# Chapter 14
## *Sorting Algorithms*

### 14.1 Types of Sorting Algorithms

In this chapter, we will explore several different algorithms that can be used to sort a sequence of elements in some predefined order. To start off, we will first look at **elementary sorting algorithms**. Elementary sorting algorithms tend to be simple and straightforward, and they can be used to illustrate different approaches that can be used to sort a collection of objects. From a complexity standpoint, elementary sorts are often not as good as their high-performance counterparts. However, despite their simplicity, they can be good enough if the data set you want to sort is small. Elementary sorts also serve as a building block for advanced sorting algorithms that may have better performance.

Before we begin, let's introduce some terminology. An **inversion** is defined as a pair of numbers in a sequence not in sorted order. That is, if you are given an array `arr` and two indices `i` and `j`, the pair `(arr[i], arr[j])` is an inversion of the array `arr` if `i < j` and `arr[i] > arr[j]` (assuming the array should be sorted in ascending order). For instance, the elements 4 and 5 form an inversion in the array below.

| 0 | 1 | 2 | 3 | **5** | **4** | 6 | 7 |
|---|---|---|---|---|---|---|---|

A sorting algorithm is considered to be **stable** if the relative order of identical elements is not modified after the completion of the sort. If two elements A and B are identical, and A comes before B before sorting, then A should still be before B after the sequence is sorted using a stable sorting algorithm. In the following example, `{14, "Banana"}` was before `{14, "Apple"}` in the initial container, so if the container were only sorted by number using a stable sort, `{14, "Banana"}` would still be before `{14, "Apple"}` upon the sort's completion.

| {14, "Banana"} | {12, "Carrot"} | {14, "Apple"} | {15, "Eggplant"} | {13, "Durian"} |
|---|---|---|---|---|

↓

| {**12**, "Carrot"} | {**13**, "Durian"} | {**14**, "Banana"} | {**14**, "Apple"} | {**15**, "Eggplant"} |
|---|---|---|---|---|

A sorting algorithm is considered to be **adaptive** if the sequence of operations performed by the algorithm differs based on the order of the data that needs to be sorted. An adaptive sort often takes advantage of the existing order of elements in its input: for instance, an adaptive sort may run more efficiently if it is given an array that is nearly sorted. Because of this, adaptive sorts often have different best-case and worst-case time complexities, since their performance can depend on the ordering of data that is passed in.

On the other hand, a **non-adaptive** sort always completes the same sequence of operations on the data it needs to sort, regardless of how the data is ordered. A non-adaptive sorting algorithm would run the same operations on an array that is nearly sorted versus an array that is not close to being sorted at all. Non-adaptive sorts are typically simpler to implement as they do not have to check the order of elements before sorting.

What dictates the efficiency of a sorting algorithm? Like with most other algorithms, the efficiency of a sorting algorithm can be measured using its asymptotic time complexity. This can be done by analyzing the number of comparisons and swaps that are completed by the algorithm. A sorting algorithm's efficiency can also be measured using its auxiliary space usage — some sorts can be done in-place, while others may require additional memory usage that depends on the input size (either through an additional container or stack frames via recursive calls).

In the next few sections, we will discuss different sorting algorithms that fall into different categories based on their implementation and behavior. As mentioned before, we will first cover elementary sorts (bubble sort, selection sort, insertion sort) before moving on to more advanced comparison sorts (heapsort, quicksort, mergesort). Lastly, we will discuss linear-time sorting algorithms that are asymptotically faster than comparison sorts, but require special assumptions about the data that needs to be sorted.

## 14.2 Bubble Sort

**Bubble sort** is an elementary sorting algorithm that sorts elements by repeatedly comparing two adjacent elements at a time and swapping them if they are in the wrong order. The following implements a standard bubble sort algorithm for a vector of integers:

```
1  void bubble_sort(std::vector<int32_t>& vec) {
2    for (size_t i = 0; i < vec.size() - 1; ++i) {
3      for (size_t j = 0; j < vec.size() - i - 1; ++j) {
4        // if out of order, swap the elements
5        if (vec[j] > vec[j + 1]) {
6          std::swap(vec[j], vec[j + 1]);
7        } // if
8      } // for j
9    } // for i
10 } // bubble_sort()
```

Let's look at how bubble sort works using the following unsorted array: `[5, 3, 9, 1, 7, 8, 2, 4, 6]`.

The bubble sort algorithm looks at two adjacent elements at a time, swapping them if they are out of order. At the end of each pass, the largest element "bubbles" to the end of the vector, as shown below:

**First pass:**

| | |
|---|---|
| [5, 3, 9, 1, 7, 8, 2, 4, 6] | 5 and 3 are out of order, so swap! |
| [3, 5, 9, 1, 7, 8, 2, 4, 6] | 5 and 9 are in the correct order, so no swap needed. |
| [3, 5, 9, 1, 7, 8, 2, 4, 6] | 9 and 1 are out of order, so swap! |
| [3, 5, 1, 9, 7, 8, 2, 4, 6] | 9 and 7 are out of order, so swap! |
| [3, 5, 1, 7, 9, 8, 2, 4, 6] | 9 and 8 are out of order, so swap! |
| [3, 5, 1, 7, 8, 9, 2, 4, 6] | 9 and 2 are out of order, so swap! |
| [3, 5, 1, 7, 8, 2, 9, 4, 6] | 9 and 4 are out of order, so swap! |
| [3, 5, 1, 7, 8, 2, 4, 9, 6] | 9 and 6 are out of order, so swap! |
| [3, 5, 1, 7, 8, 2, 4, 6, **9**] | First pass complete! 9 is fixed in position. |

Because 9 is the largest element in the array, it ends up bubbling to the end. After the first pass, we will keep 9 fixed (since we know that 9 is now in the correct position) and complete another pass of bubble sort. During this second iteration, we will bubble the second largest element up to its correct position.

**Second pass:**

| | |
|---|---|
| [3, 5, 1, 7, 8, 2, 4, 6, **9**] | 3 and 5 are in the correct order, so no swap needed. |
| [3, 5, 1, 7, 8, 2, 4, 6, **9**] | 5 and 1 are out of order, so swap! |
| [3, 1, 5, 7, 8, 2, 4, 6, **9**] | 5 and 7 are in the correct order, so no swap needed. |
| [3, 1, 5, 7, 8, 2, 4, 6, **9**] | 7 and 8 are in the correct order, so no swap needed. |
| [3, 1, 5, 7, 8, 2, 4, 6, **9**] | 8 and 2 are out of order, so swap! |
| [3, 1, 5, 7, 2, 8, 4, 6, **9**] | 8 and 4 are out of order, so swap! |
| [3, 1, 5, 7, 2, 4, 8, 6, **9**] | 8 and 6 are out of order, so swap! |
| [3, 1, 5, 7, 2, 4, 6, **8**, **9**] | Second pass complete! 8 is fixed in position. |

We can repeat this process by conducting multiple passes until the array is fully sorted. What is the time complexity of bubble sort? If we define the size of the array as $n$, then we complete $n-1$ comparisons on the first pass, $n-2$ comparisons on the second pass, $n-3$ comparisons on the third pass, and so on. The total number of comparisons we complete is equal to the sum $(n-1)+(n-2)+\ldots+1 = n(n-1)/2 = \Theta(n^2)$. Thus, our current bubble sort implementation runs in $\Theta(n^2)$ time.

**Remark:** This specific approach is not the only way to complete a bubble sort: you can also iterate from the back to the front, instead of from the front to the back. In this alternative method, the *smallest* values get bubbled to the *front* of the array.

What if the array were already sorted? The bubble sort algorithm we currently have would do $\Theta(n^2)$ work regardless of the contents of the array. However, we can prevent the algorithm from doing extra work by adding an additional check in the inner loop: if no swaps were made during a pass, the array must be fully sorted! With this check, we ensure that our bubble sort algorithm immediately terminates once the array is sorted.

```
1  // adaptive bubble sort
2  void bubble_sort(std::vector<int32_t>& vec) {
3    for (size_t i = 0; i < vec.size() - 1; ++i) {
4      bool swap_made = false;
5      for (size_t j = 0; j < vec.size() - i - 1; ++j) {
6        if (vec[j] > vec[j + 1]) {
7          std::swap(vec[j], vec[j + 1]);
8          swap_made = true;
9        } // if
10     } // for j
11     // if swapped is still false, no swaps were made during pass
12     if (swap_made = false) {
13       break;
14     } // if
15   } // for i
16 } // bubble_sort()
```

Let's see what happens when we pass the sorted array `[1, 2, 3, 4, 5]` into the adaptive bubble sort:

| | |
|---|---|
| [1, 2, 3, 4, 5] | 1 and 2 are in the correct order, so no swap needed. |
| [1, 2, 3, 4, 5] | 2 and 3 are in the correct order, so no swap needed. |
| [1, 2, 3, 4, 5] | 3 and 4 are in the correct order, so no swap needed. |
| [1, 2, 3, 4, 5] | 4 and 5 are in the correct order, so no swap needed. |

Since we completed an entire pass of the array without making a single swap, the `swap_made` variable is never set to `true`. Thus, we know that the array must be fully sorted, so we can terminate the algorithm without doing any more passes. With this implementation of bubble sort, the best-case time complexity becomes $\Theta(n)$, which occurs when the array is fully sorted.

In addition to being adaptive, bubble sort is also stable. By ensuring that equality does not lead to a swap, we can guarantee that duplicate elements end up in the same relative order after the sorting process is done. Furthermore, bubble sort can be done in-place and thus only uses $\Theta(1)$ auxiliary space. A summary of bubble sort is shown in the table below:

| Best Time | Average Time | Worst Time | Auxiliary Space | Stable? | Adaptive? |
|---|---|---|---|---|---|
| $\Theta(n)$ | $\Theta(n^2)$ | $\Theta(n^2)$ | $\Theta(1)$ | Yes | Yes |

## 14.3 Selection Sort

**Selection sort** is an elementary sorting algorithm that sorts a container by repeatedly selecting the smallest element that has not been sorted and placing it in its correct sorted position. The code is shown below:

```
1  void selection_sort(std::vector<int32_t>& vec) {
2    for (size_t i = 0; i < vec.size() - 1; ++i) {
3      // find the smallest element that is not in correct sorted position
4      size_t min_index = i;
5      for (size_t j = i + 1; j < vec.size(); ++j) {
6        if (vec[j] < vec[min_index]) {
7          min_index = j;
8        } // if
9      } // for j
10     // swap the element at min_index to its correct position
11     std::swap(vec[i], vec[min_index]);
12   } // for i
13 } // selection_sort()
```

Let's look at selection sort in action, using the same array we used earlier: `[5, 3, 9, 1, 7, 8, 2, 4, 6]`. With each pass, we find the next smallest value and place it in its correct position. In the process below, the two underlined elements are swapped, and bolded elements are fixed in their correct sorted position.

| | |
|---|---|
| [5, 3, 9, 1, 7, 8, 2, 4, 6] | 1 is the smallest element, so swap it to index 0. |
| [**1**, 3, 9, 5, 7, 8, 2, 4, 6] | 2 is the next smallest element, so swap it to index 1. |
| [**1**, **2**, 9, 5, 7, 8, 3, 4, 6] | 3 is the next smallest element, so swap it to index 2. |
| [**1**, **2**, **3**, 5, 7, 8, 9, 4, 6] | 4 is the next smallest element, so swap it to index 3. |
| [**1**, **2**, **3**, **4**, 7, 8, 9, 5, 6] | 5 is the next smallest element, so swap it to index 4. |
| [**1**, **2**, **3**, **4**, **5**, **6**, 9, 7, 8] | 6 is the next smallest element, so swap it to index 5. |
| [**1**, **2**, **3**, **4**, **5**, **6**, **7**, 9, 8] | 7 is the next smallest element, so swap it to index 6. |
| [**1**, **2**, **3**, **4**, **5**, **6**, **7**, **8**, 9] | 8 is the next smallest element, so swap it to index 7. |
| [**1**, **2**, **3**, **4**, **5**, **6**, **7**, **8**, **9**] | 9 is the last element, so we are done! |

Similar to bubble sort, we need to make $n-1$ comparisons on the first pass, $n-2$ comparisons on the second pass, and so on. This results in a $\Theta(n^2)$ time algorithm. Can we make this better?

It turns out that we cannot do much to improve the complexity of selection sort. We can add a check to prevent our algorithm from swapping an element with itself, but this does not prevent us from having to do $\Theta(n^2)$ comparisons. Because the execution of selection sort is independent of the input, selection sort is not adaptive.

Selection sort is also not stable. This is because you cannot guarantee where an element will end up after it is swapped. Consider the following unsorted array with two 4's: `[4, 3, 4', 2, 1]`, where 4' is the second 4 in the array. The first two passes of selection sort are:

| | |
|---|---|
| `[4, 3, 4', 2, 1]` | 1 is the smallest element, so swap it to index 0 |
| `[1, 3, 4', 2, 4]` | 2 is the next smallest element, so swap it to index 1 |
| ... | ... |
| `[1, 2, 3, 4', 4]` | After sorting, 4' is now before 4, so the sort is not stable! |

Notice that we have a problem when we swapped 1 to its correct position: after swapping 1, we ended up swapping 4 to a position *after* 4'. Thus, 4' will end up before 4 in the sorted array, even though 4 was positioned before 4' prior to sorting. Because you cannot predict where duplicate elements will end up (the first 4 could have been sent before *or* after 4' depending on where 1 was), selection sort is not stable. (Note that it is technically possible to make selection sort stable by modifying its behavior or by using additional memory, but for the purposes of the class, the concept of a selection sort will typically refer to the standard, unstable implementation, unless otherwise specified.)

So, not only does selection sort always run in $\Theta(n^2)$ time, it is also neither stable nor adaptive. Based on this, it might appear that selection sort is inferior to bubble sort in every way! However, selection sort does have its redeeming qualities. The most notable advantage of selection sort is that it minimizes the total number of swaps needed during the sorting process. Unlike bubble sort, which could require a swap for every pair of elements in an array, selection sort only requires $n-1$ swaps in the worst-case (given an input size of $n$). Do not confuse this with the runtime of selection sort, however; even though selection sort could perform zero swaps, it still needs to complete $\Theta(n^2)$ comparisons, which is why the best-case time complexity of selection sort is still $\Theta(n^2)$. A summary of selection sort is shown in the table below:

| Best Time | Average Time | Worst Time | Auxiliary Space | Stable? | Adaptive? |
|---|---|---|---|---|---|
| $\Theta(n^2)$ | $\Theta(n^2)$ | $\Theta(n^2)$ | $\Theta(1)$ | No | No |

## 14.4 Insertion Sort

**Insertion sort** is an elementary sorting algorithm that looks at each element sequentially and builds the sorted result one at a time. The implementation of insertion sort is shown below:

```
1   void insertion_sort(std::vector<int32_t>& vec) {
2     for (size_t i = 1; i < vec.size(); ++i) {
3       int32_t current = vec[i];
4       int32_t j = i - 1;
5       while (j >= 0 && vec[j] > current) {
6         vec[j + 1] = vec[j];
7         --j;
8       } // while()
9       vec[j + 1] = current;
10    } // for i
11  } // insertion_sort()
```

As we iterate through the array, we look at each element and compare it with the elements that come before it. If the element before the current element is out of position compared to the current element (i.e., there is an inversion), we shift that element one to the right. This allows us to identify where the current element should be placed relative to the elements that come before it.

Let's analyze insertion sort using the same unsorted array we used earlier: `[5, 3, 9, 1, 7, 8, 2, 4, 6]`. We will start from the left end of the array and consider each element one at a time.

| | |
|---|---|
| `[5, 3, 9, 1, 7, 8, 2, 4, 6]` | 5 is in the correct position relative to the elements before it (trivially). |
| `[5, 3, 9, 1, 7, 8, 2, 4, 6]` | Looking only at the elements before 3, where should 3 go? Before 5. |

At this step, we shift 5 one index to the right and insert 3 into the correct relative position, before 5.

| | |
|---|---|
| `[3, 5, 9, 1, 7, 8, 2, 4, 6]` | Looking only at the elements before 9, where should 9 go? |

9 is in the correct position relative to 3 and 5, so no shifting is needed.

| | |
|---|---|
| `[3, 5, 9, 1, 7, 8, 2, 4, 6]` | Looking only at the elements before 1, where should 1 go? Before 3. |

Compared to the elements that have been visited before it, 1 should go before 3. Thus, we would shift 9 one position to the right (to index 3), 5 one position to the right (to index 2), 3 one position to the right (to index 1), and insert 1 in its correct relative position at index 0.

| | |
|---|---|
| `[1, 3, 5, 9, 7, 8, 2, 4, 6]` | Looking only at the elements before 7, where should 7 go? Before 9. |

Thus, we would shift 9 one position to the right (to index 4) and insert 7 in its correct relative position at index 3.

| | |
|---|---|
| `[1, 3, 5, 7, 9, 8, 2, 4, 6]` | Looking only at the elements before 8, where should 8 go? Before 9. |

Thus, we would shift 9 one position to the right (to index 5) and insert 8 in its correct relative position at index 4.

`[1, 3, 5, 7, 8, 9, 2, 4, 6]` Looking only at the elements before 2, where should 2 go? Before 3.

Thus, we would shift 9 one position to the right (to index 6), 8 one position to the right (to index 5), 7 one position to the right (to index 4), 5 one position to the right (to index 3), 3 one position to the right (to index 2), and insert 2 in its correct relative position at index 1.

`[1, 2, 3, 5, 7, 8, 9, 4, 6]` Looking only at the elements before 4, where should 4 go? Before 5.

Thus, we would shift 9 one position to the right (to index 7), 8 one position to the right (to index 6), 7 one position to the right (to index 5), 5 one position to the right (to index 4), and insert 4 in its correct relative position at index 3.

`[1, 2, 3, 4, 5, 7, 8, 9, 6]` Looking only at the elements before 6, where should 6 go? Before 7.

Thus, we would shift 9 one position to the right (to index 8), 8 one position to the right (to index 7), 7 one position to the right (to index 6), and insert 6 in its correct relative position at index 5.

`[1, 2, 3, 4, 5, 6, 7, 8, 9]` The array is now fully sorted.

There is one additional optimization that we can complete to make our insertion sort even more efficient. Notice that we have the following `while` loop in our current insertion sort implementation:

```
while (j >= 0 && vec[j] > current) { ... }
```

Here, we check the expressions `j >= 0` and `vec[j] > current` a total of $\Theta(n^2)$ times. However, the `j >= 0` check seldom returns `false`! This comparison only returns `false` if the element we are considering is the smallest element we have encountered so far (which would require the element to be placed all the way to index 0). However, as long as we are not looking at the smallest element, the `vec[j] > current` check is enough to ensure that we don't index out of bounds (as `j` would never reach 0). Thus, we can save some time by moving the smallest element to the very beginning of the array before we begin our insertion sort (this element is known as a *sentinel* value). By doing this, we would only need one check in our `while` loop:

```
while (vec[j] > current) { ... }
```

The behavior of insertion sort naturally allows it to be adaptive. Since each element is only compared to the values that come before it, insertion sort would only need to do $n$ comparisons if the array were already sorted. This is because each value in a sorted array is guaranteed to be larger than the elements before it, and only a single comparison is needed per element to realize that it is already in its correct relative position. Furthermore, insertion sort is also stable, since elements are always considered in order from front to back. By shifting only when the current element is strictly lesser than the element it is being compared to, duplicates will always maintain their original relative order.

Despite being an elementary sort with an average-case $\Theta(n^2)$ time complexity, insertion sort can be fast in certain situations. In fact, the GCC implementation of `std::sort()` uses insertion sort when the number of elements to sort is fewer than 16 — this is because, for small array sizes, insertion sort is actually faster than many of the high-performance sorts that we will cover in the next few sections. A summary of insertion sort is provided in the table below:

| Best Time | Average Time | Worst Time | Auxiliary Space | Stable? | Adaptive? |
|---|---|---|---|---|---|
| $\Theta(n)$ | $\Theta(n^2)$ | $\Theta(n^2)$ | $\Theta(1)$ | Yes | Yes |

The improved version of our insertion sort algorithm is shown below:

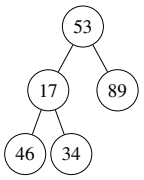
```
1  void insertion_sort(std::vector<int32_t>& vec) {
2    // find minimum element and place at index 0 (sentinel)
3    int32_t min_index = 0;
4    for (size_t i = 1; i < vec.size(); ++i) {
5      if (vec[i] < vec[min_index]) {
6        min_index = i;
7      } // if
8    } // for i
9    std::swap(vec[0], vec[min_index]);
10   // run insertion sort on remaining elements
11   for (size_t i = 2; i < vec.size(); ++i) {
12     int32_t current = vec[i];
13     int32_t j = i - 1;
14     while (vec[j] > current) {
15       vec[j + 1] = vec[j];
16       --j;
17     } // while
18     vec[j + 1] = current;
19   } // for i
20 } // insertion_sort()
```

## 14.5 Heapsort

So far, we have covered elementary sorts that tend to be simpler but have less efficient time complexities. Starting in this section, we will cover several high-performance sorts. These sorts tend to be more complex, but they are also more efficient for large input sizes. The first of these sorts is **heapsort**, which relies on the heap structure to sort elements.

To sort an array of elements using heapsort, the contents of the array are first heapified into a binary max-heap. By doing this, we ensure that the largest element ends up at the top of the heap. After this largest element is identified, we swap it with the last element in the array — this places the largest element at the end, in its correct sorted position. Then, we call fix down on the element that was swapped to the top of the heap. This process is repeated several times until the array becomes fully sorted.

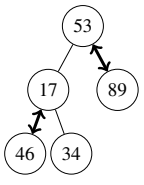
To illustrate the heapsort process, let's consider the following unsorted array: `[53, 17, 89, 46, 34]`.




Array Representation

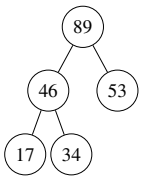
| 53 | 17 | 89 | 46 | 34 |
|---|---|---|---|---|

First, we will heapify this array into a binary max-heap so that the largest element is at the top of the heap. The bottom-up heapify approach is used since it can be done in worst-case $\Theta(n)$ time. During this process, we first fix down on 17 by swapping it with 46, and then we fix down on 53 by swapping it with 89. After heapify is complete, we have the following valid binary max-heap:




Array Representation

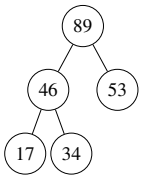
| 53 | 17 | 89 | 46 | 34 |
|---|---|---|---|---|




Array Representation

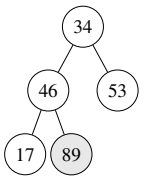
| 89 | 46 | 53 | 17 | 34 |
|---|---|---|---|---|

Because the heap is now a max-heap, we know that the element at the top (89) must be the largest element in the array, so 89's sorted position must be at the back of the array. Thus, we will move 89 to the very end of the array by swapping it with 34. Since we know 89 must now be in the correct position, we will ignore 89 for the remainder of the algorithm.




Array Representation

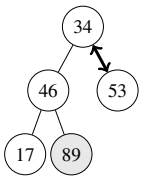
| 89 | 46 | 53 | 17 | 34 |
|---|---|---|---|---|




Array Representation

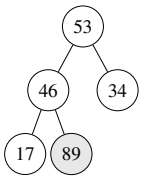
| 34 | 46 | 53 | 17 | 89 |
|---|---|---|---|---|

Since 34 was moved to the top of the heap, the heap is no longer a valid binary max-heap. However, because 34 is the only element that is out of place, we can fix it by calling fix down on 34. As a result of this fix down, 34 ends up swapping with 53.




Array Representation

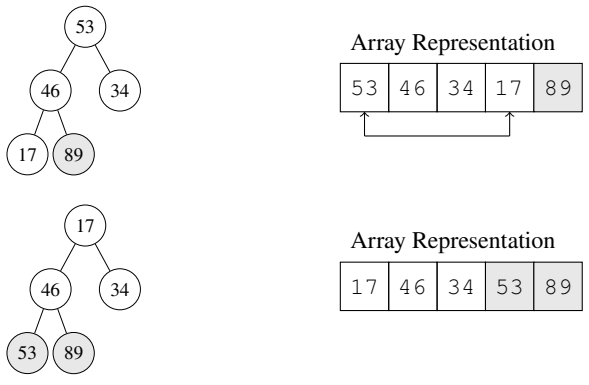
| 34 | 46 | 53 | 17 | 89 |
|---|---|---|---|---|



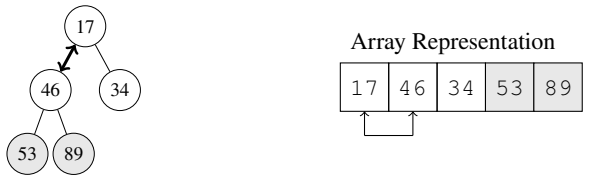

Array Representation

| 53 | 46 | 34 | 17 | 89 |
|---|---|---|---|---|

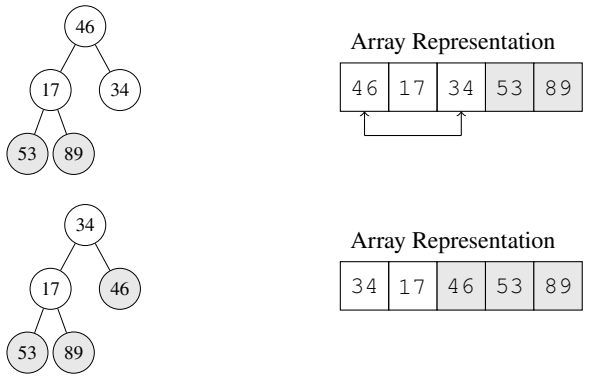
Now, 53 is at the top of the heap. Since we know that our heap is a valid max-heap (ignoring 89), 53 must be the second-largest element in the array. Thus, we can move 53 to its correct position by swapping it with the second-to-last element, 17.



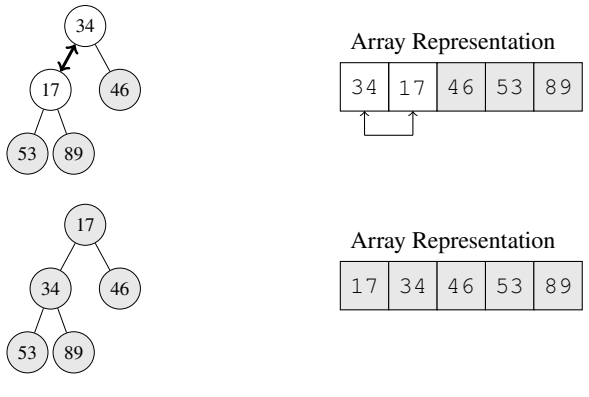

We will repeat this process until the array is fully sorted. Now, we will fix down on 17:




46 is now at the top of the heap, so it is the next largest element that has not yet been placed in the correct position. 46 is thus moved to the correct position by swapping it with 34:




Once again, we fix down on the top element, 34. Nothing ends up changing since 34 is already larger than 17. We now know that 34 is the next largest element, so it is put in place by swapping with 17. The array is now sorted!



The heapsort algorithm is outlined in the code below (using 1-indexing with a dummy element at index 0, for simplicity).

```
// assume this function calls fix down on vec[idx] using only
// a valid idx in the index range [1, numElts)
void fix_down(std::vector<int32_t>& vec, size_t idx, size_t numElts);

void heapsort(std::vector<int32_t>& vec) {
  // build heap using bottom-up heapify
  for (size_t i = vec.size() / 2; i > 0; --i) {
    fix_down(vec, i, vec.size());
  } // for i
  // loop through all elements from back to front, assumes dummy at idx 0
  for (size_t j = vec.size(); j > 0; --j) {
    // move largest element to the back
    std::swap(vec[1], vec[j]);
    // fix top element, ignoring elements that are already fixed (index > j - 1)
    fix_down(vec, 1, j - 1);
  } // for j
} // heapsort()
```

What is the time complexity of the heapsort algorithm? Heapsort consists of the following two steps:

1. Heapify the entire array at the very beginning (using bottom-up heapify).
2. Swap the element at the top of the heapified array to the back (ignoring elements that have already been fixed in position) and fix it in position. Then, call fix down on the element that was just swapped to the front. This is repeatedly done until the array is fully sorted.

The bottom-up heapify completed during the first step takes $\Theta(n)$ time. Then, for the second step, fix down is called $n$ times, once for each element. Because each fix down call takes $\Theta(\log(n))$ time, the overall time complexity of $n$ fix down calls is $n \times \Theta(n\log(n)) = \Theta(n\log(n))$. Since these two steps are completed consecutively, the overall time complexity of heapsort is $\Theta(n + n\log(n)) = \Theta(n\log(n))$.

Because there is no guarantee that duplicate elements will maintain their ordering during heap operations such as fix down, heapsort is not stable. For similar reasons, it is difficult for heapsort to be made adaptive. A summary of heapsort is provided in the table below:

| Best Time | Average Time | Worst Time | Auxiliary Space | Stable? | Adaptive? |
|---|---|---|---|---|---|
| $\Theta(n\log(n))$ | $\Theta(n\log(n))$ | $\Theta(n\log(n))$ | $\Theta(1)$ | No | No |

## 14.6 Quicksort

**Quicksort** is a divide-and-conquer algorithm that relies on a concept known as *partitioning* to sort an array. During partitioning, elements in the array are reorganized based on how large they are relative to a "pivot" element. The quicksort algorithm consists of the following two steps, which are repeated until the array is fully sorted:

1. A pivot element is selected (the method of selecting the pivot depends on the implementation, but a common strategy is to select the last element as the pivot — this is the method that will be used in these notes).
2. The remaining elements in the array are partitioned so that all elements to the left of the pivot are lesser than the pivot, and all elements to the right of the pivot are greater than the pivot.

An outline of the quicksort algorithm is shown below:

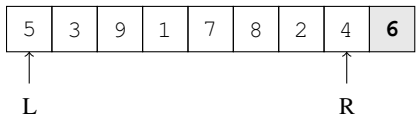
```
int32_t partition(std::vector<int32_t>& vec, int32_t left, int32_t right) {
  int32_t pivot = --right;
  while (true) {
    while (vec[left] < vec[pivot]) {
      ++left;
    } // while
    while (left < right && vec[right - 1] >= vec[pivot]) {
      --right;
    } // while
    if (left >= right) {
      break;
    } // if
    std::swap(vec[left], vec[right - 1]);
  } // while
  std::swap(vec[left], vec[pivot]);
  return left;
} // partition()

void quicksort(std::vector<int32_t>& vec, int32_t left, int32_t right) {
  if (left + 1 >= right) {
    return;
  } // if
  int32_t pivot = partition(vec, left, right);
  quicksort(vec, left, pivot);
  quicksort(vec, pivot + 1, right);
} // quicksort()
```

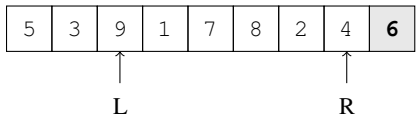
The `partition()` function takes in an index range `[left, right)` and selects the last element as the pivot. The remaining elements are partitioned so that elements to the left of the pivot are lesser, and elements to the right are greater. Let's look at what this code does using an example. Consider the following unsorted array:


First, we will choose an arbitrary pivot in this array. In our examples, we will select the last element as our pivot:

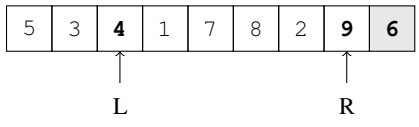

Now, we will need to partition the array so that all elements less than 6 end up to the left of 6, and all elements greater than 6 end up to the right of 6. To do so, we will keep track of two indices, `left` and `right`. The left index starts at the first element (5), while the right index starts at the last non-pivot element (4). We will refer to these two indices as L and R in the figures below.



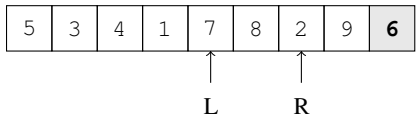

First, we increment L until the element it points to is greater than the pivot element, 6. In this case, L is incremented until it points to 9.



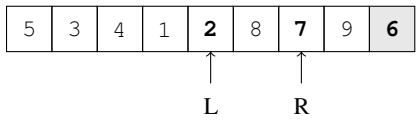

Next, we decrement R until the element it points to is less than the pivot, 6. Since the element that R is currently pointing to is already less than 6, we do not need to decrement R. Once L and R are pointing to elements that are out of place relative to the pivot, the two elements are swapped.



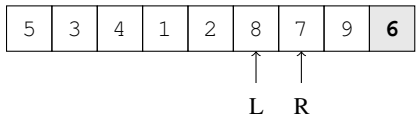

We repeat the above process until L and R meet. Increment L until it encounters a value larger than the pivot, and decrement R until it encounters a value smaller than the pivot. This happens when L is at 7 and R is at 2.



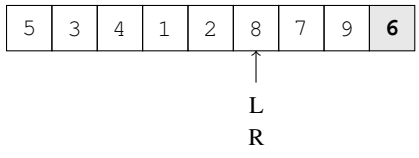

These two elements are then swapped.



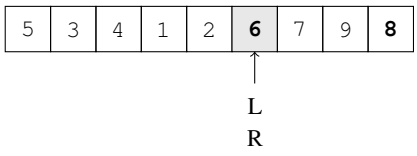

Increment L until it encounters an element greater than the pivot, 6. This happens when L is at 8:



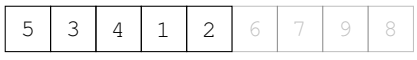

Decrement R until it encounters an element less than the pivot, 6. However, notice that the first decrement causes R to point to L.



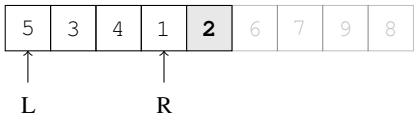
When L and R meet, swap the element at this position (in this case, 8) with the pivot value, 6.



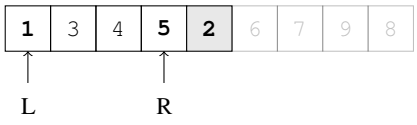

This ensures that the pivot value of 6 is now in its correct sorted position. Furthermore, all the elements to the left of 6 are smaller than 6, and all the elements to the right of 6 are larger. Since our initial array has been partitioned into two subarrays, one with elements less than 6, and one with elements greater than 6, we can recursively call quicksort on both the left and right subarrays to sort the entire array. To continue our example, let's make a recursive call and quicksort the elements to the left of 6. Note that this recursive call must run to completion before we can quicksort elements to the right of 6.




Like before, we will select the last element, 2, as our pivot, and initialize L and R:

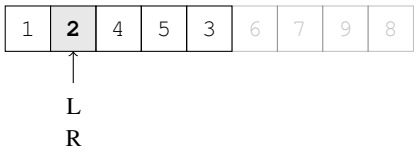



Increment L until it points to a value greater than 2, and decrement R until it points to a value less than 2 (in this case, neither L nor R needs to be moved). Then, swap the two elements.

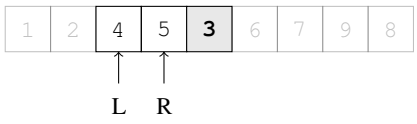



Increment L until it points to a value greater than 2, and decrement R until it points to a value less than 2. Both L and R end up at 3.

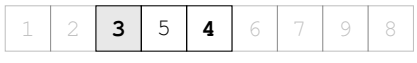

Since L and R are equal, we swap 3 with the pivot. The pivot element of 2 is now in its correct sorted position.




Once again, we will make recursive calls and quicksort elements to the left and right of 2. Since there is only one element to the left of 2, it must trivially be in the correct sorted position (we immediately exit in the base case). To recursively quicksort the elements to the right of 2, we select the last element (3) as the pivot.



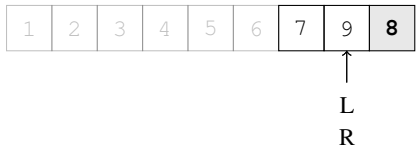

If we repeat the process of incrementing L until we reach an element greater than 3 and decrementing R until we reach an element less than 3, both L and R end up pointing to 4. We would then swap 4 with the pivot, 3:



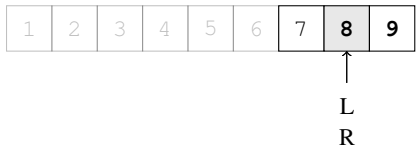

We would then recursively quicksort elements to the left and right of our pivot, 3. There are no elements to the left of 3 (since 3 is the leftmost element of the subarray we are considering), so no work is done here. When we recursively quicksort elements to the right of 3, the elements 4 and 5 end up getting swapped. After both recursive calls complete, the array looks like this:

At this point, we have finished quicksorting elements to the left of our original pivot, 6. Now that all elements to the left of 6 are sorted, we can make a recursive call to quicksort the elements to the right of 6. The process is similar to before: we select the last element, 8, as the pivot, and initialize L and R.


Increment L until it points to a value greater than the pivot value of 8. In this case, L ends up pointing to 9 since 9 is the first value greater than 8, but R is also pointing to 9.




Since L and R are equal, we swap 9 with the pivot.



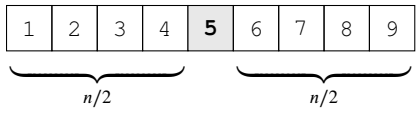

There is only one element to the left and right of 8, which are both trivially sorted. Thus, we have finished quicksorting the elements to the right of our initial pivot of 6. The entire quicksort algorithm is complete, and our array is fully sorted.


What is the runtime complexity of quicksort? Let's look at the code for quicksort again:

```
1  void quicksort(std::vector<int32_t>& vec, int32_t left, int32_t right) {
2    if (left + 1 >= right) {
3      return;
4    } // if
5    int32_t pivot = partition(vec, left, right);
6    quicksort(vec, left, pivot);
7    quicksort(vec, pivot + 1, right);
8  } // quicksort()
```

Quicksort is a recursive algorithm. On line 5, we call the `partition()` function, which ensures that the pivot value is placed in its correct sorted position, and that all elements to the left are smaller and all elements to the right are larger. This partitioning step takes $\Theta(n)$ time, since a single pass of the array is conducted when traversing using the left and right indices. Then, on lines 6 and 7, we recursively call quicksort on the elements to the left and right of the pivot.

What is the input size of these recursive calls? This actually depends on the value of the pivot! If the pivot were the median, the number of elements to the left and right of the pivot would be equal, and the input size would be $n/2$. For example, there are an equal number of elements to the left and right of the median value, 5:

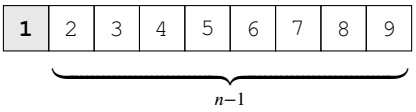



In this case, both recursive calls (on lines 6 and 7) would be called on an input size of $n/2$. If the median were chosen as the pivot every time, the recurrence relation for quicksort would be

$$T(n) = \begin{cases} 1, & \text{if } n = 0 \text{ or } 1 \\ 2T(n/2) + n, & \text{if } n > 1 \end{cases}$$

Here, the $2T(n/2)$ term comes from the two recursive quicksort calls, and the $n$ term comes from the partitioning step. Using Master's Theorem, we can see that the time complexity of quicksort is $\Theta(n\log(n))$ if the median were selected as the pivot. However, this is the best-case scenario, as it assumes that the median is chosen as the pivot every time! What would the worst-case time complexity of quicksort be?

In the worst case, either the smallest or largest element is chosen as the pivot at every step. When this happens, the pivot always leaves one side empty and the other side with all the remaining elements. For instance, if 1 were chosen as the pivot, every other element would be partitioned to the right of the pivot:

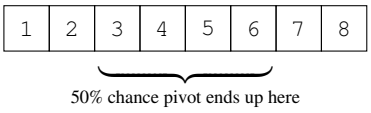



When this happens, one of the recursive calls would have an input size of 0, and the other would have an input size of $n-1$. If either the smallest or largest element is chosen as the pivot every time, the recurrence relation for quicksort becomes

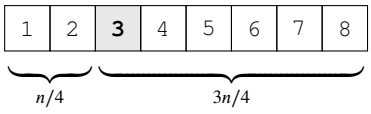
$$T(n) = \begin{cases} 1, & \text{if } n = 0 \text{ or } 1 \\ T(n-1)+n, & \text{if } n > 1 \end{cases}$$

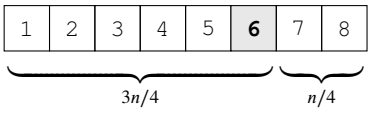
Here, the $T(n-1)$ comes from the recursive quicksort call with input size $n-1$, and the $n$ comes from the partitioning step. Using substitution, we can see that the worst-case time complexity of quicksort is $\Theta(n^2)$.

What about the average-case time complexity of quicksort? It turns out that the average-case time complexity of quicksort is also $\Theta(n\log(n))$. The following provides an intuitive explanation for why this is the case: if every element in an array has an equal chance of being selected as the pivot, then there is a 50% chance the pivot ends up in the "middle half" of the sorted values, close to the median. *Note that we are talking about the elements whose values are closest to the median value, and not the elements that are physically located in the middle of the initial array, which may be unsorted.*




If the pivot does land in this range, the worst possible partition that could happen is a 3-to-1 split, as shown below.



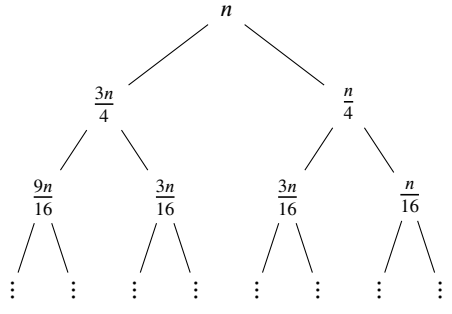




Thus, in the average-case, the recurrence relation for half of the quicksort calls is at worst

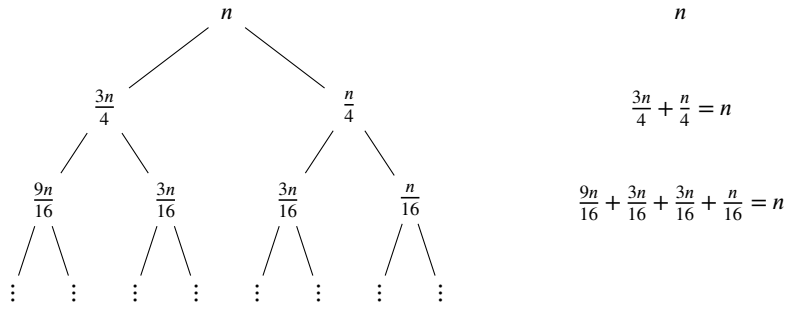
$$T(n) = \begin{cases} 1, & \text{if } n = 0 \text{ or } 1 \\ T(3n/4)+T(n/4)+n, & \text{if } n > 1 \end{cases}$$

What is the time complexity of this recurrence relation? We can use a recursion tree to gain some intuition.

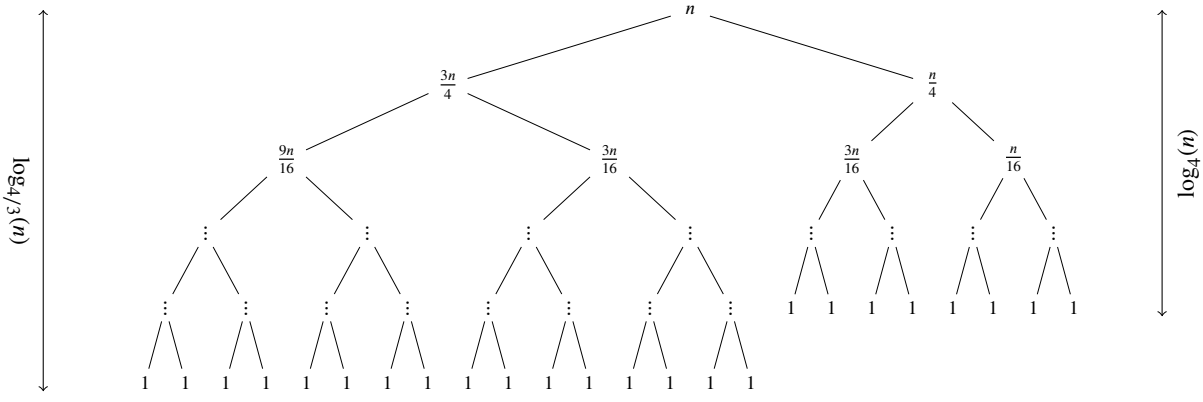
Recursion Tree for $T(n) = T(3n/4)+T(n/4)+n$



If you are not familiar with recursion trees, you will not have to worry about them for this class (however, they were briefly covered in chapter 5). Essentially, the sum of all values at each level of the recursion tree represents the total amount of work that is done at that recursion depth. To determine the total amount of work done throughout all the recursive calls, calculate the sum of all values in the tree. In this case, we can see that $\Theta(n)$ work is done for the first few recursion depths.

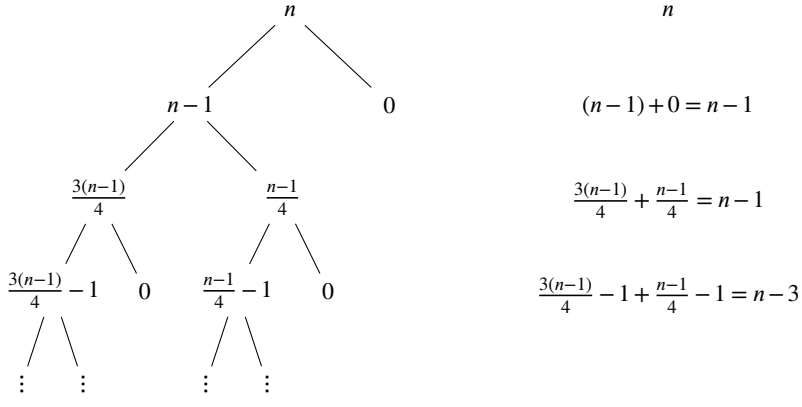



Since the amount of work done at each level of the tree is $n$, the total amount of work that is done should be equal to $n \times$ the number of levels in the tree. However, there is a catch: since the right branch represents a subproblem that is 1/3 the size of the left branch, the input size of the right branch decreases faster, and thus hits the base case first. The number of levels required for the $T(n/4)$ recursive call to hit the base case is approximately $\log_4(n)$, since the input size is reduced by a factor of 4 with each iteration. On the other hand, the number of levels required for the $T(3n/4)$ call to hit the base case is approximately $\log_{4/3}(n)$, since the input size is reduced by a factor of 4/3 with each iteration.




Because of this imbalance, there exist layers at the bottom of the tree where only the $T(3n/4)$ recursive call has any work to do (the left side of the tree in the above illustration). These layers therefore complete less than $n$ work (since $n$ is the amount of work needed if *both* recursive calls still have work to do). As a result, the total amount of work required for the entirety of quicksort must be less than $n \times \log_{4/3}(n)$ if every pivot results in a 3-1 split. This is $\Theta(n \log(n))$.

However, in the average-case, only 50% of pivots end up creating a 3-1 split or better. Let's consider what would happen if we ended up with a worst-case pivot (i.e., either the largest or smallest value) the other 50% of the time. To make our analysis easier, suppose the 3-1 and worst-case splits alternate between pivots (i.e., the first pivot creates a worst-case split, the second pivot creates a 3-1 split, the third pivot creates a worst-case split, etc.). The recursion tree would look like this:

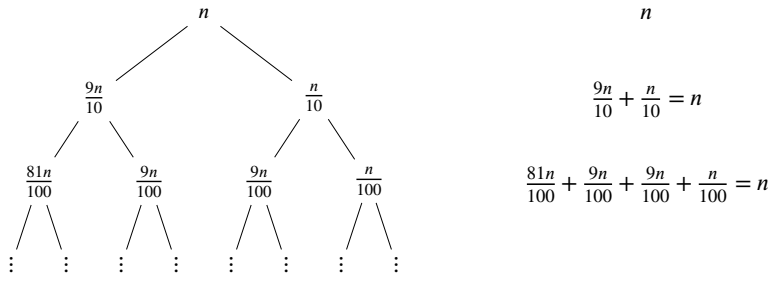



How much work is done in this scenario? While this may seem tricky to solve, there is a key insight that can make this problem easier. Notice that this tree is almost exactly the same as the one introduced right before it, where every pivot resulted in a 3-1 split. The only difference here is that the 3-1 split is done at every *other* level. Therefore, the number of levels in this tree must be double what it was before.

For example, our first recursion tree reduced the input size of the larger recursive call by a factor of 4/3 at every level, so a total of $\log_{4/3}(n)$ levels were needed before we reached the base case. When we introduced the worst-case pivot, however, the input size was reduced by a factor of 4/3 every *two* levels instead. It now takes twice as many levels (i.e., $2\log_{4/3}(n)$) to reach the base case! Since each level still takes $\Theta(n)$ time, the total work involved with the alternating 3-1 and worst-case split is $\Theta(n) \times 2\log_{4/3}(n)$, which simplifies to $\Theta(n\log(n))$.

With average-case behavior, we would expect that half of the pivots would fall within the middle half of sorted values. The example above was the worst-case scenario that could happen with this setup: the pivots that were within the middle half were as far away from the median as possible (3-1 split), and the pivots outside the middle half always ended up being the largest or smallest value in the array. Yet, even with this worst-case scenario, the time complexity of quicksort still ended up being $\Theta(n\log(n))$. Since the average-case time complexity of quicksort cannot be worse than $\Theta(n\log(n))$ (as shown above), but it also cannot be better than $\Theta(n\log(n))$ (which is the best-case time complexity), it must be precisely equal to $\Theta(n\log(n))$.

For another explanation as to why the average-case of quicksort is $\Theta(n\log(n))$, suppose you got super unlucky and every pivot you chose ended up creating a 9-1 split (i.e., 90% of elements on one side of the pivot and 10% on the other). If this were to happen, the recursion tree for quicksort would look like this:




Using the same logic as before, the number of levels needed for the longest branch to hit the base case is $\log_{10/9}(n)$. Since each level of the tree completes at most $\Theta(n)$ work, the total amount of work needed if you ended up with a 9-1 split every time is $\Theta(n) \times \log_{10/9}(n) = \Theta(n\log(n))$. Thus, even if you got super unlucky and every pivot resulted in a 9-1 split, the time complexity of quicksort would still be $\Theta(n\log(n))$. On average, you would expect to do better than a 9-1 split every time, so the average-case time complexity must intuitively also be $\Theta(n\log(n))$.

Note that these two explanations are not mathematically rigorous, and that formal proofs are significantly more complex. However, they are included to provide an intuitive description as to why quicksort runs in average-case $\Theta(n\log(n))$ time.

In summary, quicksort has a best-case time complexity of $\Theta(n\log(n))$, which occurs if the median is selected as the pivot every time (and thus partitions the array into two subarrays of roughly the same size). Quicksort has a worst-case time complexity of $\Theta(n^2)$, which occurs if either the largest or smallest element gets selected as the pivot at every step (which results in the most imbalanced partition possible, where all elements end up on one side of the pivot). In the average-case, however, quicksort runs in $\Theta(n\log(n))$ time, which we showed above.

How can we improve the performance of quicksort? If we always select the last element as the pivot, we may end up selecting a bad pivot every once in a while. Is there a way to reduce the likelihood of selecting a bad pivot?

As mentioned before, the best pivot you can choose for quicksort is the median. Thus, one method is to use the `std::nth_element()` function at every iteration of quicksort to identify the median in $\Theta(n)$ time. By doing this, the worst-case time complexity of quicksort becomes $\Theta(n\log(n))$, since the median would always be chosen as the pivot. However, this approach is never used in practice. Even though the worst-case time complexity is theoretically better, the algorithm itself runs slower in most cases. This is because the coefficient term in front of the $n\log(n)$ is large enough that the change does not translate to increased performance for reasonable input sizes.

Since it is expensive to find the median at every step, we can instead estimate the median by taking a sample of the elements we want to sort. A simple approach would be to randomly select three elements in the array and select the median of these three elements as the pivot. Randomly selecting three elements is a constant time operation, so it is not as costly as finding the exact median. The worst-case time complexity of this approach would still be $\Theta(n^2)$ since you could still get extremely unlucky with your random values, but the likelihood of this worst case happening is extremely low. In the example below, 1, 42, and 28 are randomly chosen, and the median of these three numbers, 28, is used as the pivot. To make partitioning easier, the chosen pivot, 28, can be swapped with the last element — by doing so, we can run quicksort as if the last element were always chosen as the pivot.

| 17 | 15 | 34 | 2 | 1 | 4 | 42 | 14 | 34 | 57 | 18 | 9 | 28 | 84 | 5 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

It should also be noted that quicksort uses $\Theta(\log(n))$ auxiliary space in the worst case. Consider the code for quicksort:

```
1  void quicksort(std::vector<int32_t>& vec, int32_t left, int32_t right) {
2    if (left + 1 >= right) {
3      return;
4    } // if
5    int32_t pivot = partition(vec, left, right);
6    quicksort(vec, left, pivot);
7    quicksort(vec, pivot + 1, right);
8  } // quicksort()
```

Because quicksort is not tail-recursive (there are two recursive calls on lines 6 and 7), the algorithm requires additional memory in the form of stack frames. By implementing quicksort such that the partition with the smaller input size is recursively sorted first (on line 6), we can guarantee that the partition with the larger input size is sorted as the last step of the algorithm (on line 7). This ensures that memory usage does not approach $\Theta(n)$, since the deeper recursive call is always done last and is tail recursive. The smaller recursive call will only require up to $\Theta(\log(n))$ stack frames (since if it uses any more, it cannot be the smaller recursive call). The additional stack frames required by quicksort is therefore bounded by $\Theta(\log(n))$.

```
1   void quicksort(std::vector<int32_t>& vec, int32_t left, int32_t right) {
2     if (left + 1 >= right) {
3       return;
4     } // if
5     int pivot = partition(vec, left, right);
6     if (pivot - left < right - pivot) {
7       quicksort(vec, left, pivot);
8       quicksort(vec, pivot + 1, right);
9     } // if
10    else {
11      quicksort(vec, pivot + 1, right);
12      quicksort(vec, left, pivot);
13    } // else
14  } // quicksort()
```

Overall, the efficiency of quicksort depends on which pivot is selected. If the pivot value always ends up as the largest or smallest value to be sorted, the runtime of quicksort ends up being $\Theta(n^2)$. However, this is rather uncommon, and tuning methods like random sampling can be used to prevent this worst case from happening frequently. Additionally, most implementations of quicksort are not stable due to the nature of the partitioning step, and the complex structure of the algorithm makes it non-adaptive. A summary of the quicksort algorithm is shown below:

| Best Time | Average Time | Worst Time | Auxiliary Space | Stable? | Adaptive? |
|---|---|---|---|---|---|
| $\Theta(n\log(n))$ | $\Theta(n\log(n))$ | $\Theta(n^2)$ | $\Theta(\log(n))$ | No | No |

## 14.7 Mergesort

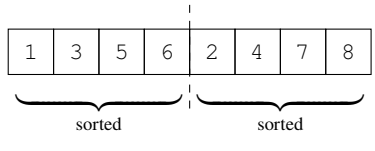
**Mergesort** is another divide-and-conquer sorting algorithm. During the mergesort process, the input array is divided into two halves. The two halves are then sorted recursively and merged together using a `merge()` function. An implementation of mergesort is shown in the code below:

```
1   void mergesort(std::vector<int32_t>& vec, int32_t left, int32_t right) {
2     // base case: if there are fewer than two items, return
3     if (right - left < 2) {
4       return;
5     } // if
6     // split the array in half
7     int32_t mid = left + (right - left) / 2;
8     // recursive merge the left and right halves
9     mergesort(vec, left, mid);
10    mergesort(vec, mid, right);
11    // merge the two sorted halves together
12    merge(vec, left, mid, right);
13  } // mergesort()
```

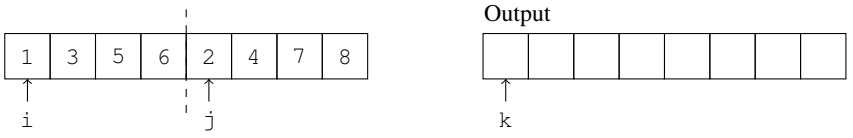
On line 12, we merge two sorted arrays together. This merging step can be done in $\Theta(n)$ time using the following process:

1. Keep track of two indices, `i` and `j`, that refer to the first element of each of the sorted halves.
2. Initialize a separate vector of size `vec.size()` to store the merged output.
3. If `vec[i] <= vec[j]`, copy `vec[i]` to the output vector and increment `i`. Otherwise, copy `vec[j]` to the output vector and increment `j`.
4. Repeat until the entire output vector is filled. If one sorted half reaches the end before the other, simply append all of the remaining elements in the unfinished half into the output vector.

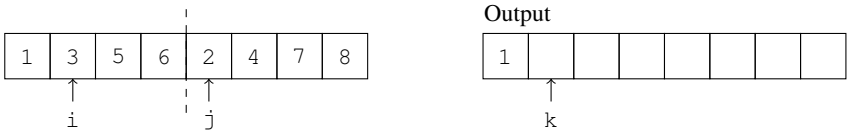
Consider the `merge()` function on the following unsorted array: `[5, 6, 3, 1, 8, 2, 4, 7]`. During the mergesort process, we first recursively mergesort the two halves of the array, `[5, 6, 3, 1]` and `[8, 2, 4, 7]` (as shown on lines 9 and 10 of the above code). This sorts the two halves of the array. After this step, the contents of the array are as follows:



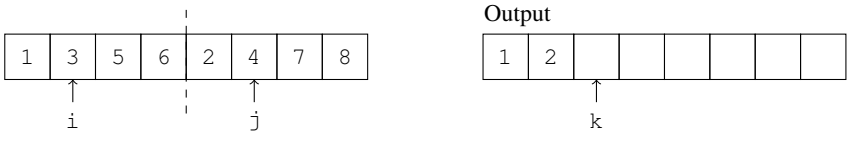
To merge the two sorted halves, we will first initialize an empty vector to store the sorted output and initialize indices `i` and `j` to refer to the first elements of the two halves. The variable `k` represents the next open slot in the output vector.



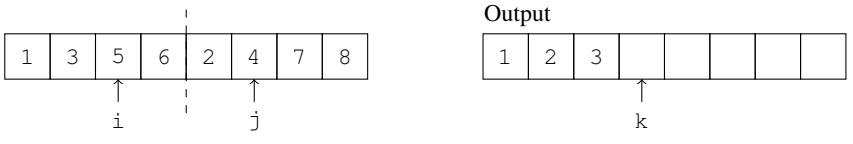

We then compare `i` and `j` and copy the smaller of the two values to the output vector. The index `k` is then incremented, along with either `i` or `j` depending on which element was copied. In this case, $1 < 2$, so we move 1 to the output vector and increment `i`.



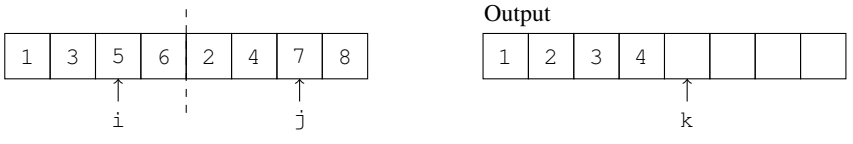

Now, `i` points to 3 and `j` points to 2. Since $2 < 3$, we copy 2 to the output vector and increment `j`.



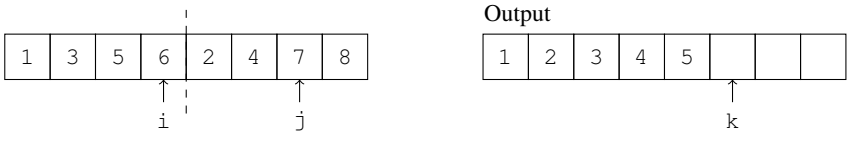

Since $3 < 4$, we copy 3 to the output vector and increment `i`.



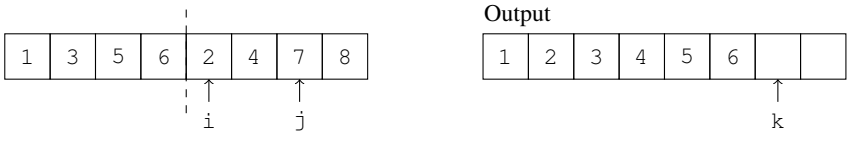

Since $4 < 5$, we copy 4 to the output vector and increment `j`.



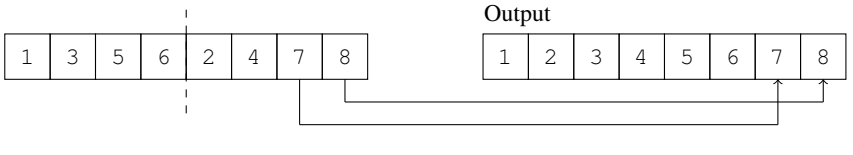

Since $5 < 7$, we copy 5 to the output vector and increment `i`.




Since $6 < 7$, we copy 6 to the output vector and increment `i`.




Notice that `i` has gone off the edge of its portion of the array! As a result, we know that every element in the first half has been added to the output vector already. This means that the remaining elements in the second half are guaranteed to be larger than the elements added so far, so they can be copied to the output vector one by one.



This merge process is implemented in the code below:

```
void merge(std::vector<int32_t>& vec, int32_t left, int32_t mid, int32_t right) {
  int32_t size = right - left;
  std::vector<int32_t> output(size);  // init output vector with 'size' elements
  for (int32_t i = left, j = mid, k = 0; k < size; ++k) {
    if (i == mid) {             // if first half is complete
      output[k] = vec[j++];     // copy all elements in second half over
    } // if
    else if (j == right) {      // if second half is complete
      output[k] = vec[i++];     // copy all elements in first half over
    } // else if
    else {
      if (vec[i] <= vec[j]) {
        output[k] = vec[i++];  // add vec[i] to output
      } // if
      else {
        output[k] = vec[j++];  // add vec[j] to output
      } // else
    } // else
  } // for i
  // copy contents of output to original vector
  std::copy(output.begin(), output.end(), vec.begin() + left);
} // merge()
```

Notice that duplicates are handled by this `merge()` function because `vec[i]` is always added before `vec[j]` if the two values are equal. This ensures that the correct number of duplicates are added to the output vector, and duplicates in the first half are copied to the output vector before duplicates in the second half. Since merging preserves the relative ordering of duplicate values, the mergesort algorithm is also stable!

What is the time complexity of mergesort? If we look at the `merge()` function implemented above, we can see that the algorithm does a single pass through all the elements in its input vector range. Thus, the merging operation takes $\Theta(n)$ time. We can now express the mergesort function as a recurrence relation (the code is reproduced below).

```
void mergesort(std::vector<int32_t>& vec, int32_t left, int32_t right) {
  // base case: if there are fewer than two items, return
  if (right - left < 2) {
    return;
  } // if
  // split the array in half
  int32_t mid = left + (right - left) / 2;
  // recursive merge the left and right halves
  mergesort(vec, left, mid);
  mergesort(vec, mid, right);
  // merge the two sorted halves together
  merge(vec, left, mid, right);
} // mergesort()
```

Since mergesort makes two recursive calls with input size $n/2$ and calls `merge()` (which takes linear time), the mergesort process can be expressed using the following recurrence relation:

$$T(n) = \begin{cases} 1, & \text{if } n = 1 \\ 2T(n/2) + n, & \text{if } n > 1 \end{cases}$$

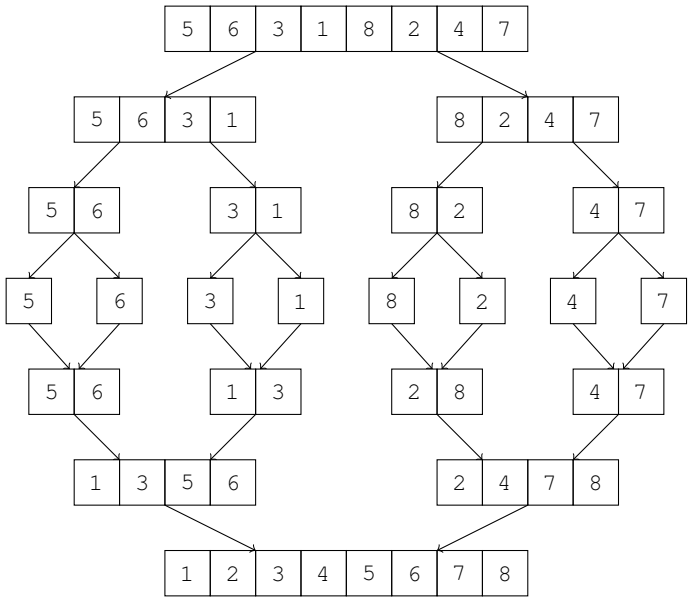
Using Master's Theorem gives us a complexity of $\Theta(n\log(n))$. Since mergesort always splits the input into two halves and merges them together, the best-case, average-case, and worst-case time complexities all involve the same recurrence relation, and thus are all $\Theta(n\log(n))$.

Additionally, since mergesort initializes an output vector to store the contents of the sorted output, the auxiliary space required by the algorithm is $\Theta(n)$, as the size of the output vector is linear on the number of elements that need to be sorted.[1]

Unlike many of the other sorting algorithms covered, mergesort does not require random access. As a result, mergesort is great for linked list sorting, external memory sorting, and parallel sorting. External memory sorting is useful if the data that needs to be sorted is so large that it cannot fit into the main memory of a computer. Mergesort can also be done concurrently; if you have a large collection of data, you can split the data into segments and send each part to a different server. Each server is then able to sort its input separately and return the output, which can then be combined together.

[1] It is actually possible to run mergesort on a *linked list* using $\Theta(1)$ auxiliary space, but we will not cover this implementation here.

The following diagram illustrates the entire mergesort process on our initial array.




The implementation of mergesort we have introduced is a *top-down* implementation, since it uses recursion to break the input array into halves that can be sorted individually. However, it is also possible to implement mergesort iteratively, without any recursive calls. The following is an implementation of this *bottom-up* mergesort:

```
1  void mergesort(std::vector<int32_t>& vec, int32_t left, int32_t right) {
2    for (int32_t size = 1; size <= right - left; size *= 2) {
3      for (int32_t i = left; i <= right - size; i += 2 * size) {
4        merge(vec, i, i + size, min(i + 2 * size, right));
5      } // for i
6    } // for size
7  } // mergesort()
```

This variation of mergesort uses nested `for` loops to simulate the behavior of the recursive calls. First, it merges groups of two elements, then it merges groups of four elements, then it merges groups of eight elements, and so on until the entire array is sorted. To illustrate this, consider the unsorted array `[5, 6, 3, 1, 8, 2, 4, 7]`. On the first pass, the algorithm considers elements in pairs of two and merges these pairs together:

| | |
|---|---|
| `[`**`5, 6,`** `3, 1, 8, 2, 4, 7]` | Merge 5 and 6 in sorted order. |
| `[5, 6,` **`1, 3,`** `8, 2, 4, 7]` | Merge 3 and 1 in sorted order. |
| `[5, 6, 1, 3,` **`2, 8,`** `4, 7]` | Merge 8 and 2 in sorted order. |
| `[5, 6, 1, 3, 2, 8,` **`4, 7`**`]` | Merge 4 and 7 in sorted order. |

On the second pass, `size` increases to two, and the algorithm merges elements in groups of four:

| | |
|---|---|
| `[`**`1, 3, 5, 6,`** `2, 8, 4, 7]` | Merge 5, 6, 1, and 3 in sorted order. |
| `[1, 3, 5, 6,` **`2, 4, 7, 8`**`]` | Merge 2, 8, 4, and 7 in sorted order. |

On the third and final pass, size gets increased to three, and the algorithm merges elements in groups of eight:

| | |
|---|---|
| `[`**`1, 2, 3, 4, 5, 6, 7, 8`**`]` | Merge 1, 3, 5, 6, 2, 4, 7, and 8 in sorted order. |

A summary of mergesort is shown in the table below:

| Best Time | Average Time | Worst Time | Auxiliary Space | Stable? | Adaptive? |
|---|---|---|---|---|---|
| $\Theta(n \log(n))$ | $\Theta(n \log(n))$ | $\Theta(n \log(n))$ | $\Theta(n)$ | Yes | No |

## 14.8 Analysis of Comparison Sorts

All of the sorting algorithms we have discussed so far are *comparison-based* sorting algorithms. These sorting algorithms sort a container by comparing elements with each other using a comparison operator.

| Sort | Best Time | Average Time | Worst Time | Auxiliary Space | Stable? | Adaptive? |
|---|---|---|---|---|---|---|
| Bubble | $\Theta(n)$ | $\Theta(n^2)$ | $\Theta(n^2)$ | $\Theta(1)$ | Yes | Yes |
| Selection | $\Theta(n^2)$ | $\Theta(n^2)$ | $\Theta(n^2)$ | $\Theta(1)$ | No | No |
| Insertion | $\Theta(n)$ | $\Theta(n^2)$ | $\Theta(n^2)$ | $\Theta(1)$ | Yes | Yes |
| Heap | $\Theta(n\log(n))$ | $\Theta(n\log(n))$ | $\Theta(n\log(n))$ | $\Theta(1)$ | No | No |
| Quick | $\Theta(n\log(n))$ | $\Theta(n\log(n))$ | $\Theta(n^2)$ | $\Theta(\log(n))$ | No | No |
| Merge | $\Theta(n\log(n))$ | $\Theta(n\log(n))$ | $\Theta(n\log(n))$ | $\Theta(n)$ | Yes | No |

Just by looking at the table, one may assume that heapsort, quicksort, and mergesort should always be used over the elementary sorts, since their time complexities are the best. However, this is not actually true. If the array you want to sort is small, insertion sort actually performs the best! How can this be? In this section, we will answer three relevant questions regarding the analysis of comparison-based sorting algorithms.

1. How is it possible for an elementary sort to perform better than an advanced sort?
2. Is it possible for a comparison sort to have a worst-case time complexity that is better than $\Theta(n\log(n))$?
3. Which sorting implementation does the STL use in its implementation of `std::sort()`?

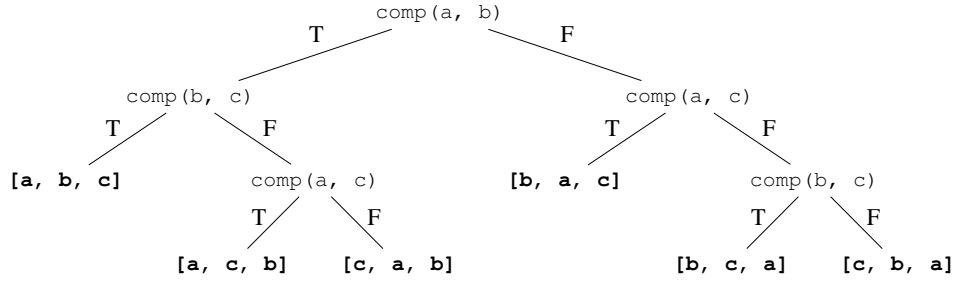
### ※ 14.8.1 *Performance of Elementary and Advanced Sorts*

For the first question, it is important to remember that time complexity is simply a measurement of how runtime scales, not the actual runtime itself. For instance, an algorithm that takes $2n$ steps is part of the same complexity class as an algorithm that takes $200n$ steps — however, the $2n$ algorithm would run much faster than the $200n$ algorithm. This can be extended to algorithms in different complexity classes. For instance, if one algorithm takes $2n^2$ steps and the other takes $200n\log(n)$ steps, the second algorithm has a better time complexity than the first one. However, this only tells us that the runtime of the second algorithm scales slower than the runtime of the first algorithm as the input size grows. This does not mean that the second algorithm is downright faster; if $n$ is small, the $2n^2$ algorithm would run faster than the $200n\log(n)$ algorithm, despite the fact that a $\Theta(n\log(n))$ algorithm has a better time complexity than a $\Theta(n^2)$ algorithm. This reasoning is precisely why insertion sort is faster than many $\Theta(n\log(n))$ sorts for small input sizes, despite its inferior complexity class. Because insertion sort is simple, it does not need to do as much work per element compared to the more advanced sorts, giving it an advantage for small input sizes. The benefit of doing more work per element only becomes advantageous as the input size becomes large.

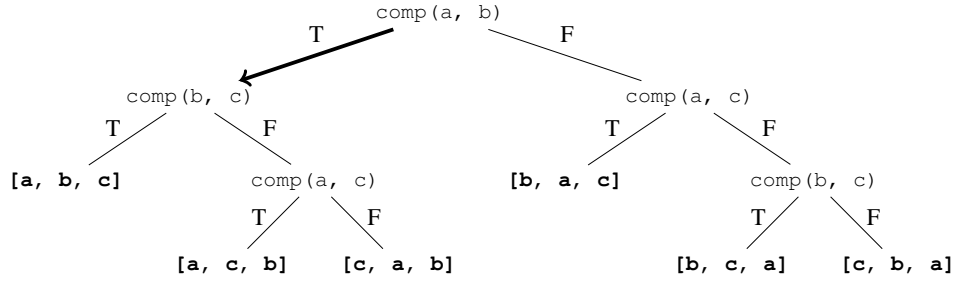
### ※ 14.8.2 *Can Comparison Sorts Do Better Than Θ(n log n) Time?*

The second question is bit tougher to answer, but the answer is no: it is not possible for a comparison sort to have a worst-case time complexity that is better than $\Theta(n\log(n))$. If you are only allowed to use comparisons to determine the sorted order of elements in an arbitrary container, you *must* make $\Omega(n\log(n))$ comparisons in the worst case if you want to ensure that your container is sorted. Let's look at why.

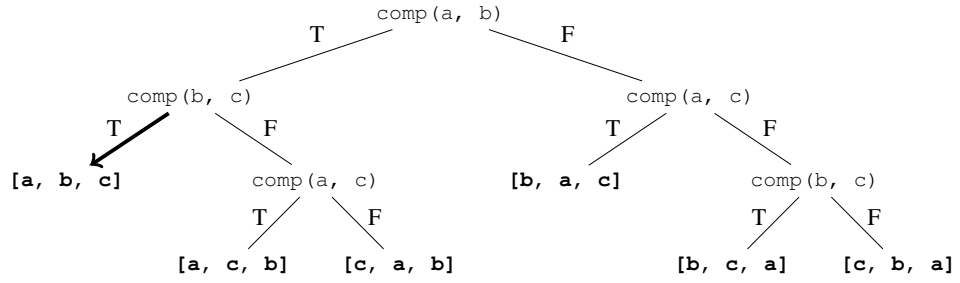
Suppose you have a comparator `comp` that can be used to compare two elements. Given two values `x` and `y`, `comp(x, y)` would return `true` if `x` $\leq$ `y` and `false` if `x` > `y`. You are then given three values, `a`, `b`, and `c`, and you are told to determine the sorted order of these three values using only the comparator `comp`. The comparisons you need to make to determine the sorted order of these three elements are shown in the following decision tree:



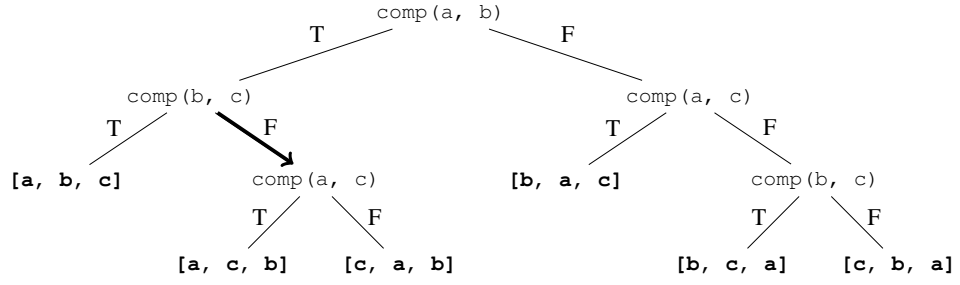

Since you are given three values, there are a total of $3! = 6$ permutations that could potentially be the true sorted order. These six permutations are bolded in the tree above. To determine which of the six permutations is the correct sorted order, you would have to compare the elements with each other and walk down the correct branch of the tree. For instance, if `comp(a, b)` returned `true`, that would mean `a` is less than or equal to `b`, and you would walk down the left branch of the tree (these are all the permutations where `a` is before `b`).



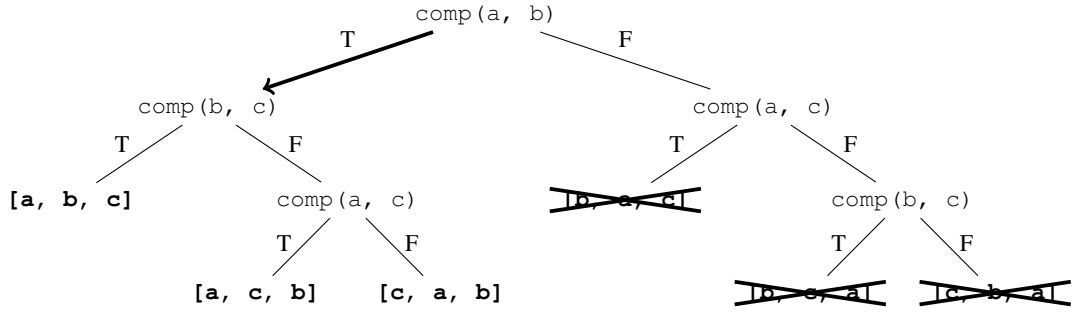
Then, you would compare `b` and `c`. If `comp(b, c)` returns `true`, you would know that the correct order is `[a, b, c]`, since `a` is less than or equal to `b`, and `b` is less than or equal to `c`.




However, if `comp(b, c)` returns `false`, you would have to compare `a` and `c` before you can determine the correct order. If `c` is less than `a`, the correct order would be `[c, a, b]`. Otherwise, it would be `[a, c, b]`.




In the worst case, how many comparisons do you need before you can determine with certainty what the correct sorted order is? Notice that every comparison you make gets rid of roughly half the possible permutations. For instance, if `comp(a, b)` returns `true`, you would be able to eliminate `[b, a, c]`, `[b, c, a]`, and `[c, b, a]` as possible orders.




If you given a container of $n$ elements, there would be a total of $n!$ possible permutations that could potentially be the sorted order. Every comparison that you make is able to get rid of half of the permutations from the set of possible orders. As a result, the number of comparisons you need in the worst case is equal to the number of times you can divide $n!$ in half before it reaches 1 (i.e., halving $n!$ permutations until a single permutation is left). We can express the number of comparisons that are needed as the variable $x$ in the following equation:

$$\frac{n!}{2^x} = 1$$

Solving for $x$, we get:

$$x = \log_2(n!)$$

Thus, $\log_2(n!)$ comparisons are needed at a minimum to sort $n$ elements in the worst case. Since we proved in chapter 4 that $\log(n!) = \Theta(n\log(n))$, the number of comparisons needed to sort a container must also be at least $\Theta(n\log(n))$. Because of this, the worst-case time complexity of any comparison-based sorting algorithm cannot be better than $\Theta(n\log(n))$.

### ※ 14.8.3 *Introsort and STL Sorting Implementations*

Lastly, the STL relies on a hybrid sorting algorithm known as **introsort** (short for *introspective sort*) to implement the `std::sort()` function. Introsort starts off with quicksort, but it switches to heapsort if the recursion depth is too high. This prevents introsort from ending up with the $\Theta(n^2)$ scenario of quicksort. Furthermore, if a quicksort partition is very small (fewer than 16 elements on GCC), introsort switches to insertion sort, which is fastest on small arrays.

Introsort is able to achieve $\Theta(n\log(n))$ average-case and worst-case performance by following this process. Thus, introsort is the perfect embodiment of an adaptive sort, as it uses many factors to determine which sorting algorithm it should use at any point in time. Since introsort relies on non-stable sorts, it itself is not stable. However, the `std::stable_sort()` function can be used to maintain stability while sorting. The STL relies on mergesort to implement `std::stable_sort()`, as mergesort is the only advanced comparison sort that maintains the relative order of duplicate elements.

## 14.9 Counting Sort

In this section, we will look at a *linear-time* sorting algorithm. Unlike comparison-based sorting algorithms, linear-time sorting algorithms may run faster than $\Theta(n\log(n))$, but they require specific conditions on the input that must be met. As a result, these sorting algorithms may not be applicable to all situations.

**Counting sort** is a linear-time sorting algorithm that can be used to sort data whose values that fall within a limited set of keys. This algorithm works by first counting the number of times each key appears, and then using arithmetic to determine where each key should be placed in the output sequence. To illustrate this process, consider the following vector of grades. Assume that a grade can only take on six possible values: A, B, C, D, E, and F.

| D | B | A | C | B | F | A | B |
|---|---|---|---|---|---|---|---|

During counting sort, we first do an initial pass of the data and count the number of times each key appears. The results from this initial pass are shown below (in this example, there are 2 A's, 3 B's, 1 C, 1 D, 0 E's, and 1 F):

| 2 | 3 | 1 | 1 | 0 | 1 |
|---|---|---|---|---|---|
| A | B | C | D | E | F |

Then, we iterate over the counters to compute offsets that determine where each key should begin in the sorted container. This is done by adding each counter with the counter values before it. Using the example, the counter for B is updated to $2 + 3 = 5$, the counter for C is updated to $2 + 3 + 1 = 6$, the counter for D is updated to $2 + 3 + 1 + 1 = 7$, and so on.

| 2 | 5 | 6 | 7 | 7 | 8 |
|---|---|---|---|---|---|
| A | B | C | D | E | F |

After this, we do another pass of the original data, but this time in reverse order (this allows counting sort to be stable). For each value we encounter during this pass, we

1. look up the value's offset in the offset vector (we'll call this value $k$)
2. place the value at index $k-1$ of an output vector
3. decrement $k$ in the offset vector

In our example, the first data value we encounter is B. We see that B has an offset value of 5. As a result, we would write B to index 5 - 1 = 4 of an output vector and decrement the offset of B.

| | | | | **B** | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

| 2 | **4** | 6 | 7 | 7 | 8 |
|---|---|---|---|---|---|
| A | B | C | D | E | F |

The next value is A. Since A has an offset value of 2, we write A to index 2 - 1 = 1 of the output vector and decrement the offset of A.

| | **A** | | | B | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

| **1** | 4 | 6 | 7 | 7 | 8 |
|---|---|---|---|---|---|
| A | B | C | D | E | F |

The next value is F. Since F has an offset value of 8, we write F to index 8 - 1 = 7 of the output vector and decrement the offset of F.

| | A | | | B | | | **F** |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

| 1 | 4 | 6 | 7 | 7 | **7** |
|---|---|---|---|---|---|
| A | B | C | D | E | F |

The next value is B. Since B has an offset value of 4, we write B to index 4 - 1 = 3 of the output vector and decrement the offset of B.

| | A | | **B** | B | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

| 1 | **3** | 6 | 7 | 7 | 7 |
|---|---|---|---|---|---|
| A | B | C | D | E | F |

The next value is C. Since C has an offset value of 6, we write C to index 6 - 1 = 5 of the output vector and decrement the offset of C.

| | A | | B | B | **C** | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

| 1 | 3 | **5** | 7 | 7 | 7 |
|---|---|---|---|---|---|
| A | B | C | D | E | F |

The next value is A. Since A has an offset value of 1, we write A to index 1 - 1 = 0 of the output vector and decrement the offset of A.

| **A** | A | | B | B | C | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

| **0** | 3 | 5 | 7 | 7 | 7 |
|---|---|---|---|---|---|
| A | B | C | D | E | F |

The next value is B. Since B has an offset value of 3, we write B to index 3 - 1 = 2 of the output vector and decrement the offset of B.

| A | A | **B** | B | B | C | | F |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

| 0 | **2** | 5 | 7 | 7 | 7 |
|---|---|---|---|---|---|
| A | B | C | D | E | F |

The next value is D. Since D has an offset value of 7, we write D to index 7 - 1 = 6 of the output vector and decrement the offset of D. After writing D, we are done with the traversal, and everything is sorted.

| A | A | B | B | B | C | **D** | F |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

| 0 | 2 | 5 | **6** | 7 | 7 |
|---|---|---|---|---|---|
| A | B | C | D | E | F |

The code for this example is provided below:

```
1  void counting_sort(std::vector<char>& grades) {
2    size_t num_keys = 6;  // six possible grade values
3    std::vector<char> output(grades.size());
4    std::vector<size_t> offsets(num_keys);
5    // first pass: iterate through data and add to counter
6    for (auto it = grades.begin(); it != grades.end(); ++it) {
7      ++offsets[static_cast<size_t>(*it - 'A')];  // A = 0, B = 1, ...
8    } // for it
9    // second pass: calculate cumulative sum for offset
10   for (size_t i = 1; i < num_keys; ++i) {
11     offsets[i] += offsets[i - 1];
12   } // for i
13   // third pass: write to output vector
14   for (auto it = grades.rbegin(); it != grades.rend(); ++it) {
15     output[--offsets[static_cast<size_t>(*it - 'A')]] = *it;
16   } // for it
17   std::swap(grades, output);
18 } // counting_sort()
```

What is the time complexity of counting sort? Notice that this algorithm does two traversals of the original data vector and one traversal of the offset vector. If we denote the size of the data vector as $n$ and the size of the offset vector as $k$, the total work completed by the algorithm is $2n+k$, or $\Theta(n+k)$. Similarly, since the algorithm allocates an additional output vector of size $n$ and an additional offset vector of size $k$, the auxiliary space used is $\Theta(n+k)$.

There is another variation of counting sort that can be used if the objects that are being counted are also the objects that need to be sorted. In this variation, the values to be sorted are written directly into the original vector, removing the need for a separate output vector. A sample implementation is shown below (using the grade example):

```
1  void in_place_counting_sort(std::vector<char>& grades) {
2    size_t num_keys = 6;  // six possible grade values
3    std::vector<size_t> counters(num_keys);
4    // count the number of times each grade appears
5    for (char grade : grades) {
6      ++counters[static_cast<size_t>(grade - 'A')];
7    } // for grade
8    // use the counters to determine how many of each grade to write to output
9    size_t current = 0;
10   for (size_t i = 0; i < num_keys; ++i) {
11     for (size_t j = 0; j < counters[i]; ++j) {
12       grades[current++] = static_cast<char>('A' + i);
13     } // for j
14   } // for i
15 } // in_place_counting_sort()
```

This implementation essentially counts the number of A's, B's, C's, D's etc. that exist in the data. After counting, the algorithm directly writes the correct number of each key to the output vector (in the example above, it would iterate through the original vector and overwrite the first two values with A's, the next three values with B's, and so on). This in-place variation of counting sort uses less auxiliary space, but it may not work in all situations depending on the type of object being sorted. It is also *not* stable, which could cause issues depending on the type of problem you are trying to solve. For example, the implementation of *radix sort* — another linear-time sorting algorithm that can be used to sort a collection of numeric values — relies on a stable implementation of counting sort to work.

| Best Time | Average Time | Worst Time | Auxiliary Space | Stable? | Adaptive? |
|---|---|---|---|---|---|
| $\Theta(n+k)$ | $\Theta(n+k)$ | $\Theta(n+k)$ | $\Theta(n+k)$ | Yes | No |

## 14.10 Radix Sort (*)

**Radix sort** is another linear-time sorting algorithm that can be used to sort a collection of numeric values or fixed-size strings. During the radix sort process, the individual digits or characters of the data values are first grouped together based on their significance position. Then, counting sort is used to sort the digits in order from the least significant position to the most significant position. To illustrate how this works, consider the following container of numbers, which will be sorted using radix sort:

| 659 | 424 | 953 | 139 | 380 | 811 | 187 | 354 | 769 | 415 | 286 | 331 |
|---|---|---|---|---|---|---|---|---|---|---|---|

First, counting sort is used to sort the numbers based on only the least significant digit (the ones digit). Since counting sort is stable, the relative ordering of numbers with the same ones digit does not change after the sort is completed (e.g., since 659 comes before 139 in the original data, 659 would still be before 139 after this initial sort).

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 65**9** | 42**4** | 95**3** | 13**9** | 38**0** | 81**1** | 18**7** | 35**4** | 76**9** | 41**5** | 28**6** | 33**1** |

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 38**0** | 81**1** | 33**1** | 95**3** | 42**4** | 35**4** | 41**5** | 28**6** | 18**7** | 65**9** | 13**9** | 76**9** |

Then, counting sort is used to sort the numbers based on only the second-least significant digit (the tens digit).

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 3**8**0 | 8**1**1 | 3**3**1 | 9**5**3 | 4**2**4 | 3**5**4 | 4**1**5 | 2**8**6 | 1**8**7 | 6**5**9 | 1**3**9 | 7**6**9 |

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 8**1**1 | 4**1**5 | 4**2**4 | 3**3**1 | 1**3**9 | 9**5**3 | 3**5**4 | 6**5**9 | 7**6**9 | 3**8**0 | 2**8**6 | 1**8**7 |

Then, counting sort is used to sort the numbers based on only the next least-significant digit (the hundreds digit).

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **8**11 | **4**15 | **4**24 | **3**31 | **1**39 | **9**53 | **3**54 | **6**59 | **7**69 | **3**80 | **2**86 | **1**87 |

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **1**39 | **1**87 | **2**86 | **3**31 | **3**54 | **3**80 | **4**15 | **4**24 | **6**59 | **7**69 | **8**11 | **9**53 |

After sorting on the most significant digit of the input, the entire container is sorted. Since the number of counting sort passes we need is equal to the number of digits in the largest value, the time complexity of radix sort is $\Theta(d(n+k))$, where $d$ is the number of digits in the largest value, $n$ is the input size, and $k$ is the number of distinct keys that need to be sorted; with integers, $k$ is the base that the input is expressed in (e.g., base 10 in the above example). This is because each pass of counting sort takes $\Theta(n+k)$ time, and $d$ passes need to be performed before the container is fully sorted. The auxiliary space used by radix sort is the same as that of counting sort, or $\Theta(n+k)$.

| Best Time | Average Time | Worst Time | Auxiliary Space | Stable? | Adaptive? |
|---|---|---|---|---|---|
| $\Theta(d(n+k))$ | $\Theta(d(n+k))$ | $\Theta(d(n+k))$ | $\Theta(n+k)$ | Yes | No |

## 14.11 Index Sorting

**Index sorting** is a method that can be used to access elements of a container in sorted order without actually sorting the container's data itself. This method is useful if you want to know the sorted order of a collection of elements without changing its original order, or if the container you want to sort contains large objects that are too expensive to move around.

Recall that an element's *index* represents its position in a container. The very first element is located at index 0, the second element is located at index 1, and so on. The goal of index sorting is to sort a *separate container of indices* that serves as a proxy for the data in the original container. The indices, however, are not sorted in ascending order themselves; the "sorted" ordering of indices instead depends on the order of elements in the original data container. For instance, if the element at index `i` of the original container is smaller than the element at index `j`, then `i` would come before `j` in the vector of indices. By sorting indices in this manner, you will be able to access the original data elements in sorted order without having to physically sort the original elements themselves. For example, consider the following vector of `double` objects, which we will call `vec`:

| vec | 7.5 | 4.4 | 6.7 | 3.2 | 5.5 | 1.3 | 9.6 | 8.3 |
|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

Suppose we wanted to know the sorted order of the `double`s in this vector without modifying the order of elements in `vec` itself. We can *index sort* this vector by initializing a separate vector of indices (e.g., `idx = [0, 1, 2, 3, 4, 5, 6, 7]`) and sort these indices based on the `double` value they reference in the original vector. For example, the smallest number in `vec` is `1.3`, which is at index 5. Thus, the value `5` ends up at index 0 in the sorted `idx` vector. The next smallest value is `3.2` at index 3, so the second value in the sorted `idx` vector is 3. Continuing this logic, the final contents of `idx` after index sorting would be `[5, 3, 1, 4, 2, 0, 7, 6]`.

This way, we can retrieve the elements of `vec` in sorted order without modifying the contents of `vec` itself. After sorting `idx`, we know that the smallest element of `vec` is located at index `idx[0]`, the second smallest element is located at index `idx[1]`, and so on. As an example, the following code can be used to print all the elements of `vec` in sorted order:

```
1  for (size_t i = 0; i < idx.size(); ++i) {
2    std::cout << vec[idx[i]] << '\n';
3  } // for i
```

How can we implement index sorting in a program? Recall that the `std::sort()` function accepts an optional comparator `comp` that can be used to determine the ordering of elements:

```
void std::sort(RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

Instead of sorting the vector of indices in ascending order (which would have been the default without a comparator passed in), we want to pass in a comparator that tells `std::sort()` to sort the indices in `idx` based on the contents of `vec`. Thus, our comparator will need to somehow know the values of our original data, so that it knows how the indices should be ordered.

This can be done by using the comparator's internal state to store a *reference* to the original data that you want to index sort. A reference is preferred because you do not want to make a separate copy of the container every time a comparator is constructed (e.g., if you had a giant vector of large objects, the comparator should not make an entirely new copy of this giant vector just to sort the indices). Because references must be initialized upon creation, you must use a constructor with an initializer list to initialize the comparator. An example is shown below:

```
template <typename T>
class IndexSortComparator {
  const std::vector<T>& data; // reference to original data to index sort
public:
  IndexSortComparator(const std::vector<T>& vec)
    : data{vec} {}
  ...
};
```

Since this is a comparator, you will have to implement `operator()`. Normally, to implement a comparator that can be used to sort a container of integers in ascending order, you would just return whether one value is less than the other (if two elements are passed into the comparator, a return value of `true` indicates that the first element comes before the second after sorting):

```
bool operator() (uint32_t i, uint32_t j) const {
  return i < j;
} // operator()
```

However, with index sort, we aren't looking at the values of `i` and `j` when determining which one should come first in the sorted vector of indices. Instead, we want to look at the elements at positions `i` and `j` of the underlying data container! That is, when an index sort is done, the index `i` should come before `j` if the element at index `i` of the underlying data container is less than the element at index `j`. This requires the following modification to our overloaded `operator()`:

```
bool operator() (uint32_t i, uint32_t j) const {
  return data[i] < data[j];
} // operator()
```

Putting this all together, we have the following comparator definition:[2]

```
template <typename T>
class IndexSortComparator {
  const std::vector<T>& data;
public:
  IndexSortComparator(const std::vector<T>& vec)
    : data{vec} {}
  bool operator() (uint32_t i, uint32_t j) const {
    return data[i] < data[j];
  } // operator()
};
```

With this comparator, we can sort the container of indices so that the sorted order of these indices is determined by the ordering of elements in the underlying data container. An example is shown below:

```
std::vector<double> vec = {7.5, 4.4, 6.7, 3.2, 5.5, 1.3, 9.6, 8.3};
// initialize vector of indices
std::vector<int32_t> idx(vec.size());
// initialize indices of idx vector
std::iota(idx.begin(), idx.end(), 0);
// initialize comparator by passing in a reference to vec
IndexSortComparator<double> idx_comp{vec};
// index sort vector of indices by passing idx_comp into sort function
std::sort(idx.begin(), idx.end(), idx_comp);
// final contents of idx vector after index sort is {5, 3, 1, 4, 2, 0, 7, 6}
```

[2]For this to work, `operator<` must be supported by the type `T`.