Lecture Notes
Data Structures and Algorithms

Sorting

Algorithm	Time complexity	Worst case	Benefits	Downsides	Common uses
Bubble	$O(n^2)$	$O(n^2)$	Simple to implement	Poor performance for large datasets	Used as a pedagogical tool, not generally used in practice
Insertion	ß$O(n^2)$	$O(n^2)$	Simple to implement	Poor performance for large datasets	Used as a pedagogical tool, not generally used in practice
Selection	$O(n^2)$	$O(n^2)$	Simple to implement	Poor performance for large datasets	Used as a pedagogical tool, not generally used in practice
Merge	O(n log n)	O(n log n)	Stable, high performance	Requires O(n) additional space	Used in many practical applications due to its performance
Quick	O(n log n)	$O(n^2)$	High performance in practice	Can be unstable	Used in many practical applications due to its performance
Heap	O(n log n)	O(n log n)	High performance	Can be unstable	Used in many practical applications due to its performance
Counting	O(n + k)	O(n + k)	Simple to implement	Only works for integers in a small range	Used in cases where the input data is restricted to a small range of integers
Radix	O(n k)	O(n k)	Simple to implement	Only works for integers	Used in cases where the input data is a list of integers

Overview

1. Introduction to sorting algorithms
    - Definition of sorting
    - Examples of why sorting is useful
    - Overview of different categories of sorting algorithms (quadratic, linear, n log n, etc.)
2. Quadratic sorting algorithms
    - Definition of quadratic time complexity
    - Examples of quadratic sorting algorithms (bubble sort, insertion sort)
    - Pros and cons of quadratic algorithms
3. Linear sorting algorithms
    - Definition of linear time complexity
    - Examples of linear sorting algorithms (counting sort, radix sort)
    - Requirements for using linear algorithms
    - Pros and cons of linear algorithms
4. O(n log n) sorting algorithms
    - Definition of n log n time complexity
    - Examples of n log n sorting algorithms (merge sort, quick sort)
    - Pros and cons of n log n algorithms
5. Heap sort
    - Overview of the heap data structure
    - How heap sort works
    - Time complexity of heap sort
    - Pros and cons of heap sort
6. Lower bound on the complexity of sorting algorithms
    - Definition of comparison-based sorting algorithms
    - Proof that any comparison-based sorting algorithm must have a time complexity of at least O(n log n)

Introduction

1. Definition of sorting: Sorting is the process of rearranging a list of items in a specific order (such as ascending or descending). The goal of sorting is to make it easier to search for and locate specific items within the list.
2. There are many practical applications for sorting algorithms. Some examples include:
    a. Organizing a list of names alphabetically
    b. Sorting a list of products by price
    c. Sorting a list of employees by job title or salary
    d. Sorting a list of students by grade point average

    Often, other algorithms depend on sorted inputs in order to improve their performance.

3. Overview of different categories of sorting algorithms: There are several different categories of sorting algorithms, which are distinguished by their time complexity (how long they take to run). Some common categories include:
    a. **Quadratic**: These algorithms have a time complexity of **$O(n^2)$**, which means that their running time increases exponentially as the size of the input data increases. Examples of quadratic sorting algorithms include **bubble sort** and **insertion sort.**
    b. **Linear**: These algorithms have a time complexity of **O(n)**, which means that their running time increases linearly as the size of the input data increases. Examples of linear sorting algorithms include counting sort and radix sort.
    c. **Loglinear (n log n)**: These algorithms have a time complexity of **O(n log n)**, which means that their running time increases more slowly as the size of the input data increases. Examples of n log n sorting algorithms include merge sort and quick sort.

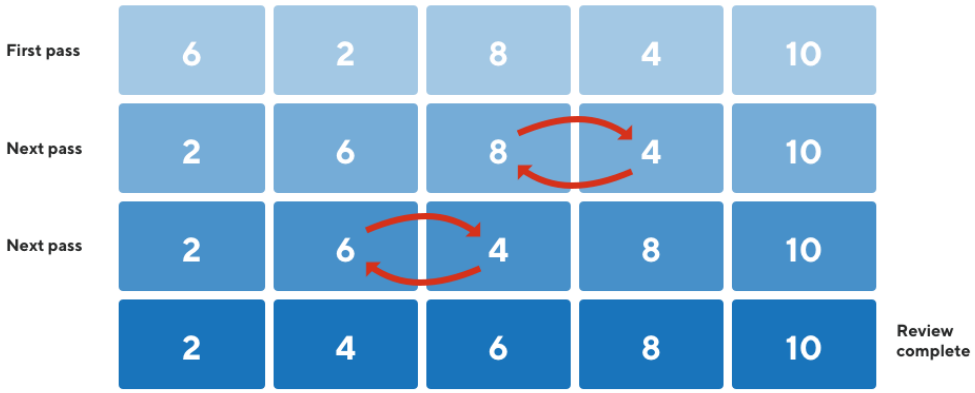
(Also called quasilinear time)

Quadratic Sorting Algorithms

Quadratic sorting algorithms are those that have a time complexity of $O(n^2)$, which means that their running time increases exponentially as the size of the input data increases. Examples of quadratic sorting algorithms include bubble sort and insertion sort.

Some examples of quadratic sorting algorithms include:

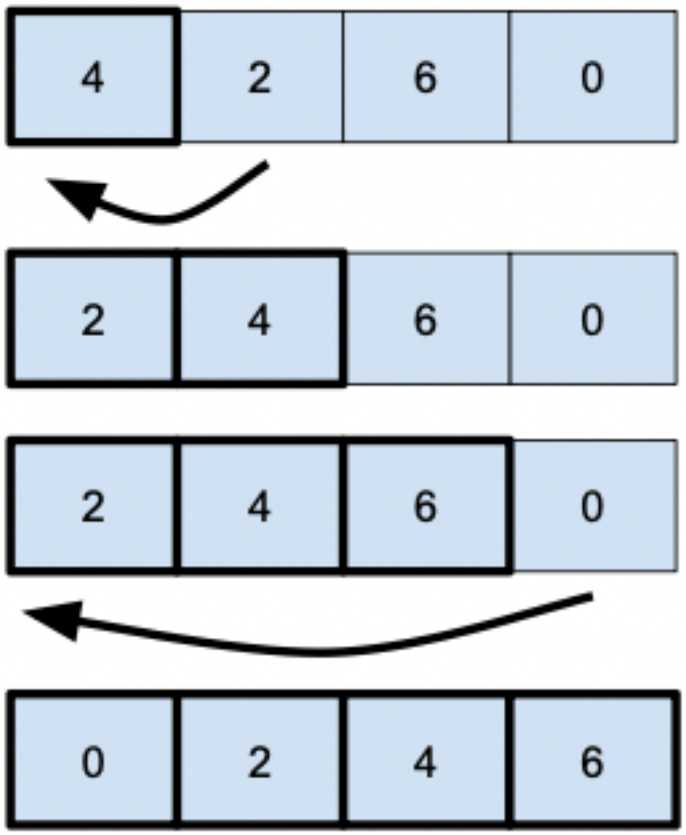
- **Bubble sort**: works by repeatedly comparing pairs of adjacent elements and swapping them if they are in the wrong order.
  It repeats this process until the list is sorted. There are ***n*** elements in the list, and for each element, the algorithm must compare it to at most ***n-1*** other elements (since it only compares elements that come after it in the list). This means that there are ***n(n-1)/2*** comparisons in total, which simplifies to $O(n^2)$.



Note that, in case the list is pre-sorted, the algorithm performs only a single loop over the input, without swapping any elements.

```java
public void bubbleSort(int[] arr) {
        int n = arr.length;
        for (int i = 0; i < n - 1; i++) {
            for (int j = 0; j < n - i - 1; j++) {
                if (arr[j] > arr[j + 1]) {
                    // swap arr[j] and arr[j+1]
                    int temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                }
            }
        }
    }
```

- **Insertion/Selection sort:** works by dividing the input list into a sorted and unsorted portion. It repeatedly takes the next element from the unsorted portion and inserts it into the correct position in the sorted portion.
  There are ***n*** elements in the list, and for each element, the algorithm must compare it to at most ***n-1*** other elements (since it only compares elements that come before it in the sorted portion). This means that there are at most ***n(n-1)/2*** comparisons in total, which simplifies to ***O(n^2)***.




```java
public void insertionSort(int[] arr) {
        int n = arr.length;
        for (int i = 1; i < n; ++i) {
            int key = arr[i];
            int j = i - 1;
            while (j >= 0 && arr[j] > key) {
                arr[j + 1] = arr[j];
                j = j - 1;
            }
            arr[j + 1] = key;
```

```
        }
    }
```

Quadratic algorithms can be simple to implement, but they are not generally used for large datasets because they can take a long time to run. For example, a quadratic algorithm might take a few seconds to sort a list of 1000 elements, but it could take hours or even days to sort a list of 10,000,000 elements.

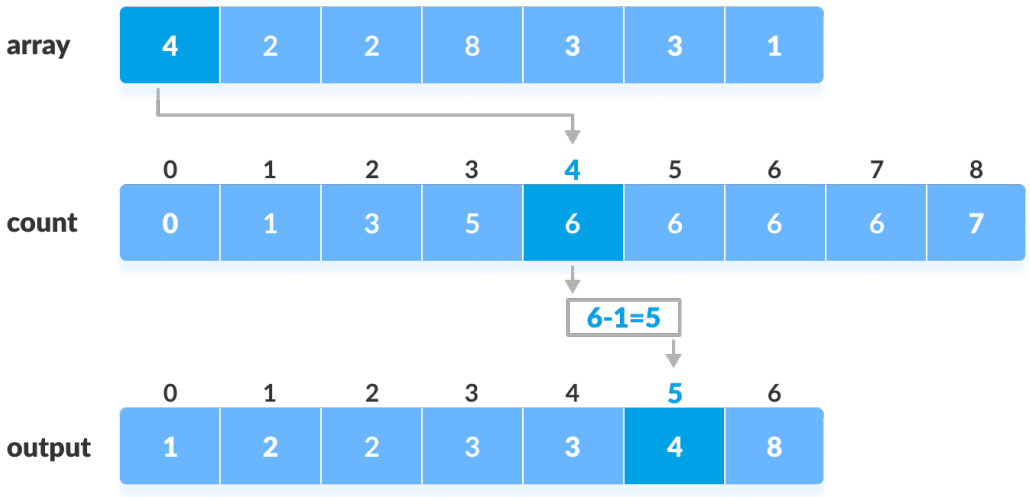
Quadratic algorithms are generally not used in practical applications because of their poor performance for large datasets. They are more commonly used as pedagogical tools to teach the basics of sorting algorithms. In some cases, they can be fast when sorting small inputs due to their simple implementation. Moreover, these algorithms can be highly effective on small inputs that are (mostly) sorted, meaning that only a few or no elements are out of place.

Linear Sorting Algorithms

Linear sorting algorithms are those that have a time complexity of ***O(n)***, which means that their running time increases linearly as the size of the input data increases. Examples of linear sorting algorithms include counting sort and radix sort.

Examples of linear sorting algorithms: Some examples of linear sorting algorithms include:

- **Counting sort**: works by counting the number of occurrences of each element in the input list, and then using this information to create a sorted list. It does this by creating an auxiliary array of size **k**, where k is the range of possible values for the elements in the input list. The algorithm then iterates through the input list, incrementing the count for each element in the auxiliary array. Finally, it iterates through the auxiliary array, creating the sorted list by adding each element to the output list the number of times indicated by its count.




```java
public void countingSort(int[] arr, int min, int max) {
    int n = arr.length;
    // Create an auxiliary array of size k
    int k = max - min + 1;
    int[] count = new int[k];
    // Initialize the count array
    for (int i = 0; i < k; i++) {
        count[i] = 0;
    }
    // Count the number of occurrences of each element
    for (int i = 0; i < n; i++) {
```

```
            count[arr[i] - min]++;
        }
        // Create the sorted list
        int j = 0;
        for (int i = min; i <= max; i++) {
            while (count[i - min]-- > 0) {
                arr[j++] = i;
            }
        }
    }
```

- **Radix sort:** works by sorting the input list one digit at a time, starting with the least significant digit and working its way up to the most significant digit. It does this by creating an auxiliary array of size n, and then repeatedly iterating through the input list and placing each element in the correct position in the auxiliary array based on its current digit. Once all the digits have been processed, the auxiliary array is copied back into the input list, resulting in a sorted list.

1	3	2
5	4	3
7	8	3
0	6	3
0	0	7
8	9	8
0	4	9

0	0	7
1	3	2
5	4	3
0	4	9
0	6	3
7	8	3
8	9	8

0	0	7
0	4	9
0	6	3
1	3	2
5	4	3
7	8	3
8	9	8

The algorithm for Radix sort uses a modified version of counting sort to sort per digit. Since this part of the algorithm is too similar to the algorithm for counting sort shown above, it is not included here. It is available on the course's GitHub.

```
public void radixSort(int[] arr) {
        // Find the maximum number to know the number of digits
        int max = Arrays.stream(arr).max().getAsInt();

        // Do counting sort for every digit.
        // Instead of passing digit number, exp is passed.
```

```
        // exp is 10^i where i is current digit number
        for (int exp = 1; max / exp > 0; exp *= 10) {
            radixCountSort(arr, exp);
        }
    }
```

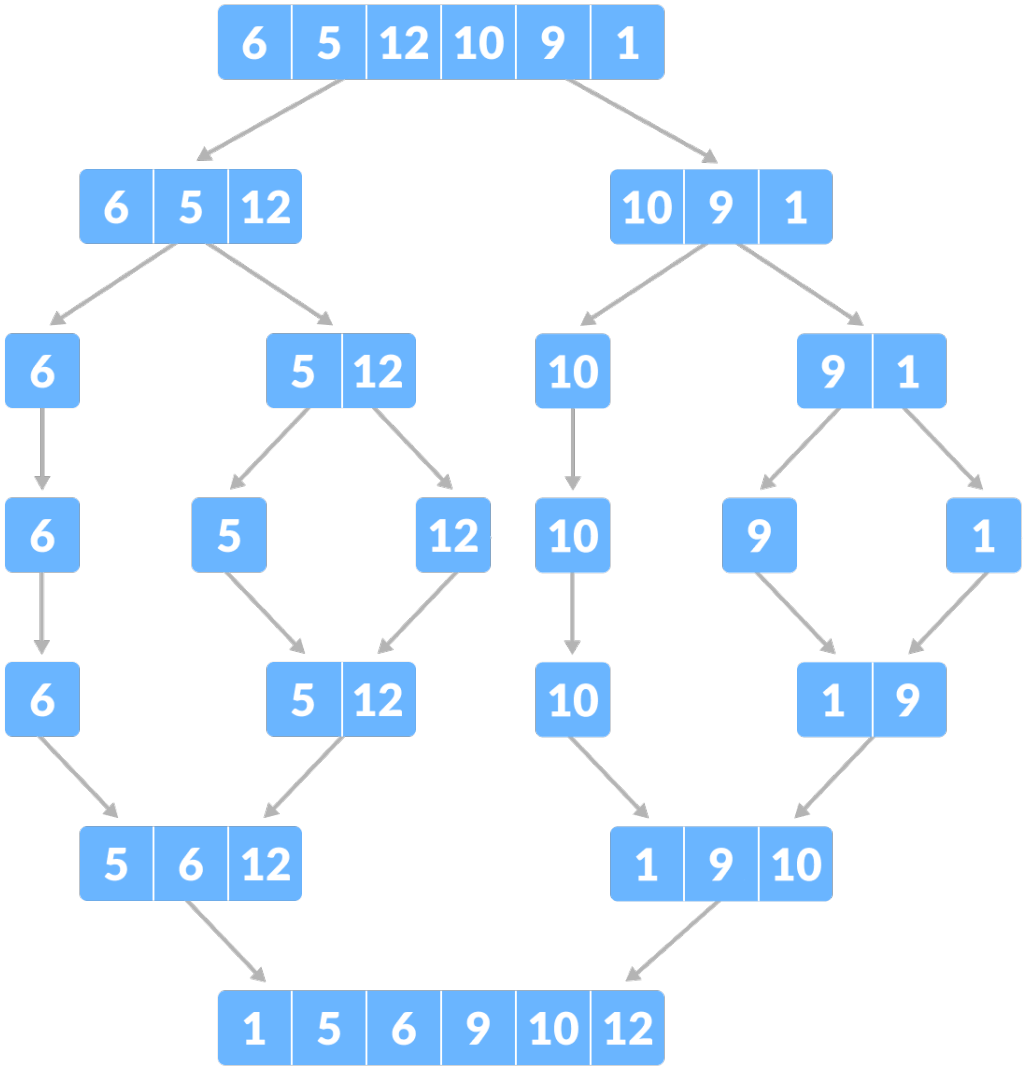
Linear algorithms are generally very fast, since their running time increases linearly with the size of the input data. However, they have some limitations. For example, counting sort and radix sort are only suitable for use with lists of integers or strings, and they may require a large amount of additional space to store the counting or radix arrays.

Linear algorithms are often used in practical applications where the input data is relatively small and the elements are integers or strings. They are also commonly used as intermediate steps in the sorting process for larger datasets.

O(n log n) Sorting Algorithms

Examples of ***O(n log n)*** sorting algorithms include:

- **Merge sort**: This algorithm works by dividing the input list into smaller and smaller sublists, until each sublist consists of a single element, and then merging the sublists back together in a sorted order. It is a stable sort, which means that the order of elements with equal values is preserved.




```
public static void mergesort(int[] arr) {

    mergesort(arr, 0, arr.length - 1);
}
```

```java
    public static void mergesort(int[] arr, int left, int right) {
        if (left >= right) {
            return;
        }
        int mid = (left + right) / 2;
        mergesort(arr, left, mid);
        mergesort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }

    public static void merge(int[] arr, int left, int mid, int right)
{
        int[] temp = new int[right - left + 1];
        int i = left;
        int j = mid + 1;
        int k = 0;
        while (i <= mid && j <= right) {
            if (arr[i] <= arr[j]) {
                temp[k] = arr[i];
                i++;
            } else {
                temp[k] = arr[j];
                j++;
            }
            k++;
        }
        while (i <= mid) {
            temp[k] = arr[i];
            i++;
            k++;
        }
        while (j <= right) {
            temp[k] = arr[j];
```

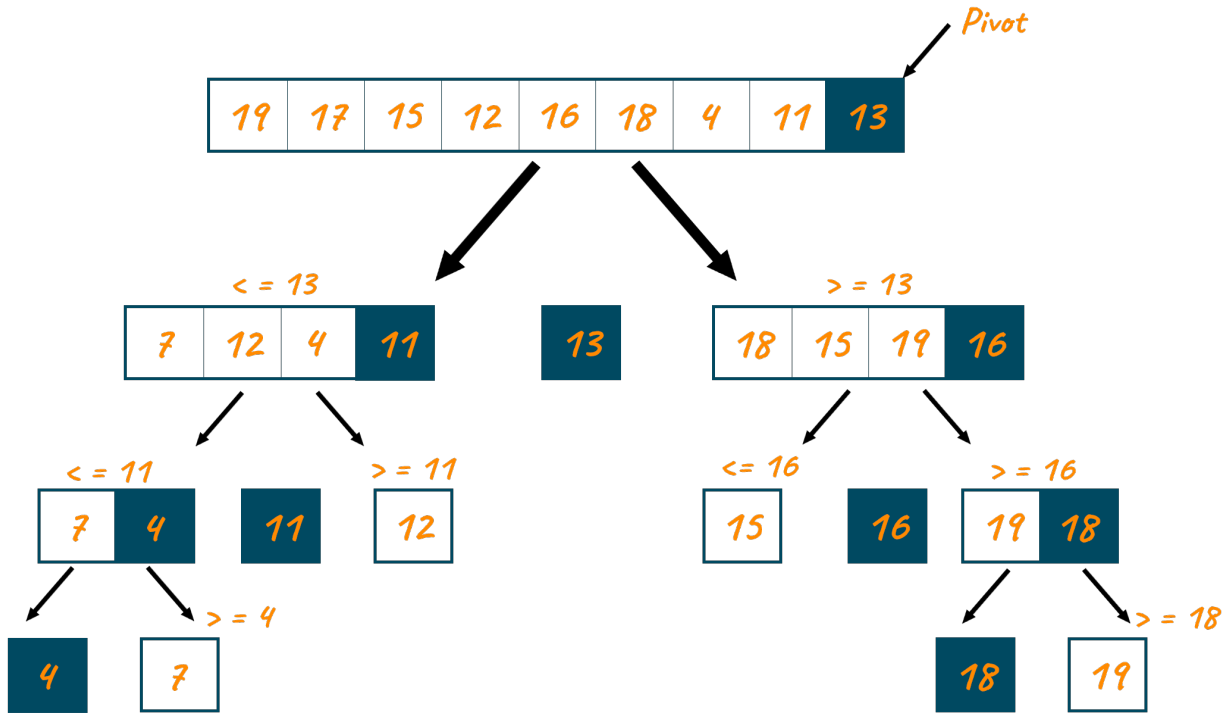
```
            j++;
            k++;
        }
        for (int m = left; m <= right; m++) {
            arr[m] = temp[m - left];
        }
    }
```

- The **mergesort()** method is the main entry point for the algorithm. It takes an array of integers as input and sorts it using the **mergesort()** method. If the input array is empty or has only one element, then it is already sorted and the method returns immediately. Otherwise, the method divides the input array into two subarrays (left and right) and recursively sorts each of them using the **mergesort()** method. Finally, the **merge()** method is called to merge the two sorted subarrays back together.
- The **mergesort()** method has two parameters: **left** and **right**. These represent the indices of the first and last elements of the subarray to be sorted. The **mid** variable is used to determine the index of the middle element in the subarray. The **mergesort()** method is then called recursively on the left and right subarrays.
- The **merge()** method takes three parameters: **left**, **mid**, and **right**. These represent the indices of the first and last elements of the left and right subarrays, respectively. The method creates a temporary array **temp** to hold the merged sublists, and uses two pointers (**i** and **j**) to traverse the left and right subarrays, respectively. If the element at index **i** in the left subarray is less than or equal to the element at index **j** in the right subarray, then the element at index **i** is added to the **temp** array and **i** is incremented. Otherwise, the element at index **j** is added to the **temp** array and **j** is incremented. This process continues until one of the subarrays is empty, at which point the remaining elements in the other subarray are added to the **temp** array. Finally, the sorted elements in the **temp** array are copied back into the original input array.

Complexity: Suppose we have an input array of length ***n*** that we want to sort using mergesort. In order to merge two sorted subarrays of length **x** and **y**, it takes a number of comparisons proportional to ***x + y***. Therefore, to merge two subarrays of length ***n/2***, it takes a number of comparisons proportional to n.
Now, let's consider the recursion tree for mergesort on an input array of length ***n*****.** The root of the tree corresponds to the original input array of length ***n***, and each node in the tree corresponds to a subarray that needs to be sorted. The leaves of the tree correspond to subarrays of length 1, which are already sorted.
At each level of the tree, we need to merge subarrays of total length n. Therefore, at each level, the total number of comparisons required is proportional to ***n***. The height of the tree is ***log n***, (although for odd inputs the tree will have one more level, but for our purposes this is not important) since each level halves the length of the subarrays. Therefore, the total number of comparisons required by mergesort is proportional to ***n log n***.

This shows that the worst-case complexity of mergesort is **O(n log n)**.

- **Quick sort**: This algorithm works by selecting a pivot element from the input list and partitioning the list into two sublists based on whether the elements are less than or greater than the pivot. It then recursively sorts the sublists, and finally merges the sorted sublists back together. Quick sort is generally faster (in terms of processing speed) than merge sort, but it is not a stable sort.




```java
public static void quicksort(int[] arr) {
    quicksort(arr, 0, arr.length - 1);
}

public static void quicksort(int[] arr, int left, int right) {
    if (left >= right) {
        return;
    }
    int pivot = arr[(left + right) / 2];
    int index = partition(arr, left, right, pivot);
    quicksort(arr, left, index - 1);
    quicksort(arr, index, right);
}
```

```
public static int partition(int[] arr, int left, int right, int
pivot) {
    int i = left;
    int j = right;
    while (i <= j) {
        while (arr[i] < pivot) {
            i++;
        }
        while (arr[j] > pivot) {
            j--;
        }
        if (i <= j) {
            swap(arr, i, j);
            i++;
            j--;
        }
    }
    return i;
}

public static void swap(int[] arr, int i, int j) {
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}
```

The pivot element is chosen as the element at the middle index of the input array. This is done in the quicksort() method, with the following line of code:

```
int pivot = arr[(left + right) / 2];
```

The **partition**() method is then called to partition the input array into two sub lists based on whether the elements are less than or greater than the pivot element. The **partition**() method does this by using two pointers, **i** and **j**, that start at the left and right ends of the input array, respectively. The pointers move towards the center of the array until they meet or cross each other. If the element at index **i** is less than the pivot, then **i** is incremented. If the element at

index **j** is greater than the pivot, then **j** is decremented. If the element at index **i** is greater than or equal to the pivot and the element at index **j** is less than or equal to the pivot, then the elements at indices **i** and **j** are swapped and the pointers are incremented and decremented, respectively.

The choice of pivot element can have a significant impact on the performance of the quick sort algorithm. A good pivot element is one that is close to the median value of the input data, as this will result in relatively balanced sublists. On the other hand, if the pivot element is either the smallest or largest element in the input array, then one of the sublists will be empty and the other will contain all of the elements. This will result in an inefficient sort, with a time complexity of O(n2).

In practice, there are a few strategies that can be used to choose a good pivot element:

- Choose the pivot element at random: By choosing a random pivot, the likelihood of encountering such a worst-case scenario is reduced, as the pivot is equally likely to divide the array into two sub-arrays of roughly equal size. One downside of choosing a random pivot is that it requires the generation of random numbers, which can be time-consuming. In practice, however, the overhead of generating random numbers is usually small compared to the overall time complexity of the algorithm.
- Choose the pivot element as the median of a small sample of elements: Instead of selecting the pivot element randomly, you can choose the pivot element as the median of a small sample of elements from the input array. This can be more efficient than selecting the pivot element randomly, as the pivot element is more likely to be close to the median value of the input data.
- Choose the pivot element as the median of the first, middle, and last elements of the input array: This is a simple and effective strategy that can be used to choose a good pivot element. To do this, you can compare the first, middle, and last elements of the input array and select the median value as the pivot element.
- Use a pivot selection algorithm: There are various pivot selection algorithms that can be used to choose a good pivot element. One such algorithm is the median-of-medians algorithm, which works by dividing the input array into small groups of elements and selecting the median of each group. The medians are then sorted and the median of the medians is chosen as the pivot element.

Complexity: The key idea is to consider the number of comparisons performed by the algorithm. In the worst case, the number of comparisons is proportional to the total number of elements in the input array.

Let's assume we have an input array of size **n**, and let's assume that the pivot is always chosen as the first element of the subarray. In the worst case, every recursive call will have to partition the array into two subarrays of size **n-1** and **0**. This means that the algorithm will make n-1 comparisons in each recursive call. Since there are n levels in the call tree (one for each element in the array), the total number of comparisons will be **(n-1)n**. This simplifies to $\mathbf{n^2 - n}$, which is $\mathbf{O(n^2)}$.

To see why this is the worst case, note that in the best case, the pivot will always divide the array into two roughly equal parts, and the algorithm will make ***log n*** recursive calls. Each recursive call will involve comparing each element in the subarray to the pivot, which gives a total of ***n log n*** comparisons. Therefore, the best-case complexity of quicksort is ***O(n log n)***.

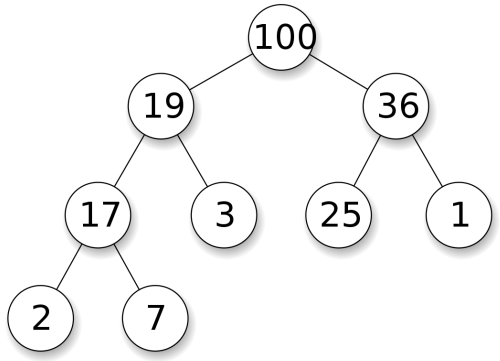
In practice, the worst case rarely occurs because pivots are typically chosen randomly or using a more sophisticated method, such as the median-of-three rule. These methods help to ensure that the pivot divides the array into reasonably balanced parts, which leads to better performance in practice.

- **Heap sort**: This algorithm works by first building a heap (a complete binary tree in which the value of each node is greater than or equal to the values of its children) from the input list, and then repeatedly extracting the root element (the maximum value) from the heap and placing it at the end of the sorted list. It is not a stable sort. More details follow in the next section.

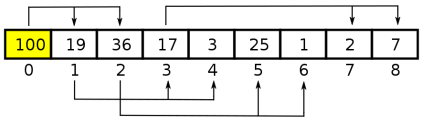
Heap Sort

The Heap Data Structure

Tree representation



Array representation



A **heap** is a specialized tree-based data structure that satisfies the heap property. The heap property states that for a ***max-heap***, the value of each node is greater than or equal to the values of its children, and for a ***min-heap***, the value of each node is less than or equal to the values of its children. A heap can be represented as an array, where the parent of a node at index i is at index ***(i-1)/2***, the left child is at index ***2i+1***, and the right child is at index ***2i+2***.

One common use of heaps is to implement a priority queue (PQ), which is an abstract data type that allows efficient access to the element with the highest priority. In a max-heap-based PQ, the element with the highest priority is the root of the heap, while in a min-heap-based PQ, the element with the lowest priority is the root of the heap.

A heap has several useful properties:

1. The height of a heap with n nodes is ***O(log n)***.
2. The maximum or minimum element of a heap can be found in constant time (i.e., ***O(1)***).
3. Inserting a new element or removing the maximum or minimum element from a heap takes **O(log n)** time in the worst case.
4. A heap is a complete binary tree, meaning that all the levels of the tree, except possibly the last one (the leaves), are completely filled and all the nodes are as far left as possible.
5. A heap satisfies the heap property, which states that the value of each node is greater than or equal to the values of its children. There are two types of heaps: ***max-heaps*** and ***min-heaps***. In a ***max-heap***, the value of each node is greater than or equal to the values of its children, and the root node has the largest value. In a min-heap, the value of each node is less than or equal to the values of its children, and the root node has the smallest value.
6. The root node of a heap is the topmost node in the tree. In a ***max-heap***, the root node has the largest value. In a ***min-heap***, the root node has the smallest value.

To store a priority queue in a heap, we can insert elements into the heap and then repeatedly extract the maximum (in the case of a max-heap-based PQ) or minimum (in the case of a min-heap-based PQ) element from the heap. When we extract the maximum or minimum element, we replace it with the last element in the heap and then perform a "**heapify-down**" operation to restore the heap property.

The **heapify-down** operation works as follows: we compare the new root node with its children and swap it with the larger (in the case of a max-heap) or smaller (in the case of a min-heap) child if necessary. We continue this process recursively until the heap property is restored. The heapify-down operation takes ***O(log n)*** time in the worst case, since we may need to swap the root node with its child all the way down to the bottom level of the heap.

Since the height of a binary tree with n nodes is at most ***log(n+1)***, a heap with n elements will have height ***log(n+1)***. This means that the time complexity of heap operations, such as insertion or removal, is ***O(log n)***, since the maximum number of exchanges required to maintain the heap property is proportional to the height of the heap, which is logarithmic in the number of elements.

Heapsort

Heap sort is a comparison-based sorting algorithm that works by building a heap data structure from the input list and extracting the elements of the heap in sorted order. It has a time complexity of ***O(n log n)*** in all cases. One of the main advantages of heap sort is that it is an in-place sorting algorithm, which means that it does not require any additional memory space to sort the input list.

```java
public static void heapsort(int[] arr) {
    int n = arr.length;
    for (int i = n / 2 - 1; i >= 0; i--) {
        heapify(arr, n, i);
    }
    for (int i = n - 1; i >= 0; i--) {
        swap(arr, 0, i);
        heapify(arr, i, 0);
    }
}

public static void heapify(int[] arr, int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < n && arr[left] > arr[largest]) {
        largest = left;
    }
    if (right < n && arr[right] > arr[largest]) {
        largest = right;
    }

    if (largest != i) {
        swap(arr, i, largest);
```

```
        heapify(arr, n, largest);
    }
}
```

Effectively, we can imagine that this algorithm inserts all elements into a priority queue and removing them one-by-one afterwards.

- **heapsort(int[] arr)**: This is the method that performs the heapsort algorithm. It takes an input array **arr** and sorts it in ascending order.
- **for (int i = n / 2 - 1; i >= 0; i--) heapify(arr, n, i);**: This builds a max heap out of the input array. It does this by iterating over all internal nodes of the binary tree representation of the heap in reverse order (from the bottom of the tree up to the root). At each node, the method **heapify()** is called to ensure that the node and its descendants satisfy the max heap property.
- **for (int i = n - 1; i >= 0; i--)**: This loop performs the sorting of the input array in ascending order. We iterate over each element of the array in reverse order.
- **heapify(arr, i, 0);**: We then call the **heapify()** method to maintain the max heap property on the remaining unsorted portion of the array (excluding the element that we just moved to its final sorted position).
- **heapify(int[] arr, int n, int i)**: This method performs the heapify operation on the input array. It takes three arguments: **arr** is the array to be heapified, **n** is the size of the heap, and **i** is the index of the node to perform the heapify operation on.
- **int largest = i;**: We set the index of the largest element to the index **i**, which is the index of the root of the current subtree.
- **int left = 2 * i + 1;**: We calculate the index of the left child of the current node.
- **int right = 2 * i + 2;**: We calculate the index of the right child of the current node.
- **if (left < n && arr[left] > arr[largest]) largest = left;**: We compare the value of the left child with the value of the current largest element. If the left child is greater, we set the index of the largest element to the index of the left child.
- **if (right < n && arr[right] > arr[largest]) largest = right;**: We do the same comparison for the right child.
- **if (largest != i) { int swap = arr[i]; arr[i] = arr[largest]; arr[largest] = swap; heapify(arr, n, largest); }**: If the largest element

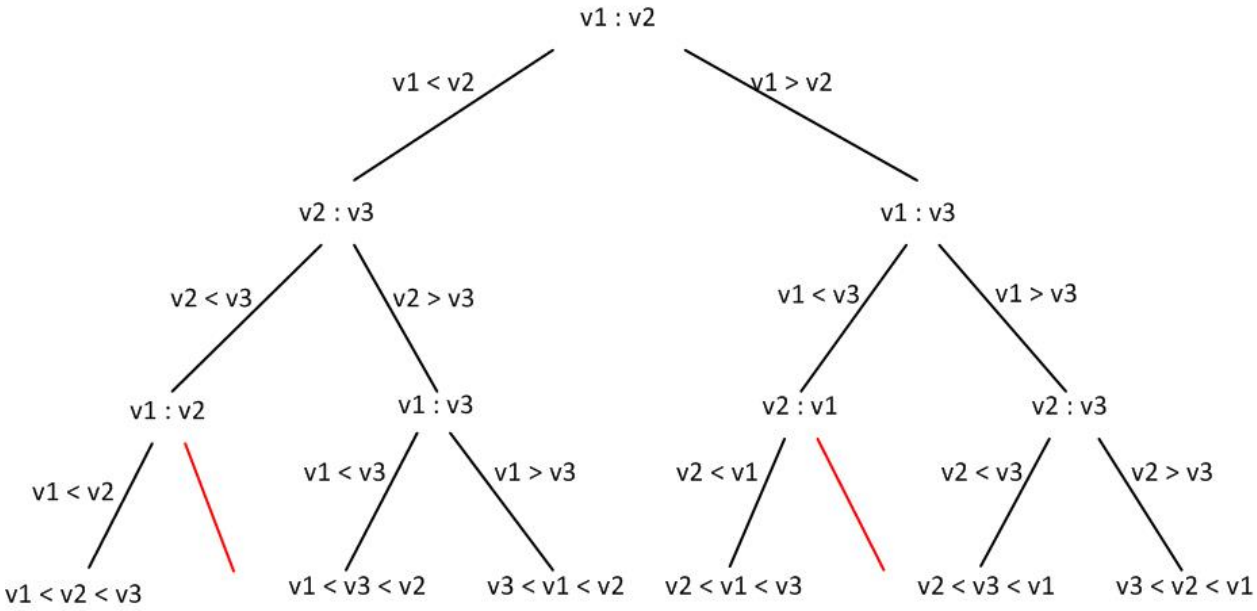
Lower-bound for Comparison-Based Sorting

Imagine you need to sort a deck of cards by rank, where the ranks are ace, 2, 3, ..., queen, king. You can only compare two cards at a time and determine which one has a higher rank.

In the worst case, you may need to compare every card in the deck with every other card to determine their relative order. For instance, if the deck is randomly shuffled, you might have to compare the ace of spades with every other card, then the 2 of spades with every other card, and so on, until you compare the king of spades with every other card. This requires n^2 comparisons, where n is the number of cards in the deck.

To sort any input, we need to consider the minimum number of comparisons required. As we can only compare two elements at a time, we can visualize this process in a decision tree where each leaf represents a unique permutation of the original data. At least one leaf in this tree corresponds to our desired output, which is the sorted list.

The fastest way to reach this leaf is by taking a direct path from the root to the leaf, without any backtracking. This means that the number of steps we need to take to reach our sorted destination is directly related to the height of the tree, assuming that we take a constant number of actions at each level.



So, how high is this decision tree? A list of ***N*** elements has ***N!*** possible permutations, and each leaf in our decision tree represents a different permutation of the original list. Since the tree covers every possible combination of comparisons that can be made, the number of leaves is ***N!***. Moreover, the decision tree is a balanced binary tree, meaning that the depth of the tree is the logarithm of the number of leaves of the tree. Therefore, the shortest path we can take in this tree requires ***log(N!)*** comparisons.

Note that:

log(N!) = log(1) + log(2) + ... + log(N)

If we keep only the last N/2 terms

log(N!) > log(N/2) + log(N/2 + 1) + ... + log(N)

Now replace all the terms by log(N/2) and the sum is even smaller:

log(N!) > log(N/2) + log(N/2) + ... + log(N/2) = 1/2 * Nlog(N/2)

Since ***log(N/2) = log(N) - 1***,

log(N!) > 1/2 * N log(N) = Ω(n log n)

The idea behind this proof is that ***n! = 1*2*3*...*n*** is a product of ***n*** numbers, each less than or equal to ***n***. Thus, it is less than ***n*** raised to the power of ***n***: n^n. Half of the numbers (i.e., ***n/2*** of them) in the ***n!*** product are greater than or equal to ***n/2***, so their product is greater than the product of ***n/2*** numbers, all equal to ***n/2***.

So: log(1) + log(2) + ... + log(n) <= log(n) + log(n) + ... + log(n)

= n*log(n)

This leads to the lower bound of Ω(n log n) for the number of comparisons needed to sort any input.