CHAPTER

8 Stacks and Queues

Linear lists in which insertions, deletions, and accesses to values occur almost always at the first or the last node are very frequently encountered, and we give them special names . . .

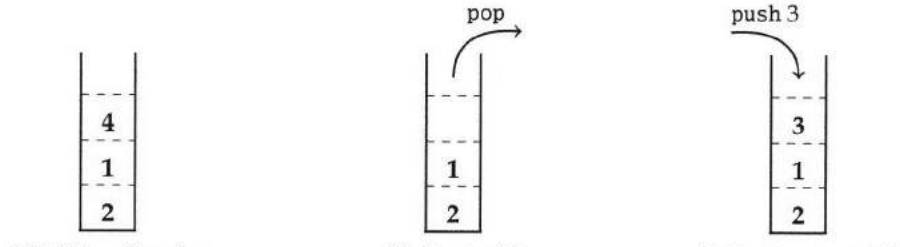
— *"The Art of Computer Programming, Volume 1,"* D. E. KNUTH, 1997

Stacks support last-in, first-out semantics for inserts and deletes, whereas queues are first-in, first-out. Stacks and queues are usually building blocks in a solution to a complex problem. As we will soon see, they can also make for stand-alone problems.

Stacks

A *stack* supports two basic operations—push and pop. Elements are added (pushed) and removed (popped) in last-in, first-out order, as shown in Figure 8.1. If the stack is empty, pop typically returns null or throws an exception.

When the stack is implemented using a linked list these operations have $O(1)$ time complexity. If it is implemented using an array, there is maximum number of entries it can have—push and pop are still $O(1)$. If the array is dynamically resized, the amortized time for both push and pop is $O(1)$. A stack can support additional operations such as peek, which returns the top of the stack without popping it.



(a) Initial configuration. (b) Popping (a). (c) Pushing 3 on to (b).

Figure 8.1: Operations on a stack.

Stacks boot camp

The last-in, first-out semantics of a stack make it very useful for creating reverse iterators for sequences which are stored in a way that would make it difficult or impossible to step back from a given element. This a program uses a stack to print the entries of a singly-linked list in reverse order.

```
def print_linked_list_in_reverse(head):
    nodes = []
    while head:
        nodes.append(head.data)
        head = head.next
    while nodes:
        print(nodes.pop())
```

The time and space complexity are $O(n)$, where n is the number of nodes in the list.

As an alternative, we could form the reverse of the list using Solution 7.2 on Page 85, iterate through the list printing entries, then perform another reverse to recover the list—this would have $O(n)$ time complexity and $O(1)$ space complexity.

Learn to recognize when the stack **LIFO** property is **applicable**. For example, **parsing** typically benefits from a stack. Consider **augmenting** the basic stack or queue data structure to support additional operations, such as finding the maximum element.

Table 8.1: Top Tips for Stacks

Know your stack libraries

Some of the problems require you to implement your own stack class; for others, use the built-in `list`-type.

- `s.append(e)` pushes an element onto the stack. Not much can go wrong with a call to push.
- `s[-1]` will retrieve, but does not remove, the element at the top of the stack.
- `s.pop()` will remove and return the element at the top of the stack.
- `len(s) == 0` tests if the stack is empty.

When called on an empty list `s`, both `s[-1]` and `s.pop()` raise an `IndexError` exception.

8.1 Implement a stack with max API

Design a stack that includes a max operation, in addition to push and pop. The max method should return the maximum value stored in the stack.

Hint: Use additional storage to track the maximum value.

Solution: The simplest way to implement a max operation is to consider each element in the stack, e.g., by iterating through the underlying array for an array-based stack. The time complexity is $O(n)$ and the space complexity is $O(1)$, where n is the number of elements currently in the stack.

The time complexity can be reduced to $O(\log n)$ using auxiliary data structures, specifically, a heap or a BST, and a hash table. The space complexity increases to $O(n)$ and the code is quite complex.

Suppose we use a single auxiliary variable, M, to record the element that is maximum in the stack. Updating M on pushes is easy: $M = \max(M, e)$, where e is the element being pushed. However, updating M on pop is very time consuming. If M is the element being popped, we have

no way of knowing what the maximum remaining element is, and are forced to consider all the remaining elements.

We can dramatically improve on the time complexity of popping by caching, in essence, trading time for space. Specifically, for each entry in the stack, we cache the maximum stored at or below that entry. Now when we pop, we evict the corresponding cached value.

```
class Stack:
    ElementWithCachedMax = collections.namedtuple('ElementWithCachedMax',
                                                   ('element', 'max'))

    def __init__(self):
        self._element_with_cached_max = []

    def empty(self):
        return len(self._element_with_cached_max) == 0

    def max(self):
        if self.empty():
            raise IndexError('max(): empty stack')
        return self._element_with_cached_max[-1].max

    def pop(self):
        if self.empty():
            raise IndexError('pop(): empty stack')
        return self._element_with_cached_max.pop().element

    def push(self, x):
        self._element_with_cached_max.append(
            self.ElementWithCachedMax(x, x if self.empty() else max(
                x, self.max())))
```

Each of the specified methods has time complexity $O(1)$. The additional space complexity is $O(n)$, regardless of the stored keys.

We can improve on the best-case space needed by observing that if an element e being pushed is smaller than the maximum element already in the stack, then e can never be the maximum, so we do not need to record it. We cannot store the sequence of maximum values in a separate stack because of the possibility of duplicates. We resolve this by additionally recording the number of occurrences of each maximum value. See Figure 8.2 on the following page for an example.

```
class Stack:
    class MaxWithCount:
        def __init__(self, max, count):
            self.max, self.count = max, count

    def __init__(self):
        self._element = []
        self._cached_max_with_count = []

    def empty(self):
        return len(self._element) == 0

    def max(self):
        if self.empty():
```

```
            raise IndexError('max(): empty stack')
        return self._cached_max_with_count[-1].max

    def pop(self):
        if self.empty():
            raise IndexError('pop(): empty stack')
        pop_element = self._element.pop()
        current_max = self._cached_max_with_count[-1].max
        if pop_element == current_max:
            self._cached_max_with_count[-1].count -= 1
            if self._cached_max_with_count[-1].count == 0:
                self._cached_max_with_count.pop()
        return pop_element

    def push(self, x):
        self._element.append(x)
        if len(self._cached_max_with_count) == 0:
            self._cached_max_with_count.append(self.MaxWithCount(x, 1))
        else:
            current_max = self._cached_max_with_count[-1].max
            if x == current_max:
                self._cached_max_with_count[-1].count += 1
            elif x > current_max:
                self._cached_max_with_count.append(self.MaxWithCount(x, 1))
```

The worst-case additional space complexity is $O(n)$, which occurs when each key pushed is greater than all keys in the primary stack. However, when the number of distinct keys is small, or the maximum changes infrequently, the additional space complexity is less, $O(1)$ in the best-case. The time complexity for each specified method is still $O(1)$.

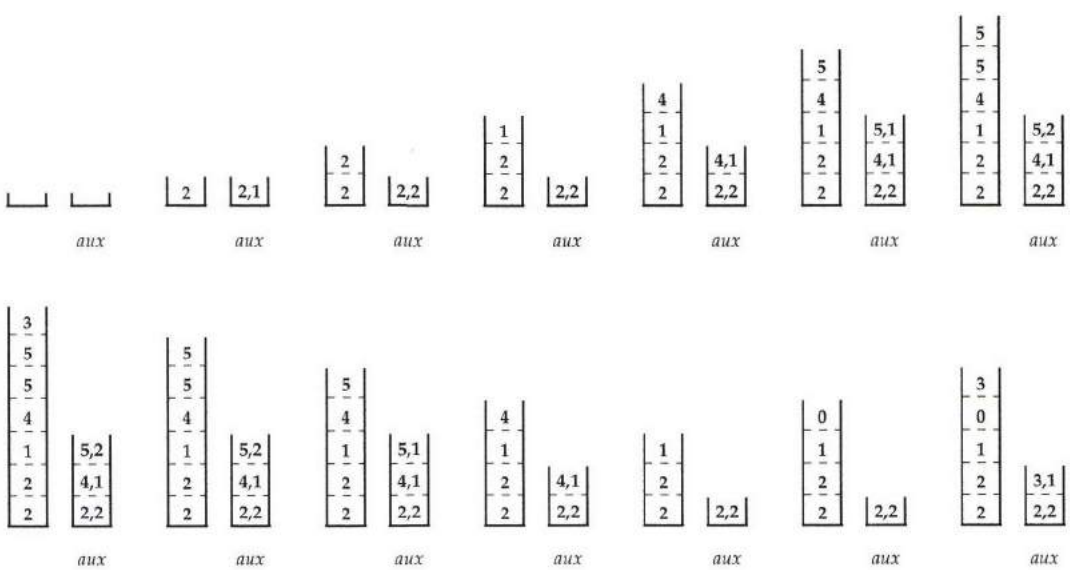


Figure 8.2: The primary and auxiliary stacks for the following operations: push 2, push 2, push 1, push 4, push 5, push 5, push 3, pop, pop, pop, pop, push 0, push 3. Both stacks are initially empty, and their progression is shown from left-to-right, then top-to-bottom. The top of the auxiliary stack holds the maximum element in the stack, and the number of times that element occurs in the stack. The auxiliary stack is denoted by *aux*.

A string is said to be an arithmetical expression in Reverse Polish notation (RPN) if:

(1.) It is a single digit or a sequence of digits, prefixed with an option −, e.g., "6", "123", "−42".

(2.) It is of the form "$A, B, \circ$" where A and B are RPN expressions and $\circ$ is one of $+, -, \times, /$.

For example, the following strings satisfy these rules: "1729", "$3, 4, +, 2, \times, 1, +$", "$1, 1, +, -2, \times$", "$-641, 6, /, 28, /$".

An RPN expression can be evaluated uniquely to an integer, which is determined recursively. The base case corresponds to Rule (1.), which is an integer expressed in base-10 positional system. Rule (2.)corresponds to the recursive case, and the RPNs are evaluated in the natural way, e.g., if A evaluates to 2 and B evaluates to 3, then "$A, B, \times$" evaluates to 6.

Write a program that takes an arithmetical expression in RPN and returns the number that the expression evaluates to.

Hint: Process subexpressions, keeping values in a stack. How should operators be handled?

Solution: Let's begin with the RPN example "$3, 4, +, 2, \times, 1, +$". The ordinary form for this is $(3 + 4) \times 2 + 1$. To evaluate this by hand, we would scan from left to right. We record 3, then 4, then applying the + to 3 and 4, and record the result, 7. Note that we never need to examine the 3 and 4 again. Next we multiply by 2, and record the result, 14. Finally, we add 1 to obtain the final result, 15.

Observe that we need to record partial results, and as we encounter operators, we apply them to the partial results. The partial results are added and removed in last-in, first-out order, which makes a stack the natural data structure for evaluating RPN expressions.

```
def evaluate(RPN_expression):
    intermediate_results = []
    DELIMITER = ','
    OPERATORS = {
        '+': lambda y, x: x + y, '-': lambda y, x: x - y, '*':
        lambda y, x: x * y, '/': lambda y, x: int(x / y)
    }

    for token in RPN_expression.split(DELIMITER):
        if token in OPERATORS:
            intermediate_results.append(OPERATORS[token](
                intermediate_results.pop(), intermediate_results.pop()))
        else:  # token is a number.
            intermediate_results.append(int(token))
    return intermediate_results[-1]
```

Since we perform $O(1)$ computation per character of the string, the time complexity is $O(n)$, where n is the length of the string.

Variant: Solve the same problem for expressions in Polish notation, i.e., when $A, B, \circ$ is replaced by $\circ, A, B$ in Rule (2.).

8.3 Test a string over "{,},(,),[,]" for well-formedness

A string over the characters "{,},(,),[,]" is said to be well-formed if the different types of brackets match in the correct order.

For example, "([]){()}" is well-formed, as is "[()[]{()()}]". However, "{)", "()", and "[()[]{()()" are not well-formed,

Write a program that tests if a string made up of the characters '(', ')', '[', ']',"{' and"}' is well-formed.

Hint: Which left parenthesis does a right parenthesis match with?

Solution: Let's begin with well-formed strings consisting solely of left and right parentheses, e.g., "()(())". If such a string is well-formed, each right parenthesis must match the closest left parenthesis to its left. Therefore, starting from the left, every time we see a left parenthesis, we store it. Each time we see a right parenthesis, we match it with a stored left parenthesis. Since there are not brackets or braces, we can simply keep a count of the number of unmatched left parentheses.

For the general case, we do the same, except that we need to explicitly store the unmatched left characters, i.e., left parenthesis, left brackets, and left braces. We cannot use three counters, because that will not tell us the last unmatched one. A stack is a perfect option for this application: we use it to record the unmatched left characters, with the most recent one at the top.

If we encounter a right character and the stack is empty or the top of the stack is a different type of left character, the right character is not matched, implying the string is not matched. For example, if the input string is "(()][)", when we encounter the first ']', the character at the top of the stack is '(', so the string is not matched. Conversely, if the input string is "(()[])", when we encounter the first ']', the character at the top of the stack is '[', so we continue on. If all characters have been processed and the stack is nonempty, there are unmatched left characters so the string is not matched.

```
def is_well_formed(s):
    left_chars, lookup = [], {'(': ')', '{': '}', '[': ']'}
    for c in s:
        if c in lookup:
            left_chars.append(c)
        elif not left_chars or lookup[left_chars.pop()] != c:
            # Unmatched right char or mismatched chars.
            return False
    return not left_chars
```

The time complexity is $O(n)$ since for each character we perform $O(1)$ operations.

8.4 Normalize pathnames

A file or directory can be specified via a string called the pathname. This string may specify an absolute path, starting from the root, e.g., `/usr/bin/gcc`, or a path relative to the current working directory, e.g., `scripts/awkscripts`.

The same directory may be specified by multiple directory paths. For example, `/usr/lib/../bin/gcc` and `scripts//./../scripts/awkscripts/././` specify equivalent absolute and relative pathnames.

Write a program which takes a pathname, and returns the shortest equivalent pathname. Assume individual directories and files have names that use only alphanumeric characters. Subdirectory names may be combined using forward slashes (/), the current directory (.), and parent directory (..).

Hint: Trace the cases. How should . and .. be handled? Watch for invalid paths.

Solution: It is natural to process the string from left-to-right, splitting on forward slashes (/s). We record directory and file names. Each time we encounter a .., we delete the most recent name, which corresponds to going up directory hierarchy. Since names are processed in a last-in, first-out order, it is natural to store them in a stack. Individual periods (.s) are skipped.

If the string begins with /, then we cannot go up from it. We record this in the stack. If the stack does not begin with /, we may encounter an empty stack when processing .., which indicates a path that begins with an ancestor of the current working path. We need to record this in order to give the shortest equivalent path. The final state of the stack directly corresponds to the shortest equivalent directory path.

For example, if the string is sc//./../tc/awk/././, the stack progression is as follows: ⟨sc⟩, ⟨⟩, ⟨tc⟩, ⟨tc, awk⟩. Note that we skip three .s and the / after sc/.

```
def shortest_equivalent_path(path):
    if not path:
        raise ValueError('Empty string is not a valid path.')

    path_names = []  # Uses list as a stack.
    # Special case: starts with '/', which is an absolute path.
    if path[0] == '/':
        path_names.append('/')

    for token in (token for token in path.split('/') if token not in ['.', '']):
        if token == '..':
            if not path_names or path_names[-1] == '..':
                path_names.append(token)
            else:
                if path_names[-1] == '/':
                    raise ValueError('Path error')
                path_names.pop()
        else:  # Must be a name.
            path_names.append(token)

    result = '/'.join(path_names)
    return result[result.startswith('//'):]  # Avoid starting '//'.
```

The time complexity is $O(n)$, where n is the length of the pathname.

8.5 Compute buildings with a sunset view

You are given a series of buildings that have windows facing west. The buildings are in a straight line, and any building which is to the east of a building of equal or greater height cannot view the sunset.

Design an algorithm that processes buildings in east-to-west order and returns the set of buildings which view the sunset. Each building is specified by its height.

Hint: When does a building not have a sunset view?

Solution: A brute-force approach is to store all buildings in an array. We then do a reverse scan of this array, tracking the running maximum. Any building whose height is less than or equal to the running maximum does not have a sunset view.

The time and space complexity are both $O(n)$, where n is the number of buildings.

Note that if a building is to the east of a taller building, it cannot view the sunset. This suggests a way to reduce the space complexity. We record buildings which potentially have a view. Each new building may block views from the existing set. We determine which such buildings are blocked by comparing the new building's height to that of the buildings in the existing set. We can store the existing set as a hash set—this requires us to iterate over all buildings each time a new building is processed.

If a new building is shorter than a building in the current set, then all buildings in the current set which are further to the east cannot be blocked by the new building. This suggests keeping the buildings in a last-in, first-out manner, so that we can terminate earlier.

Specifically, we use a stack to record buildings that have a view. Each time a building b is processed, if it is taller than the building at the top of the stack, we pop the stack until the top of the stack is taller than b—all the buildings thus removed lie to the east of a taller building.

Although some individual steps may require many pops, each building is pushed and popped at most once. Therefore, the run time to process n buildings is $O(n)$, and the stack always holds precisely the buildings which currently have a view.

The memory used is $O(n)$, and the bound is tight, even when only one building has a view—consider the input where the west-most building is the tallest, and the remaining $n - 1$ buildings decrease in height from east to west. However, in the best-case, e.g., when buildings appear in increasing height, we use $O(1)$ space. In contrast, the brute-force approach always uses $O(n)$ space.

```
def examine_buildings_with_sunset(sequence):
    BuildingWithHeight = collections.namedtuple('BuildingWithHeight',
                                                ('id', 'height'))
    candidates = []
    for building_idx, building_height in enumerate(sequence):
        while candidates and building_height >= candidates[-1].height:
            candidates.pop()
        candidates.append(BuildingWithHeight(building_idx, building_height))
    return [candidate.id for candidate in reversed(candidates)]
```

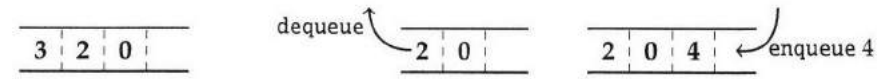
Variant: Solve the problem subject to the same constraints when buildings are presented in west-to-east order.

Queues

A *queue* supports two basic operations—enqueue and dequeue. (If the queue is empty, dequeue typically returns null or throws an exception.) Elements are added (enqueued) and removed (dequeued) in first-in, first-out order. The most recently inserted element is referred to as the tail

or back element, and the item that was inserted least recently is referred to as the head or front element.

A queue can be implemented using a linked list, in which case these operations have $O(1)$ time complexity. The queue API often includes other operations, e.g., a method that returns the item at the head of the queue without removing it, a method that returns the item at the tail of the queue without removing it, etc. A queue can also be implemented using an array; see Problem 8.7 on Page 107 for details.



(a) Initial configuration. (b) Queue (a) after dequeue. (c) Queue (b) after enqueuing 4.

Figure 8.3: Examples of enqueuing and dequeuing.

A *deque*, also sometimes called a double-ended queue, is a doubly linked list in which all insertions and deletions are from one of the two ends of the list, i.e., at the head or the tail. An insertion to the front is commonly called a push, and an insertion to the back is commonly called an inject. A deletion from the front is commonly called a pop, and a deletion from the back is commonly called an eject. (Different languages and libraries may have different nomenclature.)

Queues boot camp

In the following program, we implement the basic queue API—enqueue and dequeue—as well as a max-method, which returns the maximum element stored in the queue. The basic idea is to use composition: add a private field that references a library queue object, and forward existing methods (enqueue and dequeue in this case) to that object.

```
class Queue:
    def __init__(self):
        self._data = collections.deque()

    def enqueue(self, x):
        self._data.append(x)

    def dequeue(self):
        return self._data.popleft()

    def max(self):
        return max(self._data)
```

The time complexity of enqueue and dequeue are the same as that of the library queue, namely, $O(1)$. The time complexity of finding the maximum is $O(n)$, where n is the number of entries. In Solution 8.9 on Page 109 we show how to improve the time complexity of maximum to $O(1)$ with a more customized approach.

Learn to recognize when the queue FIFO property is applicable. For example, queues are ideal when order needs to be preserved.

Table 8.2: Top Tips for Queues

Know your queue libraries

Some of the problems require you to implement your own queue class; for others, use the `collections.deque` class.

- `q.append(e)` pushes an element onto the queue. Not much can go wrong with a call to push.
- `q[0]` will retrieve, but not remove, the element at the front of the queue. Similarly, `q[-1]` will retrieve, but not remove, the element at the back of the queue.
- `q.popleft()` will remove and return the element at the front of the queue.

Dequeing or accessing the head/tail of an empty collection results in an `IndexError` exception being raised.

8.6 Compute binary tree nodes in order of increasing depth

Binary trees are formally defined in Chapter 9. In particular, each node in a binary tree has a depth, which is its distance from the root.

Given a binary tree, return an array consisting of the keys at the same level. Keys should appear in the order of the corresponding nodes' depths, breaking ties from left to right. For example, you should return $\langle\langle 314\rangle, \langle 6, 6\rangle, \langle 271, 561, 2, 271\rangle, \langle 28, 0, 3, 1, 28\rangle, \langle 17, 401, 257\rangle, \langle 641\rangle\rangle$ for the binary tree in Figure 9.1 on Page 112.

Hint: First think about solving this problem with a pair of queues.

Solution: A brute force approach might be to write the nodes into an array while simultaneously computing their depth. We can use preorder traversal to compute this array—by traversing a node's left child first we can ensure that nodes at the same depth are sorted from left to right. Now we can sort this array using a stable sorting algorithm with node depth being the sort key. The time complexity is dominated by the time taken to sort, i.e., $O(n \log n)$, and the space complexity is $O(n)$, which is the space to store the node depths.

Intuitively, since nodes are already presented in a somewhat ordered fashion in the tree, it should be possible to avoid a full-blow sort, thereby reducing time complexity. Furthermore, by processing nodes in order of depth, we do not need to label every node with its depth.

In the following, we use a queue of nodes to store nodes at depth i and a queue of nodes at depth $i + 1$. After all nodes at depth i are processed, we are done with that queue, and can start processing the queue with nodes at depth $i + 1$, putting the depth $i + 2$ nodes in a new queue.

```
def binary_tree_depth_order(tree):
    result = []
    if not tree:
        return result

    curr_depth_nodes = [tree]
    while curr_depth_nodes:
        result.append([curr.data for curr in curr_depth_nodes])
        curr_depth_nodes = [
            child
            for curr in curr_depth_nodes for child in (curr.left, curr.right)
            if child
        ]
    return result
```

Since each node is enqueued and dequeued exactly once, the time complexity is $O(n)$. The space complexity is $O(m)$, where m is the maximum number of nodes at any single depth.

Variant: Write a program which takes as input a binary tree and returns the keys in top down, alternating left-to-right and right-to-left order, starting from left-to-right. For example, if the input is the tree in Figure 9.1 on Page 112, your program should return $\langle\langle 314\rangle, \langle 6, 6\rangle, \langle 271, 561, 2, 271\rangle, \langle 28, 1, 3, 0, 28\rangle, \langle 17, 401, 257\rangle, \langle 641\rangle\rangle$.

Variant: Write a program which takes as input a binary tree and returns the keys in a bottom up, left-to-right order. For example, if the input is the tree in Figure 9.1 on Page 112, your program should return $\langle\langle 641\rangle, \langle 17, 401, 257\rangle, \langle 28, 0, 3, 1, 28\rangle, \langle 271, 561, 2, 271\rangle, \langle 6, 6\rangle, \langle 314\rangle\rangle$.

Variant: Write a program which takes as input a binary tree with integer keys, and returns the average of the keys at each level. For example, if the input is the tree in Figure 9.1 on Page 112, your program should return $\langle 314, 6, 276.25, 12, 225, 641\rangle$.

8.7 Implement a circular queue

A queue can be implemented using an array and two additional fields, the beginning and the end indices. This structure is sometimes referred to as a circular queue. Both enqueue and dequeue have $O(1)$ time complexity. If the array is fixed, there is a maximum number of entries that can be stored. If the array is dynamically resized, the total time for m combined enqueue and dequeue operations is $O(m)$.

Implement a queue API using an array for storing elements. Your API should include a constructor function, which takes as argument the initial capacity of the queue, enqueue and dequeue functions, and a function which returns the number of elements stored. Implement dynamic resizing to support storing an arbitrarily large number of elements.

Hint: Track the head and tail. How can you differentiate a full queue from an empty one?

Solution: A brute-force approach is to use an array, with the head always at index 0. An additional variable tracks the index of the tail element. Enqueue has $O(1)$ time complexity. However dequeue's time complexity is $O(n)$, where n is the number of elements in the queue, since every element has to be left-shifted to fill up the space created at index 0.

A better approach is to keep one more variable to track the head. This way, dequeue can also be performed in $O(1)$ time. When performing an enqueue into a full array, we need to resize the array. We cannot only resize, because this results in queue elements not appearing contiguously. For example, if the array is $\langle e, b, c, d\rangle$, with e being the tail and b the head, if we resize to get $\langle e, b, c, d, \sqcup, \sqcup, \sqcup, \sqcup\rangle$, we cannot enqueue without overwriting or moving elements.

```
class Queue:
    SCALE_FACTOR = 2

    def __init__(self, capacity):
        self._entries = [None] * capacity
        self._head = self._tail = self._num_queue_elements = 0

    def enqueue(self, x):
```

```
        if self._num_queue_elements == len(self._entries):  # Needs to resize.
            # Makes the queue elements appear consecutively.
            self._entries = (
                self._entries[self._head:] + self._entries[:self._head])
            # Resets head and tail.
            self._head, self._tail = 0, self._num_queue_elements
            self._entries += [None] * (
                len(self._entries) * Queue.SCALE_FACTOR - len(self._entries))

        self._entries[self._tail] = x
        self._tail = (self._tail + 1) % len(self._entries)
        self._num_queue_elements += 1

    def dequeue(self):
        if not self._num_queue_elements:
            raise IndexError('empty queue')
        self._num_queue_elements -= 1
        ret = self._entries[self._head]
        self._head = (self._head + 1) % len(self._entries)
        return ret

    def size(self):
        return self._num_queue_elements
```

The time complexity of dequeue is $O(1)$, and the amortized time complexity of enqueue is $O(1)$.

8.8 Implement a queue using stacks

Queue insertion and deletion follows first-in, first-out semantics; stack insertion and deletion is last-in, first-out.

How would you implement a queue given a library implementing stacks?

Hint: It is impossible to solve this problem with a single stack.

Solution: A straightforward implementation is to enqueue by pushing the element to be enqueued onto one stack. The element to be dequeued is then the element at the bottom of this stack, which can be achieved by first popping all its elements and pushing them to another stack, then popping the top of the second stack (which was the bottom-most element of the first stack), and finally popping the remaining elements back to the first stack.

The primary problem with this approach is that every dequeue takes two pushes and two pops of *every* element, i.e., dequeue has $O(n)$ time complexity, where n is the number of stored elements. (Enqueue takes $O(1)$ time.)

The intuition for improving the time complexity of dequeue is that after we move elements from the first stack to the second stack, any further dequeues are trivial, until the second stack is empty. This is true even if we need to enqueue, as long as we enqueue onto the first stack. When the second stack becomes empty, and we need to perform a dequeue, we simply repeat the process of transferring from the first stack to the second stack. In essence, we are using the first stack for enqueue and the second for dequeue.

```
class Queue:
```

```
def __init__(self):
    self._enq, self._deq = [], []

def enqueue(self, x):
    self._enq.append(x)

def dequeue(self):
    if not self._deq:
        # Transfers the elements in _enq to _deq.
        while self._enq:
            self._deq.append(self._enq.pop())

    if not self._deq:  # _deq is still empty!
        raise IndexError('empty queue')
    return self._deq.pop()
```

This approach takes $O(m)$ time for m operations, which can be seen from the fact that each element is pushed no more than twice and popped no more than twice.

8.9 Implement a queue with max API

Implement a queue with enqueue, dequeue, and max operations. The max operation returns the maximum element currently stored in the queue.

Hint: When can an element never be returned by max, regardless of future updates?

Solution: A brute-force approach is to track the current maximum. The current maximum has to be updated on both enqueue and dequeue. Updating the current maximum on enqueue is trivial and fast—just compare the enqueued value with the current maximum. However, updating the current maximum on dequeue is slow—we must examine every single remaining element, which takes $O(n)$ time, where n is the size of the queue.

Consider an element s in the queue that has the property that it entered the queue before a later element, b, which is greater than s. Since s will be dequeued before b, s can never in the future become the maximum element stored in the queue, regardless of the subsequent enqueues and dequeues.

The key to a faster implementation of a queue-with-max is to eliminate elements like s from consideration. We do this by maintaining the set of entries in the queue that have no later entry in the queue greater than them in a separate deque. Elements in the deque will be ordered by their position in the queue, with the candidate closest to the head of the queue appearing first. Since each entry in the deque is greater than or equal to its successors, the largest element in the queue is at the head of the deque.

We now briefly describe how to update the deque on queue updates. If the queue is dequeued, and if the element just dequeued is at the deque's head, we pop the deque from its head; otherwise the deque remains unchanged. When we add an entry to the queue, we iteratively evict from the deque's tail until the element at the tail is greater than or equal to the entry being enqueued, and then add the new entry to the deque's tail. These operations are illustrated in Figure 8.4 on the following page.

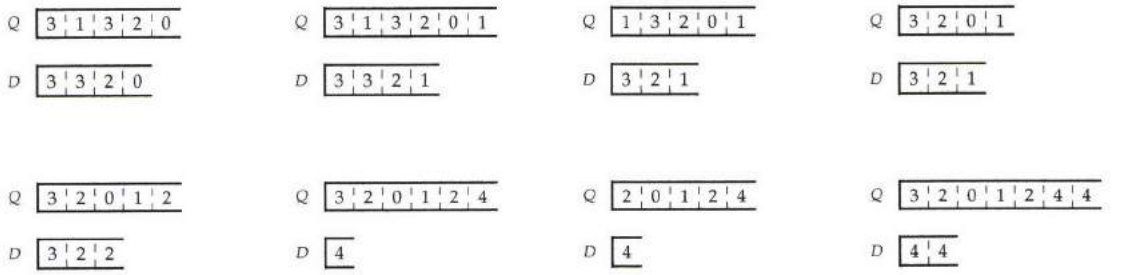


Figure 8.4: The queue with max for the following operations: enqueue 1, dequeue, dequeue, enqueue 2, enqueue 4, dequeue, enqueue 4. The queue initially contains 3, 1, 3, 2, and 0 in that order. The deque D corresponding to queue Q is immediately below Q. The progression is shown from left-to-right, then top-to-bottom. The head of each queue and deque is on the left. Observe how the head of the deque holds the maximum element in the queue.

```
class QueueWithMax:
    def __init__(self):
        self._entries = collections.deque()
        self._candidates_for_max = collections.deque()

    def enqueue(self, x):
        self._entries.append(x)
        # Eliminate dominated elements in _candidates_for_max.
        while self._candidates_for_max and self._candidates_for_max[-1] < x:
            self._candidates_for_max.pop()
        self._candidates_for_max.append(x)

    def dequeue(self):
        if self._entries:
            result = self._entries.popleft()
            if result == self._candidates_for_max[0]:
                self._candidates_for_max.popleft()
            return result
        raise IndexError('empty queue')

    def max(self):
        if self._candidates_for_max:
            return self._candidates_for_max[0]
        raise IndexError('empty queue')
```

Each dequeue operation has time $O(1)$ complexity. A single enqueue operation may entail many ejections from the deque. However, the amortized time complexity of n enqueues and dequeues is $O(n)$, since an element can be added and removed from the deque no more than once. The max operation is $O(1)$ since it consists of returning the element at the head of the deque.

An alternate solution that is often presented is to use reduction. Specifically, we know how to solve the stack-with-max problem efficiently (Solution 8.1 on Page 98) and we also know how to efficiently model a queue with two stacks (Solution 8.8 on Page 108), so we can solve the queue-with-max design by modeling a queue with two stacks-with-max. This approach feels unnatural compared to the one presented above.

```
class QueueWithMax:
    def __init__(self):
```

```
        self._enqueue = Stack()
        self._dequeue = Stack()

    def enqueue(self, x):
        self._enqueue.push(x)

    def dequeue(self):
        if self._dequeue.empty():
            while not self._enqueue.empty():
                self._dequeue.push(self._enqueue.pop())
        if not self._dequeue.empty():
            return self._dequeue.pop()
        raise IndexError('empty queue')

    def max(self):
        if not self._enqueue.empty():
            return self._enqueue.max() if self._dequeue.empty() else max(
                self._enqueue.max(), self._dequeue.max())
        if not self._dequeue.empty():
            return self._dequeue.max()
        raise IndexError('empty queue')
```

Since the stack-with-max has $O(1)$ amortized time complexity for push, pop, and max, and the queue from two stacks has $O(1)$ amortized time complexity for enqueue and dequeue, this approach has $O(1)$ amortized time complexity for enqueue, dequeue, and max.

CHAPTER

9 Binary Trees

The method of solution involves the development of a theory of finite automata operating on infinite trees.

— *"Decidability of Second Order Theories and Automata on Trees,"* M. O. RABIN, 1969

Formally, a binary tree is either empty, or a *root* node r together with a left binary tree and a right binary tree. The subtrees themselves are binary trees. The left binary tree is sometimes referred to as the *left subtree* of the root, and the right binary tree is referred to as the *right subtree* of the root.

Binary trees most commonly occur in the context of binary search trees, wherein keys are stored in a sorted fashion (Chapter 14 on Page 197). However, there are many other applications of binary trees: at a high level, *binary tree* are appropriate when dealing with hierarchies.

Figure 9.1 gives a graphical representation of a binary tree. Node A is the root. Nodes B and I are the left and right children of A.

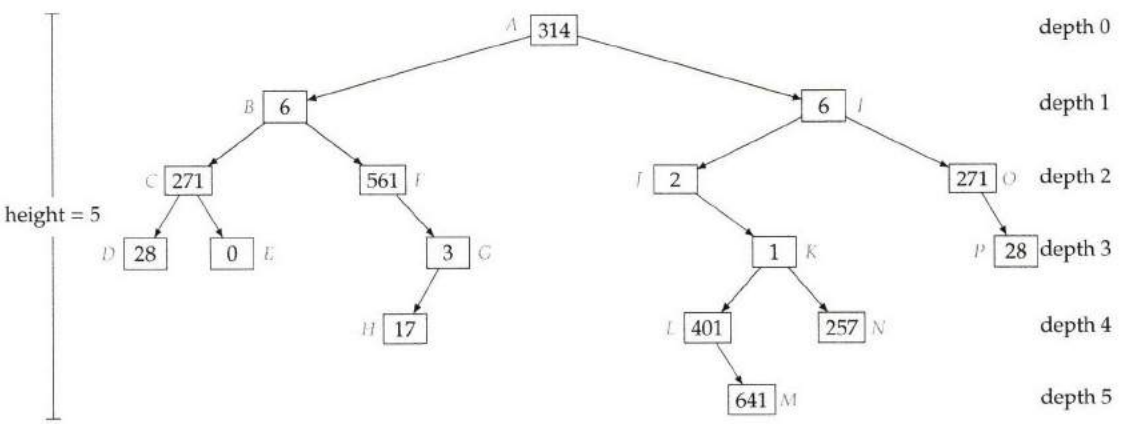


Figure 9.1: Example of a binary tree. The node depths range from 0 to 5. Node M has the highest depth (5) of any node in the tree, implying the height of the tree is 5.

Often the node stores additional data. Its prototype is listed as follows:

```
class BinaryTreeNode:
    def __init__(self, data=None, left=None, right=None):
        self.data = data
        self.left = left
        self.right = right
```

Each node, except the root, is itself the root of a left subtree or a right subtree. If l is the root of p's left subtree, we will say l is the *left child* of p, and p is the *parent* of l; the notion of *right child* is

similar. If a node is a left or a right child of p, we say it is a *child* of p. Note that with the exception of the root, every node has a unique parent. Usually, but not universally, the node object definition includes a parent field (which is null for the root). Observe that for any node there exists a unique sequence of nodes from the root to that node with each node in the sequence being a child of the previous node. This sequence is sometimes referred to as the *search path* from the root to the node.

The parent-child relationship defines an ancestor-descendant relationship on nodes in a binary tree. Specifically, a node is an *ancestor* of d if it lies on the search path from the root to d. If a node is an ancestor of d, we say d is a *descendant* of that node. Our convention is that a node is an ancestor and descendant of itself. A node that has no descendants except for itself is called a *leaf*.

The *depth* of a node n is the number of nodes on the search path from the root to n, not including n itself. The *height* of a binary tree is the maximum depth of any node in that tree. A *level* of a tree is all nodes at the same depth. See Figure 9.1 on the preceding page for an example of the depth and height concepts.

As concrete examples of these concepts, consider the binary tree in Figure 9.1 on the facing page. Node I is the parent of J and O. Node G is a descendant of B. The search path to L is $\langle A, I, J, K, L\rangle$. The depth of N is 4. Node M is the node of maximum depth, and hence the height of the tree is 5. The height of the subtree rooted at B is 3. The height of the subtree rooted at H is 0. Nodes D, E, H, M, N, and P are the leaves of the tree.

A *full binary tree* is a binary tree in which every node other than the leaves has two children. A *perfect binary tree* is a full binary tree in which all leaves are at the same depth, and in which every parent has two children. A *complete binary tree* is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible. (This terminology is not universal, e.g., some authors use complete binary tree where we write perfect binary tree.) It is straightforward to prove using induction that the number of nonleaf nodes in a full binary tree is one less than the number of leaves. A perfect binary tree of height h contains exactly $2^{h+1} - 1$ nodes, of which 2^h are leaves. A complete binary tree on n nodes has height $\lfloor \log n \rfloor$. A left-skewed tree is a tree in which no node has a right child; a right-skewed tree is a tree in which no node has a left child. In either case, we refer to the binary tree as being skewed.

A key computation on a binary tree is *traversing* all the nodes in the tree. (Traversing is also sometimes called *walking*.) Here are some ways in which this visit can be done.

- Traverse the left subtree, visit the root, then traverse the right subtree (an *inorder* traversal). An inorder traversal of the binary tree in Figure 9.1 on the preceding page visits the nodes in the following order: $\langle D, C, E, B, F, H, G, A, J, L, M, K, N, I, O, P\rangle$.
- Visit the root, traverse the left subtree, then traverse the right subtree (a *preorder* traversal). A preorder traversal of the binary tree in Figure 9.1 on the facing page visits the nodes in the following order: $\langle A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P\rangle$.
- Traverse the left subtree, traverse the right subtree, and then visit the root (a *postorder* traversal). A postorder traversal of the binary tree in Figure 9.1 on the preceding page visits the nodes in the following order: $\langle D, E, C, H, G, F, B, M, L, N, K, J, P, O, I, A\rangle$.

Let T be a binary tree of n nodes, with height h. Implemented recursively, these traversals have $O(n)$ time complexity and $O(h)$ additional space complexity. (The space complexity is dictated by the maximum depth of the function call stack.) If each node has a parent field, the traversals can be done with $O(1)$ additional space complexity.

The term tree is overloaded, which can lead to confusion; see Page 275 for an overview of the common variants.

Binary trees boot camp

A good way to get up to speed with binary trees is to implement the three basic traversals—inorder, preorder, and postorder.

```
def tree_traversal(root):
    if root:
        # Preorder: Processes the root before the traversals of left and right
        # children.
        print('Preorder: %d' % root.data)
        tree_traversal(root.left)
        # Inorder: Processes the root after the traversal of left child and
        # before the traversal of right child.
        print('Inorder: %d' % root.data)
        tree_traversal(root.right)
        # Postorder: Processes the root after the traversals of left and right
        # children.
        print('Postorder: %d' % root.data)
```

The time complexity of each approach is $O(n)$, where n is the number of nodes in the tree. Although no memory is explicitly allocated, the function call stack reaches a maximum depth of h, the height of the tree. Therefore, the space complexity is $O(h)$. The minimum value for h is $\log n$ (complete binary tree) and the maximum value for h is n (skewed tree).

Recursive algorithms are well-suited to problems on trees. Remember to include space implicitly allocated on the **function call stack** when doing space complexity analysis. Please read the introduction to Chapter 15 if you need a refresher on recursion.
Some tree problems have simple brute-force solutions that use $O(n)$ space, but subtler solutions that use the **existing tree nodes** to reduce space complexity to $O(1)$.
Consider **left- and right-skewed trees** when doing complexity analysis. Note that $O(h)$ complexity, where h is the tree height, translates into $O(\log n)$ complexity for balanced trees, but $O(n)$ complexity for skewed trees.
If each node has a **parent field**, use it to make your code simpler, and to reduce time and space complexity.
It's easy to make the **mistake** of treating a node that has a **single child** as a leaf.

Table 9.1: Top Tips for Binary Trees

9.1 Test if a binary tree is height-balanced

A binary tree is said to be height-balanced if for each node in the tree, the difference in the height of its left and right subtrees is at most one. A perfect binary tree is height-balanced, as is a complete binary tree. A height-balanced binary tree does not have to be perfect or complete—see Figure 9.2 on the next page for an example.

Write a program that takes as input the root of a binary tree and checks whether the tree is height-balanced.

Hint: Think of a classic binary tree algorithm.

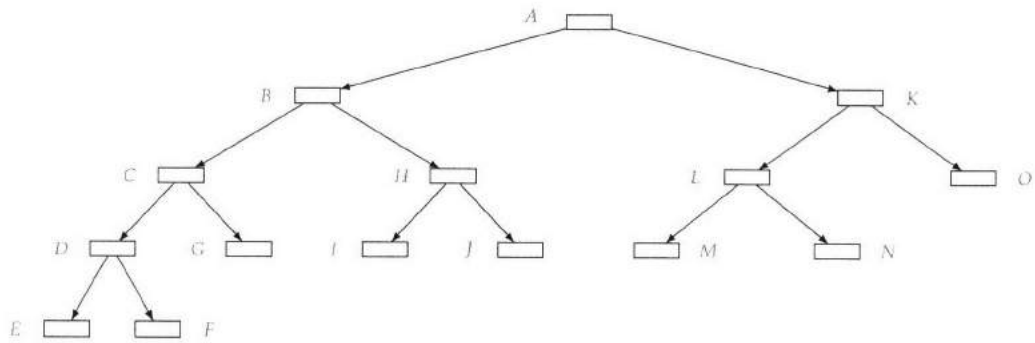


Figure 9.2: A height-balanced binary tree of height 4.

Solution: Here is a brute-force algorithm. Compute the height for the tree rooted at each node x recursively. The basic computation is to compute the height for each node starting from the leaves, and proceeding upwards. For each node, we check if the difference in heights of the left and right children is greater than one. We can store the heights in a hash table, or in a new field in the nodes. This entails $O(n)$ storage and $O(n)$ time, where n is the number of nodes of the tree.

We can solve this problem using less storage by observing that we do not need to store the heights of all nodes at the same time. Once we are done with a subtree, all we need to know is whether it is height-balanced, and if so, what its height is—we do not need any information about descendants of the subtree's root.

```
def is_balanced_binary_tree(tree):
    BalancedStatusWithHeight = collections.namedtuple(
        'BalancedStatusWithHeight', ('balanced', 'height'))

    # First value of the return value indicates if tree is balanced, and if
    # balanced the second value of the return value is the height of tree.
    def check_balanced(tree):
        if not tree:
            return BalancedStatusWithHeight(True, -1)  # Base case.

        left_result = check_balanced(tree.left)
        if not left_result.balanced:
            # Left subtree is not balanced.
            return BalancedStatusWithHeight(False, 0)

        right_result = check_balanced(tree.right)
        if not right_result.balanced:
            # Right subtree is not balanced.
            return BalancedStatusWithHeight(False, 0)

        is_balanced = abs(left_result.height - right_result.height) <= 1
        height = max(left_result.height, right_result.height) + 1
        return BalancedStatusWithHeight(is_balanced, height)

    return check_balanced(tree).balanced
```

The program implements a postorder traversal with some calls possibly being eliminated because of early termination. Specifically, if any left subtree is not height-balanced we do not need to visit

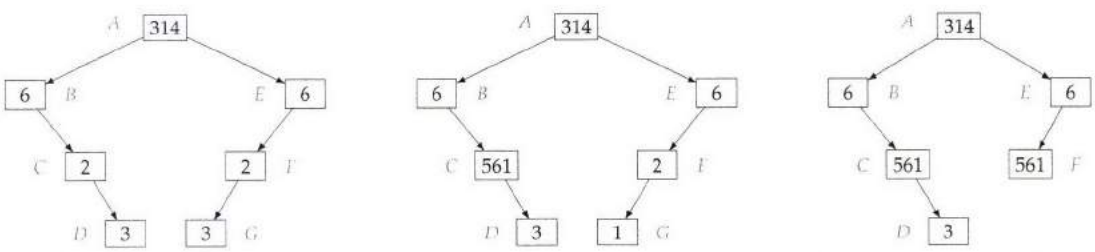
the corresponding right subtree. The function call stack corresponds to a sequence of calls from the root through the unique path to the current node, and the stack height is therefore bounded by the height of the tree, leading to an $O(h)$ space bound. The time complexity is the same as that for a postorder traversal, namely $O(n)$.

Variant: Write a program that returns the size of the largest subtree that is complete.

Variant: Define a node in a binary tree to be k-balanced if the difference in the number of nodes in its left and right subtrees is no more than k. Design an algorithm that takes as input a binary tree and positive integer k, and returns a node in the binary tree such that the node is not k-balanced, but all of its descendants are k-balanced. For example, when applied to the binary tree in Figure 9.1 on Page 112, if $k = 3$, your algorithm should return Node J.

9.2 Test if a binary tree is symmetric

A binary tree is symmetric if you can draw a vertical line through the root and then the left subtree is the mirror image of the right subtree. The concept of a symmetric binary tree is illustrated in Figure 9.3.



(a) A symmetric binary tree. (b) An asymmetric binary tree. (c) An asymmetric binary tree.

Figure 9.3: Symmetric and asymmetric binary trees. The tree in (a) is symmetric. The tree in (b) is structurally symmetric, but not symmetric, because symmetry requires that corresponding nodes have the same keys; here C and F as well as D and G break symmetry. The tree in (c) is asymmetric because there is no node corresponding to D.

Write a program that checks whether a binary tree is symmetric.

Hint: The definition of symmetry is recursive.

Solution: We can test if a tree is symmetric by computing its mirror image and seeing if the mirror image is equal to the original tree. Computing the mirror image of a tree is as simple as swapping the left and right subtrees, and recursively continuing. The time and space complexity are both $O(n)$, where n is the number of nodes in the tree.

The insight to a better algorithm is that we do not need to construct the mirrored subtrees. All that is important is whether a pair of subtrees are mirror images. As soon as a pair fails the test, we can short circuit the check to false. This is shown in the code below.

```
def is_symmetric(tree):
    def check_symmetric(subtree_0, subtree_1):
        if not subtree_0 and not subtree_1:
            return True
        elif subtree_0 and subtree_1:
            return (subtree_0.data == subtree_1.data
```

```
                and check_symmetric(subtree_0.left, subtree_1.right)
                and check_symmetric(subtree_0.right, subtree_1.left))
        # One subtree is empty, and the other is not.
        return False

    return not tree or check_symmetric(tree.left, tree.right)
```

The time complexity and space complexity are $O(n)$ and $O(h)$, respectively, where n is the number of nodes in the tree and h is the height of the tree.

9.3 Compute the lowest common ancestor in a binary tree

Any two nodes in a binary tree have a common ancestor, namely the root. The lowest common ancestor (LCA) of any two nodes in a binary tree is the node furthest from the root that is an ancestor of both nodes. For example, the LCA of M and N in Figure 9.1 on Page 112 is K.

Computing the LCA has important applications. For example, it is an essential calculation when rendering web pages, specifically when computing the Cascading Style Sheet (CSS) that is applicable to a particular Document Object Model (DOM) element.

Design an algorithm for computing the LCA of two nodes in a binary tree in which nodes do not have a parent field.

Hint: When is the root the LCA?

Solution: A brute-force approach is to see if the nodes are in different immediate subtrees of the root, or if one of the nodes is the root. In this case, the root must be the LCA. If both nodes are in the left subtree of the root, or the right subtree of the root, we recurse on that subtree. The time complexity is $O(n^2)$, where n is the number of nodes. The worst-case is a skewed tree with the two nodes at the bottom of the tree.

The insight to a better time complexity is that we do not need to perform multiple passes. If the two nodes are in a subtree, we can compute the LCA directly, instead of simply returning a Boolean indicating that both nodes are in that subtree. The program below returns an object with two fields—the first is an integer indicating how many of the two nodes were present in that subtree, and the second is their LCA, if both nodes were present.

```
def lca(tree, node0, node1):
    Status = collections.namedtuple('Status', ('num_target_nodes', 'ancestor'))

    # Returns an object consisting of an int and a node. The int field is 0,
    # 1, or 2 depending on how many of {node0, node1} are present in tree. If
    # both are present in tree, when ancestor is assigned to a non-null value,
    # it is the LCA.
    def lca_helper(tree, node0, node1):
        if not tree:
            return Status(0, None)

        left_result = lca_helper(tree.left, node0, node1)
        if left_result.num_target_nodes == 2:
            # Found both nodes in the left subtree.
            return left_result
        right_result = lca_helper(tree.right, node0, node1)
        if right_result.num_target_nodes == 2:
```

```
            # Found both nodes in the right subtree.
            return right_result
        num_target_nodes = (
            left_result.num_target_nodes + right_result.num_target_nodes + int(
                tree is node0) + int(tree is node1))
        return Status(num_target_nodes, tree if num_target_nodes == 2 else None)

    return lca_helper(tree, node0, node1).ancestor
```

The algorithm is structurally similar to a recursive postorder traversal, and the complexities are the same. Specifically, the time complexity and space complexity are $O(n)$ and $O(h)$, respectively, where h is the height of the tree.

9.4 Compute the LCA when nodes have parent pointers

Given two nodes in a binary tree, design an algorithm that computes their LCA. Assume that each node has a parent pointer.

Hint: The problem is easy if both nodes are the same distance from the root.

Solution: A brute-force approach is to store the nodes on the search path from the root to one of the nodes in a hash table. This is easily done since we can use the parent field. Then we go up from the second node, stopping as soon as we hit a node in the hash table. The time and space complexity are both $O(h)$, where h is the height of the tree.

We know the two nodes have a common ancestor, namely the root. If the nodes are at the same depth, we can move up the tree in tandem from both nodes, stopping at the first common node, which is the LCA. However, if they are not the same depth, we need to keep the set of traversed nodes to know when we find the first common node. We can circumvent having to store these nodes by ascending from the deeper node to get the same depth as the shallower node, and then performing the tandem upward movement.

For example, for the tree in Figure 9.1 on Page 112, nodes M and P are depths 5 and 3, respectively. Their search paths are $\langle A, I, J, K, L, M\rangle$ and $\langle A, I, O, P\rangle$. If we ascend to depth 3 from M, we get to K. Now when we move upwards in tandem, the first common node is I, which is the LCA of M and P.

Computing the depth is straightforward since we have the parent field—the time complexity is $O(h)$ and the space complexity is $O(1)$. Once we have the depths we can perform the tandem move to get the LCA.

```
def lca(node_0, node_1):
    def get_depth(node):
        depth = 0
        while node:
            depth += 1
            node = node.parent
        return depth

    depth_0, depth_1 = get_depth(node_0), get_depth(node_1)
    # Makes node_0 as the deeper node in order to simplify the code.
    if depth_1 > depth_0:
        node_0, node_1 = node_1, node_0

    # Ascends from the deeper node.
```

```
    depth_diff = abs(depth_0 - depth_1)
    while depth_diff:
        node_0 = node_0.parent
        depth_diff -= 1

    # Now ascends both nodes until we reach the LCA.
    while node_0 is not node_1:
        node_0, node_1 = node_0.parent, node_1.parent
    return node_0
```

The time and space complexity are that of computing the depth, namely $O(h)$ and $O(1)$, respectively.

9.5 Sum the root-to-leaf paths in a binary tree

Consider a binary tree in which each node contains a binary digit. A root-to-leaf path can be associated with a binary number—the MSB is at the root. As an example, the binary tree in Figure 9.4 represents the numbers $(1000)_2$, $(1001)_2$, $(10110)_2$, $(110011)_2$, $(11000)_2$, and $(1100)_2$.

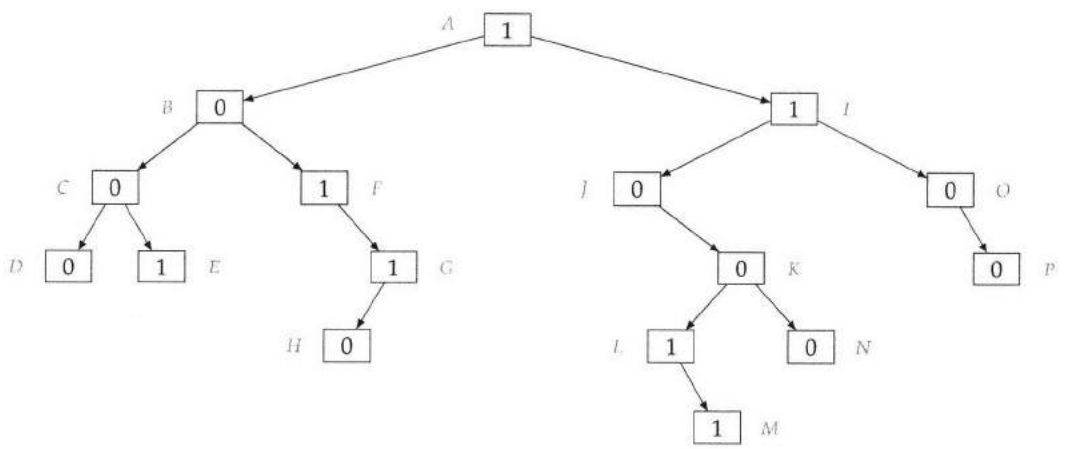


Figure 9.4: Binary tree encoding integers.

Design an algorithm to compute the sum of the binary numbers represented by the root-to-leaf paths.

Hint: Think of an appropriate way of traversing the tree.

Solution: Here is a brute-force algorithm. We compute the leaves, and store the child-parent mapping in a hash table, e.g., via an inorder walk. Afterwards, we traverse from each of the leaves to the root using the child-parent map. Each leaf-to-root path yields a binary integer, with the leaf's bit being the LSB. We sum these integers to obtain the result. The time complexity is $O(Lh)$, where L is the number of root-to-leaf paths (which equals the number of leaves), and h is the tree height. The space complexity is dominated by the hash table, namely $O(n)$, where n is the number of nodes.

The insight to improving complexity is to recognize that paths share nodes and that it is not necessary to repeat computations across the shared nodes. To compute the integer for the path from the root to any node, we take the integer for the node's parent, double it, and add the bit at that node. For example, the integer for the path from A to L is $2 \times (1100)_2 + 1 = (11001)_2$.

Therefore, we can compute the sum of all root to leaf node as follows. Each time we visit a node, we compute the integer it encodes using the number for its parent. If the node is a leaf we return its integer. If it is not a leaf, we return the sum of the results from its left and right children.

```
def sum_root_to_leaf(tree, partial_path_sum=0):
    if not tree:
        return 0

    partial_path_sum = partial_path_sum * 2 + tree.data
    if not tree.left and not tree.right:  # Leaf.
        return partial_path_sum
    # Non-leaf.
    return (sum_root_to_leaf(tree.left, partial_path_sum) + sum_root_to_leaf(
        tree.right, partial_path_sum))
```

The time complexity and space complexity are $O(n)$ and $O(h)$, respectively.

9.6 Find a root to leaf path with specified sum

You are given a binary tree where each node is labeled with an integer. The path weight of a node in such a tree is the sum of the integers on the unique path from the root to that node. For the example shown in Figure 9.1 on Page 112, the path weight of E is 591.

Write a program which takes as input an integer and a binary tree with integer node weights, and checks if there exists a leaf whose path weight equals the given integer.

Hint: What do you need to know about the rest of the tree when checking a specific subtree?

Solution: The brute-force algorithm in Solution 9.5 on the preceding page can be directly applied to this problem, and it has the same time complexity, namely, $O(Lh)$, where L is the number of root-to-leaf paths (which equals the number of leaves), and h is the tree height. The space complexity is dominated by the hash table, namely $O(n)$, where n is the number of nodes.

The inefficiency in the brute-force algorithm stems from the fact that we have overlapping paths, and we do not share the summation computation across those overlaps.

A better approach is to traverse the tree, keeping track of difference of the root-to-node path sum and the target value—call this the remaining weight. As soon as we encounter a leaf and the remaining weight is equal to the leaf's weight, we return true. Short circuit evaluation of the check ensures that we do not process additional leaves.

```
def has_path_sum(tree, remaining_weight):
    if not tree:
        return False
    if not tree.left and not tree.right:  # Leaf.
        return remaining_weight == tree.data
    # Non-leaf.
    return (has_path_sum(tree.left, remaining_weight - tree.data)
            or has_path_sum(tree.right, remaining_weight - tree.data))
```

The time complexity and space complexity are $O(n)$ and $O(h)$, respectively.

Variant: Write a program which takes the same inputs as in Problem 9.6 and returns all the paths to leaves whose weight equals s. For example, if $s = 619$, you should return $\langle\langle A, B, C, D\rangle, \langle A, I, O, P\rangle\rangle$.

9.7 Implement an inorder traversal without recursion

This problem is concerned with traversing nodes in a binary tree in an inorder fashion. See Page 113 for details and examples of these traversals. Generally speaking, a traversal computation is easy to implement if recursion is allowed.

Write a program which takes as input a binary tree and performs an inorder traversal of the tree. Do not use recursion. Nodes do not contain parent references.

Hint: Simulate the function call stack.

Solution: The recursive solution is trivial—first traverse the left subtree, then visit the root, and finally traverse the right subtree. This algorithm can be converted into a iterative algorithm by using an explicit stack. Several implementations are possible; the one below is noteworthy in that it pushes the current node, and not its right child. Furthermore, it does not use a visited field.

```
def inorder_traversal(tree):
    s, result = [], []

    while s or tree:
        if tree:
            s.append(tree)
            # Going left.
            tree = tree.left
        else:
            # Going up.
            tree = s.pop()
            result.append(tree.data)
            # Going right.
            tree = tree.right
    return result
```

For the binary tree in Figure 9.1 on Page 112, the first few stack states are $\langle A \rangle$, $\langle A, B \rangle$, $\langle A, B, C \rangle$, $\langle A, B, C, D \rangle$, $\langle A, B, C \rangle$, $\langle A, B, D \rangle$, $\langle A, B \rangle$, $\langle A \rangle$, $\langle A, F \rangle$.

The time complexity is $O(n)$, since the total time spent on each node is $O(1)$. The space complexity is $O(h)$, where h is the height of the tree. This space is allocated dynamically, specifically it is the maximum depth of the function call stack for the recursive implementation. See Page 113 for a definition of tree height.

9.8 Implement a preorder traversal without recursion

This problem is concerned with traversing nodes in a binary tree in preorder fashion. See Page 113 for details and examples of these traversals. Generally speaking, a traversal computation is easy to implement if recursion is allowed.

Write a program which takes as input a binary tree and performs a preorder traversal of the tree. Do not use recursion. Nodes do not contain parent references.

Solution:

We can get intuition as to the best way to perform a preorder traversal without recursion by noting that a preorder traversal visits nodes in a last in, first out order. We can perform the preorder traversal using a stack of tree nodes. The stack is initialized to contain the root. We visit a node by

popping it, adding first its right child, and then its left child to the stack. (We add the left child after the right child, since we want to continue with the left child.)

For the binary tree in Figure 9.1 on Page 112, the first few stack states are $\langle A \rangle$, $\langle I, B \rangle$, $\langle I, F, C \rangle$, $\langle I, F, E, D \rangle$, $\langle I, F, E \rangle$, $\langle I, F \rangle$, $\langle I, G \rangle$, $\langle I, H \rangle$, and $\langle I \rangle$. (The top of the stack is the rightmost node in the sequences.)

```
def preorder_traversal(tree):
    path, result = [tree], []
    while path:
        curr = path.pop()
        if curr:
            result.append(curr.data)
            path += [curr.right, curr.left]
    return result
```

Since we push and pop each node exactly once, the time complexity is $O(n)$, where n is the number of nodes. The space complexity is $O(h)$, where h is the height of the tree, since, with the possible exception of the top of the stack, the nodes in the stack correspond to the right children of the nodes on a path beginning at the root.

9.9 Compute the kth node in an inorder traversal

It is trivial to find the kth node that appears in an inorder traversal with $O(n)$ time complexity, where n is the number of nodes. However, with additional information on each node, you can do better.

Write a program that efficiently computes the kth node appearing in an inorder traversal. Assume that each node stores the number of nodes in the subtree rooted at that node.

Hint: Use the divide and conquer principle.

Solution: The brute-force approach is to perform an inorder walk, keeping track of the number of visited nodes, stopping when the node being visited is the kth one. The time complexity is $O(n)$. (Consider for example, a left-skewed tree—to get the first node ($k = 1$) we have to pass through all the nodes.)

Looking carefully at the brute-force algorithm, observe that it does not take advantage of the information present in the node. For example, if k is greater than the number of nodes in the left subtree, the kth node cannot lie in the left subtree. More precisely, if the left subtree has L nodes, then the kth node in the original tree is the $(k - L)$th node when we skip the left subtree. Conversely, if $k \leq L$, the desired node lies in the left subtree. For example, the left subtree in Figure 9.1 on Page 112 has seven nodes, so the tenth node cannot be in the left subtree. Instead it is the third node if we skip the left subtree. This observation leads to the following program.

```
def find_kth_node_binary_tree(tree, k):
    while tree:
        left_size = tree.left.size if tree.left else 0
        if left_size + 1 < k:  # k-th node must be in right subtree of tree.
            k -= left_size + 1
            tree = tree.right
        elif left_size == k - 1:  # k-th is iter itself.
            return tree
        else:  # k-th node must be in left subtree of iter.
            tree = tree.left
```

```
    return None  # If k is between 1 and the tree size, this is unreachable.
```

Since we descend the tree in each iteration, the time complexity is $O(h)$, where h is the height of the tree.

9.10 Compute the successor

The successor of a node in a binary tree is the node that appears immediately after the given node in an inorder traversal. For example, in Figure 9.1 on Page 112, the successor of G is A, and the successor of A is J.

Design an algorithm that computes the successor of a node in a binary tree. Assume that each node stores its parent.

Hint: Study the node's right subtree. What if the node does not have a right subtree?

Solution: The brute-force algorithm is to perform the inorder walk, stopping immediately at the first node to be visited after the given node. The time complexity is that of an inorder walk, namely $O(n)$, where n is the number of nodes.

Looking more carefully at the structure of the tree, observe that if the given node has a nonempty right subtree, its successor must lie in that subtree, and the rest of the nodes are immaterial. For example, in Figure 9.1 on Page 112, regardless of the structure of A's left subtree, A's successor must lie in the subtree rooted at I. Similarly, B's successor must lie in the subtree rooted at F. Furthermore, when a node has a nonempty right subtree, its successor is the first node visited when performing an inorder traversal on that subtree. This node is the "left-most" node in that subtree, and can be computed by following left children exclusively, stopping when there is no left child to continue from.

The challenge comes when the given node does not have a right subtree, e.g., H in Figure 9.1 on Page 112. If the node is its parent's left child, the parent will be the next node we visit, and hence is its successor, e.g., G is H's successor. If the node is its parent's right child, e.g., G, then we have already visited the parent. We can determine the next visited node by iteratively following parents, stopping when we move up from a left child. For example, from G we traverse F, then B, then A. We stop at A, since B is the left child of A—the successor of G is A.

Note that we may reach the root without ever moving up from a left child. This happens when the given node is the last node visited in an inorder traversal, and hence has no successor. Node P in Figure 9.1 on Page 112 illustrates this scenario.

```
def find_successor(node):
    if node.right:
        # Successor is the leftmost element in node's right subtree.
        node = node.right
        while node.left:
            node = node.left
        return node

    # Find the closest ancestor whose left subtree contains node.
    while node.parent and node.parent.right is node:
        node = node.parent

    # A return value of None means node does not have successor, i.e., node is
```

```
    # the rightmost node in the tree.
    return node.parent
```

Since the number of edges followed cannot be more than the tree height, the time complexity is $O(h)$, where h is the height of the tree.

9.11 Implement an inorder traversal with $O(1)$ space

The direct implementation of an inorder traversal using recursion has $O(h)$ space complexity, where h is the height of the tree. Recursion can be removed with an explicit stack, but the space complexity remains $O(h)$.

Write a nonrecursive program for computing the inorder traversal sequence for a binary tree. Assume nodes have parent fields.

Hint: How can you tell whether a node is a left child or right child of its parent?

Solution: The standard idiom for an inorder traversal is traverse-left, visit-root, traverse-right. When we complete traversing a subtree we need to return to its parent. What we do after that depends on whether the subtree we returned from was the left subtree or right subtree of the parent. In the former, we visit the parent, and then its right subtree; in the latter, we return from the parent itself.

One way to do this traversal without recursion is to record the parent node for each node we begin a traversal from. This can be done with a hash table, and entails $O(n)$ time and space complexity for the hash table, where n is the number of nodes, and h the height of the tree. The space complexity can be reduced to $O(h)$ by evicting a node from the hash table when we complete traversing the subtree rooted at it.

For the given problem, since each node stores its parent, we do not need the hash table, which improves the space complexity to $O(1)$.

To complete this algorithm, we need to know when we return to a parent if the just completed subtree was the parent's left child (in which case we need to visit the parent and then traverse its right subtree) or a right subtree (in which case we have completed traversing the parent). We achieve this by recording the subtree's root before we move to the parent. We can then compare the subtree's root with the parent's left child. For example, for the tree in Figure 9.1 on Page 112, after traversing the subtree rooted at C, when we return to B, we record C. Since C is B's left child, we still need to traverse B's right child. When we return from F to B, we record F. Since F is not B's left child, it must be B's right child, and we are done traversing B.

```
def inorder_traversal(tree):
    prev, result = None, []
    while tree:
        if prev is tree.parent:
            # We came down to tree from prev.
            if tree.left:  # Keep going left.
                next = tree.left
            else:
                result.append(tree.data)
                # Done with left, so go right if right is not empty. Otherwise,
                # go up.
                next = tree.right or tree.parent
```

```
        elif tree.left is prev:
            # We came up to tree from its left child.
            result.append(tree.data)
            # Done with left, so go right if right is not empty. Otherwise, go
            # up.
            next = tree.right or tree.parent
        else:  # Done with both children, so move up.
            next = tree.parent

        prev, tree = tree, next
    return result
```

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

Alternatively, since the successor of a node is the node appearing after it in an inorder visit sequence, we could start with the left-most node, and keep calling successor. This enables us to reuse Solution 9.10 on Page 123.

Variant: How would you perform preorder and postorder traversals iteratively using $O(1)$ additional space? Your algorithm cannot modify the tree. Nodes have an explicit parent field.

9.12 Reconstruct a binary tree from traversal data

Many different binary trees yield the same sequence of keys in an inorder, preorder, or postorder traversal. However, given an inorder traversal and one of any two other traversal orders of a binary tree, there exists a unique binary tree that yields those orders, assuming each node holds a distinct key. For example, the unique binary tree whose inorder traversal sequence is $\langle F, B, A, E, H, C, D, I, G\rangle$ and whose preorder traversal sequence is $\langle H, B, F, E, A, C, D, G, I\rangle$ is given in Figure 9.5.

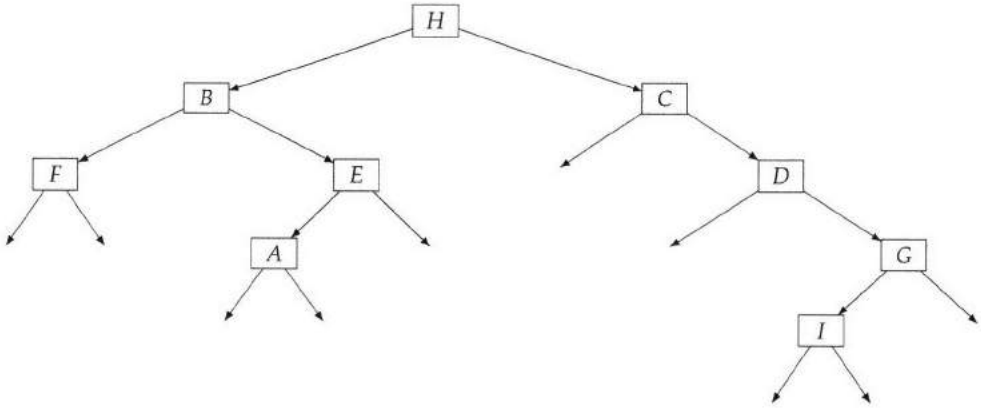


Figure 9.5: A binary tree—edges that do not terminate in nodes denote empty subtrees.

Given an inorder traversal sequence and a preorder traversal sequence of a binary tree write a program to reconstruct the tree. Assume each node has a unique key.

Hint: Focus on the root.

Solution: A truly brute-force approach is to enumerate every binary tree on the inorder traversal sequence, and check if the preorder sequence from that tree is the given one. The complexity is enormous.

Looking more carefully at the example, the preorder sequence gives us the key of the root node—it is the first node in the sequence. This in turn allows us to split the inorder sequence into an inorder sequence for the left subtree, followed by the root, followed by the right subtree.

The next insight is that we can use the left subtree inorder sequence to compute the preorder sequence for the left subtree from the preorder sequence for the entire tree. A preorder traversal sequence consists of the root, followed by the preorder traversal sequence of the left subtree, followed by the preorder traversal sequence of the right subtree. We know the number k of nodes in the left subtree from the location of the root in the inorder traversal sequence. Therefore, the subsequence of k nodes after the root in the preorder traversal sequence is the preorder traversal sequence for the left subtree.

As a concrete example, for the inorder traversal sequence $\langle F, B, A, E, H, C, D, I, G\rangle$ and preorder traversal sequence $\langle H, B, F, E, A, C, D, G, I\rangle$ (in Figure 9.5 on the preceding page) the root is the first node in the preorder traversal sequence, namely H. From the inorder traversal sequence, we know the inorder traversal sequence for the root's left subtree is $\langle F, B, A, E\rangle$. Therefore the sequence $\langle B, F, E, A\rangle$, which is the four nodes after the root, H, in the preorder traversal sequence $\langle H, B, F, E, A, C, D, G, I\rangle$ is the preorder traversal sequence for the root's left subtree. A similar construction applies to the root's right subtree. This construction is continued recursively till we get to the leaves.

Implemented naively, the above algorithm has a time complexity of $O(n^2)$. The worst-case corresponds to a skewed tree. Finding the root within the inorder sequences takes time $O(n)$. We can improve the time complexity by initially building a hash table from keys to their positions in the inorder sequence. This is the approach described below.

```
def binary_tree_from_preorder_inorder(preorder, inorder):
    node_to_inorder_idx = {data: i for i, data in enumerate(inorder)}

    # Builds the subtree with preorder[preorder_start:preorder_end] and
    # inorder[inorder_start:inorder_end].
    def binary_tree_from_preorder_inorder_helper(preorder_start, preorder_end,
                                                 inorder_start, inorder_end):
        if preorder_end <= preorder_start or inorder_end <= inorder_start:
            return None

        root_inorder_idx = node_to_inorder_idx[preorder[preorder_start]]
        left_subtree_size = root_inorder_idx - inorder_start
        return BinaryTreeNode(
            preorder[preorder_start],
            # Recursively builds the left subtree.
            binary_tree_from_preorder_inorder_helper(
                preorder_start + 1, preorder_start + 1 + left_subtree_size,
                inorder_start, root_inorder_idx),
            # Recursively builds the right subtree.
            binary_tree_from_preorder_inorder_helper(
                preorder_start + 1 + left_subtree_size, preorder_end,
                root_inorder_idx + 1, inorder_end))

    return binary_tree_from_preorder_inorder_helper(0,
```

```
                                    len(preorder), 0,
                                    len(inorder))
```

The time complexity is $O(n)$—building the hash table takes $O(n)$ time and the recursive reconstruction spends $O(1)$ time per node. The space complexity is $O(n + h) = O(n)$—the size of the hash table plus the maximum depth of the function call stack.

Variant: Solve the same problem with an inorder traversal sequence and a postorder traversal sequence.

Variant: Let A be an array of n distinct integers. Let the index of the maximum element of A be m. Define the max-tree on A to be the binary tree on the entries of A in which the root contains the maximum element of A, the left child is the max-tree on $A[0, m-1]$ and the right child is the max-tree on $A[m+1, n-1]$. Design an $O(n)$ algorithm for building the max-tree of A.

9.13 Reconstruct a binary tree from a preorder traversal with markers

Many different binary trees have the same preorder traversal sequence.

In this problem, the preorder traversal computation is modified to mark where a left or right child is empty. For example, the binary tree in Figure 9.5 on Page 125 yields the following preorder traversal sequence:

$$\langle H, B, F, \texttt{null}, \texttt{null}, E, A, \texttt{null}, \texttt{null}, \texttt{null}, C, \texttt{null}, D, \texttt{null}, G, I, \texttt{null}, \texttt{null}, \texttt{null} \rangle$$

Design an algorithm for reconstructing a binary tree from a preorder traversal visit sequence that uses `null` to mark empty children.

Hint: It's difficult to solve this problem by examining the preorder traversal visit sequence from left-to-right.

Solution: One brute-force approach is to enumerate all binary trees and compare the resulting preorder sequence with the given one. This approach will have unacceptable time complexity.

The intuition for a better algorithm is the recognition that the first node in the sequence is the root, and the sequence for the root's left subtree appears before all the nodes in the root's right subtree. It is not easy to see where the left subtree sequence ends. However, if we solve the problem recursively, we can assume that the routine correctly computes the left subtree, which will also tell us where the right subtree begins.

```
def reconstruct_preorder(preorder):
    def reconstruct_preorder_helper(preorder_iter):
        subtree_key = next(preorder_iter)
        if subtree_key is None:
            return None

        # Note that reconstruct_preorder_helper updates preorder_iter. So the
        # order of following two calls are critical.
        left_subtree = reconstruct_preorder_helper(preorder_iter)
        right_subtree = reconstruct_preorder_helper(preorder_iter)
        return BinaryTreeNode(subtree_key, left_subtree, right_subtree)

    return reconstruct_preorder_helper(iter(preorder))
```

The time complexity is $O(n)$, where n is the number of nodes in the tree.

Variant: Solve the same problem when the sequence corresponds to a postorder traversal sequence. Is this problem solvable when the sequence corresponds to an inorder traversal sequence?

9.14 Form a linked list from the leaves of a binary tree

In some applications of a binary tree, only the leaf nodes contain actual information. For example, the outcomes of matches in a tennis tournament can be represented by a binary tree where leaves are players. The internal nodes correspond to matches, with a single winner advancing. For such a tree, we can link the leaves to get a list of participants.

Given a binary tree, compute a linked list from the leaves of the binary tree. The leaves should appear in left-to-right order. For example, when applied to the binary tree in Figure 9.1 on Page 112, your function should return $\langle D, E, H, M, N, P \rangle$.

Hint: Build the list incrementally—it's easy if the partial list is a global.

Solution: A fairly direct approach is to use two passes—one to compute the leaves, and the other to assign ranks to the leaves, with the left-most leaf getting the lowest rank. The final result is the leaves sorted by ascending order of rank.

In fact, it is not necessary to make two passes—if we process the tree from left to right, the leaves occur in the desired order, so we can incrementally add them to the result. This idea is shown below.

```
def create_list_of_leaves(tree):
    if not tree:
        return []
    if not tree.left and not tree.right:
        return [tree]
    # First do the left subtree, and then do the right subtree.
    return create_list_of_leaves(tree.left) + create_list_of_leaves(tree.right)
```

The time complexity is $O(n)$, where n is the number of nodes.

9.15 Compute the exterior of a binary tree

The exterior of a binary tree is the following sequence of nodes: the nodes from the root to the leftmost leaf, followed by the leaves in left-to-right order, followed by the nodes from the rightmost leaf to the root. (By leftmost (rightmost) leaf, we mean the leaf that appears first (last) in an inorder traversal.) For example, the exterior of the binary tree in Figure 9.1 on Page 112 is $\langle A, B, C, D, E, H, M, N, P, O, I \rangle$.

Write a program that computes the exterior of a binary tree.

Hint: Handle the root's left child and right child in mirror fashion.

Solution: A brute-force approach is to use a case analysis. We need the nodes on the path from the root to the leftmost leaf, the nodes on the path from the root to the rightmost leaf, and the leaves in left-to-right order.

We already know how to compute the leaves in left-to-right order (Solution 9.14 on the facing page). The path from root to leftmost leaf is computed by going left if a left child exists, and otherwise going right. When we reach a leaf, it must be the leftmost leaf. A similar computation yields the nodes on the path from the root to the rightmost leaf. The time complexity is $O(h + n + h) = O(n)$, where n and h are the number of nodes and the height of the tree, respectively. The implementation is a little tricky, because some nodes appear in multiple sequences. For example, in Figure 9.1 on Page 112, the path from the root to the leftmost leaf is $\langle A, B, C, D\rangle$, the leaves in left-to-right order are $\langle D, E, H, M, N, P\rangle$, and the path from the root to the rightmost leaf is $\langle A, I, O, P\rangle$. Note the leftmost leaf, D, the rightmost leaf, P, and the root, A, appear in two sequences.

We can simplify the above approach by computing the nodes on the path from the root to the leftmost leaf and the leaves in the left subtree in one traversal. After that, we find the leaves in the right subtree followed by the nodes from the rightmost leaf to the root with another traversal. This is the program shown below. For the tree in Figure 9.1 on Page 112, the first traversal returns $\langle B, C, D, E, H\rangle$, and the second traversal returns $\langle M, N, P, O, I\rangle$. We append the first and then the second sequences to $\langle A\rangle$.

```
def exterior_binary_tree(tree):
    def is_leaf(node):
        return not node.left and not node.right

    # Computes the nodes from the root to the leftmost leaf followed by all
    # the leaves in subtree.
    def left_boundary_and_leaves(subtree, is_boundary):
        if not subtree:
            return []
        return (([subtree] if is_boundary
                 or is_leaf(subtree) else []) + left_boundary_and_leaves(
                     subtree.left, is_boundary) + left_boundary_and_leaves(
                         subtree.right, is_boundary and not subtree.left))

    # Computes the leaves in left-to-right order followed by the rightmost
    # leaf to the root path in subtree.
    def right_boundary_and_leaves(subtree, is_boundary):
        if not subtree:
            return []
        return (right_boundary_and_leaves(subtree.left, is_boundary
                                          and not subtree.right) +
                right_boundary_and_leaves(subtree.right, is_boundary) +
                ([subtree] if is_boundary or is_leaf(subtree) else []))

    return ([tree] + left_boundary_and_leaves(tree.left, is_boundary=True) +
            right_boundary_and_leaves(tree.right, is_boundary=True)
            if tree else [])
```

The time complexity is $O(n)$.

9.16 Compute the right sibling tree

For this problem, assume that each binary tree node has a extra field, call it level-next, that holds a binary tree node (this field is distinct from the fields for the left and right children). The level-next

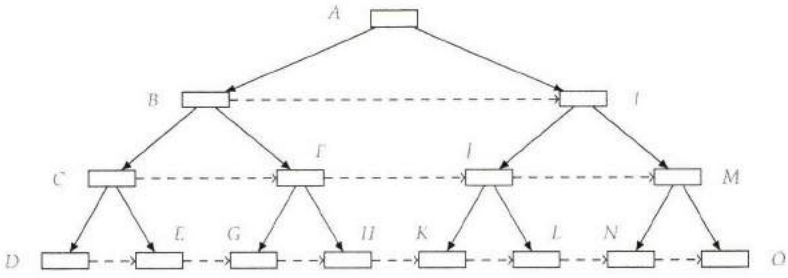


Figure 9.6: Assigning each node's level-next field to its right sibling in a perfect binary tree. A dashed arrow indicates the value held by the level-next field after the update. No dashed arrow is shown for the nodes on the path from the root to the rightmost leaf, i.e., A, I, M, and O, since these nodes have no right siblings.

field will be used to compute a map from nodes to their right siblings. The input is assumed to be perfect binary tree. See Figure 9.6 for an example.

Write a program that takes a perfect binary tree, and sets each node's level-next field to the node on its right, if one exists.

Hint: Think of an appropriate traversal order.

Solution: A brute-force approach is to compute the depth of each node, which is stored in a hash table. Next we order nodes at the same depth using inorder visit times. Then we set each node's level-next field according to this order. The time and space complexity are $O(n)$, where n is the number of nodes.

The key insight into solving this problem with better space complexity is to use the structure of the tree. Since it is a perfect binary tree, for a node which is a left child, its right sibling is just its parent's right child. For a node which is a right child, its right sibling is its parent's right sibling's left child. For example in Figure 9.6, since C is B's left child, C's right sibling is B's right child, i.e., F. Since Node F is B's right child, F's right sibling is B's right sibling's left child, i.e., J.

For this approach to work, we need to ensure that we process nodes level-by-level, left-to-right. Traversing a level in which the level-next field is set is trivial. As we do the traversal, we set the level-next fields for the nodes on the level below using the principle outlined above. To get to the next level, we record the starting node for each level. When we complete that level, the next level is the starting node's left child.

```
def construct_right_sibling(tree):
    def populate_children_next_field(start_node):
        while start_node and start_node.left:
            # Populate left child's next field.
            start_node.left.next = start_node.right
            # Populate right child's next field if iter is not the last node of
            # level.
            start_node.right.next = start_node.next and start_node.next.left
            start_node = start_node.next

    while tree and tree.left:
        populate_children_next_field(tree)
        tree = tree.left
```

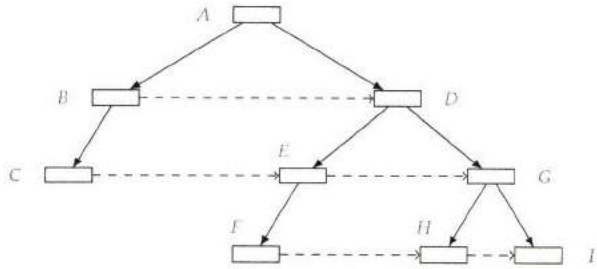


Figure 9.7: Assigning each node's level-next field to its right sibling in a general binary tree. A dashed arrow indicates the value held by the level-next field after the update.

Since we perform $O(1)$ computation per node, the time complexity is $O(n)$. The space complexity is $O(1)$.

Variant: Solve the same problem when there is no level-next field. Your result should be stored in the right child field.

Variant: Solve the same problem for a general binary tree. See Figure 9.7 for an example.

CHAPTER

10 Heaps

Using F-heaps we are able to obtain improved running times for several network optimization algorithms.

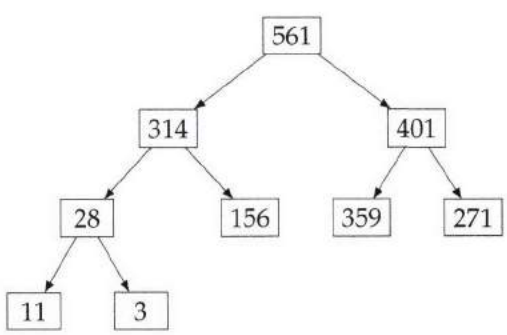
— "Fibonacci heaps and their uses,"
M. L. FREDMAN AND R. E. TARJAN, 1987

A *heap* is a specialized binary tree. Specifically, it is a complete binary tree as defined on Page 113. The keys must satisfy the *heap property*—the key at each node is at least as great as the keys stored at its children. See Figure 10.1(a) for an example of a max-heap. A max-heap can be implemented as an array; the children of the node at index i are at indices $2i + 1$ and $2i + 2$. The array representation for the max-heap in Figure 10.1(a) is $\langle 561, 314, 401, 28, 156, 359, 271, 11, 3 \rangle$.

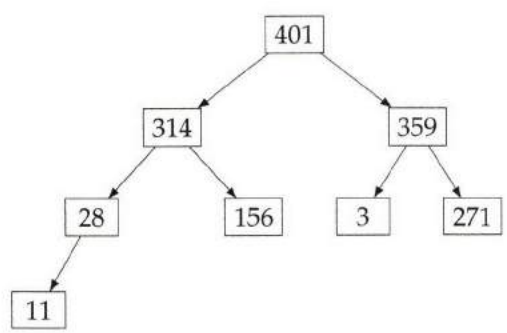
A max-heap supports $O(\log n)$ insertions, $O(1)$ time lookup for the max element, and $O(\log n)$ deletion of the max element. The extract-max operation is defined to delete and return the maximum element. See Figure 10.1(b) for an example of deletion of the max element. Searching for arbitrary keys has $O(n)$ time complexity.

A heap is sometimes referred to as a priority queue because it behaves like a queue, with one difference: each element has a "priority" associated with it, and deletion removes the element with the highest priority.

The *min-heap* is a completely symmetric version of the data structure and supports $O(1)$ time lookups for the minimum element.



(a) A max-heap. Note that the root holds the maximum key, 561.



(b) After the deletion of max of the heap in (a). Deletion is performed by replacing the root's key with the key at the last leaf and then recovering the heap property by repeatedly exchanging keys with children.

Figure 10.1: A max-heap and deletion on that max-heap.

Heaps boot camp

Suppose you were asked to write a program which takes a sequence of strings presented in "streaming" fashion: you cannot back up to read an earlier value. Your program must compute the k longest strings in the sequence. All that is required is the k longest strings—it is not required to order these strings.

As we process the input, we want to track the k longest strings seen so far. Out of these k strings, the string to be evicted when a longer string is to be added is the shortest one. A min-heap (not a max-heap!) is the right data structure for this application, since it supports efficient find-min, remove-min, and insert. In the program below we use a heap with a custom compare function, wherein strings are ordered by length.

```
def top_k(k, stream):
    # Entries are compared by their lengths.
    min_heap = [(len(s), s) for s in itertools.islice(stream, k)]
    heapq.heapify(min_heap)
    for next_string in stream:
        # Push next_string and pop the shortest string in min_heap.
        heapq.heappushpop(min_heap, (len(next_string), next_string))
    return [p[1] for p in heapq.nsmallest(k, min_heap)]
```

Each string is processed in $O(\log k)$ time, which is the time to add and to remove the minimum element from the heap. Therefore, if there are n strings in the input, the time complexity to process all of them is $O(n \log k)$.

We could improve best-case time complexity by first comparing the new string's length with the length of the string at the top of the heap (getting this string takes $O(1)$ time) and skipping the insert if the new string is too short to be in the set.

Use a heap when **all you care about** is the **largest** or **smallest** elements, and you **do not need** to support fast lookup, delete, or search operations for arbitrary elements. A heap is a good choice when you need to compute the k **largest or** k **smallest** elements in a collection. For the former, use a min-heap, for the latter, use a max-heap.

Table 10.1: Top Tips for Heaps

Know your heap libraries

Heap functionality in Python is provided by the `heapq` module. The operations and functions we will use are

- `heapq.heapify(L)`, which transforms the elements in `L` into a heap in-place,
- `heapq.nlargest(k, L)` (`heapq.nsmallest(k, L)`) returns the k largest (smallest) elements in `L`,
- `heapq.heappush(h, e)`, which pushes a new element on the heap,
- `heapq.heappop(h)`, which pops the smallest element from the heap,
- `heapq.heappushpop(h, a)`, which pushes `a` on the heap and then pops and returns the smallest element, and
- `e = h[0]`, which returns the smallest element on the heap without popping it.
-

It's very important to remember that `heapq` only provides min-heap functionality. If you need to build a max-heap on integers or floats, insert their negative to get the effect of a max-heap using

heapq. For objects, implement `__lt()__` appropriately. Problem 10.4 on Page 137 illustrates how use a max-heap.

10.1 Merge sorted files

This problem is motivated by the following scenario. You are given 500 files, each containing stock trade information for an S&P 500 company. Each trade is encoded by a line in the following format: `1232111,AAPL,30,456.12`.

The first number is the time of the trade expressed as the number of milliseconds since the start of the day's trading. Lines within each file are sorted in increasing order of time. The remaining values are the stock symbol, number of shares, and price. You are to create a single file containing all the trades from the 500 files, sorted in order of increasing trade times. The individual files are of the order of 5–100 megabytes; the combined file will be of the order of five gigabytes. In the abstract, we are trying to solve the following problem.

Write a program that takes as input a set of sorted sequences and computes the union of these sequences as a sorted sequence. For example, if the input is $\langle 3,5,7\rangle$, $\langle 0,6\rangle$, and $\langle 0,6,28\rangle$, then the output is $\langle 0,0,3,5,6,6,7,28\rangle$.

Hint: Which part of each sequence is significant as the algorithm executes?

Solution: A brute-force approach is to concatenate these sequences into a single array and then sort it. The time complexity is $O(n \log n)$, assuming there are n elements in total.

The brute-force approach does not use the fact that the individual sequences are sorted. We can take advantage of this fact by restricting our attention to the first remaining element in each sequence. Specifically, we repeatedly pick the smallest element amongst the first element of each of the remaining part of each of the sequences.

A min-heap is ideal for maintaining a collection of elements when we need to add arbitrary values and extract the smallest element.

For ease of exposition, we show how to merge sorted arrays, rather than files. As a concrete example, suppose there are three sorted arrays to be merged: $\langle 3,5,7\rangle$, $\langle 0,6\rangle$, and $\langle 0,6,28\rangle$. For simplicity, we show the min-heap as containing entries from these three arrays. In practice, we need additional information for each entry, namely the array it is from, and its index in that array. (In the file case we do not need to explicitly maintain an index for next unprocessed element in each sequence—the file I/O library tracks the first unread entry in the file.)

The min-heap is initialized to the first entry of each array, i.e., it is $\{3,0,0\}$. We extract the smallest entry, 0, and add it to the output which is $\langle 0\rangle$. Then we add 6 to the min-heap which is $\{3,0,6\}$ now. (We chose the 0 entry corresponding to the third array arbitrarily, it would be perfectly acceptable to choose from the second array.) Next, extract 0, and add it to the output which is $\langle 0,0\rangle$; then add 6 to the min-heap which is $\{3,6,6\}$. Next, extract 3, and add it to the output which is $\langle 0,0,3\rangle$; then add 5 to the min-heap which is $\{5,6,6\}$. Next, extract 5, and add it to the output which is $\langle 0,0,3,5\rangle$; then add 7 to the min-heap which is $\{7,6,6\}$. Next, extract 6, and add it to the output which is $\langle 0,0,3,5,6\rangle$; assuming 6 is selected from the second array, which has no remaining elements, the min-heap is $\{7,6\}$. Next, extract 6, and add it to the output which is $\langle 0,0,3,5,6,6\rangle$; then add 28 to the min-heap which is $\{7,28\}$. Next, extract 7, and add it to the output which is $\langle 0,0,3,5,6,6,7\rangle$; the min-heap is $\{28\}$. Next, extract 28, and add it to the output which is $\langle 0,0,3,5,6,6,7,28\rangle$; now, all elements are processed and the output stores the sorted elements.

```
def merge_sorted_arrays(sorted_arrays):
```

```
    min_heap = []
    # Builds a list of iterators for each array in sorted_arrays.
    sorted_arrays_iters = [iter(x) for x in sorted_arrays]

    # Puts first element from each iterator in min_heap.
    for i, it in enumerate(sorted_arrays_iters):
        first_element = next(it, None)
        if first_element is not None:
            heapq.heappush(min_heap, (first_element, i))

    result = []
    while min_heap:
        smallest_entry, smallest_array_i = heapq.heappop(min_heap)
        smallest_array_iter = sorted_arrays_iters[smallest_array_i]
        result.append(smallest_entry)
        next_element = next(smallest_array_iter, None)
        if next_element is not None:
            heapq.heappush(min_heap, (next_element, smallest_array_i))
    return result


# Pythonic solution, uses the heapq.merge() method which takes multiple inputs.
def merge_sorted_arrays_pythonic(sorted_arrays):
    return list(heapq.merge(*sorted_arrays))
```

Let k be the number of input sequences. Then there are no more than k elements in the min-heap. Both extract-min and insert take $O(\log k)$ time. Hence, we can do the merge in $O(n \log k)$ time. The space complexity is $O(k)$ beyond the space needed to write the final result. In particular, if the data comes from files and is written to a file, instead of arrays, we would need only $O(k)$ additional storage.

Alternatively, we could recursively merge the k files, two at a time using the merge step from merge sort. We would go from k to $k/2$ then $k/4$, etc. files. There would be $\log k$ stages, and each has time complexity $O(n)$, so the time complexity is the same as that of the heap-based approach, i.e., $O(n \log k)$. The space complexity of any reasonable implementation of merge sort would end up being $O(n)$, which is considerably worse than the heap based approach when $k \ll n$.

10.2 Sort an increasing-decreasing array

An array is said to be k-increasing-decreasing if elements repeatedly increase up to a certain index after which they decrease, then again increase, a total of k times. This is illustrated in Figure 10.2.

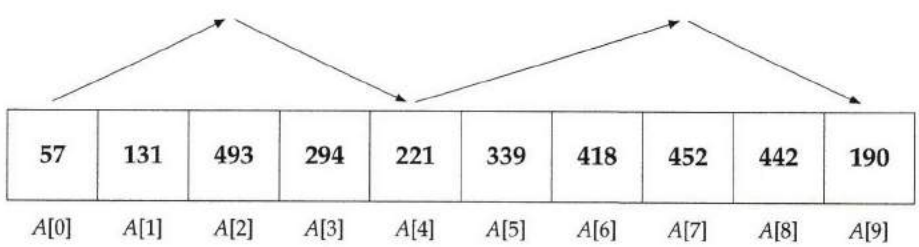


Figure 10.2: A 4-increasing-decreasing array.

Design an efficient algorithm for sorting a k-increasing-decreasing array.

Hint: Can you cast this in terms of combining k sorted arrays?

Solution: The brute-force approach is to sort the array, without taking advantage of the k-increasing-decreasing property. Sorting algorithms run in time $O(n \log n)$, where n is the length of the array.

If k is significantly smaller than n we can do better. For example, if $k = 2$, the input array consists of two subarrays, one increasing, the other decreasing. Reversing the second subarray yields two sorted arrays, and the result is their merge. It is fairly easy to merge two sorted arrays in $O(n)$ time.

Generalizing, we could first reverse the order of each of the decreasing subarrays. For the example in Figure 10.2 on the preceding page, we would decompose A into four sorted arrays—$\langle 57, 131, 493 \rangle$, $\langle 221, 294 \rangle$, $\langle 339, 418, 452 \rangle$, and $\langle 190, 442 \rangle$. Now we can use the techniques in Solution 10.1 on Page 134 to merge these.

```
def sort_k_increasing_decreasing_array(A):
    # Decomposes A into a set of sorted arrays.
    sorted_subarrays = []
    INCREASING, DECREASING = range(2)
    subarray_type = INCREASING
    start_idx = 0
    for i in range(1, len(A) + 1):
        if (i == len(A) or  # A is ended. Adds the last subarray.
            (A[i - 1] < A[i] and subarray_type == DECREASING) or
            (A[i - 1] >= A[i] and subarray_type == INCREASING)):
            sorted_subarrays.append(A[start_idx:i] if subarray_type ==
                                    INCREASING else A[i - 1:start_idx - 1:-1])
            start_idx = i
            subarray_type = (DECREASING
                             if subarray_type == INCREASING else INCREASING)
    return merge_sorted_arrays(sorted_subarrays)


# Pythonic solution, uses a stateful object to trace the monotonic subarrays.
def sort_k_increasing_decreasing_array_pythonic(A):
    class Monotonic:
        def __init__(self):
            self._last = float('-inf')

        def __call__(self, curr):
            res = curr < self._last
            self._last = curr
            return res

    return merge_sorted_arrays([
        list(group)[::-1 if is_decreasing else 1]
        for is_decreasing, group in itertools.groupby(A, Monotonic())
    ])
```

Just as in Solution 10.1 on Page 134, the time complexity is $O(n \log k)$ time.

10.3 Sort an almost-sorted array

Often data is almost-sorted—for example, a server receives timestamped stock quotes and earlier quotes may arrive slightly after later quotes because of differences in server loads and network routes. In this problem we address efficient ways to sort such data.

Write a program which takes as input a very long sequence of numbers and prints the numbers in sorted order. Each number is at most k away from its correctly sorted position. (Such an array is sometimes referred to as being k-sorted.) For example, no number in the sequence $\langle 3, -1, 2, 6, 4, 5, 8 \rangle$ is more than 2 away from its final sorted position.

Hint: How many numbers must you read after reading the ith number to be sure you can place it in the correct location?

Solution: The brute-force approach is to put the sequence in an array, sort it, and then print it. The time complexity is $O(n \log n)$, where n is the length of the input sequence. The space complexity is $O(n)$.

We can do better by taking advantage of the almost-sorted property. Specifically, after we have read $k+1$ numbers, the smallest number in that group must be smaller than all following numbers. For the given example, after we have read the first 3 numbers, $3, -1, 2$, the smallest, -1, must be globally the smallest. This is because the sequence was specified to have the property that every number is at most 2 away from its final sorted location and the smallest number is at index 0 in sorted order. After we read in the 4, the second smallest number must be the minimum of $3, 2, 4$, i.e., 2.

To solve this problem in the general setting, we need to store $k+1$ numbers and want to be able to efficiently extract the minimum number and add a new number. A min-heap is exactly what we need. We add the first k numbers to a min-heap. Now, we add additional numbers to the min-heap and extract the minimum from the heap. (When the numbers run out, we just perform the extraction.)

```
def sort_approximately_sorted_array(sequence, k):
    result = []
    min_heap = []
    # Adds the first k elements into min_heap. Stop if there are fewer than k
    # elements.
    for x in itertools.islice(sequence, k):
        heapq.heappush(min_heap, x)

    # For every new element, add it to min_heap and extract the smallest.
    for x in sequence:
        smallest = heapq.heappushpop(min_heap, x)
        result.append(smallest)

    # sequence is exhausted, iteratively extracts the remaining elements.
    while min_heap:
        smallest = heapq.heappop(min_heap)
        result.append(smallest)

    return result
```

The time complexity is $O(n \log k)$. The space complexity is $O(k)$.

10.4 Compute the k closest stars

Consider a coordinate system for the Milky Way, in which Earth is at $(0, 0, 0)$. Model stars as points, and assume distances are in light years. The Milky Way consists of approximately 10^{12} stars, and their coordinates are stored in a file.

How would you compute the k stars which are closest to Earth?

Hint: Suppose you know the k closest stars in the first n stars. If the $(n+1)$th star is to be added to the set of k closest stars, which element in that set should be evicted?

Solution: If RAM was not a limitation, we could read the data into an array, and compute the k smallest elements using sorting. Alternatively, we could use Solution 11.8 on Page 153 to find the kth smallest element, after which it is easy to find the k smallest elements. For both, the space complexity is $O(n)$, which, for the given dataset, cannot be stored in RAM.

Intuitively, we only care about stars close to Earth. Therefore, we can keep a set of candidates, and iteratively update the candidate set. The candidates are the k closest stars we have seen so far. When we examine a new star, we want to see if it should be added to the candidates. This entails comparing the candidate that is furthest from Earth with the new star. To find this candidate efficiently, we should store the candidates in a container that supports efficiently extracting the maximum and adding a new member.

A max-heap is perfect for this application. Conceptually, we start by adding the first k stars to the max-heap. As we process the stars, each time we encounter a new star that is closer to Earth than the star which is the furthest from Earth among the stars in the max-heap, we delete from the max-heap, and add the new one. Otherwise, we discard the new star and continue. We can simplify the code somewhat by simply adding each star to the max-heap, and discarding the maximum element from the max-heap once it contains $k+1$ elements.

```
class Star:
    def __init__(self, x, y, z):
        self.x, self.y, self.z = x, y, z

    @property
    def distance(self):
        return math.sqrt(self.x**2 + self.y**2 + self.z**2)

    def __lt__(self, rhs):
        return self.distance < rhs.distance



def find_closest_k_stars(stars, k):
    # max_heap to store the closest k stars seen so far.
    max_heap = []
    for star in stars:
        # Add each star to the max-heap. If the max-heap size exceeds k, remove
        # the maximum element from the max-heap.
        # As python has only min-heap, insert tuple (negative of distance, star)
        # to sort in reversed distance order.
        heapq.heappush(max_heap, (-star.distance, star))
        if len(max_heap) == k + 1:
            heapq.heappop(max_heap)

    # Iteratively extract from the max-heap, which yields the stars sorted
    # according from furthest to closest.
    return [s[1] for s in heapq.nlargest(k, max_heap)]
```

The time complexity is $O(n \log k)$ and the space complexity is $O(k)$.

Variant: Design an $O(n \log k)$ time algorithm that reads a sequence of n elements and for each element, starting from the kth element, prints the kth largest element read up to that point. The

length of the sequence is not known in advance. Your algorithm cannot use more than $O(k)$ additional storage. What are the worst-case inputs for your algorithm?

10.5 Compute the median of online data

You want to compute the running median of a sequence of numbers. The sequence is presented to you in a streaming fashion—you cannot back up to read an earlier value, and you need to output the median after reading in each new element. For example, if the input is $1, 0, 3, 5, 2, 0, 1$ the output is $1, 0.5, 1, 2, 2, 1.5, 1$.

Design an algorithm for computing the running median of a sequence.

Hint: Avoid looking at all values each time you read a new value.

Solution: The brute-force approach is to store all the elements seen so far in an array and compute the median using, for example Solution 11.8 on Page 153 for finding the kth smallest entry in an array. This has time complexity $O(n^2)$ for computing the running median for the first n elements.

The shortcoming of the brute-force approach is that it is not incremental, i.e., it does not take advantage of the previous computation. Note that the median of a collection divides the collection into two equal parts. When a new element is added to the collection, the parts can change by at most one element, and the element to be moved is the largest of the smaller half or the smallest of the larger half.

We can use two heaps, a max-heap for the smaller half and a min-heap for the larger half. We will keep these heaps balanced in size. The max-heap has the property that we can efficiently extract the largest element in the smaller part; the min-heap is similar.

For example, let the input values be $1, 0, 3, 5, 2, 0, 1$. Let L and H be the contents of the min-heap and the max-heap, respectively. Here is how they progress:

(1.) Read in 1: $L = [1], H = []$, median is 1.
(2.) Read in 0: $L = [1], H = [0]$, median is $(1 + 0)/2 = 0.5$.
(3.) Read in 3: $L = [1, 3], H = [0]$, median is 1.
(4.) Read in 5: $L = [3, 5], H = [1, 0]$, median is $(3 + 1)/2 = 2$.
(5.) Read in 2: $L = [2, 3, 5], H = [1, 0]$, median is 2.
(6.) Read in 0: $L = [2, 3, 5], H = [1, 0, 0]$, median is $(2 + 1)/2 = 1.5$.
(7.) Read in 1: $L = [1, 2, 3, 5], H = [1, 0, 0]$, median is 1.

```
def online_median(sequence):
    # min_heap stores the larger half seen so far.
    min_heap = []
    # max_heap stores the smaller half seen so far.
    # values in max_heap are negative
    max_heap = []
    result = []

    for x in sequence:
        heapq.heappush(max_heap, -heapq.heappushpop(min_heap, x))
        # Ensure min_heap and max_heap have equal number of elements if an even
        # number of elements is read; otherwise, min_heap must have one more
        # element than max_heap.
        if len(max_heap) > len(min_heap):
            heapq.heappush(min_heap, -heapq.heappop(max_heap))

        result.append(0.5 * (min_heap[0] + (-max_heap[0])))
```

```
                    if len(min_heap) == len(max_heap) else min_heap[0])
    return result
```

The time complexity per entry is $O(\log n)$, corresponding to insertion and extraction from a heap.

10.6 Compute the k largest elements in a max-heap

A heap contains limited information about the ordering of elements, so unlike a sorted array or a balanced BST, naive algorithms for computing the k largest elements have a time complexity that depends linearly on the number of elements in the collection.

Given a max-heap, represented as an array A, design an algorithm that computes the k largest elements stored in the max-heap. You cannot modify the heap. For example, if the heap is the one shown in Figure 10.1(a) on Page 132, then the array representation is $\langle 561, 314, 401, 28, 156, 359, 271, 11, 3\rangle$, the four largest elements are 561, 314, 401, and 359.

Solution: The brute-force algorithm is to perform k extract-max operations. The time complexity is $O(k \log n)$, where n is the number of elements in the heap. Note that this algorithm entails modifying the heap.

Another approach is to use an algorithm for finding the kth smallest element in an array, such as the one described in Solution 11.8 on Page 153. That has time complexity almost certain $O(n)$, and it too modifies the heap.

The following algorithm is based on the insight that the heap has partial order information, specifically, a parent node always stores value greater than or equal to the values stored at its children. Therefore, the root, which is stored in $A[0]$, must be one of the k largest elements—in fact, it is the largest element. The second largest element must be the larger of the root's children, which are $A[1]$ and $A[2]$—this is the index we continue processing from.

The ideal data structure for tracking the index to process next is a data structure which support fast insertions, and fast extract-max, i.e., in a max-heap. So our algorithm is to create a max-heap of candidates, initialized to hold the index 0, which serves as a reference to $A[0]$. The indices in the max-heap are ordered according to corresponding value in A. We then iteratively perform k extract-max operations from the max-heap. Each extraction of an index i is followed by inserting the indices of i's left child, $2i + 1$, and right child, $2i + 2$, to the max-heap, assuming these children exist.

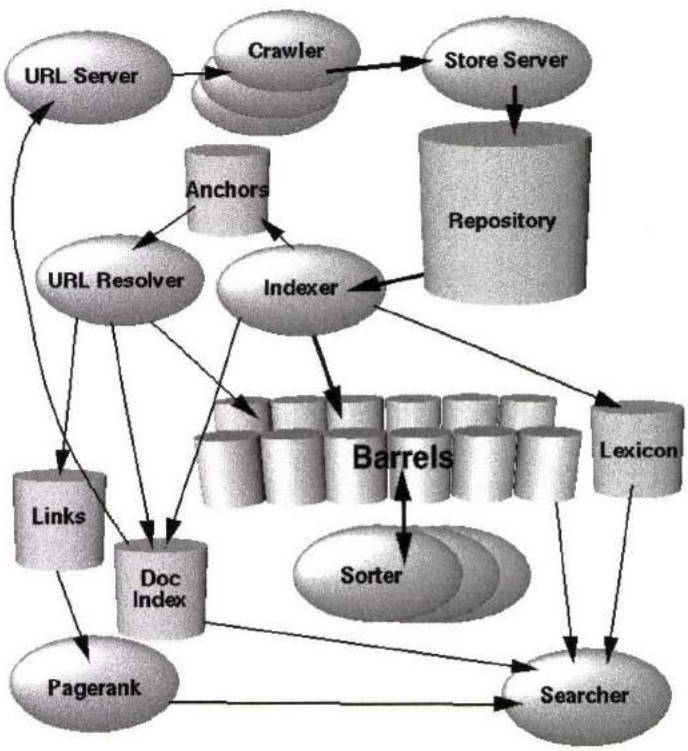
```
def k_largest_in_binary_heap(A, k):
    if k <= 0:
        return []

    # Stores the (-value, index)-pair in candidate_max_heap. This heap is
    # ordered by value field. Uses the negative of value to get the effect of
    # a max heap.
    candidate_max_heap = []
    # The largest element in A is at index 0.
    candidate_max_heap.append((-A[0], 0))
    result = []
    for _ in range(k):
        candidate_idx = candidate_max_heap[0][1]
        result.append(-heapq.heappop(candidate_max_heap)[0])

        left_child_idx = 2 * candidate_idx + 1
        if left_child_idx < len(A):
```

```
            heapq.heappush(candidate_max_heap, (-A[left_child_idx],
                                                left_child_idx))
        right_child_idx = 2 * candidate_idx + 2
        if right_child_idx < len(A):
            heapq.heappush(candidate_max_heap, (-A[right_child_idx],
                                                right_child_idx))
    return result
```

The total number of insertion and extract-max operations is $O(k)$, yielding an $O(k \log k)$ time complexity, and an $O(k)$ additional space complexity. This algorithm does not modify the original heap.

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 $O(n \log m)$.)

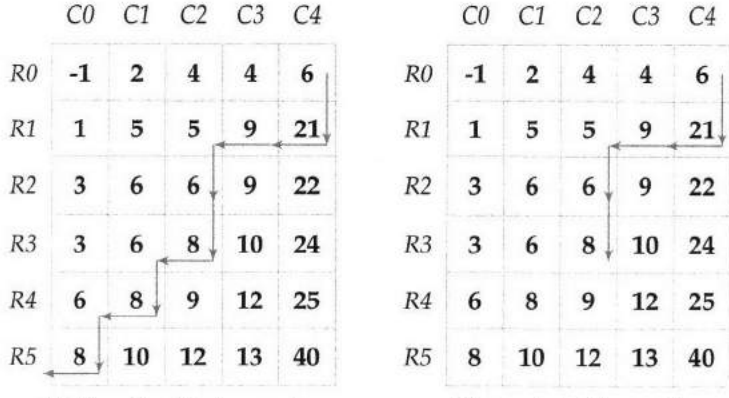
Note that the above approach fails to take advantage of the fact that both rows and columns are sorted—it treats separate rows independently of each other. For example, if $x < A[0][0]$ then no row or column can contain x—the sortedness property guarantees that $A[0][0]$ is the smallest element in A.

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 $O(1)$.

CHAPTER

12 Hash Tables

The new methods are intended to reduce the amount of space required to contain the hash-coded information from that associated with conventional methods. The reduction in space is accomplished by exploiting the possibility that a small fraction of errors of commission may be tolerable in some applications.

— *"Space/time trade-offs in hash coding with allowable errors,"* B. H. BLOOM, 1970

A hash table is a data structure used to store keys, optionally, with corresponding values. Inserts, deletes and lookups run in $O(1)$ time on average.

The underlying idea is to store keys in an array. A key is stored in the array locations ("slots") based on its "hash code". The hash code is an integer computed from the key by a hash function. If the hash function is chosen well, the objects are distributed uniformly across the array locations.

If two keys map to the same location, a "collision" is said to occur. The standard mechanism to deal with collisions is to maintain a linked list of objects at each array location. If the hash function does a good job of spreading objects across the underlying array and take $O(1)$ time to compute, on average, lookups, insertions, and deletions have $O(1 + n/m)$ time complexity, where n is the number of objects and m is the length of the array. If the "load" n/m grows large, rehashing can be applied to the hash table. A new array with a larger number of locations is allocated, and the objects are moved to the new array. Rehashing is expensive ($O(n + m)$ time) but if it is done infrequently (for example, whenever the number of entries doubles), its amortized cost is low.

A hash table is qualitatively different from a sorted array—keys do not have to appear in order, and randomization (specifically, the hash function) plays a central role. Compared to binary search trees (discussed in Chapter 14), inserting and deleting in a hash table is more efficient (assuming rehashing is infrequent). One disadvantage of hash tables is the need for a good hash function but this is rarely an issue in practice. Similarly, rehashing is not a problem outside of realtime systems and even for such systems, a separate thread can do the rehashing.

A hash function has one hard requirement—equal keys should have equal hash codes. This may seem obvious, but is easy to get wrong, e.g., by writing a hash function that is based on address rather than contents, or by including profiling data.

A softer requirement is that the hash function should "spread" keys, i.e., the hash codes for a subset of objects should be uniformly distributed across the underlying array. In addition, a hash function should be efficient to compute.

A common mistake with hash tables is that a key that's present in a hash table will be updated. The consequence is that a lookup for that key will now fail, even though it's still in the hash table. If you have to update a key, first remove it, then update it, and finally, add it back—this ensures it's moved to the correct array location. As a rule, you should avoid using mutable objects as keys.

Now we illustrate the steps for designing a hash function suitable for strings. First, the hash function should examine all the characters in the string. It should give a large range of values, and should not let one character dominate (e.g., if we simply cast characters to integers and multiplied them, a single 0 would result in a hash code of 0). We would also like a rolling hash function, one in which if a character is deleted from the front of the string, and another added to the end, the new hash code can be computed in $O(1)$ time (see Solution 6.13 on Page 80). The following function has these properties:

```
def string_hash(s, modulus):
    MULT = 997
    return functools.reduce(lambda v, c: (v * MULT + ord(c)) % modulus, s, 0)
```

A hash table is a good data structure to represent a dictionary, i.e., a set of strings. In some applications, a trie, which is a tree data structure that is used to store a dynamic set of strings, has computational advantages. Unlike a BST, nodes in the tree do not store a key. Instead, the node's position in the tree defines the key which it is associated with.

Hash tables boot camp

We introduce hash tables with two examples—one is an application that benefits from the algorithmic advantages of hash tables, and the other illustrates the design of a class that can be used in a hash table.

An application of hash tables

Anagrams are popular word play puzzles, whereby rearranging letters of one set of words, you get another set of words. For example, "eleven plus two" is an anagram for "twelve plus one". Crossword puzzle enthusiasts and Scrabble players benefit from the ability to view all possible anagrams of a given set of letters.

Suppose you were asked to write a program that takes as input a set of words and returns groups of anagrams for those words. Each group must contain at least two words.

For example, if the input is "debitcard", "elvis", "silent", "badcredit", "lives", "freedom", "listen", "levis", "money" then there are three groups of anagrams: (1.) "debitcard", "badcredit"; (2.) "elvis", "lives", "levis"; (3.) "silent", "listen". (Note that "money" does not appear in any group, since it has no anagrams in the set.)

Let's begin by considering the problem of testing whether one word is an anagram of another. Since anagrams do not depend on the ordering of characters in the strings, we can perform the test by sorting the characters in the string. Two words are anagrams if and only if they result in equal strings after sorting. For example, sort("logarithmic") and sort("algorithmic") are both "acghiilmort", so "logarithmic" and "algorithmic" are anagrams.

We can form the described grouping of strings by iterating through all strings, and comparing each string with all other remaining strings. If two strings are anagrams, we do not consider the second string again. This leads to an $O(n^2 m \log m)$ algorithm, where n is the number of strings and m is the maximum string length.

Looking more carefully at the above computation, note that the key idea is to map strings to a representative. Given any string, its sorted version can be used as a unique identifier for the anagram group it belongs to. What we want is a map from a sorted string to the anagrams it

corresponds to. Anytime you need to store a set of strings, a hash table is an excellent choice. Our final algorithm proceeds by adding sort(s) for each string s in the dictionary to a hash table. The sorted strings are keys, and the values are arrays of the corresponding strings from the original input.

```
def find_anagrams(dictionary):
    sorted_string_to_anagrams = collections.defaultdict(list)
    for s in dictionary:
        # Sorts the string, uses it as a key, and then appends the original
        # string as another value into hash table.
        sorted_string_to_anagrams[''.join(sorted(s))].append(s)

    return [
        group for group in sorted_string_to_anagrams.values() if len(group) >= 2
    ]
```

The computation consists of n calls to sort and n insertions into the hash table. Sorting all the keys has time complexity $O(nm \log m)$. The insertions add a time complexity of $O(nm)$, yielding $O(nm \log m)$ time complexity in total.

Variant: Design an $O(nm)$ algorithm for the same problem, assuming strings are made up of lower case English characters.

Design of a hashable class

Consider a class that represents contacts. For simplicity, assume each contact is a string. Suppose it is a hard requirement that the individual contacts are to be stored in a list and it's possible that the list contains duplicates. Two contacts should be equal if they contain the same set of strings, regardless of the ordering of the strings within the underlying list. Multiplicity is not important, i.e., three repetitions of the same contact is the same as a single instance of that contact. In order to be able to store contacts in a hash table, we first need to explicitly define equality, which we can do by forming sets from the lists and comparing the sets.

In our context, this implies that the hash function should depend on the strings present, but not their ordering; it should also consider only one copy if a string appears in duplicate form. It should be pointed out that the hash function and equals methods below are very inefficient. In practice, it would be advisable to cache the underlying set and the hash code, remembering to void these values on updates.

```
class ContactList:
    def __init__(self, names):
        '''
        names is a list of strings.
        '''
        self.names = names

    def __hash__(self):
        # Conceptually we want to hash the set of names. Since the set type is
        # mutable, it cannot be hashed. Therefore we use frozenset.
        return hash(frozenset(self.names))

    def __eq__(self, other):
        return set(self.names) == set(other.names)
```

```
def merge_contact_lists(contacts):
    '''
    contacts is a list of ContactList.
    '''
    return list(set(contacts))
```

The time complexity of computing the hash is $O(n)$, where n is the number of strings in the contact list. Hash codes are often cached for performance, with the caveat that the cache must be cleared if object fields that are referenced by the hash function are updated.

Hash tables have the **best theoretical and real-world performance** for lookup, insert and delete. Each of these operations has $O(1)$ time complexity. The $O(1)$ time complexity for insertion is for the average case—a single insert can take $O(n)$ if the hash table has to be resized.
Consider using a hash code as a **signature** to enhance performance, e.g., to filter out candidates.
Consider using a precomputed `lookup table` instead of boilerplate if-then code for mappings, e.g., from character to value, or character to character.
When defining your own type that will be put in a hash table, be sure you understand the relationship between **logical equality** and the fields the hash function must inspect. Specifically, anytime equality is implemented, it is imperative that the correct hash function is also implemented, otherwise when objects are placed in hash tables, logically equivalent objects may appear in different buckets, leading to lookups returning false, even when the searched item is present.
Sometimes you'll need a `multimap`, i.e., a map that contains multiple values for a single key, or a bi-directional map. If the language's standard libraries do not provide the functionality you need, learn how to implement a multimap using lists as values, or find a **third party library** that has a multimap.

Table 12.1: Top Tips for Hash Tables

Know your hash table libraries

There are multiple hash table-based data structures commonly used in Python—`set`, `dict`, `collections.defaultdict`, and `collections.Counter`. The difference between `set` and the other three is that is `set` simply stores keys, whereas the others store key-value pairs. All have the property that they do not allow for duplicate keys, unlike, for example, `list`.

In a `dict`, accessing value associated with a key that is not present leads to a `KeyError` exception. However, a `collections.defaultdict` returns the default value of the type that was specified when the collection was instantiated, e.g., if `d = collections.defaultdict(list)`, then if `k not in d` then `d[k]` is `[]`. A `collections.Counter` is used for counting the number of occurrences of keys, with a number of set-like operations, as illustrated below.

```
c = collections.Counter(a=3, b=1)
d = collections.Counter(a=1, b=2)
# add two counters together:  c[x] + d[x], collections.Counter({'a': 4, 'b': 3})
c + d
```

```
# subtract (keeping only positive counts), collections.Counter({'a': 2})
c - d
# intersection:  min(c[x], d[x]), collections.Counter({'a': 1, 'b': 1})
c & d
# union:  max(c[x], d[x]), collections.Counter({'a': 3, 'b': 2})
c | d
```

The most important operations for `set` are `s.add(42)`, `s.remove(42)`, `s.discard(123)`, `x in s`, as well as `s <= t` (is `s` a subset of `t`), and `s - t` (elements in `s` that are not in `t`).

The basic operations on the three key-value collections are similar to those on `set`. One difference is with iterators—iteration over a key-value collection yields the keys. To iterate over the key-value pairs, iterate over `items()`; to iterate over values, use `values()`. (The `keys()` method returns an iterator to the keys.)

Not every type is "hashable", i.e., can be added to a `set` or used as a key in a `dict`. In particular, mutable containers are not hashable—this is to prevent a client from modifying an object after adding it to the container, since the lookup will then fail to find it if the slot that the modified object hashes to is different.

Note that the built-in `hash()` function can greatly simplify the implementation of a hash function for a user-defined class, i.e., implementing `__hash(self)__`.

12.1 Test for palindromic permutations

A palindrome is a string that reads the same forwards and backwards, e.g., "level", "rotator", and "foobaraboof".

Write a program to test whether the letters forming a string can be permuted to form a palindrome. For example, "edified" can be permuted to form "deified".

Hint: Find a simple characterization of strings that can be permuted to form a palindrome.

Solution: A brute-force approach is to compute all permutations of the string, and test each one for palindromicity. This has a very high time complexity. Examining the approach in more detail, one thing to note is that if a string begins with say 'a', then we only need consider permutations that end with 'a'. This observation can be used to prune the permutation-based algorithm. However, a more powerful conclusion is that all characters must occur in pairs for a string to be permutable into a palindrome, with one exception, if the string is of odd length. For example, for the string "edified", which is of odd length (7) there are two 'e', two 'd's, two 'i's, and one 'f'—this is enough to guarantee that "edified" can be permuted into a palindrome.

More formally, if the string is of even length, a necessary and sufficient condition for it to be a palindrome is that each character in the string appears an even number of times. If the length is odd, all but one character should appear an even number of times. Both these cases are covered by testing that at most one character appears an odd number of times, which can be checked using a hash table mapping characters to frequencies.

```
def can_form_palindrome(s):
    # A string can be permuted to form a palindrome if and only if the number
    # of chars whose frequencies is odd is at most 1.
    return sum(v % 2 for v in collections.Counter(s).values()) <= 1
```

The time complexity is $O(n)$, where n is the length of the string. The space complexity is $O(c)$, where c is the number of distinct characters appearing in the string.

12.2 Is an anonymous letter constructible?

Write a program which takes text for an anonymous letter and text for a magazine and determines if it is possible to write the anonymous letter using the magazine. The anonymous letter can be written using the magazine if for each character in the anonymous letter, the number of times it appears in the anonymous letter is no more than the number of times it appears in the magazine.

Hint: Count the number of distinct characters appearing in the letter.

Solution: A brute force approach is to count for each character in the character set the number of times it appears in the letter and in the magazine. If any character occurs more often in the letter than the magazine we return false, otherwise we return true. This approach is potentially slow because it iterates over all characters, including those that do not occur in the letter or magazine. It also makes multiple passes over both the letter and the magazine—as many passes as there are characters in the character set.

A better approach is to make a single pass over the letter, storing the character counts for the letter in a single hash table—keys are characters, and values are the number of times that character appears. Next, we make a pass over the magazine. When processing a character c, if c appears in the hash table, we reduce its count by 1; we remove it from the hash when its count goes to zero. If the hash becomes empty, we return true. If we reach the end of the letter and the hash is nonempty, we return false—each of the characters remaining in the hash occurs more times in the letter than the magazine.

```
def is_letter_constructible_from_magazine(letter_text, magazine_text):
    # Compute the frequencies for all chars in letter_text.
    char_frequency_for_letter = collections.Counter(letter_text)

    # Checks if characters in magazine_text can cover characters in
    # char_frequency_for_letter.
    for c in magazine_text:
        if c in char_frequency_for_letter:
            char_frequency_for_letter[c] -= 1
            if char_frequency_for_letter[c] == 0:
                del char_frequency_for_letter[c]
                if not char_frequency_for_letter:
                    # All characters for letter_text are matched.
                    return True

    # Empty char_frequency_for_letter means every char in letter_text can be
    # covered by a character in magazine_text.
    return not char_frequency_for_letter


# Pythonic solution that exploits collections.Counter. Note that the
# subtraction only keeps keys with positive counts.
def is_letter_constructible_from_magazine_pythonic(letter_text, magazine_text):
    return (not collections.Counter(letter_text) -
```

```
        collections.Counter(magazine_text))
```

In the worst-case, the letter is not constructible or the last character of the magazine is essentially required. Therefore, the time complexity is $O(m + n)$ where m and n are the number of characters in the letter and magazine, respectively. The space complexity is the size of the hash table constructed in the pass over the letter, i.e., $O(L)$, where L is the number of distinct characters appearing in the letter.

If the characters are coded in ASCII, we could do away with the hash table and use a 256 entry integer array A, with $A[i]$ being set to the number of times the character i appears in the letter.

12.3 Implement an ISBN cache

The International Standard Book Number (ISBN) is a unique commercial book identifier. It is a string of length 10. The first 9 characters are digits; the last character is a check character. The check character is the sum of the first 9 digits, mod 11, with 10 represented by 'X'. (Modern ISBNs use 13 digits, and the check digit is taken mod 10; this problem is concerned with 10-digit ISBNs.)

Create a cache for looking up prices of books identified by their ISBN. You implement lookup, insert, and remove methods. Use the Least Recently Used (LRU) policy for cache eviction. If an ISBN is already present, insert should not change the price, but it should update that entry to be the most recently used entry. Lookup should also update that entry to be the most recently used entry.

Hint: Amortize the cost of deletion. Alternatively, use an auxiliary data structure.

Solution: Hash tables are ideally suited for fast lookups. We can use a hash table to quickly lookup price by using ISBNs as keys. Along with each key, we store a value, which is the price and the most recent time a lookup was done on that key.

This yields $O(1)$ lookup times on cache hits. Inserts into the cache are also $O(1)$ time, until the cache is full. Once the cache fills up, to add a new entry we have to find the LRU entry, which will be evicted to make place for the new entry. Finding this entry takes $O(n)$ time, where n is the cache size.

One way to improve performance is to use lazy garbage collection. Specifically, let's say we want the cache to be of size n. We do not delete any entries from the hash table until it grows to $2n$ entries. At this point we iterate through the entire hash table, and find the median age of items. Subsequently we discard everything below the median. The worst-case time to delete becomes $O(n)$ but it will happen at most once every n operations. Therefore, the amortized time to delete is $O(1)$. The drawback of this approach is the $O(n)$ time needed for some lookups that miss on a full cache, and the $O(n)$ increase in memory.

An alternative is to maintain a separate queue of keys. In the hash table we store for each key a reference to its location in the queue. Each time an ISBN is looked up and is found in the hash table, it is moved to the front of the queue. (This requires us to use a linked list implementation of the queue, so that items in the middle of the queue can be moved to the head.) When the length of the queue exceeds n, when a new element is added to the cache, the item at the tail of the queue is deleted from the cache, i.e., from the queue and the hash table.

```
class LRUCache:
    def __init__(self, capacity):
```

```
        self._isbn_price_table = collections.OrderedDict()
        self._capacity = capacity

    def lookup(self, isbn):
        if isbn not in self._isbn_price_table:
            return -1
        price = self._isbn_price_table.pop(isbn)
        self._isbn_price_table[isbn] = price
        return price

    def insert(self, isbn, price):
        # We add the value for key only if key is not present - we don't update
        # existing values.
        if isbn in self._isbn_price_table:
            price = self._isbn_price_table.pop(isbn)
        elif self._capacity <= len(self._isbn_price_table):
            self._isbn_price_table.popitem(last=False)
        self._isbn_price_table[isbn] = price

    def erase(self, isbn):
        return self._isbn_price_table.pop(isbn, None) is not None
```

The time complexity for each lookup is $O(1)$ for the hash table lookup and $O(1)$ for updating the queue, i.e., $O(1)$ overall.

12.4 Compute the LCA, optimizing for close ancestors

Problem 9.4 on Page 118 is concerned with computing the LCA in a binary tree with parent pointers in time proportional to the height of the tree. The algorithm presented in Solution 9.4 on Page 118 entails traversing all the way to the root even if the nodes whose LCA is being computed are very close to their LCA.

Design an algorithm for computing the LCA of two nodes in a binary tree. The algorithm's time complexity should depend only on the distance from the nodes to the LCA.

Hint: Focus on the extreme case described in the problem introduction.

Solution: The brute-force approach is to traverse upwards from the one node to the root, recording the nodes on the search path, and then traversing upwards from the other node, stopping as soon as we see a node on the path from the first node. The problem with this approach is that if the two nodes are far from the root, we end up traversing all the way to the root, even if the LCA is the parent of the two nodes, i.e., they are siblings. This is illustrated in by L and N in Figure 9.1 on Page 112.

Intuitively, the brute-force approach is suboptimal because it potentially processes nodes well above the LCA. We can avoid this by alternating moving upwards from the two nodes and storing the nodes visited as we move up in a hash table. Each time we visit a node we check to see if it has been visited before.

```
def lca(node_0, node_1):
    iter_0, iter_1 = node_0, node_1
    nodes_on_path_to_root = set()
```

```
    while iter_0 or iter_1:
        # Ascend tree in tandem for these two nodes.
        if iter_0:
            if iter_0 in nodes_on_path_to_root:
                return iter_0
            nodes_on_path_to_root.add(iter_0)
            iter_0 = iter_0.parent
        if iter_1:
            if iter_1 in nodes_on_path_to_root:
                return iter_1
            nodes_on_path_to_root.add(iter_1)
            iter_1 = iter_1.parent
    raise ValueError('node_0 and node_1 are not in the same tree')
```

Note that we are trading space for time. The algorithm for Solution 9.4 on Page 118 used $O(1)$ space and $O(h)$ time, whereas the algorithm presented above uses $O(D0 + D1)$ space and time, where $D0$ is the distance from the LCA to the first node, and $D1$ is the distance from the LCA to the second node. In the worst-case, the nodes are leaves whose LCA is the root, and we end up using $O(h)$ space and time, where h is the height of the tree.

12.5 Find the nearest repeated entries in an array

People do not like reading text in which a word is used multiple times in a short paragraph. You are to write a program which helps identify such a problem.

Write a program which takes as input an array and finds the distance between a closest pair of equal entries. For example, if $s = \langle$"All", "work", "and", "no", "play", "makes", "for", "no", "work", "no", "fun", "and", "no", "results"$\rangle$, then the second and third occurrences of "no" is the closest pair.

Hint: Each entry in the array is a candidate.

Solution: The brute-force approach is to iterate over all pairs of entries, check if they are the same, and if so, if the distance between them is less than the smallest such distance seen so far. The time complexity is $O(n^2)$, where n is the array length.

We can improve upon the brute-force algorithm by noting that when examining an entry, we do not need to look at every other entry—we only care about entries which are the same. We can store the set of indices corresponding to a given value using a hash table and iterate over all such sets. However, there is a better approach—when processing an entry, all we care about is the closest previous equal entry. Specifically, as we scan through the array, for each value seen so far, we store in a hash table the latest index at which it appears. When processing the element, we use the hash table to see the latest index less than the current index holding the same value.

For the given example, when processing the element at index 9, which is "no", the hash table tells us the most recent previous occurrence of "no" is at index 7, so we update the distance of the closest pair of equal entries seen so far to 2.

```
def find_nearest_repetition(paragraph):
    word_to_latest_index, nearest_repeated_distance = {}, float('inf')
    for i, word in enumerate(paragraph):
        if word in word_to_latest_index:
```

```
            latest_equal_word = word_to_latest_index[word]
            nearest_repeated_distance = min(nearest_repeated_distance,
                                            i - latest_equal_word)
        word_to_latest_index[word] = i
    return nearest_repeated_distance if nearest_repeated_distance != float(
        'inf') else -1
```

The time complexity is $O(n)$, since we perform a constant amount of work per entry. The space complexity is $O(d)$, where d is the number of distinct entries in the array.

12.6 Find the smallest subarray covering all values

When you type keywords in a search engine, the search engine will return results, and each result contains a digest of the web page, i.e., a highlighting within that page of the keywords that you searched for. For example, a search for the keywords "Union" and "save" on a page with the text of the Emancipation Proclamation should return the result shown in Figure 12.1.

> My paramount object in this struggle is to **save the Union**, and is not either to save or to destroy slavery. If I could save the Union without freeing any slave I would do it, and if I could save it by freeing all the slaves I would do it; and if I could save it by freeing some and leaving others alone I would also do that.

Figure 12.1: Search result with digest in boldface and search keywords underlined.

The digest for this page is the text in boldface, with the keywords underlined for emphasis. It is the shortest substring of the page which contains all the keywords in the search. The problem of computing the digest is abstracted as follows.

Write a program which takes an array of strings and a set of strings, and return the indices of the starting and ending index of a shortest subarray of the given array that "covers" the set, i.e., contains all strings in the set.

Hint: What is the maximum number of minimal subarrays that can cover the query?

Solution: The brute force approach is to iterate over all subarrays, testing if the subarray contains all strings in the set. If the array length is n, there are $O(n^2)$ subarrays. Testing whether the subarray contains each string in the set is an $O(n)$ operation using a hash table to record which strings are present in the subarray. The overall time complexity is $O(n^3)$.

We can improve the time complexity to $O(n^2)$ by growing the subarrays incrementally. Specifically, we can consider all subarrays starting at i in order of increasing length, stopping as soon as the set is covered. We use a hash table to record which strings in the set remain to be covered. Each time we increment the subarray length, we need $O(1)$ time to update the set of remaining strings.

We can further improve the algorithm by noting that when we move from i to $i+1$ we can reuse the work performed from i. Specifically, let's say the smallest subarray starting at i covering the set ends at j. There is no point in considering subarrays starting at $i+1$ and ending before j, since we know they cannot cover the set. When we advance to $i+1$, either we still cover the set, or we have to advance j to cover the set. We continuously advance one of i or j, which implies an $O(n)$ time complexity.

As a concrete example, consider the array ⟨*apple, banana, apple, apple, dog, cat, apple, dog, banana, apple, cat, dog*⟩ and the set {*banana, cat*}. The smallest subarray covering the set starting at 0 ends at 5. Next, we advance to 1. Since the element at 0 is not in the set, the smallest subarray covering the set still ends at 5. Next, we advance to 2. Now we do not cover the set, so we advance from 5 to 8—now the subarray from 2 to 8 covers the set. We update the start index from 2 to 3 to 4 to 5 and continue to cover the set. When we advance to 6, we no longer cover the set, so we advance the end index till we get to 10. We can advance the start index to 8 and still cover the set. After we move past 8, we cannot cover the set. The shortest subarray covering the set is from 8 to 10.

```
Subarray = collections.namedtuple('Subarray', ('start', 'end'))


def find_smallest_subarray_covering_set(paragraph, keywords):
    keywords_to_cover = collections.Counter(keywords)
    result = Subarray(-1, -1)
    remaining_to_cover = len(keywords)
    left = 0
    for right, p in enumerate(paragraph):
        if p in keywords:
            keywords_to_cover[p] -= 1
            if keywords_to_cover[p] >= 0:
                remaining_to_cover -= 1

        # Keeps advancing left until keywords_to_cover does not contain all
        # keywords.
        while remaining_to_cover == 0:
            if result == (-1, -1) or right - left < result[1] - result[0]:
                result = (left, right)
            pl = paragraph[left]
            if pl in keywords:
                keywords_to_cover[pl] += 1
                if keywords_to_cover[pl] > 0:
                    remaining_to_cover += 1
            left += 1
    return result
```

The complexity is $O(n)$, where n is the length of the array, since for each of the two indices we spend $O(1)$ time per advance, and each is advanced at most $n - 1$ times.

The disadvantage of this approach is that we need to keep the subarrays in memory. We can achieve a streaming algorithm by keeping track of latest occurrences of query keywords as we process A. We use a doubly linked list L to store the last occurrence (index) of each keyword in Q, and hash table H to map each keyword in Q to the corresponding node in L. Each time a word in Q is encountered, we remove its node from L (which we find by using H), create a new node which records the current index in A, and append the new node to the end of L. We also update H. By doing this, each keyword in L is ordered by its order in A; therefore, if L has n_Q words (i.e., all keywords are shown) and the current index minus the index stored in the first node in L is less than current best, we update current best. The complexity is still $O(n)$.

```
def find_smallest_subarray_covering_subset(stream, query_strings):
    class DoublyLinkedListNode:
        def __init__(self, data=None):
```

```
        self.data = data
        self.next = self.prev = None

class LinkedList:
    def __init__(self):
        self.head = self.tail = None
        self._size = 0

    def __len__(self):
        return self._size

    def insert_after(self, value):
        node = DoublyLinkedListNode(value)
        node.prev = self.tail
        if self.tail:
            self.tail.next = node
        else:
            self.head = node
        self.tail = node
        self._size += 1

    def remove(self, node):
        if node.next:
            node.next.prev = node.prev
        else:
            self.tail = node.prev
        if node.prev:
            node.prev.next = node.next
        else:
            self.head = node.next
        node.next = node.prev = None
        self._size -= 1

# Tracks the last occurrence (index) of each string in query_strings.
loc = LinkedList()
d = {s: None for s in query_strings}
result = Subarray(-1, -1)
for idx, s in enumerate(stream):
    if s in d:  # s is in query_strings.
        it = d[s]
        if it is not None:
            # Explicitly remove s so that when we add it, it's the string most
            # recently added to loc.
            loc.remove(it)
        loc.insert_after(idx)
        d[s] = loc.tail

        if len(loc) == len(query_strings):
            # We have seen all strings in query_strings, let's get to work.
            if (result == (-1, -1)
                    or idx - loc.head.data < result[1] - result[0]):
                result = (loc.head.data, idx)
return result
```

Variant: Given an array A, find a shortest subarray $A[i, j]$ such that each distinct value present in A is also present in the subarray.

Variant: Given an array A, rearrange the elements so that the shortest subarray containing all the distinct values in A has maximum possible length.

Variant: Given an array A and a positive integer k, rearrange the elements so that no two equal elements are k or less apart.

12.7 Find smallest subarray sequentially covering all values

In Problem 12.6 on Page 168 we did not differentiate between the order in which keywords appeared. If the digest has to include the keywords in the order in which they appear in the search textbox, we may get a different digest. For example, for the search keywords "Union" and "save", in that order, the digest would be "**Union, and is not either to save**".

Write a program that takes two arrays of strings, and return the indices of the starting and ending index of a shortest subarray of the first array (the "paragraph" array) that "sequentially covers", i.e., contains all the strings in the second array (the "keywords" array), in the order in which they appear in the keywords array. You can assume all keywords are distinct. For example, let the paragraph array be $\langle \text{apple}, \text{banana}, \text{cat}, \text{apple} \rangle$, and the keywords array be $\langle \text{banana}, \text{apple} \rangle$. The paragraph subarray starting at index 0 and ending at index 1 does not fulfill the specification, even though it contains all the keywords, since they do not appear in the specified order. On the other hand, the subarray starting at index 1 and ending at index 3 does fulfill the specification.

Hint: For each index in the paragraph array, compute the shortest subarray ending at that index which fulfills the specification.

Solution: The brute-force approach is to iterate over all subarrays of the paragraph array. To check whether a subarray of the paragraph array sequentially covers the keyword array, we search for the first occurrence of the first keyword. We never need to consider a later occurrence of the first keyword, since subsequent occurrences do not give us any additional power to cover the keywords. Next we search for the first occurrence of the second keyword that appears after the first occurrence of the first keyword. No earlier occurrence of the second keyword is relevant, since those occurrences can never appear in the correct order. This observation leads to an $O(n)$ time algorithm for testing whether a subarray fulfills the specification, where n is the length of the paragraph array. Since there are $O(n^2)$ subarrays of the paragraph array, the overall time complexity is $O(n^3)$.

The brute-force algorithm repeats work. We can improve the time complexity to $O(n^2)$ by computing for each index, the shortest subarray starting at that index which sequentially covers the keyword array. The idea is that we can compute the desired subarray by advancing from the start index and marking off the keywords in order.

The improved algorithm still repeats work—as we advance through the paragraph array, we can reuse our computation of the earliest occurrences of keywords. To do this, we need auxiliary data structures to record previous results.

Specifically, we use a hash table to map keywords to their most recent occurrences in the paragraph array as we iterate through it, and a hash table mapping each keyword to the length of the shortest subarray ending at the most recent occurrence of that keyword.

These two hash tables give us is the ability to determine the shortest subarray sequentially covering the first k keywords given the shortest subarray sequentially covering the first $k-1$ keywords.

When processing the ith string in the paragraph array, if that string is the jth keyword, we update the most recent occurrence of that keyword to i. The shortest subarray ending at i which sequentially covers the first j keywords consists of the shortest subarray ending at the most recent occurrence of the first $j-1$ keywords plus the elements from the most recent occurrence of the $(j-1)$th keyword to i. This computation is implemented below.

```
Subarray = collections.namedtuple('Subarray', ('start', 'end'))


def find_smallest_sequentially_covering_subset(paragraph, keywords):
    # Maps each keyword to its index in the keywords array.
    keyword_to_idx = {k: i for i, k in enumerate(keywords)}

    # Since keywords are uniquely identified by their indices in keywords
    # array, we can use those indices as keys to lookup in an array.
    latest_occurrence = [-1] * len(keywords)
    # For each keyword (identified by its index in keywords array), the length
    # of the shortest subarray ending at the most recent occurrence of that
    # keyword that sequentially cover all keywords up to that keyword.
    shortest_subarray_length = [float('inf')] * len(keywords)

    shortest_distance = float('inf')
    result = Subarray(-1, -1)
    for i, p in enumerate(paragraph):
        if p in keyword_to_idx:
            keyword_idx = keyword_to_idx[p]
            if keyword_idx == 0:  # First keyword.
                shortest_subarray_length[keyword_idx] = 1
            elif shortest_subarray_length[keyword_idx - 1] != float('inf'):
                distance_to_previous_keyword = (
                    i - latest_occurrence[keyword_idx - 1])
                shortest_subarray_length[keyword_idx] = (
                    distance_to_previous_keyword +
                    shortest_subarray_length[keyword_idx - 1])
            latest_occurrence[keyword_idx] = i

            # Last keyword, for improved subarray.
            if (keyword_idx == len(keywords) - 1
                    and shortest_subarray_length[-1] < shortest_distance):
                shortest_distance = shortest_subarray_length[-1]
                result = Subarray(i - shortest_distance + 1, i)
    return result
```

Processing each entry of the paragraph array entails a constant number of lookups and updates, leading to an $O(n)$ time complexity, where n is the length of the paragraph array. The additional space complexity is dominated by the three hash tables, i.e., $O(m)$, where m is the number of keywords.

Write a program that takes an array and returns the length of a longest subarray with the property that all its elements are distinct. For example, if the array is $\langle f, s, f, e, t, w, e, n, w, e \rangle$ then a longest subarray all of whose elements are distinct is $\langle s, f, e, t, w \rangle$.

Hint: What should you do if the subarray from indices i to j satisfies the property, but the subarray from i to $j+1$ does not?

Solution: We begin with a brute-force approach. For each subarray, we test if all its elements are distinct using a hash table. The time complexity is $O(n^3)$, where n is the array length since there are $O(n^2)$ subarrays, and their average length is $O(n)$.

We can improve on the brute-force algorithm by noting that if a subarray contains duplicates, every array containing that subarray will also contain duplicates. Therefore, for any given starting index, we can compute the longest subarray starting at that index containing no duplicates in time $O(n)$, since we can incrementally add elements to the hash table of elements from the starting index. This leads to an $O(n^2)$ algorithm. As soon as we get a duplicate, we cannot find a longer beginning at the same initial index that is duplicate-free.

We can improve the time complexity by reusing previous computation as we iterate through the array. Suppose we know the longest duplicate-free subarray ending at a given index. The longest duplicate-free subarray ending at the next index is either the previous subarray appended with the element at the next index, if that element does not appear in the longest duplicate-free subarray at the current index. Otherwise it is the subarray beginning at the most recent occurrence of the element at the next index to the next index. To perform this case analysis as we iterate, all we need is a hash table storing the most recent occurrence of each element, and the longest duplicate-free subarray ending at the current element.

For the given example, $\langle f, s, f, e, t, w, e, n, w, e \rangle$, when we process the element at index 2, the longest duplicate-free subarray ending at index 1 is from 0 to 1. The hash table tells us that the element at index 2, namely f, appears in that subarray, so we update the longest subarray ending at index 2 to being from index 1 to 2. Indices 3–5 introduce fresh elements. Index 6 holds a repeated value, e, which appears within the longest subarray ending at index 5; specifically, it appears at index 3. Therefore, the longest subarray ending at index 6 to start at index 4.

```
def longest_subarray_with_distinct_entries(A):
    # Records the most recent occurrences of each entry.
    most_recent_occurrence = {}
    longest_dup_free_subarray_start_idx = result = 0
    for i, a in enumerate(A):
        # Defer updating dup_idx until we see a duplicate.
        if a in most_recent_occurrence:
            dup_idx = most_recent_occurrence[a]
            # A[i] appeared before. Did it appear in the longest current
            # subarray?
            if dup_idx >= longest_dup_free_subarray_start_idx:
                result = max(result, i - longest_dup_free_subarray_start_idx)
                longest_dup_free_subarray_start_idx = dup_idx + 1
        most_recent_occurrence[a] = i
    return max(result, len(A) - longest_dup_free_subarray_start_idx)
```

The time complexity is $O(n)$, since we perform a constant number of operations per element.

12.9 Find the length of a longest contained interval

Write a program which takes as input a set of integers represented by an array, and returns the size of a largest subset of integers in the array having the property that if two integers are in the subset, then so are all integers between them. For example, if the input is $\langle 3, -2, 7, 9, 8, 1, 2, 0, -1, 5, 8\rangle$, the largest such subset is $\{-2, -1, 0, 1, 2, 3\}$, so you should return 6.

Hint: Do you really need a total ordering on the input?

Solution: The brute-force algorithm is to sort the array and then iterate through it, recording for each entry the largest subset with the desired property ending at that entry.

On closer inspection we see that sorting is not essential to the functioning of the algorithm. We do not need the total ordering—all we care are about is whether the integers adjacent to a given value are present. This suggests using a hash table to store the entries. Now we iterate over the entries in the array. If an entry e is present in the hash table, we compute the largest interval including e such that all values in the interval are contained in the hash table. We do this by iteratively searching entries in the hash table of the form $e+1, e+2, \ldots$, and $e-1, e-2, \ldots$. When we are done, to avoid doing duplicated computation we remove all the entries in the computed interval from the hash table, since all these entries are in the same largest contained interval.

As a concrete example, consider $A = \langle 10, 5, 3, 11, 6, 100, 4\rangle$. We initialize the hash table to $\{6, 10, 3, 11, 5, 100, 4\}$. The first entry in A is 10, and we find the largest interval contained in A including 10 by expanding from 10 in each direction by doing lookups in the hash table. The largest set is $\{10, 11\}$ and is of size 2. This computation updates the hash table to $\{6, 3, 5, 100, 4\}$. The next entry in A is 5. Since it is contained in the hash table, we know that the largest interval contained in A including 5 has not been computed yet. Expanding from 5, we see that $3, 4, 6$ are all in the hash table, and 2 and 7 are not in the hash table, so the largest set containing 5 is $\{3, 4, 5, 6\}$, which is of size 4. We update the hash table to $\{100\}$. The three entries after 5, namely $3, 11, 6$ are not present in the hash table, so we know we have already computed the longest intervals in A containing each of these. Then we get to 100, which cannot be extended, so the largest set containing it is $\{100\}$, which is of size 1. We update the hash table to $\{\}$. Since 4 is not in the hash table, we can skip it. The largest of the three sets is $\{3, 4, 5, 6\}$, i.e., the size of the largest contained interval is 4.

```
def longest_contained_range(A):
    # unprocessed_entries records the existence of each entry in A.
    unprocessed_entries = set(A)

    max_interval_size = 0
    while unprocessed_entries:
        a = unprocessed_entries.pop()

        # Finds the lower bound of the largest range containing a.
        lower_bound = a - 1
        while lower_bound in unprocessed_entries:
            unprocessed_entries.remove(lower_bound)
            lower_bound -= 1
```

```
        # Finds the upper bound of the largest range containing a.
        upper_bound = a + 1
        while upper_bound in unprocessed_entries:
            unprocessed_entries.remove(upper_bound)
            upper_bound += 1

        max_interval_size = max(max_interval_size,
                                upper_bound - lower_bound - 1)
    return max_interval_size
```

The time complexity of this approach is $O(n)$, where n is the array length, since we add and remove array elements in the hash table no more than once.

12.10 Compute all string decompositions

This problem is concerned with taking a string (the "sentence" string) and a set of strings (the "words"), and finding the substrings of the sentence which are the concatenation of all the words (in any order). For example, if the sentence string is "amanaplanacanal" and the set of words is {"can", "apl", "ana"}, "aplanacan" is a substring of the sentence that is the concatenation of all words.

Write a program which takes as input a string (the "sentence") and an array of strings (the "words"), and returns the starting indices of substrings of the sentence string which are the concatenation of all the strings in the words array. Each string must appear exactly once, and their ordering is immaterial. Assume all strings in the words array have equal length. It is possible for the words array to contain duplicates.

Hint: Exploit the fact that the words have the same length.

Solution: Let's begin by considering the problem of checking whether a string is the concatenation strings in words. We can solve this problem recursively—we find a string from words that is a prefix of the given string, and recurse with the remaining words and the remaining suffix.

When all strings in words have equal length, say n, only one distinct string in words can be a prefix of the given string. So we can directly check the first n characters of the string to see if they are in words. If not, the string cannot be the concatenation of words. If it is, we remove that string from words and continue with the remainder of the string and the remaining words.

To find substrings in the sentence string that are the concatenation of the strings in words, we can use the above process for each index in the sentence as the starting index.

```
def find_all_substrings(s, words):
    def match_all_words_in_dict(start):
        curr_string_to_freq = collections.Counter()
        for i in range(start, start + len(words) * unit_size, unit_size):
            curr_word = s[i:i + unit_size]
            it = word_to_freq[curr_word]
            if it == 0:
                return False
            curr_string_to_freq[curr_word] += 1
            if curr_string_to_freq[curr_word] > it:
                # curr_word occurs too many times for a match to be possible.
```

```
            return False
    return True

word_to_freq = collections.Counter(words)
unit_size = len(words[0])
return [
    i for i in range(len(s) - unit_size * len(words) + 1)
    if match_all_words_in_dict(i)
]
```

We analyze the time complexity as follows. Let m be the number of words and n the length of each word. Let N be the length of the sentence. For any fixed i, to check if the string of length nm starting at an offset of i in the sentence is the concatenation of all words has time complexity $O(nm)$, assuming a hash table is used to store the set of words. This implies the overall time complexity is $O(Nnm)$. In practice, the individual checks are likely to be much faster because we can stop as soon as a mismatch is detected.

The problem is made easier, complexity-wise and implementation-wise, by the fact that the words are all the same length—it makes testing if a substring equals the concatenation of words straightforward.

12.11 Test the Collatz conjecture

The Collatz conjecture is the following: Take any natural number. If it is odd, triple it and add one; if it is even, halve it. Repeat the process indefinitely. No matter what number you begin with, you will eventually arrive at 1.

As an example, if we start with 11 we get the sequence $11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1$. Despite intense efforts, the Collatz conjecture has not been proved or disproved.

Suppose you were given the task of checking the Collatz conjecture for the first billion integers. A direct approach would be to compute the convergence sequence for each number in this set.

Test the Collatz conjecture for the first n positive integers.

Hint: How would you efficiently check the conjecture for n assuming it holds for all $m < n$?

Solution: Often interview questions are open-ended with no definite good solution—all you can do is provide a good heuristic and code it well.

The Collatz hypothesis can fail in two ways—a sequence returns to a previous number in the sequence, which implies it will loop forever, or a sequence goes to infinity. The latter cannot be tested with a fixed integer word length, so we simply flag overflows.

The general idea is to iterate through all numbers and for each number repeatedly apply the rules till you reach 1. Here are some of the ideas that you can try to accelerate the check:

- Reuse computation by storing all the numbers you have already proved to converge to 1; that way, as soon as you reach such a number, you can assume it would reach 1.
- To save time, skip even numbers (since they are immediately halved, and the resulting number must have already been checked).
- If you have tested every number up to k, you can stop the chain as soon as you reach a number that is less than or equal to k. You do not need to store the numbers below k in the hash table.

- If multiplication and division are expensive, use bit shifting and addition.
- Partition the search set and use many computers in parallel to explore the subsets, as show in Solution 19.9 on Page 299.

Since the numbers in a sequence may grow beyond 32 bits, you should use 64-bit integer and keep testing for overflow; alternately, you can use arbitrary precision integers.

```
def test_collatz_conjecture(n):
    # Stores odd numbers already tested to converge to 1.
    verified_numbers = set()

    # Starts from 3, hypothesis holds trivially for 1.
    for i in range(3, n + 1):
        sequence = set()
        test_i = i
        while test_i >= i:
            if test_i in sequence:
                # We previously encountered test_i, so the Collatz sequence has
                # fallen into a loop. This disproves the hypothesis, so we
                # short-circuit, returning False.
                return False
            sequence.add(test_i)

            if test_i % 2:  # Odd number.
                if test_i in verified_numbers:
                    break  # test_i has already been verified to converge to 1.
                verified_numbers.add(test_i)
                test_i = 3 * test_i + 1  # Multiply by 3 and add 1.
            else:
                test_i //= 2  # Even number, halve it.
    return True
```

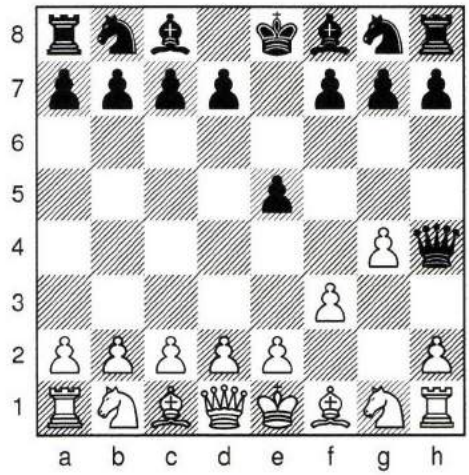
We cannot say much about time complexity beyond the obvious, namely that it is at least proportional to n.

12.12 Implement a hash function for chess

The state of a game of chess is determined by what piece is present on each square, as illustrated in Figure 12.2 on the following page. Each square may be empty, or have one of six classes of pieces; each piece may be black or white. Thus $\lceil \log(1+6\times 2) \rceil = 4$ bits suffice per square, which means that a total of $64 \times 4 = 256$ bits can represent the state of the chessboard. (The actual state of the game is slightly more complex, as it needs to capture which side is to move, castling rights, *en passant*, etc., but we will use the simpler model for this question.)

Chess playing computers need to store sets of states, e.g., to determine if a particular state has been evaluated before, or is known to be a winning state. To reduce storage, it is natural to apply a hash function to the 256 bits of state, and ignore collisions. The hash code can be computed by a conventional hash function for strings. However, since the computer repeatedly explores nearby states, it is advantageous to consider hash functions that can be efficiently computed based on incremental changes to the board.

Design a hash function for chess game states. Your function should take a state and the hash code for that state, and a move, and efficiently compute the hash code for the updated state.



1. f3, e5 2. g4, ♕h4

Figure 12.2: Chessboard corresponding to the fastest checkmate, *Fool's Mate*.

Hint: XOR is associative, commutative, and fast to compute. Additionally, $a \oplus a = 0$.

Solution: A straightforward hash function is to treat the board as a sequence of 64 base 13 digits. There is one digit per square, with the squares numbered from 0 to 63. Each digit encodes the state of a square: blank, white pawn, white rook,. . . ,white king, black pawn, . . . , black king. We use the hash function $\sum_{i=0}^{63} c_i p^i$, where c_i is the digit in location i, and p is a prime (see on Page 159 for more details).

Note that this hash function has some ability to be updated incrementally. If, for example, a black knight taken by a white bishop the new hash code can be computed by subtracting the terms corresponding to the initial location of the knight and bishop, and adding a term for a blank at the initial location of the bishop and a term for the bishop at the knight's original position.

Now we describe a hash function which is much faster to update. It is based on creating a random 64-bit integer code for each of the 13 states that each of the 64 squares can be in. These $13 \times 64 = 832$ random codes are constants in the program. The hash code for the state of the chessboard is the XOR of the code for each location. Updates are very fast—for the example above, we XOR the code for black knight on i_1, white bishop on i_2, white bishop on i_1, and blank on i_2.

Incremental updates to the first hash function entail computing terms like p^i which is more expensive than computing an XOR, which is why the second hash function is preferable. The maximum number of word-level XORs performed is 4, for a capture or a castling.

As an example, consider a simpler game played on a 2×2 board, with at most two pieces, P and Q present on the board. At most one piece can be present at a board position. Denote the board positions by $(0,0)$, $(0,1)$, $(1,0)$, and $(1,1)$. We use the following random 7-bit codes for each individual position:

- For $(0,0)$: $(1100111)_2$ for blank, $(1011000)_2$ for P, $(1100010)_2$ for Q.
- For $(0,1)$: $(1111100)_2$ for blank, $(1000001)_2$ for P, $(0001111)_2$ for Q.
- For $(1,0)$: $(1100101)_2$ for blank, $(1101101)_2$ for P, $(0011101)_2$ for Q.
- For $(1,1)$: $(0100001)_2$ for blank, $(0101100)_2$ for P, $(1001011)_2$ for Q.

Consider the following state: P is present at $(0, 0)$ and Q at $(1, 1)$, with the remaining positions blank. The hash code for this state is $(1011000)_2 \oplus (1111100)_2 \oplus (1100101)_2 \oplus (1001011)_2 = (0001010)_2$. Now to compute the code for the state where Q moves to $(0, 1)$, we XOR the code for the current state with $(1001011)_2$ (removes Q from $(1, 1)$), $(0100001)_2$ (adds blank at $(1, 1)$), $(1111100)_2$ (removes blank from $(0, 1)$), and $(0001111)_2$ (adds Q at $(0, 1)$). Note that, regardless of the size of the board and the number of pieces, this approach uses four XORs to get the updated state.

Variant: How can you include castling rights and *en passant* information in the state?

CHAPTER

13 Sorting

PROBLEM 14 *(Meshing). Two monotone sequences S, T, of lengths n, m, respectively, are stored in two systems of $n(p+1), m(p+1)$ consecutive memory locations, respectively: $s, s+1, \ldots, s+n(p+1)-1$ and $t, t+1, \ldots, t+m(p+1)-1$. ... It is desired to find a monotone permutation R of the sum $[S, T]$, and place it at the locations $r, r+1, \ldots, r+(n+m)(p+1)-1$.*

— *"Planning And Coding Of Problems For An Electronic Computing Instrument,"* H. H. GOLDSTINE AND J. VON NEUMANN, 1948

Sorting—rearranging a collection of items into increasing or decreasing order—is a common problem in computing. Sorting is used to preprocess the collection to make searching faster (as we saw with binary search through an array), as well as identify items that are similar (e.g., students are sorted on test scores).

Naive sorting algorithms run in $O(n^2)$ time. A number of sorting algorithms run in $O(n \log n)$ time—heapsort, merge sort, and quicksort are examples. Each has its advantages and disadvantages: for example, heapsort is in-place but not stable; merge sort is stable but not in-place; quicksort runs $O(n^2)$ time in worst-case. (An in-place sort is one which uses $O(1)$ space; a stable sort is one where entries which are equal appear in their original order.)

A well-implemented quicksort is usually the best choice for sorting. We briefly outline alternatives that are better in specific circumstances.

For short arrays, e.g., 10 or fewer elements, insertion sort is easier to code and faster than asymptotically superior sorting algorithms. If every element is known to be at most k places from its final location, a min-heap can be used to get an $O(n \log k)$ algorithm (Solution 10.3 on Page 137). If there are a small number of distinct keys, e.g., integers in the range [0..255], counting sort, which records for each element, the number of elements less than it, works well. This count can be kept in an array (if the largest number is comparable in value to the size of the set being sorted) or a BST, where the keys are the numbers and the values are their frequencies. If there are many duplicate keys we can add the keys to a BST, with linked lists for elements which have the same key; the sorted result can be derived from an in-order traversal of the BST.

Most sorting algorithms are not stable. Merge sort, carefully implemented, can be made stable. Another solution is to add the index as an integer rank to the keys to break ties.

Most sorting routines are based on a compare function. However, it is also possible to use numerical attributes directly, e.g., in radix sort.

The heap data structure is discussed in detail in Chapter 10. Briefly, a max-heap (min-heap) stores keys drawn from an ordered set. It supports $O(\log n)$ inserts and $O(1)$ time lookup for the maximum (minimum) element; the maximum (minimum) key can be deleted in $O(\log n)$ time. Heaps can be helpful in sorting problems, as illustrated by Problems 10.1 on Page 134, 10.2 on Page 135, and 10.3 on Page 136.

Sorting boot camp

It's important to know how to use effectively the sort functionality provided by your language's library. Let's say we are given a student class that implements a compare method that compares students by name. Then by default, the array sort library routine will sort by name. To sort an array of students by GPA, we have to explicitly specify the compare function to the sort routine.

```
class Student(object):
    def __init__(self, name, grade_point_average):
        self.name = name
        self.grade_point_average = grade_point_average

    def __lt__(self, other):
        return self.name < other.name


students = [
    Student('A', 4.0),
    Student('C', 3.0),
    Student('B', 2.0),
    Student('D', 3.2)
]

# Sort according to __lt__ defined in Student. students remained unchanged.
students_sort_by_name = sorted(students)

# Sort students in-place by grade_point_average.
students.sort(key=lambda student: student.grade_point_average)
```

The time complexity of any reasonable library sort is $O(n \log n)$ for an array with n entries. Most library sorting functions are based on quicksort, which has $O(1)$ space complexity.

Sorting problems come in two flavors: (1.) **use sorting to make subsequent steps in an algorithm simpler**, and (2.) design a **custom sorting routine**. For the former, it's fine to use a library sort function, possibly with a custom comparator. For the latter, use a data structure like a BST, heap, or array indexed by values. Certain problems become easier to understand, as well as solve, when the input is sorted. The most natural reason to sort is if the inputs have a **natural ordering**, and sorting can be used as a preprocessing step to **speed up searching**. For **specialized input**, e.g., a very small range of values, or a small number of values, it's possible to sort in $O(n)$ time rather than $O(n \log n)$ time. It's often the case that sorting can be implemented in **less space** than required by a brute-force approach. Sometimes it is not obvious what to sort on, e.g., should a collection of intervals be sorted on starting points or endpoints? (Problem 13.5 on Page 186)

Table 13.1: Top Tips for Sorting

Know your sorting libraries

To sort a list in-place, use the `sort()` method; to sort an iterable, use the function `sorted()`.

- The `sort()` method implements a stable in-place sort for list objects. It returns `None`—the calling list itself is updated. It takes two arguments, both optional: `key=None`, and `reverse=False` If `key` is not `None`, it is assumed to be a function which takes list elements and maps them to objects which are comparable—this function defines the sort order. For example, if `a=[1, 2, 4, 3, 5, 0, 11, 21, 100]` then `a.sort(key=lambda x: str(x))` maps integers to strings, and `a` is updated to `[0, 1, 100, 11, 2, 21, 3, 4, 5]`, i.e., the entries are sorted according to the lexicographical ordering of their string representation. If `reverse` is set to `True`, the sort is in descending order; otherwise it is in ascending order.
- The `sorted` function takes an iterable and return a new list containing all items from the iterable in ascending order. The original list is unchanged. For example, `b = sorted(a, key=lambda x: str(x))` leaves `a` unchanged, and assigns `b` to `[0, 1, 100, 11, 2, 21, 3, 4, 5]`. The optional arguments `key` and `reverse` work identically to `sort`.

13.1 Compute the intersection of two sorted arrays

A natural implementation for a search engine is to retrieve documents that match the set of words in a query by maintaining an inverted index. Each page is assigned an integer identifier, its *document-ID*. An inverted index is a mapping that takes a word w and returns a sorted array of page-ids which contain w—the sort order could be, for example, the page rank in descending order. When a query contains multiple words, the search engine finds the sorted array for each word and then computes the intersection of these arrays—these are the pages containing all the words in the query. The most computationally intensive step of doing this is finding the intersection of the sorted arrays.

Write a program which takes as input two sorted arrays, and returns a new array containing elements that are present in both of the input arrays. The input arrays may have duplicate entries, but the returned array should be free of duplicates. For example, the input is $\langle 2,3,3,5,5,6,7,7,8,12\rangle$ and $\langle 5,5,6,8,8,9,10,10\rangle$, your output should be $\langle 5,6,8\rangle$.

Hint: Solve the problem if the input array lengths differ by orders of magnitude. What if they are approximately equal?

Solution: The brute-force algorithm is a "loop join", i.e., traversing through all the elements of one array and comparing them to the elements of the other array. Let m and n be the lengths of the two input arrays.

```
def intersect_two_sorted_arrays(A, B):
    return [a for i, a in enumerate(A) if (i == 0 or a != A[i - 1]) and a in B]
```

The brute-force algorithm has $O(mn)$ time complexity.

Since both the arrays are sorted, we can make some optimizations. First, we can iterate through the first array and use binary search in array to test if the element is present in the second array.

```
def intersect_two_sorted_arrays(A, B):
    def is_present(k):
        i = bisect.bisect_left(B, k)
        return i < len(B) and B[i] == k

    return [
        a for i, a in enumerate(A)
        if (i == 0 or a != A[i - 1]) and is_present(a)
    ]
```

The time complexity is $O(m \log n)$, where m is the length of the array being iterated over. We can further improve our run time by choosing the shorter array for the outer loop since if n is much smaller than m, then $n \log(m)$ is much smaller than $m \log(n)$.

This is the best solution if one set is much smaller than the other. However, it is not the best when the array lengths are similar because we are not exploiting the fact that both arrays are sorted. We can achieve linear runtime by simultaneously advancing through the two input arrays in increasing order. At each iteration, if the array elements differ, the smaller one can be eliminated. If they are equal, we add that value to the intersection and advance both. (We handle duplicates by comparing the current element with the previous one.) For example, if the arrays are $A = \langle 2,3,3,5,7,11 \rangle$ and $B = \langle 3,3,7,15,31 \rangle$, then we know by inspecting the first element of each that 2 cannot belong to the intersection, so we advance to the second element of A. Now we have a common element, 3, which we add to the result, and then we advance in both arrays. Now we are at 3 in both arrays, but we know 3 has already been added to the result since the previous element in A is also 3. We advance in both again without adding to the intersection. Comparing 5 to 7, we can eliminate 5 and advance to the fourth element in A, which is 7, and equal to the element that B's iterator holds, so it is added to the result. We then eliminate 11, and since no elements remain in A, we return $\langle 3,7 \rangle$.

```
def intersect_two_sorted_arrays(A, B):
    i, j, intersection_A_B = 0, 0, []
    while i < len(A) and j < len(B):
        if A[i] == B[j]:
            if i == 0 or A[i] != A[i - 1]:
                intersection_A_B.append(A[i])
            i, j = i + 1, j + 1
        elif A[i] < B[j]:
            i += 1
        else:  # A[i] > B[j].
            j += 1
    return intersection_A_B
```

Since we spend $O(1)$ time per input array element, the time complexity for the entire algorithm is $O(m+n)$.

13.2 Merge two sorted arrays

Suppose you are given two sorted arrays of integers. If one array has enough empty entries at its end, it can be used to store the combined entries of the two arrays in sorted order. For example, consider $\langle 5, 13, 17, \sqcup, \sqcup, \sqcup, \sqcup, \sqcup \rangle$ and $\langle 3, 7, 11, 19 \rangle$, where $\sqcup$ denotes an empty entry. Then the combined sorted entries can be stored in the first array as $\langle 3, 5, 7, 11, 13, 17, 19, \sqcup \rangle$.

Write a program which takes as input two sorted arrays of integers, and updates the first to the combined entries of the two arrays in sorted order. Assume the first array has enough empty entries at its end to hold the result.

Hint: Avoid repeatedly moving entries.

Solution: The challenge in this problem lies in writing the result back into the first array—if we had a third array to store the result it, we could solve by iterating through the two input arrays in tandem, writing the smaller of the entries into the result. The time complexity is $O(m+n)$, where m and n are the number of entries initially in the first and second arrays.

We cannot use the above approach with the first array playing the role of the result and still keep the time complexity $O(m+n)$. The reason is that if an entry in the second array is smaller than some entry in the first array, we will have to shift that and all subsequent entries in the first array to the right by 1. In the worst-case, each entry in the second array is smaller than every entry in the first array, and the time complexity is $O(mn)$.

We do have spare space at the end of the first array. We take advantage of this by filling the first array from its end. The last element in the result will be written to index $m+n-1$. For example, if $A = \langle 5, 13, 17, \_, \_, \_, \_, \_ \rangle$ and $B = \langle 3, 7, 11, 19 \rangle$, then A is updated in the following manner: $\langle 5, 13, 17, \_, \_, \_, 19, \_ \rangle$, $\langle 5, 13, 17, \_, \_, 17, 19, \_ \rangle$, $\langle 5, 13, 17, \_, 13, 17, 19, \_ \rangle$, $\langle 5, 13, 17, 11, 13, 17, 19, \_ \rangle$, $\langle 5, 13, 7, 11, 13, 17, 19, \_ \rangle$, $\langle 5, 5, 7, 11, 13, 17, 19, \_ \rangle$, $\langle 3, 5, 7, 11, 13, 17, 19, \_ \rangle$.

Note that we will never overwrite an entry in the first array that has not already been processed. The reason is that even if every entry of the second array is larger than each element of the first array, all elements of the second array will fill up indices m to $m+n-1$ inclusive, which does not conflict with entries stored in the first array. This idea is implemented in the program below. Note the resemblance to Solution 6.4 on Page 71, where we also filled values from the end.

```
def merge_two_sorted_arrays(A, m, B, n):
    a, b, write_idx = m - 1, n - 1, m + n - 1
    while a >= 0 and b >= 0:
        if A[a] > B[b]:
            A[write_idx] = A[a]
            a -= 1
        else:
            A[write_idx] = B[b]
            b -= 1
        write_idx -= 1
    while b >= 0:
        A[write_idx] = B[b]
        write_idx, b = write_idx - 1, b - 1
```

The time complexity is $O(m+n)$. It uses $O(1)$ additional space.

13.3 Remove first-name duplicates

Design an efficient algorithm for removing all first-name duplicates from an array. For example, if the input is $\langle$(Ian, Botham), (David, Gower), (Ian, Bell), (Ian, Chappell)$\rangle$, one result could be $\langle$(Ian, Bell), (David, Gower)$\rangle$; $\langle$(David, Gower), (Ian, Botham)$\rangle$ would also be acceptable.

Hint: Bring equal items close together.

Solution: A brute-force approach is to use a hash table. For the names example, we would need a hash function and an equals function which use the first name only. We first create the hash table and then iterate over it to write entries to the result array. The time complexity is $O(n)$, where n is the number of items. The hash table has a worst-case space complexity of $O(n)$.

We can avoid the additional space complexity if we can reuse the input array for storing the final result. First we sort the array, which brings equal elements together. Sorting can be done in $O(n \log n)$ time. The subsequent elimination of duplicates takes $O(n)$ time. Note that sorting an array requires that its elements are comparable.

```
class Name:
```

```
    def __init__(self, first_name, last_name):
        self.first_name, self.last_name = first_name, last_name


    def __eq__(self, other):
        return self.first_name == other.first_name

    def __lt__(self, other):
        return (self.first_name < other.first_name
                if self.first_name != other.first_name else
                self.last_name < other.last_name)



def eliminate_duplicate(A):
    A.sort()  # Makes identical elements become neighbors.
    write_idx = 1
    for cand in A[1:]:
        if cand != A[write_idx - 1]:
            A[write_idx] = cand
            write_idx += 1
    del A[write_idx:]
```

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

13.4 Smallest nonconstructible value

Given a set of coins, there are some amounts of change that you may not be able to make with them, e.g., you cannot create a change amount greater than the sum of the your coins. For example, if your coins are $1, 1, 1, 1, 1, 5, 10, 25$, then the smallest value of change which cannot be made is 21.

Write a program which takes an array of positive integers and returns the smallest number which is not to the sum of a subset of elements of the array.

Hint: Manually solve for a few short arrays.

Solution: A brute-force approach would be to enumerate all possible values, starting from 1, testing each value to see if it is the sum of array elements. However, there is no simple efficient algorithm for checking if a given number is the sum of a subset of entries in the array. Heuristics may be employed, but the program will be unwieldy.

We can get a great deal of insight from some small concrete examples. Observe that $\langle 1, 2 \rangle$ produces $1, 2, 3$, and $\langle 1, 3 \rangle$ produces $1, 3, 4$. A trivial observation is that the smallest element in the array sets a lower bound on the change amount that can be constructed from that array, so if the array does not contain a 1, it cannot produce 1. However, it may be possible to produce 2 without a 2 being present, since there can be 2 or more 1s present.

Continuing with a larger example, $\langle 1, 2, 4 \rangle$ produces $1, 2, 3, 4, 5, 6, 7$, and $\langle 1, 2, 5 \rangle$ produces $1, 2, 3, 5, 6, 7, 8$. Generalizing, suppose a collection of numbers can produce every value up to and including V, but not $V + 1$. Now consider the effect of adding a new element u to the collection. If $u \leq V + 1$, we can still produce every value up to and including $V + u$ and we cannot produce $V + u + 1$. On the other hand, if $u > V + 1$, then even by adding u to the collection we cannot produce $V + 1$.

Another observation is that the order of the elements within the array makes no difference to the amounts that are constructible. However, by sorting the array allows us to stop when we reach a value that is too large to help, since all subsequent values are at least as large as that value. Specifically, let $M[i-1]$ be the maximum constructible amount from the first i elements of the sorted array. If the next array element x is greater than $M[i-1]+1$, $M[i-1]$ is still the maximum constructible amount, so we stop and return $M[i-1]+1$ as the result. Otherwise, we set $M[i] = M[i-1] + x$ and continue with element $(i+1)$.

To illustrate, suppose we are given $\langle 12, 2, 1, 15, 2, 4\rangle$. This sorts to $\langle 1, 2, 2, 4, 12, 15\rangle$. The maximum constructible amount we can make with the first element is 1. The second element, 2, is less than or equal to $1+1$, so we can produce all values up to and including 3 now. The third element, 2, allows us to produce all values up to and including 5. The fourth element, 4, allows us to produce all values up to 9. The fifth element, 12 is greater than $9+1$, so we cannot produce 10. We stop—10 is the smallest number that cannot be constructed.

The code implementing this approach is shown below.

```
def smallest_nonconstructible_value(A):
    max_constructible_value = 0
    for a in sorted(A):
        if a > max_constructible_value + 1:
            break
        max_constructible_value += a
    return max_constructible_value + 1
```

The time complexity as a function of n, the length of the array, is $O(n \log n)$ to sort and $O(n)$ to iterate, i.e., $O(n \log n)$.

13.5 Render a calendar

Consider the problem of designing an online calendaring application. One component of the design is to render the calendar, i.e., display it visually.

Suppose each day consists of a number of events, where an event is specified as a start time and a finish time. Individual events for a day are to be rendered as nonoverlapping rectangular regions whose sides are parallel to the X- and Y-axes. Let the X-axis correspond to time. If an event starts at time b and ends at time e, the upper and lower sides of its corresponding rectangle must be at b and e, respectively. Figure 13.1 on the facing page represents a set of events.

Suppose the Y-coordinates for each day's events must lie between 0 and L (a pre-specified constant), and each event's rectangle must have the same "height" (distance between the sides parallel to the X-axis). Your task is to compute the maximum height an event rectangle can have. In essence, this is equivalent to the following problem.

Write a program that takes a set of events, and determines the maximum number of events that take place concurrently.

Hint: Focus on endpoints.

Solution: The number of events scheduled for a given time changes only at times that are start or end times of an event. This leads the following brute-force algorithm. For each endpoint, compute the number of events that contain it. The maximum number of concurrent events is the maximum of this quantity over all endpoints. If there are n intervals, the total number of endpoints is $2n$.

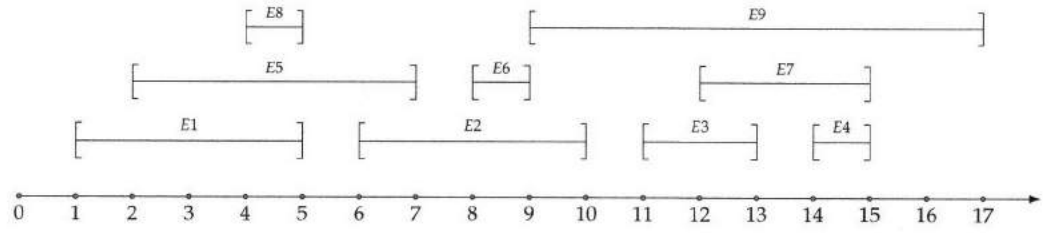


Figure 13.1: A set of nine events. The earliest starting event begins at time 1; the latest ending event ends at time 17. The maximum number of concurrent events is 3, e.g., $\{E1, E5, E8\}$ as well as others.

Computing the number of events containing an endpoint takes $O(n)$ time, since checking whether an interval contains a point takes $O(1)$ time. Therefore, the overall time complexity is $O(2n \times n) = O(n^2)$.

The inefficiency in the brute-force algorithm lies in the fact that it does not take advantage of locality, i.e., as we move from one endpoint to another. Intuitively, we can improve the run time by sorting the set of all the endpoints in ascending order. (If two endpoints have equal times, and one is a start time and the other is an end time, the one corresponding to a start time comes first. If both are start or finish times, we break ties arbitrarily.)

Now as we proceed through endpoints we can incrementally track the number of events taking place at that endpoint using a counter. For each endpoint that is the start of an interval, we increment the counter by 1, and for each endpoint that is the end of an interval, we decrement the counter by 1. The maximum value attained by the counter is maximum number of overlapping intervals.

For the example in Figure 13.1, the first seven endpoints are 1(start), 2(start), 4(start), 5(end), 5(end), 6(start), 7(end). The counter values are updated to 1, 2, 3, 2, 1, 2, 1.

```
# Event is a tuple (start_time, end_time)
Event = collections.namedtuple('Event', ('start', 'finish'))

# Endpoint is a tuple (start_time, 0) or (end_time, 1) so that if times
# are equal, start_time comes first
Endpoint = collections.namedtuple('Endpoint', ('time', 'is_start'))


def find_max_simultaneous_events(A):
    # Builds an array of all endpoints.
    E = ([Endpoint(event.start, True)
          for event in A] + [Endpoint(event.finish, False) for event in A])
    # Sorts the endpoint array according to the time, breaking ties by putting
    # start times before end times.
    E.sort(key=lambda e: (e.time, not e.is_start))

    # Track the number of simultaneous events, record the maximum number of
    # simultaneous events.
    max_num_simultaneous_events, num_simultaneous_events = 0, 0
    for e in E:
        if e.is_start:
            num_simultaneous_events += 1
            max_num_simultaneous_events = max(num_simultaneous_events,
                                              max_num_simultaneous_events)
        else:
            num_simultaneous_events -= 1
    return max_num_simultaneous_events
```

Sorting the endpoint array takes $O(n \log n)$ time; iterating through the sorted array takes $O(n)$ time, yielding an $O(n \log n)$ time complexity. The space complexity is $O(n)$, which is the size of the endpoint array.

Variant: Users $1, 2, \ldots, n$ share an Internet connection. User i uses b_i bandwidth from time s_i to f_i, inclusive. What is the peak bandwidth usage?

13.6 Merging intervals

Suppose the time during the day that a person is busy is stored as a set of disjoint time intervals. If an event is added to the person's calendar, the set of busy times may need to be updated.

In the abstract, we want a way to add an interval to a set of disjoint intervals and represent the new set as a set of disjoint intervals. For example, if the initial set of intervals is $[-4, -1], [0, 2], [3, 6], [7, 9], [11, 12], [14, 17]$, and the added interval is $[1, 8]$, the result is $[-4, -1], [0, 9], [11, 12], [14, 17]$.

Write a program which takes as input an array of disjoint closed intervals with integer endpoints, sorted by increasing order of left endpoint, and an interval to be added, and returns the union of the intervals in the array and the added interval. Your result should be expressed as a union of disjoint intervals sorted by left endpoint.

Hint: What is the union of two closed intervals?

Solution: A brute-force approach is to find the smallest left endpoint and the largest right endpoint in the set of intervals in the array and the added interval. We then form the result by testing every integer between these two values for membership in an interval. The time complexity is $O(Dn)$, where D is the difference between the two extreme values and n is the number of intervals. Note that D may be much larger than n. For example, the brute-force approach will iterate over all integers from 0 to 1000000 if the array is $\langle [0, 1], [999999, 1000000] \rangle$ and the added interval is $[10, 20]$.

The brute-force approach examines values that are not endpoints, which is wasteful, since if an integer point p is not an endpoint, it must lie in the same interval as $p - 1$ does. A better approach is to focus on endpoints, and use the sorted property to quickly process intervals in the array.

Specifically, processing an interval in the array takes place in three stages:

(1.) First, we iterate through intervals which appear completely before the interval to be added—all these intervals are added directly to the result.

(2.) As soon as we encounter an interval that intersects the interval to be added, we compute its union with the interval to be added. This union is itself an interval. We iterate through subsequent intervals, as long as they intersect with the union we are forming. This single union is added to the result.

(3.) Finally, we iterate through the remaining intervals. Because the array was originally sorted, none of these can intersect with the interval to be added, so we add these intervals to the result.

Suppose the sorted array of intervals is $[-4, -1], [0, 2], [3, 6], [7, 9], [11, 12], [14, 17]$, and the added interval is $[1, 8]$. We begin in Stage 1. Interval $[-4, -1]$ does not intersect $[1, 8]$, so we add it directly to the result. Next we proceed to $[0, 2]$. Since $[0, 2]$ intersects $[1, 8]$, we are now in Stage 2 of the algorithm. We add the union of the two, $[0, 8]$, to the result. Now we process $[3, 6]$—it lies completely

in $[0,8]$, so we proceed to $[7,9]$. It intersects $[1,8]$ but is not completely contained in it, so we update the most recently added interval to the result, $[1,8]$ to $[0,9]$. Next we proceed to $[11,12]$. It does not intersect the most recently added interval to the result, $[0,9]$, so we are in Stage 3. We add it and all subsequent intervals to the result, which is now $[-4,-1],[0,9],[11,12],[14,17]$. Note how the algorithm operates "locally"—sortedness guarantees that we do not miss any combinations of intervals.

The program implementing this idea is given below.

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


def add_interval(disjoint_intervals, new_interval):
    i, result = 0, []

    # Processes intervals in disjoint_intervals which come before new_interval.
    while (i < len(disjoint_intervals)
           and new_interval.left > disjoint_intervals[i].right):
        result.append(disjoint_intervals[i])
        i += 1

    # Processes intervals in disjoint_intervals which overlap with new_interval.
    while (i < len(disjoint_intervals)
           and new_interval.right >= disjoint_intervals[i].left):
        # If [a, b] and [c, d] overlap, union is [min(a, c), max(b, d)].
        new_interval = Interval(
            min(new_interval.left, disjoint_intervals[i].left),
            max(new_interval.right, disjoint_intervals[i].right))
        i += 1
    # Processes intervals in disjoint_intervals which come after new_interval.
    return result + [new_interval] + disjoint_intervals[i:]
```

Since the program spends $O(1)$ time per entry, its time complexity is $O(n)$.

13.7 Compute the union of intervals

In this problem we consider sets of intervals with integer endpoints; the intervals may be open or closed at either end. We want to compute the union of the intervals in such sets. A concrete example is given in Figure 13.2.

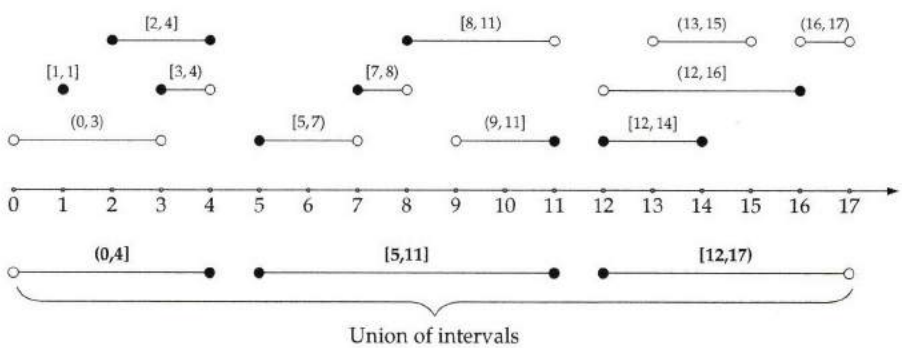


Figure 13.2: A set of intervals and their union.

Design an algorithm that takes as input a set of intervals, and outputs their union expressed as a set of disjoint intervals.

Hint: Do a case analysis.

Solution: The brute-force approach of considering every number from the minimum left endpoint to the maximum right endpoint described at the start of Solution 13.6 on Page 188 will work for this problem too. As before, its time complexity is $O(Dn)$, where D is the difference between the two extreme values and n is the number of intervals, which is unacceptable when D is large.

We can improve the run time by focusing on intervals instead of individual values. We perform the following iteratively: select an interval arbitrarily, and find all intervals it intersects with. If it does not intersect any interval, remove it from the set and add it to the result. Otherwise, take its union with all the intervals it intersects with (the union must itself be an interval), remove it and all the intervals it intersects with from the result, and add the union to the set. Since we remove at least one interval from the set each time, and the time to process each interval (test intersection and form unions) is $O(n)$, the time complexity is $O(n^2)$.

A faster implementation of the above approach is to process the intervals in sorted order, so that we can limit our attention to a subset of intervals as we proceed. Specifically, we begin by sorting the intervals on their left endpoints. The idea is that this allows us to not have to revisit intervals which are entirely to the left of the interval currently being processed.

We refer to an interval which does not include its left endpoint as being left-open. Left-closed, right-open, and right-closed are defined similarly. When sorting, if two intervals have the same left endpoint, we put intervals which are left-closed first. We break ties arbitrarily.

As we iterate through the sorted array of intervals, we have the following cases:

- The interval most recently added to the result does not intersect the current interval, nor does its right endpoint equal the left endpoint of the current interval. In this case, we simply add the current interval to the end of the result array as a new interval.
- The interval most recently added to the result intersects the current interval. In this case, we update the most recently added interval to the union of it with the current interval.
- The interval most recently added to the result has its right endpoint equal to the left endpoint of the current interval, and one (or both) of these endpoints are closed. In this case too, we update the most recently added interval to the union of it with the current interval.

For the example in Figure 13.2 on the preceding page, the result array updates in the following way: $\langle(0,3)\rangle$, $\langle(0,4]\rangle$, $\langle(0,4],[5,7)\rangle$, $\langle(0,4],[5,8)\rangle$, $\langle(0,4],[5,11)\rangle$, $\langle(0,4],[5,11]\rangle$, $\langle(0,4],[5,11],[12,14]\rangle$, $\langle(0,4],[5,11],[12,16]\rangle$, $\langle(0,4],[5,11],[12,17)\rangle$.

```
Endpoint = collections.namedtuple('Endpoint', ('is_closed', 'val'))


class Interval:
    def __init__(self, left, right):
        self.left = left
        self.right = right

    def __lt__(self, other):
        if self.left.val != other.left.val:
            return self.left.val < other.left.val
        # Left endpoints are equal, so now see if one is closed and the other open
        # - closed intervals should appear first.
```

```
        return self.left.is_closed and not other.left.is_closed


def union_of_intervals(intervals):
    # Empty input.
    if not intervals:
        return []

    # Sort intervals according to left endpoints of intervals.
    intervals.sort()
    result = [intervals[0]]
    for i in intervals:
        if intervals and (i.left.val < result[-1].right.val or
                          (i.left.val == result[-1].right.val and
                           (i.left.is_closed or result[-1].right.is_closed))):
            if (i.right.val > result[-1].right.val or
                (i.right.val == result[-1].right.val and i.right.is_closed)):
                result[-1].right = i.right
        else:
            result.append(i)
    return result
```

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

13.8 Partitioning and sorting an array with many repeated entries

Suppose you need to reorder the elements of a very large array so that equal elements appear together. For example, if the array is $\langle b,a,c,b,d,a,b,d \rangle$ then $\langle a,a,b,b,b,c,d,d \rangle$ is an acceptable reordering, as is $\langle d,d,c,a,a,b,b,b \rangle$.

If the entries are integers, this reordering can be achieved by sorting the array. If the number of distinct integers is very small relative to the size of the array, an efficient approach to sorting the array is to count the number of occurrences of each distinct integer and write the appropriate number of each integer, in sorted order, to the array. When array entries are objects, with multiple fields, only one of which is to be used as a key, the problem is harder to solve.

You are given an array of student objects. Each student has an integer-valued age field that is to be treated as a key. Rearrange the elements of the array so that students of equal age appear together. The order in which different ages appear is not important. How would your solution change if ages have to appear in sorted order?

Hint: Count the number of students for each age.

Solution: The brute-force approach is to sort the array, comparing on age. If the array length is n, the time complexity is $O(n \log n)$ and space complexity is $O(1)$. The inefficiency in this approach stems from the fact that it does more than is required—the specification simply asks for students of equal age to be adjacent.

We use the approach described in the introduction to the problem. However, we cannot apply it directly, since we need to write objects, not integers—two students may have the same age but still be different.

For example, consider the array $\langle$(Greg, 14), (John, 12), (Andy, 11), (Jim, 13), (Phil, 12), (Bob, 13), (Chip, 13), (Tim, 14)$\rangle$. We can iterate through the array and record the number of students of each

age in a hash. Specifically, keys are ages, and values are the corresponding counts. For the given example, on completion of the iteration, the hash is $(14, 2), (12, 2), (11, 1), (13, 3)$. This tells us that we need to write two students of age 14, two students of age 12, one student of age 11 and three students of age 13. We can write these students in any order, as long as we keep students of equal age adjacent.

If we had a new array to write to, we can write the two students of age 14 starting at index 0, the two students of age 12 starting at index $0 + 2 = 2$, the one student of age 11 at index $2 + 2 = 4$, and the three students of age 13 starting at index $4 + 1 = 5$. We would iterate through the original array, and write each entry into the new array according to these offsets. For example, after the first four iterations, the new array would be $\langle (\text{Greg}, 14), \_, (\text{John}, 12), \_, (\text{Andy}, 11), (\text{Jim}, 13), \_, \_ \rangle$.

The time complexity of this approach is $O(n)$, but it entails $O(n)$ additional space for the result array. We can avoid having to allocate a new array by performing the updates in-place. The idea is to maintain a subarray for each of the different types of elements. Each subarray marks out entries which have not yet been assigned elements of that type. We swap elements across these subarrays to move them to their correct position.

In the program below, we use two hash tables to track the subarrays. One is the starting offset of the subarray, the other its size. As soon as the subarray becomes empty, we remove it.

```
Person = collections.namedtuple('Person', ('age', 'name'))


def group_by_age(people):
    age_to_count = collections.Counter([person.age for person in people])
    age_to_offset, offset = {}, 0
    for age, count in age_to_count.items():
        age_to_offset[age] = offset
        offset += count

    while age_to_offset:
        from_age = next(iter(age_to_offset))
        from_idx = age_to_offset[from_age]
        to_age = people[from_idx].age
        to_idx = age_to_offset[people[from_idx].age]
        people[from_idx], people[to_idx] = people[to_idx], people[from_idx]
        # Use age_to_count to see when we are finished with a particular age.
        age_to_count[to_age] -= 1
        if age_to_count[to_age]:
            age_to_offset[to_age] = to_idx + 1
        else:
            del age_to_offset[to_age]
```

The time complexity is $O(n)$, since the first pass entails n hash table inserts, and the second pass spends $O(1)$ time to move one element to its proper location. The additional space complexity is dictated by the hash table, i.e., $O(m)$, where m is the number of distinct ages.

If the entries are additionally required to appear sorted by age, we can use a BST-based map (Chapter 14) to map ages to counts, since the BST-based map keeps ages in sorted order. For our example, the age-count pairs would appear in the order $(11, 1), (12, 2), (13, 3), (14, 2)$. The time complexity becomes $O(n + m \log m)$, since BST insertion takes time $O(\log m)$. Such a sort is often referred to as a counting sort.

You are a photographer for a soccer meet. You will be taking pictures of pairs of opposing teams. All teams have the same number of players. A team photo consists of a front row of players and a back row of players. A player in the back row must be taller than the player in front of him, as illustrated in Figure 13.3. All players in a row must be from the same team.

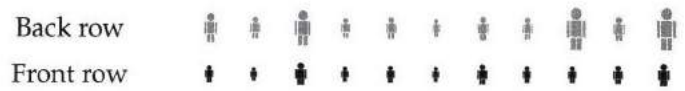


Figure 13.3: A team photo. Each team has 11 players, and each player in the back row is taller than the corresponding player in the front row.

Design an algorithm that takes as input two teams and the heights of the players in the teams and checks if it is possible to place players to take the photo subject to the placement constraint.

Hint: First try some concrete inputs, then make a general conclusion.

Solution: A brute-force approach is to consider every permutation of one array, and compare it against the other array, element by element. Suppose there are n players in each team. It takes $O(n!)$ time to enumerate every possible permutation of a team, and testing if a permutation leads to a satisfactory arrangement takes $O(n)$ time. Therefore, the time complexity is $O(n! \times n)$, clearly unacceptable.

Intuitively, we should narrow the search by focusing on the hardest to place players. Suppose we want to place Team A behind Team B. If A's tallest player is not taller than the tallest player in B, then it's not possible to place Team A behind Team B and satisfy the placement constraint. Conversely, if Team A's tallest player is taller than the tallest player in B, we should place him in front of the tallest player in B, since the tallest player in B is the hardest to place. Applying the same logic to the remaining players, the second tallest player in A should be taller than the second tallest player in B, and so on.

We can efficiently check whether A's tallest, second tallest, etc. players are each taller than B's tallest, second tallest, etc. players by first sorting the arrays of player heights. Figure 13.4 shows the teams in Figure 13.3 sorted by their heights.

Figure 13.4: The teams from Figure 13.3 in sorted order.

The program below uses this idea to test if a given team can be placed in front of another team.

```
class Team:
    Player = collections.namedtuple('Player', ('height'))

    def __init__(self, height):
        self._players = [Team.Player(h) for h in height]

    # Checks if A can be placed in front of B.
    @staticmethod
```

```
    def valid_placement_exists(A, B):
        return all(a < b
                   for a, b in zip(sorted(A._players), sorted(B._players)))
```

The time complexity is that of sorting, i.e., $O(n \log n)$.

13.10 Implement a fast sorting algorithm for lists

Implement a routine which sorts lists efficiently. It should be a stable sort, i.e., the relative positions of equal elements must remain unchanged.

Hint: In what respects are lists superior to arrays?

Solution: The brute-force approach is to repeatedly delete the smallest element in the list and add it to the end of a new list. The time complexity is $O(n^2)$ and the additional space complexity is $O(n)$, where n is the number of nodes in the list. We can refine the simple algorithm to run in $O(1)$ space by reordering the nodes, instead of creating new ones.

```
def insertion_sort(L):
    dummy_head = ListNode(0, L)
    # The sublist consisting of nodes up to and including iter is sorted in
    # increasing order. We need to ensure that after we move to L.next this
    # property continues to hold. We do this by swapping L.next with its
    # predecessors in the list till it's in the right place.
    while L and L.next:
        if L.data > L.next.data:
            target, pre = L.next, dummy_head
            while pre.next.data < target.data:
                pre = pre.next
            temp, pre.next, L.next = pre.next, target, target.next
            target.next = temp
        else:
            L = L.next
    return dummy_head.next
```

The time complexity is $O(n^2)$, which corresponds to the case where the list is reverse-sorted to begin with. The space complexity is $O(1)$.

To improve on runtime, we can gain intuition from considering arrays. Quicksort is the best all round sorting algorithm for arrays—it runs in time $O(n \log n)$, and is in-place. However, it is not stable. Mergesort applied to arrays is a stable $O(n \log n)$ algorithm. However, it is not in-place, since there is no way to merge two sorted halves of an array in-place in linear time.

Unlike arrays, lists can be merged in-place—conceptually, this is because insertion into the middle of a list is an $O(1)$ operation. The following program implements a mergesort on lists. We decompose the list into two equal-sized sublists around the node in the middle of the list. We find this node by advancing two iterators through the list, one twice as fast as the other. When the fast iterator reaches the end of the list, the slow iterator is at the middle of the list. We recurse on the sublists, and use Solution 7.1 on Page 84 (merge two sorted lists) to combine the sorted sublists.

```
def stable_sort_list(L):
    # Base cases: L is empty or a single node, nothing to do.
    if not L or not L.next:
        return L
```

```
    # Find the midpoint of L using a slow and a fast pointer.
    pre_slow, slow, fast = None, L, L
    while fast and fast.next:
        pre_slow = slow
        fast, slow = fast.next.next, slow.next
    pre_slow.next = None  # Splits the list into two equal-sized lists.
    return merge_two_sorted_lists(stable_sort_list(L), stable_sort_list(slow))
```

The time complexity is the same as that of mergesort, i.e., $O(n \log n)$. Though no memory is explicitly allocated, the space complexity is $O(\log n)$. This is the maximum function call stack depth, since each recursive call is with an argument that is half as long.

13.11 Compute a salary threshold

You are working in the finance office for ABC corporation. ABC needs to cut payroll expenses to a specified target. The chief executive officer wants to do this by putting a cap on last year's salaries. Every employee who earned more than the cap last year will be paid the cap this year; employees who earned no more than the cap will see no change in their salary.

For example, if there were five employees with salaries last year were \$90, \$30, \$100, \$40, and \$20, and the target payroll this year is \$210, then 60 is a suitable salary cap, since 60+30+60+40+20 = 210.

Design an algorithm for computing the salary cap, given existing salaries and the target payroll.

Hint: How does the payroll vary with the cap?

Solution: Brute-force is not much use—there are an infinite number of possibilities for the cap.

The cap lies between 0 and the maximum current salary. The payroll increases with the cap, which suggests using binary search in this range—if a cap is too high, no higher cap will work; the same is true if the cap is too low.

Suppose there are n employees. Let the array holding salary data be A. The payroll, $P(c)$, implied by a cap of c is $\sum_{i=0}^{n-1} \min(A[i], c)$. Each step of the binary search evaluating $P(c)$ which takes time $O(n)$. As in Solution 11.5 on Page 149, the number of binary search steps depends on the largest salary and the desired accuracy.

We can use a slightly more analytical method to avoid the need for a specified tolerance. The intuition is that as we increase the cap, as long as it does not exceed someone's salary, the payroll increases linearly. This suggests iterating through the salaries in increasing order. Assume the salaries are given by an array A, which is sorted. Suppose the cap for a total payroll of T is known to lie between the kth and $(k+1)$th salaries. We want $\sum_{i=0}^{k-1} A[i] + (n-k)c$ to equal $= T$, which solves to $c = (T - \sum_{i=0}^{k-1} A[i])/(n-k)$.

For the given example, $A = \langle 20, 30, 40, 90, 100 \rangle$, and $T = 210$. The payrolls for caps equal to the salaries in A are $\langle 100, 140, 170, 270, 280 \rangle$. Since $T = 210$ lies between 170 and 270, the cap lies between the 40 and 90. For any cap c between 40 and 90, the implied payroll is $20 + 30 + 40 + 2c$. We want this to be 210, so we solve $20 + 30 + 40 + 2c = 210$ for c, yielding $c = 60$.

```
def find_salary_cap(target_payroll, current_salaries):
    current_salaries.sort()
    unadjusted_salary_sum = 0.0
    for i, current_salary in enumerate(current_salaries):
        adjusted_people = len(current_salaries) - i
```

```
        adjusted_salary_sum = current_salary * adjusted_people
        if unadjusted_salary_sum + adjusted_salary_sum >= target_payroll:
            return (target_payroll - unadjusted_salary_sum) / adjusted_people
        unadjusted_salary_sum += current_salary
    # No solution, since target_payroll > existing payroll.
    return -1.0
```

The most expensive operation for this entire solution is sorting A, hence the run time is $O(n \log n)$. Once we have A sorted, we simply iterate through its entries looking for the first entry which implies a payroll that exceeds the target, and then solve for the cap using an arithmetical expression.

If we are given the salary array sorted in advance as well as its prefix sums, then for a given value of T, we can use binary search to get the cap in $O(\log n)$ time.

Variant: Solve the same problem using only $O(1)$ space.

CHAPTER

14 Binary Search Trees

The number of trees which can be formed with $n+1$ given knots $\alpha, \beta, \gamma, \ldots = (n+1)^{n-1}$.

— *"A Theorem on Trees,"*
A. CAYLEY, 1889

BSTs are a workhorse of data structures and can be used to solve almost every data structures problem reasonably efficiently. They offer the ability to efficiently search for a key as well as find the *min* and *max* elements, look for the successor or predecessor of a search key (which itself need not be present in the BST), and enumerate the keys in a range in sorted order.

BSTs are similar to arrays in that the stored values (the "keys") are stored in a sorted order. However, unlike with a sorted array, keys can be added to and deleted from a BST efficiently. Specifically, a BST is a binary tree as defined in Chapter 9 in which the nodes store keys that are comparable, e.g., integers or strings. The keys stored at nodes have to respect the BST property—the key stored at a node is greater than or equal to the keys stored at the nodes of its left subtree and less than or equal to the keys stored in the nodes of its right subtree. Figure 14.1 on the following page shows a BST whose keys are the first 16 prime numbers.

Key lookup, insertion, and deletion take time proportional to the height of the tree, which can in worst-case be $O(n)$, if insertions and deletions are naively implemented. However, there are implementations of insert and delete which guarantee that the tree has height $O(\log n)$. These require storing and updating additional data at the tree nodes. Red-black trees are an example of height-balanced BSTs and are widely used in data structure libraries.

A common mistake with BSTs is that an object that's present in a BST is not updated. The consequence is that a lookup for that object returns false, even though it's still in the BST. As a rule, avoid putting mutable objects in a BST. Otherwise, when a mutable object that's in a BST is to be updated, always first remove it from the tree, then update it, then add it back.

The BST prototype is as follows:

```
class BSTNode:
    def __init__(self, data=None, left=None, right=None):
        self.data, self.left, self.right = data, left, right
```

Binary search trees boot camp

Searching is the single most fundamental application of BSTs. Unlike a hash table, a BST offers the ability to find the min and max elements, and find the next largest/next smallest element. These operations, along with lookup, delete and find, take time $O(\log n)$ for library implementations of BSTs. Both BSTs and hash tables use $O(n)$ space—in practice, a BST uses slightly more space.

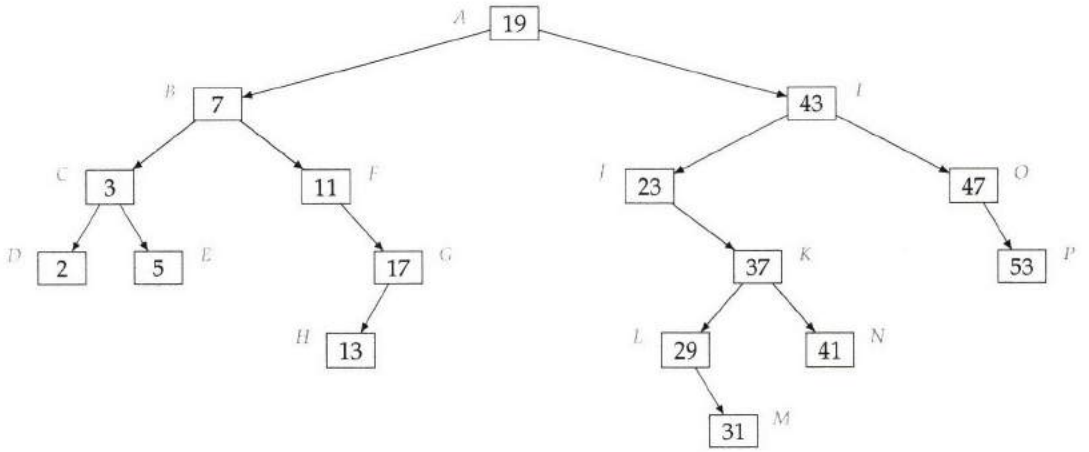


Figure 14.1: An example of a BST.

The following program demonstrates how to check if a given value is present in a BST. It is a nice illustration of the power of recursion when operating on BSTs.

```
def search_bst(tree, key):
    return (tree
            if not tree or tree.data == key else search_bst(tree.left, key)
            if key < tree.data else search_bst(tree.right, key))
```

Since the program descends tree with in each step, and spends $O(1)$ time per level, the time complexity is $O(h)$, where h is the height of the tree.

With a BST you can **iterate** through elements in **sorted order** in time $O(n)$ (regardless of whether it is balanced).
Some problems need a **combination of a BST and a hashtable**. For example, if you insert student objects into a BST and entries are ordered by GPA, and then a student's GPA needs to be updated and all we have is the student's name and new GPA, we cannot find the student by name without a full traversal. However, with an additional hash table, we can directly go to the corresponding entry in the tree.
Sometimes, it's necessary to **augment** a BST to make it possible to manipulate more complicated data, e.g., intervals, and efficiently support more complex queries, e.g., the number of elements in a range (on Page 214).
The BST property is a **global property**—a binary tree may have the property that each node's key is greater than the key at its left child and smaller than the key at its right child, but it may not be a BST.

Table 14.1: Top Tips for Binary Search Trees

Know your binary search tree libraries

Some of the problems in this chapter entail writing a BST class; for others, you can use a BST library. Python does not come with a built-in BST library.

The `sortedcontainers` module the best-in-class module for sorted sets and sorted dictionaries—it is performant, has a clean API that is well documented, with a responsive community. The underlying data structure is a sorted list of sorted lists. Its asymptotic time complexity for inserts and deletes is $O(\sqrt{n})$) since these operations entail insertion into a list of length roughly $\sqrt{n}$, rather than the $O(\log n)$) of balanced BSTs. In practice, this is not an issue, since CPUs are highly optimized for block data movements.

In the interests of pedagogy, we have elected to use the `bintrees` module which implements sorted sets and sorted dictionaries using balanced BSTs. However, any reasonable interviewer should accept `sortedcontainers` wherever we use `bintrees`.

Below, we describe the functionalities added by `bintrees`.

- `insert(e)` inserts new element e in the BST.
- `discard(e)` removes e in the BST if present.
- `min_item()/max_item()` yield the smallest and largest key-value pair in the BST.
- `min_key()/max_key()` yield the smallest and largest key in the BST.
- `pop_min()/pop_max()` remove the return the smallest and largest key-value pair in the BST.

It's particularly important to note that these operations take $O(\log n)$, since they are backed by the underlying tree.

The following program illustrates the use of `bintrees`.

```
t = bintrees.RBTree([(5, 'Alfa'), (2, 'Bravo'), (7, 'Charlie'), (3, 'Delta'),
                     (6, 'Echo')])

print(t[2])  # 'Bravo'

print(t.min_item(), t.max_item())  # (2, 'Bravo'), (7, 'Charlie')

# {2: 'Bravo', 3: 'Delta', 5: 'Alfa', 6: 'Echo', 7: 'Charlie', 9: 'Golf'}
t.insert(9, 'Golf')
print(t)

print(t.min_key(), t.max_key())  # 2, 9

t.discard(3)
print(t)  # {2: 'Bravo', 5: 'Alfa', 6: 'Echo', 7: 'Charlie', 9: 'Golf'}

# a = (2: 'Bravo')
a = t.pop_min()
print(t)  # {5: 'Alfa', 6: 'Echo', 7: 'Charlie', 9: 'Golf'}

# b = (9, 'Golf')
b = t.pop_max()
print(t)  # {5: 'Alfa', 6: 'Echo', 7: 'Charlie'}
```

14.1 Test if a binary tree satisfies the BST property

Write a program that takes as input a binary tree and checks if the tree satisfies the BST property.

Hint: Is it correct to check for each node that its key is greater than or equal to the key at its left child and less than or equal to the key at its right child?

Solution: A direct approach, based on the definition of a BST, is to begin with the root, and compute the maximum key stored in the root's left subtree, and the minimum key in the root's right subtree. We check that the key at the root is greater than or equal to the maximum from the left subtree and less than or equal to the minimum from the right subtree. If both these checks pass, we recursively check the root's left and right subtrees. If either check fails, we return false.

Computing the minimum key in a binary tree is straightforward: we take the minimum of the key stored at its root, the minimum key of the left subtree, and the minimum key of the right subtree. The maximum key is computed similarly. Note that the minimum can be in either subtree, since a general binary tree may not satisfy the BST property.

The problem with this approach is that it will repeatedly traverse subtrees. In the worst-case, when the tree is a BST and each node's left child is empty, the complexity is $O(n^2)$, where n is the number of nodes. The complexity can be improved to $O(n)$ by caching the largest and smallest keys at each node; this requires $O(n)$ additional storage for the cache.

We now present two approaches which have $O(n)$ time complexity.

The first approach is to check constraints on the values for each subtree. The initial constraint comes from the root. Every node in its left (right) subtree must have a key less than or equal (greater than or equal) to the key at the root. This idea generalizes: if all nodes in a tree must have keys in the range $[l, u]$, and the key at the root is w (which itself must be between $[l, u]$, otherwise the requirement is violated at the root itself), then all keys in the left subtree must be in the range $[l, w]$, and all keys stored in the right subtree must be in the range $[w, u]$.

As a concrete example, when applied to the BST in Figure 14.1 on Page 198, the initial range is $[-\infty, \infty]$. For the recursive call on the subtree rooted at B, the constraint is $[-\infty, 19]$; the 19 is the upper bound required by A on its left subtree. For the recursive call starting at the subtree rooted at F, the constraint is $[7, 19]$. For the recursive call starting at the subtree rooted at K, the constraint is $[23, 43]$. The binary tree in Figure 9.1 on Page 112 is identified as not being a BST when the recursive call reaches C—the constraint is $[-\infty, 6]$, but the key at F is 271, so the tree cannot satisfy the BST property.

```
def is_binary_tree_bst(tree, low_range=float('-inf'), high_range=float('inf')):
    if not tree:
        return True
    elif not low_range <= tree.data <= high_range:
        return False
    return (is_binary_tree_bst(tree.left, low_range, tree.data)
            and is_binary_tree_bst(tree.right, tree.data, high_range))
```

The time complexity is $O(n)$, and the additional space complexity is $O(h)$, where h is the height of the tree.

Alternatively, we can use the fact that an inorder traversal visits keys in sorted order. Furthermore, if an inorder traversal of a binary tree visits keys in sorted order, then that binary tree must be a BST. (This follows directly from the definition of a BST and the definition of an inorder walk.) Thus we can check the BST property by performing an inorder traversal, recording the key stored at the last visited node. Each time a new node is visited, its key is compared with the key of the previously visited node. If at any step in the walk, the key at the previously visited node is greater than the node currently being visited, we have a violation of the BST property.

All these approaches explore the left subtree first. Therefore, even if the BST property does not hold at a node which is close to the root (e.g., the key stored at the right child is less than the key

stored at the root), their time complexity is still $O(n)$.

We can search for violations of the BST property in a BFS manner, thereby reducing the time complexity when the property is violated at a node whose depth is small.

Specifically, we use a queue, where each queue entry contains a node, as well as an upper and a lower bound on the keys stored at the subtree rooted at that node. The queue is initialized to the root, with lower bound $-\infty$ and upper bound ∞. We iteratively check the constraint on each node. If it violates the constraint we stop—the BST property has been violated. Otherwise, we add its children along with the corresponding constraint.

For the example in Figure 14.1 on Page 198, we initialize the queue with $(A, [-\infty, \infty])$. Each time we pop a node, we first check the constraint. We pop the first entry, $(A, [-\infty, \infty])$, and add its children, with the corresponding constraints, i.e., $(B, [-\infty, 19])$ and $(I, [19, \infty])$. Next we pop $(B, [-\infty, 19])$, and add its children, i.e., $(C, [-\infty, 7])$ and $(D, [7, 19])$. Continuing through the nodes, we check that all nodes satisfy their constraints, and thus verify the tree is a BST.

If the BST property is violated in a subtree consisting of nodes within a particular depth, the violation will be discovered without visiting any nodes at a greater depth. This is because each time we enqueue an entry, the lower and upper bounds on the node's key are the tightest possible.

```
def is_binary_tree_bst(tree):
    QueueEntry = collections.namedtuple('QueueEntry', ('node', 'lower',
                                                       'upper'))
    bfs_queue = collections.deque(
        [QueueEntry(tree, float('-inf'), float('inf'))])

    while bfs_queue:
        front = bfs_queue.popleft()
        if front.node:
            if not front.lower <= front.node.data <= front.upper:
                return False
            bfs_queue += [
                QueueEntry(front.node.left, front.lower, front.node.data),
                QueueEntry(front.node.right, front.node.data, front.upper)
            ]
    return True
```

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

14.2 Find the first key greater than a given value in a BST

Write a program that takes as input a BST and a value, and returns the first key that would appear in an inorder traversal which is greater than the input value. For example, when applied to the BST in Figure 14.1 on Page 198 you should return 29 for input 23.

Hint: Perform binary search, keeping some additional state.

Solution: We can find the desired node in $O(n)$ time, where n is the number of nodes in the BST, by doing an inorder walk. This approach does not use the BST property.

A better approach is to use the BST search idiom. We store the best candidate for the result and update that candidate as we iteratively descend the tree, eliminating subtrees by comparing the keys stored at nodes with the input value. Specifically, if the current subtree's root holds a value less than or equal to the input value, we search the right subtree. If the current subtree's root stores

a key that is greater than the input value, we search in the left subtree, updating the candidate to the current root. Correctness follows from the fact that whenever we first set the candidate, the desired result must be within the tree rooted at that node.

For example, when searching for the first node whose key is greater than 23 in the BST in Figure 14.1 on Page 198, the node sequence is A, I, J, K, L. Since L has no left child, its key, 29, is the result.

```
def find_first_greater_than_k(tree, k):
    subtree, first_so_far = tree, None
    while subtree:
        if subtree.data > k:
            first_so_far, subtree = subtree, subtree.left
        else:  # Root and all keys in left subtree are <= k, so skip them.
            subtree = subtree.right
    return first_so_far
```

The time complexity is $O(h)$, where h is the height of the tree. The space complexity is $O(1)$.

Variant: Write a program that takes as input a BST and a value, and returns the node whose key equals the input value and appears first in an inorder traversal of the BST. For example, when applied to the BST in Figure 14.2, your program should return Node B for 108, Node G for 285, and null for 143.

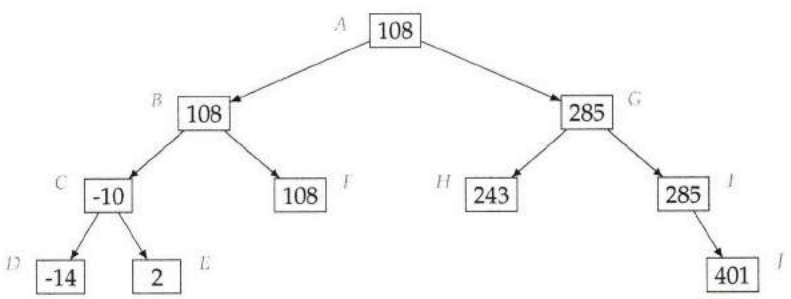


Figure 14.2: A BST with duplicate keys.

14.3 Find the k largest elements in a BST

A BST is a sorted data structure, which suggests that it should be possible to find the k largest keys easily.

Write a program that takes as input a BST and an integer k, and returns the k largest elements in the BST in decreasing order. For example, if the input is the BST in Figure 14.1 on Page 198 and $k = 3$, your program should return $\langle 53, 47, 43 \rangle$.

Hint: What does an inorder traversal yield?

Solution: The brute-force approach is to do an inorder traversal, which enumerates keys in ascending order, and return the last k visited nodes. A queue is ideal for storing visited nodes, since it makes it easy to evict nodes visited more than k steps previously. A drawback of this approach is

that it potentially processes many nodes that cannot possibly be in the result, e.g., if k is small and the left subtree is large.

A better approach is to begin with the desired nodes, and work backwards. We do this by recursing first on the right subtree and then on the left subtree. This amounts to a reverse-inorder traversal. For the BST in Figure 14.1 on Page 198, the reverse inorder visit sequence is $\langle P, O, I, N, K, M, L, J, A, G, H, F, B, E, C, D \rangle$.

As soon as we visit k nodes, we can halt. The code below uses a dynamic array to store the desired keys. As soon as the array has k elements, we return. We store newer nodes at the end of the array, as per the problem specification.

To find the five biggest keys in the tree in Figure 14.1 on Page 198, we would recurse on A, I, O, P, in that order. Returning from recursive calls, we would visit P, O, I, in that order, and add their keys to the result. Then we would recurse on J, K, N, in that order. Finally, we would visit N and then K, adding their keys to the result. Then we would stop, since we have five keys in the array.

```
def find_k_largest_in_bst(tree, k):
    def find_k_largest_in_bst_helper(tree):
        # Perform reverse inorder traversal.
        if tree and len(k_largest_elements) < k:
            find_k_largest_in_bst_helper(tree.right)
            if len(k_largest_elements) < k:
                k_largest_elements.append(tree.data)
                find_k_largest_in_bst_helper(tree.left)

    k_largest_elements = []
    find_k_largest_in_bst_helper(tree)
    return k_largest_elements
```

The time complexity is $O(h + k)$, which can be much better than performing a conventional inorder walk, e.g., when the tree is balanced and k is small. The complexity bound comes from the observation that the number of times the program descends in the tree can be at most h more than the number of times it ascends the tree, and each ascent happens after we visit a node in the result. After k nodes have been added to the result, the program stops.

14.4 Compute the LCA in a BST

Since a BST is a specialized binary tree, the notion of lowest common ancestor, as expressed in Problem 9.4 on Page 118, holds for BSTs too.

In general, computing the LCA of two nodes in a BST is no easier than computing the LCA in a binary tree, since structurally a binary tree can be viewed as a BST where all the keys are equal. However, when the keys are distinct, it is possible to improve on the LCA algorithms for binary trees.

Design an algorithm that takes as input a BST and two nodes, and returns the LCA of the two nodes. For example, for the BST in Figure 14.1 on Page 198, and nodes C and G, your algorithm should return B. Assume all keys are distinct. Nodes do not have references to their parents.

Hint: Take advantage of the BST property.

Solution: In Solution 9.3 on Page 117 we presented an algorithm for this problem in the context of binary trees. The idea underlying that algorithm was to do a postorder traversal—the LCA is the

first node visited after the two nodes whose LCA we are to compute have been visited. The time complexity was $O(n)$, where n is the number of nodes in the tree.

This approach can be improved upon when operating on BSTs with distinct keys. Consider the BST in Figure 14.1 on Page 198 and nodes C and G. Since both C and G hold keys that are smaller than A's key, their LCA must lie in A's left subtree. Examining B, since C's key is less than B's key, and B's key is less than G's key. B must be the LCA of C and G.

Let s and b be the two nodes whose LCA we are to compute, and without loss of generality assume the key at s is smaller. (Since the problem specified keys are distinct, it cannot be that s and b hold equal keys.) Consider the key stored at the root of the BST. There are four possibilities:

- If the root's key is the same as that stored at s or at b, we are done—the root is the LCA.
- If the key at s is smaller than the key at the root, and the key at b is greater than the key at the root, the root is the LCA.
- If the keys at s and b are both smaller than that at the root, the LCA must lie in the left subtree of the root.
- If both keys are larger than that at the root, then the LCA must lie in the right subtree of the root.

```
# Input nodes are nonempty and the key at s is less than or equal to that at b.
def find_LCA(tree, s, b):
    while tree.data < s.data or tree.data > b.data:
        # Keep searching since tree is outside of [s, b].
        while tree.data < s.data:
            tree = tree.right  # LCA must be in tree's right child.
        while tree.data > b.data:
            tree = tree.left  # LCA must be in tree's left child.
    # Now, s.data <= tree.data && tree.data <= b.data.
    return tree
```

Since we descend one level with each iteration, the time complexity is $O(h)$, where h is the height of the tree.

14.5 Reconstruct a BST from traversal data

As discussed in Problem 9.12 on Page 125 there are many different binary trees that yield the same sequence of visited nodes in an inorder traversal. This is also true for preorder and postorder traversals. Given the sequence of nodes that an inorder traversal sequence visits and either of the other two traversal sequences, there exists a unique binary tree that yields those sequences. Here we study if it is possible to reconstruct the tree with less traversal information when the tree is known to be a BST.

It is critical that the elements stored in the tree be unique. If the root contains key v and the tree contains more occurrences of v, we cannot always identify from the sequence whether the subsequent vs are in the left subtree or the right subtree. For example, for the tree rooted at G in Figure 14.2 on Page 202 the preorder traversal sequence is $285, 243, 285, 401$. The same preorder traversal sequence is seen if 285 appears in the left subtree as the right child of the node with key 243 and 401 is at the root's right child.

Suppose you are given the sequence in which keys are visited in an inorder traversal of a BST, and all keys are distinct. Can you reconstruct the BST from the sequence? If so, write a program to do so. Solve the same problem for preorder and postorder traversal sequences.

Hint: Draw the five BSTs on the keys $1, 2, 3$, and the corresponding traversal orders.

Solution: First, with some experimentation, we see the sequence of keys generated by an inorder traversal is not enough to reconstruct the tree. For example, the key sequence $\langle 1, 2, 3 \rangle$ corresponds to five distinct BSTs as shown in Figure 14.3.

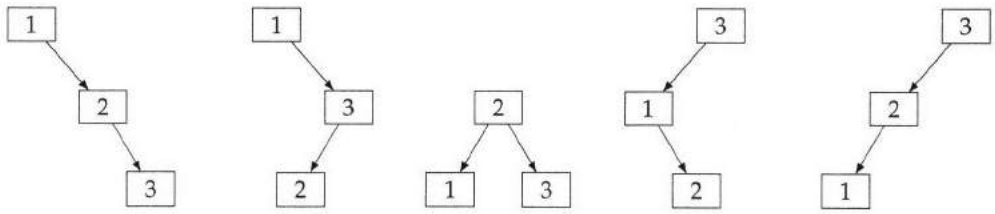


Figure 14.3: Five distinct BSTs for the traversal sequence $\langle 1, 2, 3 \rangle$.

However, the story for a preorder sequence is different. As an example, consider the preorder key sequence $\langle 43, 23, 37, 29, 31, 41, 47, 53 \rangle$. The root must hold 43, since it's the first visited node. The left subtree contains keys less than 43, i.e., $23, 37, 29, 31, 41$, and the right subtree contains keys greater than 43, i.e., $47, 53$. Furthermore, $\langle 23, 37, 29, 31, 41 \rangle$ is exactly the preorder sequence for the left subtree and $\langle 47, 53 \rangle$ is exactly the preorder sequence for the right subtree. We can recursively reason that 23 and 47 are the roots of the left and right subtree, and continue to build the entire tree, which is exactly the subtree rooted at Node I in Figure 14.1 on Page 198.

Generalizing, in any preorder traversal sequence, the first key corresponds to the root. The subsequence which begins at the second element and ends at the last key less than the root, corresponds to the preorder traversal of the root's left subtree. The final subsequence, consisting of keys greater than the root corresponds to the preorder traversal of the root's right subtree. We recursively reconstruct the BST by recursively reconstructing the left and right subtrees from the two subsequences then adding them to the root.

```
def rebuild_bst_from_preorder(preorder_sequence):
    if not preorder_sequence:
        return None

    transition_point = next((i for i, a in enumerate(preorder_sequence)
                             if a > preorder_sequence[0]),
                            len(preorder_sequence))
    return BSTNode(
        preorder_sequence[0],
        rebuild_bst_from_preorder(preorder_sequence[1:transition_point]),
        rebuild_bst_from_preorder(preorder_sequence[transition_point:]))
```

The worst-case input for this algorithm is the pre-order sequence corresponding to a left-skewed tree. The worst-case time complexity satisfies the recurrence $W(n) = W(n-1) + O(n)$, which solves to $O(n^2)$. The best-case input is a sequence corresponding to a right-skewed tree, and the corresponding time complexity is $O(n)$. When the sequence corresponds to a balanced BST, the time complexity is given by $B(n) = 2B(n/2) + O(n)$, which solves to $O(n \log n)$.

The implementation above potentially iterates over nodes multiple times, which is wasteful. A better approach is to reconstruct the left subtree in the same iteration as identifying the nodes which lie in it. The code shown below takes this approach. The intuition is that we do not want to iterate from first entry after the root to the last entry smaller than the root, only to go back and

partially repeat this process for the root's left subtree. We can avoid repeated passes over nodes by including the range of keys we want to reconstruct the subtrees over. For example, looking at the preorder key sequence $\langle 43, 23, 37, 29, 31, 41, 47, 53 \rangle$, instead of recursing on $\langle 23, 37, 29, 31, 41 \rangle$ (which would involve an iteration to get the last element in this sequence). We can directly recur on $\langle 23, 37, 29, 31, 41, 47, 53 \rangle$, with the constraint that we are building the subtree on nodes whose keys are less than 43.

```
def rebuild_bst_from_preorder(preorder_sequence):
    def rebuild_bst_from_preorder_on_value_range(lower_bound, upper_bound):
        if root_idx[0] == len(preorder_sequence):
            return None

        root = preorder_sequence[root_idx[0]]
        if not lower_bound <= root <= upper_bound:
            return None
        root_idx[0] += 1
        # Note that rebuild_bst_from_preorder_on_value_range updates root_idx[0].
        # So the order of following two calls are critical.
        left_subtree = rebuild_bst_from_preorder_on_value_range(
            lower_bound, root)
        right_subtree = rebuild_bst_from_preorder_on_value_range(
            root, upper_bound)
        return BSTNode(root, left_subtree, right_subtree)

    root_idx = [0]  # Tracks current subtree.
    return rebuild_bst_from_preorder_on_value_range(float('-inf'), float('inf'))
```

The worst-case time complexity is $O(n)$, since it performs a constant amount of work per node. Note the similarity to Solution 24.20 on Page 377.

A postorder traversal sequence also uniquely specifies the BST, and the algorithm for reconstructing the BST is very similar to that for the preorder case.

14.6 Find the closest entries in three sorted arrays

Design an algorithm that takes three sorted arrays and returns one entry from each such that the minimum interval containing these three entries is as small as possible. For example, if the three arrays are $\langle 5, 10, 15 \rangle$, $\langle 3, 6, 9, 12, 15 \rangle$, and $\langle 8, 16, 24 \rangle$, then 15, 15, 16 lie in the smallest possible interval.

Hint: How would you proceed if you needed to pick three entries in a single sorted array?

Solution: The brute-force approach is to try all possible triples, e.g., with three nested for loops. The length of the minimum interval containing a set of numbers is simply the difference of the maximum and the minimum values in the triple. The time complexity is $O(lmn)$, where l, m, n are the lengths of each of the three arrays.

The brute-force approach does not take advantage of the sortedness of the input arrays. For the example in the problem description, the smallest intervals containing $(5, 3, 16)$ and $(5, 3, 24)$ must be larger than the smallest interval containing $(5, 3, 8)$ (since 8 is the maximum of 5, 3, 8, and $8 < 16 < 24$).

Let's suppose we begin with the triple consisting of the smallest entries in each array. Let s be the minimum value in the triple and t the maximum value in the triple. Then the smallest interval