reduce(lambda v, i: v ^ i[0] ^ i[1],
                                    enumerate(A), 0)

    # We need to find a bit that's set to 1 in miss_XOR_dup. Such a bit must
    # exist if there is a single missing number and a single duplicated number
    # in A.
    #
    # The bit-fiddling assignment below sets all of bits in differ_bit
    # to 0 except for the least significant bit in miss_XOR_dup that's 1.
    differ_bit, miss_or_dup = miss_XOR_dup & (~(miss_XOR_dup - 1)), 0
    for i, a in enumerate(A):
        # Focus on entries and numbers in which the differ_bit-th bit is 1.
        if i & differ_bit:
            miss_or_dup ^= i
        if a & differ_bit:
            miss_or_dup ^= a

    # miss_or_dup is either the missing value or the duplicated entry.
    if miss_or_dup in A:
        # miss_or_dup is the duplicate.
        return DuplicateAndMissing(miss_or_dup, miss_or_dup ^ miss_XOR_dup)
    # miss_or_dup is the missing value.
    return DuplicateAndMissing(miss_or_dup ^ miss_XOR_dup, miss_or_dup)
```

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