17

Sort and Search

Introduction to Sort and Search

When sorting and searching items in data structures, efficiency is key. This chapter covers common sorting methods and their roles in efficient searching.

First, we present a comparison of the time and space complexities for various sorting algorithms. Below, n denotes the number of elements in the data structure:

Algorithm	Time complexity			Space complexity
	Best case	Average case	Worst case	
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Merge sort	$O(n\log(n))$	$O(n\log(n))$	$O(n\log(n))$	$O(n)$
Quicksort	$O(n\log(n))$	$O(n\log(n))$	$O(n^2)$	$O(\log(n))$ (average) $O(n)$ (worst)
Heapsort	$O(n\log(n))$	$O(n\log(n))$	$O(n\log(n))$	$O(1)$
Counting sort[1]	$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(n+k)$
Bucket sort[2]	$O(n+k)$	$O(n+k)$	$O(n^2)$	$O(n+k)$
Radix sort[3]	$O(d\cdot(n+k))$	$O(d\cdot(n+k))$	$O(d\cdot(n+k))$	$O(n+k)$

This chapter focuses on merge sort, quicksort, and counting sort. There is additional information on the other algorithms in the references below, as well as a tool for visualizing how these algorithms work.

Fundamental concepts for sorting algorithms

1 In counting sort, k represents the range of the values in the input array.

2 In bucket sort, k represents the number of buckets used.

3 In radix sort, k represents the range of the inputs and d represents the number of digits in the maximum element.

Here's a list of fundamental attributes of sorting algorithms you should be familiar with:

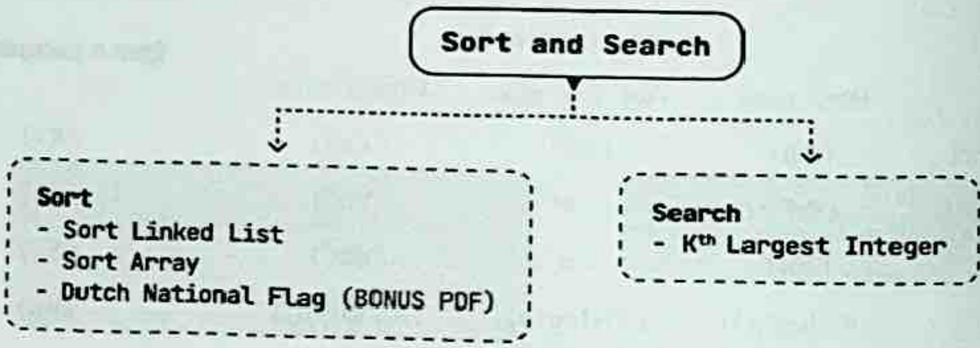
- **Stability:** A sorting algorithm is considered stable if it preserves the relative order of equal elements in the sorted output. If two elements have equal values, their order in the sorted output is the same as in the input.
- **In-place sorting:** An in-place sorting algorithm transforms the input using a constant amount of extra storage space. It involves sorting the elements within the original data structure.
- **Comparison-based sorting:** Comparison-based sorting algorithms sort elements by comparing them pairwise. These algorithms typically have a lower bound of $O(n\log(n))$, whereas non-comparison-based sorting algorithms can achieve linear time complexity, but require specific assumptions about the input data.

These concepts are referenced throughout the problems in this chapter.

Real-world Example

Sorting products by category: When users search for products, the platform often sorts the results based on various criteria such as lowest to highest price, highest to lowest rating, or even relevance to the search query. Efficient sorting algorithms ensure large datasets of products can be quickly arranged according to user preferences.

Chapter Outline



References

[1] Insertion sort: https://en.wikipedia.org/wiki/Insertion_sort

[2] Selection sort: https://en.wikipedia.org/wiki/Selection_sort

[3] Bubble sort: https://en.wikipedia.org/wiki/Bubble_sort

[4] Heapsort: https://en.wikipedia.org/wiki/Heapsort

[5] Bucket sort: https://en.wikipedia.org/wiki/Bucket_sort

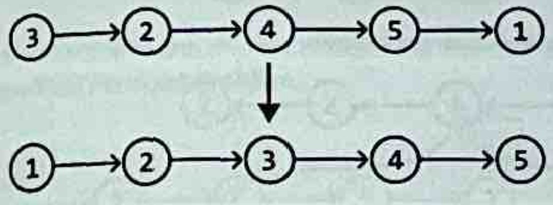
[6] Radix sort: https://en.wikipedia.org/wiki/Radix_sort

[7] Visualize sorting algorithms: https://www.toptal.com/developers/sorting-algorithms

Sort Linked List

Given the head of a singly linked list, sort the linked list in ascending order.

Example:



Intuition

Let's start by finding a sorting algorithm that allows us to sort a linked list.

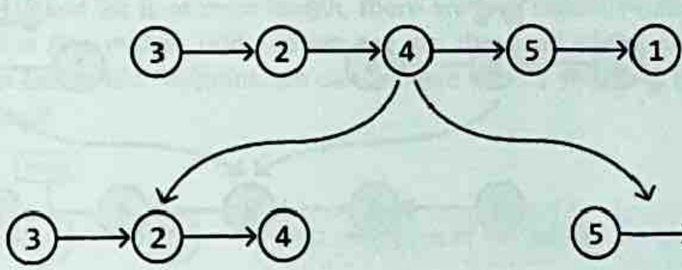
Choosing a sorting algorithm

We're tasked with sorting a linked list, not an array. This distinction is crucial because algorithms like quicksort rely on random access through indexing, which linked lists don't support. Merge sort is a great $O(n \log(n))$ time option, where n denotes the length of the linked list, because it does not require random access and works well with linked lists, as we'll see in this explanation.

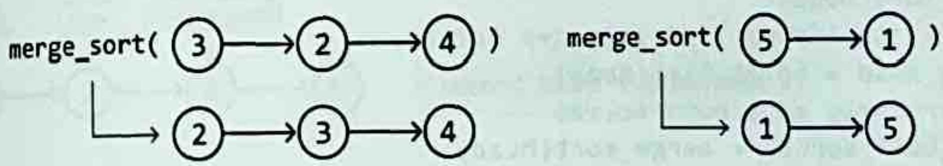
Merge sort

The merge sort algorithm uses a divide and conquer strategy. At a high level, it can be broken down into three steps:

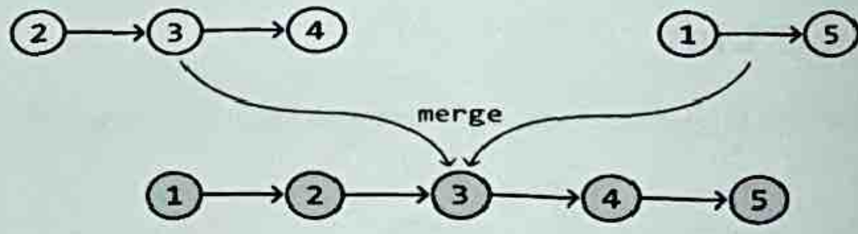
1. Split the linked list into two halves:



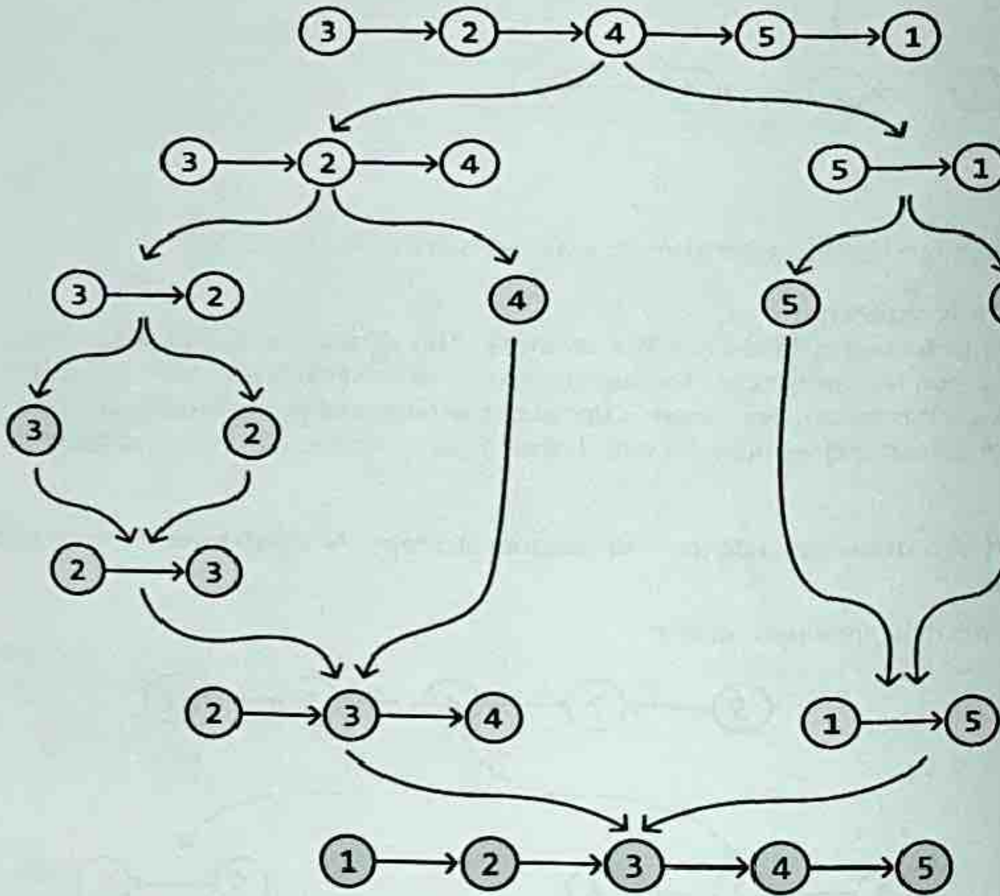
2. Recursively sort both halves:



3. Merge the halves back together in a sorted manner to form a single sorted list:



We can see what the entire process looks like in the diagram below:



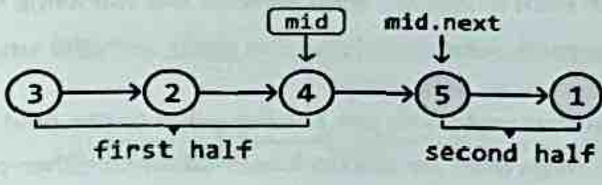
This is what this process looks like as pseudocode:

```
def merge_sort(head):
    # Split the linked list into two halves.
    second_head = split_list(head)
    # Recursively sort both halves.
    first_half_sorted = merge_sort(head)
    second_half_sorted = merge_sort(second_head)
    # Merge the sorted sublists.
    return merge(first_half_sorted, second_half_sorted)
```

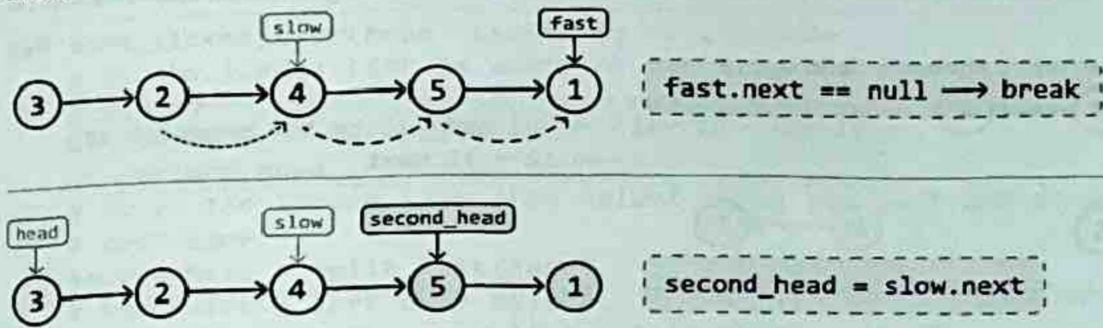
Let's discuss in more detail how to split a linked list, and how to merge two sorted linked lists.

Splitting the linked list in half

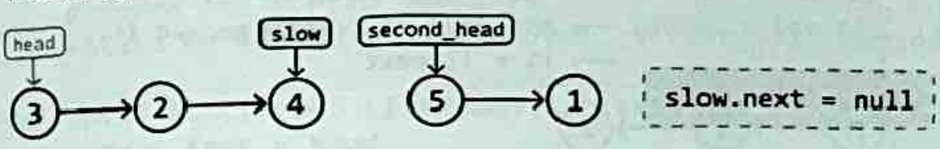
To split a linked list in half, we need access to its middle node because the node next to the middle node can represent the head of the second linked list:



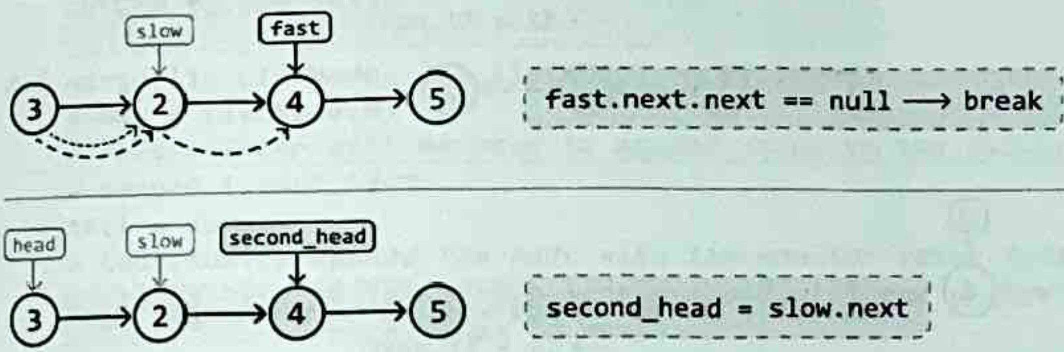
We can retrieve the middle node using the **fast and slow pointer** technique, as described in the *Linked List Midpoint* problem:



Then, we just need to disconnect the two halves by setting `slow.next` to null:

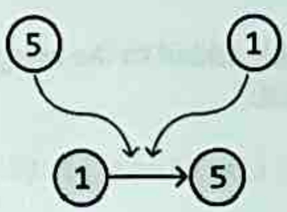


Note that when the linked list is of even length, there are two middle nodes. We want the slow pointer to stop at the first middle node so we can get the head of the second half more easily. As mentioned in *Linked List Midpoint*, we can achieve this by stopping the fast pointer when `fast.next.next` is null:



Merging two sorted linked lists

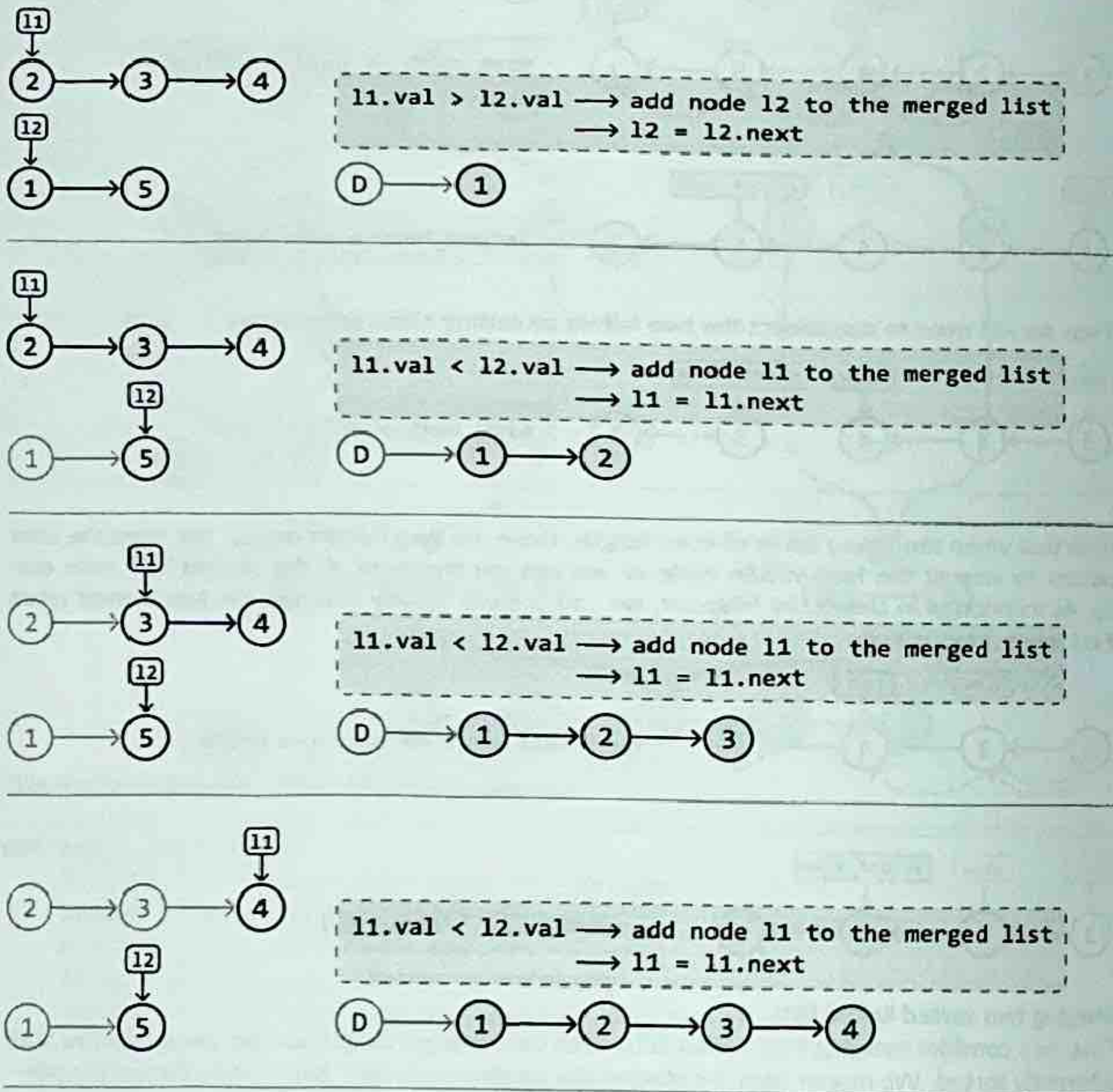
First, let's consider merging two linked lists, each containing a single node, meaning they're both inherently sorted. We merge them by placing the smaller node first, followed by the other node:



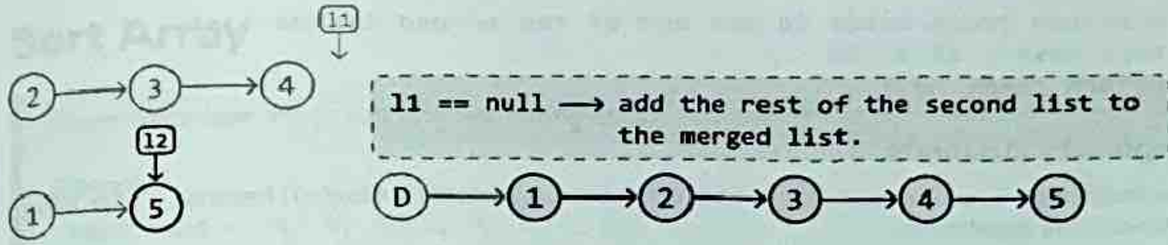
With that established, how would we merge two longer linked lists? To do this, we can set two pointers, one at the start of each linked list, then perform the following steps:

1. Compare the nodes at each pointer and add the node with the smaller value to the merged linked list.
2. Advance the pointer at that node with the smaller value to the next node in its linked list.
3. Repeat the above two steps until we can no longer advance either pointer.

Before discussing the final step, observe how these steps are applied to the following two sorted linked lists. We can use a dummy node to point to the head of the merged linked list:



If one of the linked lists has been entirely added to the merged list, we can just add the rest of the other linked list to the merged linked list:



Implementation

```python
def sort_linked_list(head: ListNode) -> ListNode:
    # If the linked list is empty or has only one element, it's already
    # sorted.
    if not head or not head.next:
        return head
    # Split the linked list into halves using the fast and slow pointer
    # technique.
    second_head = split_list(head)
    # Recursively sort both halves.
    first_half_sorted = sort_linked_list(head)
    second_half_sorted = sort_linked_list(second_head)
    # Merge the sorted sublists.
    return merge(first_half_sorted, second_half_sorted)

def split_list(head: ListNode) -> ListNode:
    slow = fast = head
    while fast.next and fast.next.next:
        slow = slow.next
        fast = fast.next.next
    second_head = slow.next
    slow.next = None
    return second_head

def merge(l1: ListNode, l2: ListNode) -> ListNode:
    dummy = ListNode(0)
    # This pointer will be used to append nodes to the tail of the
    # merged linked list.
    tail = dummy
    # Continually append the node with the smaller value from each
    # linked list to the merged linked list until one of the linked
    # lists has no more nodes to merge.
    while l1 and l2:
        if l1.val < l2.val:
            tail.next = l1
            l1 = l1.next
        else:
            tail.next = l2
            l2 = l2.next
        tail = tail.next
    # One of the two linked lists could still have nodes remaining.
```

```
    # Attach those nodes to the end of the merged linked list.
    tail.next = l1 or l2
    return dummy.next
```

Complexity Analysis

Time complexity: The time complexity of `sort_linked_list` is $O(n \log(n))$ because it uses merge sort. Here's the breakdown:

- The linked list is recursively split until each sublist contains only one node. This splitting process happens about $\log_2(n)$ times because each split reduces the size of the linked list by half.
- At each level, we merge the split linked lists. Merging all elements at one level takes about n operations.
- Since there are $\log_2(n)$ levels of splitting and merging, and there are n operations at each level, the time complexity is $O(n \log(n))$.

Space complexity: The space complexity is $O(\log(n))$ due to the recursive call stack, which can grow up to $\log_2(n)$ in height.

Stable Sorting

Merge sort is a stable sorting algorithm. This is important in scenarios where the original order of equal elements must be preserved. For example, if we are sorting a list of nodes that share the same value, but have an additional attribute storing the time they were created, then it's important for the relative order of these nodes to stay the same. Stability can be important for database systems; for instance, where stable sorting is required to maintain data integrity and consistency across multiple operations.

Sort Array

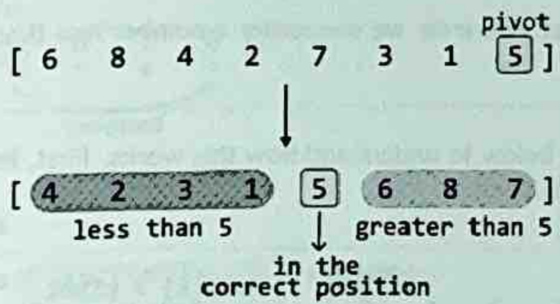
> Given an integer array, sort it in ascending order.
>
> **Example:**
>
> Input: nums = [6, 8, 4, 2, 7, 3, 1, 5]
> Output: [1, 2, 3, 4, 5, 6, 7, 8]

Intuition

This problem is quite open-ended because there are many sorting algorithms we could use to sort an array, each with its own pros and cons. In this explanation, we focus on the quicksort algorithm, which runs in $O(n \log(n))$ time on average, where n denotes the length of the array.

Quicksort

Conceptually, the goal of quicksort is to sort the array by **placing each number in its sorted position one at a time**. To correctly position a number, we move all numbers smaller than it to its left, and all numbers larger than it to its right. At each step, we call this number the pivot. We discuss how a pivot is selected later in the explanation.



This process is called **partitioning** because we're dividing the array into two parts around the pivot:

1. The left part with elements smaller than the pivot.
2. The right part with elements larger than the pivot.

Note that neither the left nor the right part of the pivot value need to be sorted. All that matters is that the pivot is in the correct position.

After partitioning, we just need to **sort the left and right parts**. We can do this by recursively calling quicksort on both of them. This makes quicksort a divide and conquer strategy. The pseudocode for this is provided below:

```
def quicksort(nums, left, right):
    # Partition the array and obtain the index of the pivot.
    pivot_index = partition(nums, left, right)
    # Sort the left and right parts.
    quicksort(nums, left, pivot_index - 1)
    quicksort(nums, pivot_index + 1, right)
```

Let's discuss the partitioning process in more detail.

Partitioning

There are two primary steps to partitioning:

1. Selecting the pivot.
2. Rearranging the elements so elements smaller than the pivot are on its left, and elements larger than the pivot are on its right.

<u>Selecting the pivot</u>

We can actually choose any number as the pivot. The method of selecting a pivot can be optimized, which is discussed later, but for simplicity, we can choose the rightmost number as the pivot:

```
                                 pivot
[ 6    8    4    2    7    3    1   [5] ]
  0    1    2    3    4    5    6    7
```

<u>Rearranging elements around the pivot</u>

Let's see if we can do this in linear time without using extra space.

If we can ensure all numbers less than the pivot are placed to the left, then numbers greater than or equal to the pivot will consequently be placed to the right. This can be done using two pointers:

- One pointer at the left of the array (`lo`) to position the numbers less than the pivot.
- One pointer to iterate through the array (`i`), looking for numbers less than the pivot.

The main idea here is that whenever we encounter a number less than the pivot, swap it with `nums[lo]`.

Let's look at the example below to understand how this works. First, keep advancing `i` until we find a number less than the pivot:

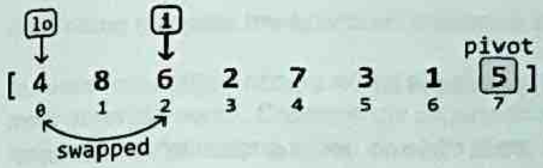
```
 lo i
  ↓↓                                 pivot
[ 6    8    4    2    7    3    1   [5] ]     nums[i] > pivot
  0    1    2    3    4    5    6    7          ──→ continue
```

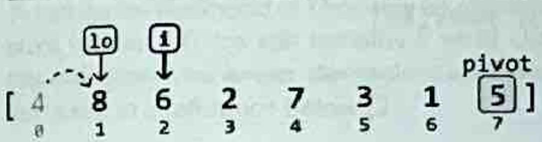
```
 lo   i
  ↓   ↓                              pivot
[ 6    8    4    2    7    3    1   [5] ]     nums[i] > pivot
  0    1    2    3    4    5    6    7          ──→ continue
```

```
 lo        i
  ↓        ↓                         pivot
[ 6    8    4    2    7    3    1   [5] ]
  0    1    2    3    4    5    6    7
```

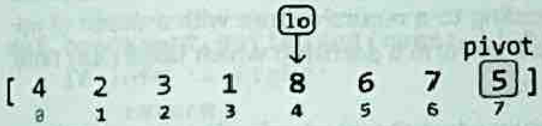
Once a number less than the pivot is found, move it to the left by swapping it with `nums[lo]`. Then, increment `lo` so it points to where the next number less than the pivot should be placed:

```
 lo        i
  ↓        ↓                         pivot     nums[i] < pivot
[ 6    8    4    2    7    3    1   [5] ]     ──→ swap(nums[i], nums[lo])
  0    1    2    3    4    5    6    7         ──→ lo += 1
```

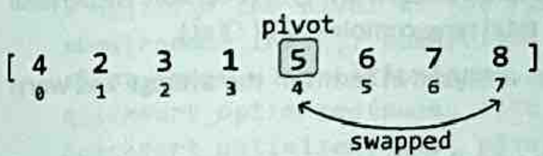




Continue this process until all numbers less than the pivot are on the left of the array:



The last step is to move the pivot to the correct position by swapping the pivot with `nums[lo]`:



Implementation

```
def sort_array(nums: List[int]) -> List[int]:
    quicksort(nums, 0, len(nums) - 1)
    return nums

def quicksort(nums: List[int], left: int, right: int) -> None:
    # Base case: if the subarray has 0 or 1 element, it's already
    # sorted.
    if left >= right:
        return
    # Partition the array and retrieve the pivot index.
    pivot_index = partition(nums, left, right)
    # Call quicksort on the left and right parts to recursively sort
    # them.
    quicksort(nums, left, pivot_index - 1)
    quicksort(nums, pivot_index + 1, right)

def partition(nums: List[int], left: int, right: int) -> int:
    pivot = nums[right]
    lo = left
    # Move all numbers less than the pivot to the left, which
    # consequently positions all numbers greater than or equal to the
    # pivot to the right.
```

```python
    for i in range(left, right):
        if nums[i] < pivot:
            nums[lo], nums[i] = nums[i], nums[lo]
            lo += 1
    # After partitioning, 'lo' will be positioned where the pivot should
    # be. So, swap the pivot number with the number at the 'lo' pointer.
    nums[lo], nums[right] = nums[right], nums[lo]
    return lo
```

Complexity Analysis

Time complexity: The time complexity of `sort_array` can be analyzed in terms of the average and worst cases:

- Average case: $O(n\log(n))$. In the average case, quicksort effectively divides the array into two roughly equal parts after each partition, leading to a recursive tree with a depth of approximately $\log_2(n)$. For each of these levels, we perform a partition which takes $O(n)$ time, resulting in a total time complexity of $O(n\log(n))$.
- Worst case: $O(n^2)$. The worst-case scenario occurs when the pivot selection consistently results in extremely unbalanced partitions, such as when the smallest or largest element is always chosen as a pivot, which is explained in more detail in the optimization. Uneven partitioning can result in a recursive depth as deep as n. For each of these n levels of recursion, we perform a partition which takes $O(n)$ time, resulting in a total time complexity of $O(n^2)$.

Space complexity: The space complexity can also be analyzed in terms of the average and worst cases:

- **Average case:** $O(\log(n))$. In the average case, the depth of the recursive call stack is approximately $\log_2(n)$.
- **Worst case:** $O(n)$. In the worst case, the depth of the recursive call stack can be as deep as n.

Optimization

As mentioned in the above complexity analysis, it's possible for a worst-case time complexity of $O(n^2)$ to occur. Let's dive into when this can happen. Consider the following array, which is already sorted:

```
[ 1   2   3   4   5 ]
```

If we perform quicksort on this array, we choose the rightmost element as the pivot and partition the other elements around this pivot. However, since this pivot is the largest element, there will be $n-1$ elements less than the pivot and 0 elements greater than or equal to it:

```
                     pivot
[ 1   2   3   4   [5] ]
     left part        └→ no right part
```

This creates quite an uneven partition. When we call quicksort on the left part, the same imbalance will occur, but with a left part consisting of $n-2$ elements:

```
                 pivot
[ 1   2   3   [4] ]
   left part      └→ no right part
```

Continuing this until the quicksort process is complete results in a recursion depth of *n*.

An uneven partition occurs when we choose an **extreme pivot**: one that's larger or smaller than most other elements. Consistently picking an extreme pivot can occur when the array is sorted in increasing or decreasing order, or when there are many duplicates in the array.

To reduce the likelihood of choosing an extreme pivot, we can modify quicksort to choose a **random pivot** instead. There still remains a small chance of consistently picking an extreme pivot, but this outcome is no longer dependent on the order of the input array. A more detailed answer is discussed in a reference below [2].

We can integrate this change into our current solution by randomly selecting an index and swapping its element with the rightmost element, before performing the partition. This way, we don't have to modify the partition function, as it uses the rightmost element:

```python
def quicksort_optimized(nums: List[int], left: int, right: int) -> None:
    if left >= right:
        return
    # Choose a pivot at a random index.
    random_index = random.randint(left, right)
    # Swap the randomly chosen pivot with the rightmost element to
    # position the pivot at the rightmost index.
    nums[random_index], nums[right] = nums[right], nums[random_index]
    pivot_index = partition(nums, left, right)
    quicksort_optimized(nums, left, pivot_index - 1)
    quicksort_optimized(nums, pivot_index + 1, right)
```

Interview Follow-Up

Let's say the interviewer introduces the following constraints to the initial sorting problem:

- The input array does not contain negative values.
- All values in the input array are less than or equal to 10^3.

Does our approach to the problem change, and should we still use quicksort? Considering that all values in our array now fall within the limited range of $[0, 10^3]$, a counting sort approach becomes appropriate.

Counting sort

Counting sort is a non-comparison-based sorting algorithm that works by counting the number of occurrences of each element in the array, then using these counts to place each element in its correct sorted position:

```
nums = [ 2  1  0  0  2  4  2 ]

counts = [ 2  1  3  0  1 ]
           0  1  2  3  4

res = [ 0  0  1  2  2  2  4 ]
```

We can do this in two steps:

- Count occurrences: create a `counts` array, where each of its indexes represents an element from the original array. Increment the value at each index based on how many times the corresponding element appears in the original array.
- Build sorted array (`res`): iterate through each index of the `counts` array and add that index (`i`) to the sorted array as many times as its value (`counts[i]`) indicates.

Counting sort is efficient here because we know the largest possible number in the array is at most 10^3, which means our `counts` array will have a maximum size of $10^3 + 1$. However, if this problem constraint is not specified and the maximum value in the array may be very large, then a counting sort solution might not be appropriate, due to the potentially large size of the `counts` array.

Implementation

Note that there's another common method for implementing counting sort, which is detailed in the reference provided [1].

```python
def sort_array_counting_sort(nums: List[int]) -> List[int]:
    if not nums:
        return []
    res = []
    # Count occurrences of each element in 'nums'.
    counts = [0] * (max(nums) + 1)
    for num in nums:
        counts[num] += 1
    # Build the sorted array by appending each index 'i' to it a total
    # of 'counts[i]' times.
    for i, count in enumerate(counts):
        res.extend([i] * count)
    return res
```

Complexity Analysis

Time complexity: The time complexity of `sort_array_counting_sort` is $O(n+k)$, where k denotes the maximum value of `nums`. This is because it takes $O(n)$ time to count the occurrences of each element and $O(k)$ time to build the sorted array.

Space complexity: The space complexity is $O(n + k)$, since the `res` array occupies $O(n)$ space, and the `counts` array takes up $O(k)$ space. Note that `res` is considered in the space complexity, as counting sort is not an in-place sorting algorithm requiring an additional array to store the sorted result.

Interview Tip

Tip: Quicksort is useful for in-place sorting.

While quicksort isn't a stable sorting algorithm, it is an in-place sorting algorithm, meaning it requires less additional space compared to an algorithm such as merge sort, which takes $O(n)$ space when used to sort an array.

References

[1] Counting sort: https://en.wikipedia.org/wiki/Counting_sort

[2] What is the advantage of Randomized Quicksort?: https://cs.stackexchange.com/questions/7582/what-is-the-advantage-of-randomized-quicksort

K^{th} Largest Integer

> Return the k^{th} largest integer in an array.
>
> **Example:**
>
> Input: `nums = [5, 2, 4, 3, 1, 6], k = 3`
> Output: 4
>
> **Constraints:**
> - The array contains no duplicates.
> - The array contains at least one element.
> - $1 \le k \le n$, where n denotes the length of the array.

Intuition – Min-Heap

A straightforward solution to this problem is to sort the array in reverse order and return the number at the $(k - 1)^{th}$ index:

```
                                      kth largest
k = 3                                    ↓
[ 5  2  4  3  1  6 ] --sort--> [ 6  5  4  3  2  1 ]
  0  1  2  3  4  5               0  1  2  3  4  5
```

This solution takes $O(n \log(n))$ time, but as we only need the k^{th} largest integer, sorting all the elements in the input array might not be necessary. Let's explore some solutions which take this into account.

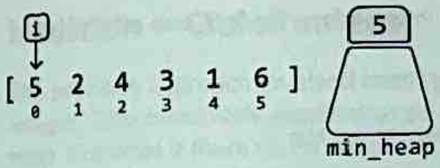
An interesting thing to realize is that if we know what the top k largest integers are, we'd know that the smallest of these would be the k^{th} largest integer:

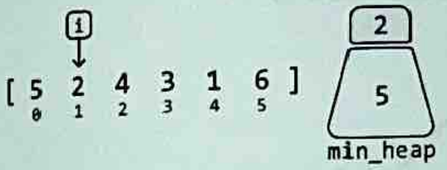
```
                  [ 5  2  4  3  1  6 ]

top k largest: [ 5  4  6 ]
                    ↓
                   min
```

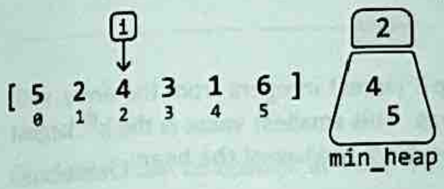
This leads to the idea that instead of sorting the entire array, we can keep track of the top k largest integers in the array. Is there a way to maintain the top k largest integers, while having access to the smallest of these integers?

A min-heap seems like it should work well for this purpose because it provides efficient access to the smallest element. Let's explore how to use it to maintain the top k integers in the array.

Consider the below example with k = 3. Let's begin by pushing the first k integers to the heap:



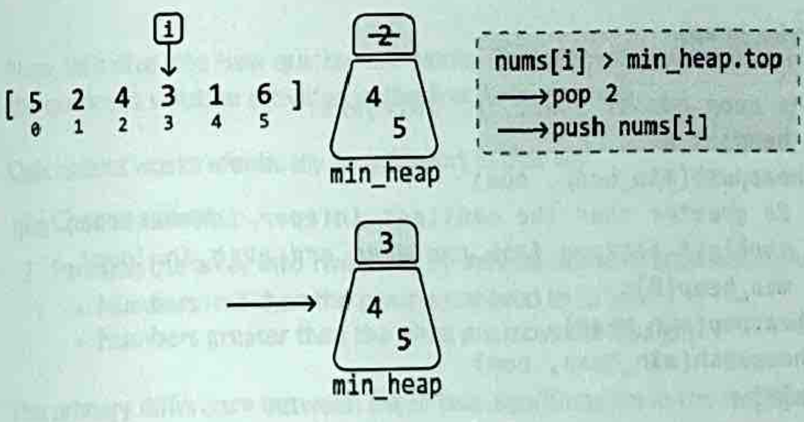




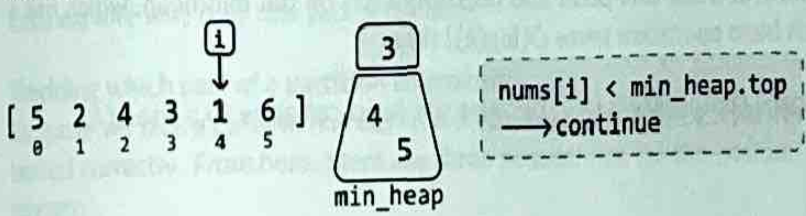
We shouldn't immediately push the next element, 3, in the heap because the heap already contains k integers. This gives us a choice:

- Skip 3, or:
- Remove an integer from the heap and push 3 in.

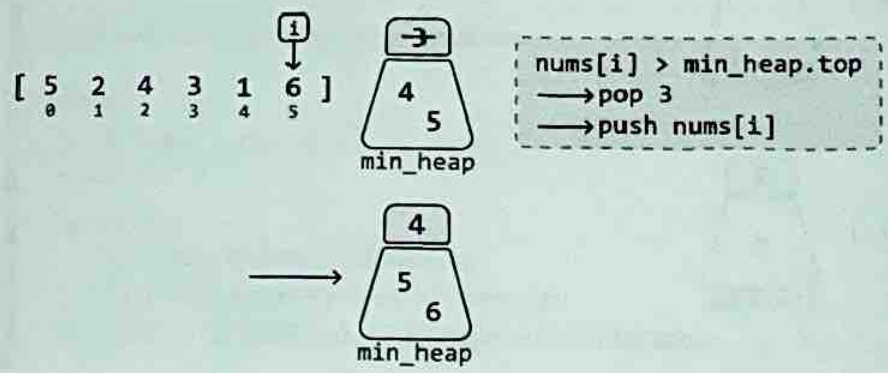
The smallest integer in the heap is 2. Integer 3 is more likely to belong in the top k integers than 2 is, so let's pop 2 from the heap and push 3 in:



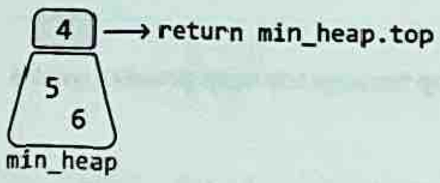
The next integer is 1, which is smaller than the smallest integer in the heap. So, we know it's not among the top k integers:



The final integer is 6. It's larger than the smallest integer in the heap. So, let's pop off the top of the heap and push in 6:



At this point, the k integers in the heap represent the top k largest integers from the array, with the smallest of these integers located at the top of the heap. This smallest value is the kth largest integer in the entire array, so we can just return this integer from the top of the heap:



Implementation - Min-Heap

```
def kth_largest_integer_min_heap(nums: List[int], k: int) -> int:
    min_heap = []
    heapq.heapify(min_heap)
    for num in nums:
        # Ensure the heap has at least 'k' integers.
        if len(min_heap) < k:
            heapq.heappush(min_heap, num)
        # If 'num' is greater than the smallest integer in the heap, pop
        # off this smallest integer from the heap and push in 'num'.
        elif num > min_heap[0]:
            heapq.heappop(min_heap)
            heapq.heappush(min_heap, num)
    return min_heap[0]
```

Complexity Analysis

Time complexity: The time complexity of `kth_largest_integer_min_heap` is $O(n \log(k))$ because for each integer, we perform at most one push and pop operation on the min-heap, which has a size no larger than k. Each heap operation takes $O(\log(k))$ time.

Space complexity: The space complexity is $O(k)$ because the heap can grow to a size of k.

Intuition – Quickselect

The previous approach involved keeping track of the k largest integers to identify the k^{th} largest integer. This effectively required us to keep these k values partially sorted through the use of a heap. But what if there's a way to position only the k^{th} largest value in its expected sorted position, without sorting the other numbers?

```
                                 sort the whole array:
                                 [ 1  2  3  4  5  6 ]   O(nlog(n))
                                   0  1  2  3  4  5

                                 sort the k largest integers:
[ 5  2  4  3  1  6 ]  ---------> [ •  •  •  4  5  6 ]   O(nlog(k))
  0  1  2  3  4  5                 0  1  2  3  4  5

                                 only position the kth largest
                                 integer in its sorted index:
                                 [ •  •  •  4  •  • ]   (faster?)
                                   0  1  2  3  4  5
```

Quickselect can be used to achieve this.

Quickselect is an algorithm that **leverages the partition step of quicksort**, which positions a value in its sorted position. Quickselect is generally used to find the k^{th} *smallest* element, whereas, in this problem, we're asked to find the k^{th} *largest* element. To utilize quickselect for this purpose, we can instead **find the $(n - k)^{th}$ smallest integer**, which is equivalent to finding the k^{th} largest element:

```
    kth largest == (n - k)th smallest

    [ 5  2  4  3  1  6 ]  quickselect(n - k)  [ 2  1  3  4  5  6 ]
      0  1  2  3  4  5   ------------------>    0  1  2  3  4  5
                                                        ↑
                                                  index n - k
```

Now, let's dive into how quickselect works. To understand quickselect, we recommend you study the quicksort solution provided in the *Sort Array* problem.

Quickselect works identically to quicksort in that we:

1. Choose a pivot.
2. Partition the array into two parts by moving elements around the pivot so that:
   - Numbers less than the pivot are moved to its left.
   - Numbers greater than the pivot are moved to its right.

The primary difference between these two algorithms lies in the recursion step:

- In quicksort, we recursively process both the left and right parts.
- In quickselect, **we only need to recursively process one of these parts.**

Let's explore why only one part is chosen.

Deciding which part of a partition to process

Suppose we pick a random number as a pivot. After performing a partition, this pivot will be positioned correctly. From here, there are three possibilities for the position of the $(n - k)^{th}$ smallest integer:

- If the pivot is positioned before index `n - k`, it means the `(n - k)`th smallest integer must be somewhere to the right of the pivot. So, perform quickselect on the right part:

```
if pivot_index < n - k:
   pivot
[ · [·]  ·  ·  ·  · ]   perform quickselect on the right part
               ↑
          index n - k
```

- If the pivot is positioned after index `n - k`, perform quickselect on the left part:

```
if pivot_index > n - k:
                    pivot
[ ·  ·  ·  ·  ·  [·] ]   perform quickselect on the left part
               ↑
          index n - k
```

- If the pivot is positioned exactly at index `n - k`, the pivot itself is the `(n - k)`th smallest integer. So, we just return the pivot:

```
if pivot_index == n - k:
               pivot
[ ·  ·  ·  ·  [·]  · ]   return nums[pivot_index]
               ↑
          index n - k
```

Implementation – Quickselect

Note that the partition function in this implementation is identical to the partition function in *Sort Array*. Also, this implementation selects a random pivot, as per the optimization discussed in *Sort Array*.

```python
def kth_largest_integer_quickselect(nums: List[int], k: int) -> int:
    return quickselect(nums, 0, len(nums) - 1, k)

def quickselect(nums: List[int], left: int, right: int, k: int) -> None:
    n = len(nums)
    if left >= right:
        return nums[left]
    random_index = random.randint(left, right)
    nums[random_index], nums[right] = nums[right], nums[random_index]
    pivot_index = partition(nums, left, right)
    # If the pivot comes before 'n - k', the ('n - k')th smallest
    # integer is somewhere to its right. Perform quickselect on the
    # right part.
    if pivot_index < n - k:
        return quickselect(nums, pivot_index + 1, right, k)
    # If the pivot comes after 'n - k', the ('n - k')th smallest integer
    # is somewhere to its left. Perform quickselect on the left part.
    elif pivot_index > n - k:
        return quickselect(nums, left, pivot_index - 1, k)
    # If the pivot is at index 'n - k', it's the ('n - k')th smallest
    # integer.
```

```
    else:
        return nums[pivot_index]

def partition(nums: List[int], left: int, right: int) -> int:
    pivot = nums[right]
    lo = left
    for i in range(left, right):
        if nums[i] < pivot:
            nums[lo], nums[i] = nums[i], nums[lo]
            lo += 1
    nums[lo], nums[right] = nums[right], nums[lo]
    return lo
```

Complexity Analysis

Time complexity: The time complexity of `kth_largest_integer_quickselect` can be analyzed in terms of the average and worst cases:

- Average case: $O(n)$. In the average case, quickselect partitions the array and reduces the problem size by approximately half each time by performing a recursive call on only one part of each partition. A linear partition is performed during each of these recursive calls. This results in a total time complexity of $O(n) + O(\frac{n}{2}) + O(\frac{n}{4}) + \cdots$ until the base case of quickselect is reached. This can be simplified to an $O(n)$ time complexity.
- Worst case: $O(n^2)$. The worst-case scenario occurs when the pivot selection consistently results in extremely unbalanced partitions. This can result in the problem size only being reduced by one element after each partition, leading to a total time complexity of $O(n) + O(n-1) + O(n-2) + \cdots$, which can be simplified to an $O(n^2)$ time complexity.

Space complexity: The space complexity can also be analyzed in terms of the average and worst cases:

- Average Case: $O(\log(n))$. In the average case, the depth of the recursive call stack is approximately $\log_2(n)$.
- Worst Case: $O(n)$. In the worst case, the depth of the recursive call stack can be as deep as n.