Sorting Algorithms

Introduction to Sorting

- Goal of sorting: rearrange N items such that their keys are in ascending (increasing) order.
- Sorting algorithms are based on the following operations:
  - **Compare**: compare two items and determine their relative order.
  - **Exchange**: swap two items.
- The preimitive operation in sorting is the **comparison**.
- Comparing items: the Java `Comparable` interface:
  - Sorting requires that 2 items can be compared.
  - Java defines the `Comparable<T>` interface to represent this requirement:

    ```
    public Interface Comparable<T> {
        public int  compareTo(T o);
    }
    ```

  - Class that implements `Comparable<T>` are comparable classes, and they must implement the `compareTo()` method.
  - Requirement of the `compareTo()` method:
    - `a.compareTo(b)` returns a negative value if `a` is less than `b`.
    - `a.compareTo(b)` returns zero if `a` is equal to `b`.
    - `a.compareTo(b)` returns a positive value if `a` is greater than `b`.
- Exchanging two items (objects) in an array:

  ```
  public static void exch(DataType[] a, int i, int j) {
      DataType help;
      help = a[i];
      a[i] = a[j];
      a[j] = help;
  }
  ```

  - The `exch()` method will only work on objects of the type `DataType`.
  - We can define a generic (parameterized) `exch()` so that it will only work on subclasses of `Comparable<T>`:

```java
    public static <T extends Comparable<T>> void exch(T[] a, int i, int j) {
        T help;

        help = a[i];
        a[i] = a[j];
        a[j] = help;
    }
```

Selection Sort

```java
public static <T extends Comparable<T>> void selectionSort(T[] list) {
    for (int i = 0; i < list.length-1; i++) {
        // Find the minimum in the list[i..list.length-1]
        T min = list[i]; // Assume first element is min
        int minIndex = i; // index where min is found

        for (int k = minIndex+1; k < list.length; k++) {
            if (list[k].compareTo(min) < 0) { // compare list[k] and min
                min = list[k]; // update min value
                minIndex = k; // update min index
            }
        }

        // Swap list[i] with list[minIndex] if necessary
        if (minIndex != i) {
            exch(list, i, minIndex);
        }
    }
}
```

- Runtime analysis:
    - The algorithm can be simplified as follows:

```java
for (int i = 0; i < n-1; i++) {
    for (int k = i + 1; k < n; k++) {
        doPrimitive();
    }
}
```

i=	**0**	**1**	**2**	**...**	**n-3**	**n-2**
k=	1	2	3	...	n-2	n-1
k=	2	3	4	...	n-1	
...	...	...	...			
k=	n-2	n-1				
k=	n-1					

    - So, number of iterations is $(n-1)+(n-2)+(n-3)+\cdots+2+1$, which is the triangular sum.
    - Then, we know number of iterations=$\frac{n(n-1)}{2}$.
    - This indicates that the runtime is $\mathcal{O}(n^2)$.
- Additional properties of sorting algorithms:
  - **In-place**: a sorting algorithm is in-place if it does not require another array to execute the algorithm.
  - **Stable**: a sorting algorithm is stable if it preserves the relative order of equal keys in the array.
- ***Selection sort is in-place but not stable.***

Bubble Sort

- The bubble sort algorithm will:
  - Compare every pair of adjacent element
  - Exchange(=swap) them if they are out of order.
- Example:

2	**9**	**5**	**4**	**8**	**1**	**Orignal array**
2	**9**	5	4	8	1	2 < 9 --> OK, no swap
2	**9**	**5**	4	8	1	9 > 5 --> swap
2	5	**9**	**4**	8	1	9 > 4 --> swap
2	5	4	**9**	**8**	1	9 > 8 --> swap
2	5	4	8	**9**	**1**	9 > 1 --> swap

2	**9**	**5**	**4**	**8**	**1**	**Orignal array**
2	5	4	8	1	**9**	Finish first iteration

- Note that after 1 iteration, the largest element is at the end of the array (the correct position). Therefore, if we repeat $n-1$ times, the array will be sorted.

```
public static <T extends Comparable<T>> void bubbleSort(T[] arr) {
    int n = arr.length;

    for (int i = 0; i < n-1; i++) { // repeat n-1 times
        // compare every adjacent pair of elements
        for (int j = 0; j < n-1-i; j++) {
            if (arr[j].compareTo(arr[j+1]) > 0) { // if out of order
                exch(arr, j, j+1); // swap
            }
        }
    }
}
```

- Runtime analysis:
    - The loop structure of bubble sort can be simplified as

```
for (int i = 0; i < n-1; i++) {
    for (int j = 0; j < n-1-i; j++) {
        doPrimitive();
    }
}
```

i=	**0**	**1**	**2**	**...**	**n-3**	**n-2**
j=	0	0	0	...	0	0
j=	1	1	1	...	1	
...	...	...	...			
j=	n-4	n-4	n-4			
j=	n-3	n-3				
j=	n-2					

        - So, number of iterations is $(n-1)+(n-2)+(n-3)+\cdots+2+1$, which is the triangular sum.
        - Then, we know number of iterations=$\frac{n(n-1)}{2}$.
        - This indicates that the runtime is $\mathcal{O}(n^2)$.
- In fact, we can improve the basic bubble sort algorithms.
- The idea is that if we find that the array is already sorted, we can stop the algorithm.
    - Facts:
        - In each iteration, the bubble sort algorithm will compare every pair of elements and swap only out-of-place pairs.
        - When the array is sorted: an (entire) iteration in bubble sort will not perform any exchange operation.
    - So, we can add a variable to track the number of swaps in each iteration. If we find that the number of swaps is zero, we can stop the algorithm.

```java
public static <T extends Comparable<T>> void bubbleSort(T[] arr) {
    int n = arr.length;
    boolean swapped;

    for (int i = 0; i < n-1; i++) { // repeat n-1 times
        swapped = false; // reset swapped to false
        // compare every adjacent pair of elements
        for (int j = 0; j < n-1-i; j++) {
            if (arr[j].compareTo(arr[j+1]) > 0) { // out of order
                exch(arr, j, j+1); // swap
                swapped = true; // set swapped to true
            }
        }

        if (swap == false) { // if no swap in this iteration
            break; // stop the algorithm
        }
    }
}
```

- ***The bubble sort algorithm is an in-place and stable sorting algorithm.***

Insertion Sort

- The insertion sort algorithm will:

    - Selections the next unsorted element/key in the array
    - Insert(=exchange) it towards the front into its correct position
- Example:

2	**9**	**5**	**4**	**8**	**1**	**Original array**
2	9	5	4	8	1	2 is in the correct position
2	**9**	5	4	8	1	9 is in the correct position
2	9	**5**	4	8	1	5 is not in the correct position, swap
2	5	9	**4**	8	1	5 is not in the correct position, swap
2	4	5	9	**8**	1	8 is not in the correct position, swap
2	4	5	8	9	**1**	9 is not in the correct position, swap
1	2	4	5	8	9	Finished!

```java
public static <T extends Comparable<T>> void insertionSort(T[] arr) {
    int n = arr.length;

    for (int i = 1; i < n; i++) { // Repeat n-1 times
        // Compare adjacent pairs starting at (i-1, i)
        for (int j = i; j < 0; j--) {
            if (arr[j-1].compareTo(arr[j]) > 0) { // out of order
                exch(arr, j-1, j); // swap
            } else {
                break; // stop the inner loop
            }
        }
    }
}
```

- Runtime analysis:
    - The loop structure of the insertion sort algorithm can be simplified as:

```java
for (int i = 1; i < n; i++) {
    for (int j = i; j > 0; j--) {
        doPrimitive();
    }
}
```

i=	1	2	3	...	n-2	n-1
j=	1	2	3	...	n-2	n-1
j=		1	2	...	n-3	n-2
...	...	...	...			
j=						1

    - So, number of iterations is $1+2+3+\cdots+(n-2)+(n-1)$, which is the triangular sum.
    - Then, we know number of iterations=$\frac{n(n-1)}{2}$.
    - This indicates that the runtime is $\mathcal{O}(n^2)$.
- ***The selection sort algorithm is an in-place and stable sorting algorithm.***

Merge Sort

Merge Algorithm

- We are given two sorted array portions that are adjacent to each other:

```
Input:
A[]:    ...... A[s] ... A[m-1] A[m] ... A[e-1] ...
               <-------------> <------------->
                    sorted          sorted
```

- We will design an efficient algorithm to merge the 2 sorted array portions. The result will then occupy the same portion in the array.
- Parameters of the merge algorithm:
    - They merge algorithm has 3(index) parameters: `s` , `m` , and `e` .
    - `s` : the starting index of the first sorted array portion.
    - `m` : the ending index of the first sorted array portion.
    - `e` : the ending index of the second sorted array portion.
- The merge algorithm will:
    - only merge the array elements inside `A[s] ... A[e-1]`
    - not affect the array elements outside the range.
- Variables used in the merge algorithm:
    - We use 2 indices `i` and `j` to point to the current elements in each sorted array portion:

    - The element `A[i]` is always the smallest value in the left (sorted) portion.
    - The element `A[j]` is always the smallest value in the right (sorted) portion.
  - We also use a helper array variable `H[]` array to perform the merge operation.
    - We will repeatedly copy the smallest value from both arrays to `H[]` .
    - If `A[j] < A[i]` , we will copy `A[j]` to `H[]` and increment `j` .
    - If `A[i] < A[j]` , we will copy `A[i]` to `H[]` and increment `i` .
    - When `i == m` , we know the left portion is exhausted, so we can copy the remaining elements in the right portion to `H[]` .
    - When `i == m` and `j == e` , both portions are exhausted, and `H[]` now contains the merged result. We will then copy `H[]` back to `A[]` .

```
public static <T extends Comparable<T>> void merge(T[] A, int s, int m, int e, T[] H) {
    // The merge() method will be invoked repeatedly.
    // If we create a helper array inside merge(), we would repeatedly allocated and de
    // That is inefficient.
    // So, we create a helper array outside merge() and pass it as a parameter.
    int i = s, j = m; // current elements in left and right portions
    int k = 0; // current element in helper array

    while (i < m || j < e) { // loop as long as there are unprocessed items
        if (i < m && j < e) {
            // Case 1: both portions have unprocessed elements
            if (A[i].compareTo(A[j]) < 0) {
                H[k++] = A[i++];
            } else {
                H[k++] = A[j++];
            }
        } else if (i == m) {
            // Case 2: the left portion is exhausted
            H[k++] == A[j++];
        } else if (j == e) {
            // Case e: the right portion is exhausted
            H[k++] = A[i++];
        }
    }

    // Copy H[] back to A[]
    for (i = s, k = 0; i < e; i++, k++) {
        A[i] = H[k];
    }
}
```

- Run time analysis:
  - We can simplify the while-loop into:

    ```
    while (i < m || j < e) {
        doPrimitive();
        // Then do either:
        i++; // --> i = s; i < m; i++ = m-s times
        // Or do:
        j++; // --> j = m; j < e; j++ = e-m times
    }
    // Copy
    for (int i = s, k = 0; i < e; i++, k++) {
        // i = s; i < e; i++ = e-s times
        doPrimitive();
    }
    ```

    - The while loop will be executed $(m-s)+(e-m)=e-m=n$ times.
    - The for loop will be executed $e-s=n$ times.
    - In total, we have $2n$ primitive operations.
    - The running time of the `merge()` algorithm is $2n$, which is $\mathcal{O}(n)$.

Merge Sort Algorithm

- The Merge sort algorithm will:
  - Split the array `a[]` into 2 halves:
    - an array `left[]` containing `a[0] ... a[n/2]`
    - an array `right[]` containing `a[n/2+1] ... a[n-1]`
  - Sort both halves of the arrays (by repeating the above step)
  - Merge the sorted arrays into the final sorted array
- The recursive structure (illustration using `selectinoSort`):

```java
public static <T extends Comparable<T>> void sort(T[] a, int s, int e, T[] H) {
    if (e-s <= 1) { // A[s]..A[e] has 0 or 1 elements
        return; // nothing to sort
    }
    int m = (s+e)/2; // m is the middle index

    /*
     * Sort the left portion A[s]..A[m-1]
     * Sort the right portion A[m]..A[e-1]
     * Merge the sorted portions
     */
    selectionSort(A, s, m);
    selectionSort(A, m, e);

    // Merge both sorted arrays
    merge(A, s, m, e, H);
}
```

- Now, the real merge sort algorithm is simply to replace `selectionSort()` with `mergeSort()` .

```java
public static <T extends Comparable<T>> void mergeSort(T[] a, int s, int e, T[] H) {
    if (e-s <= 1) { // A[s]..A[e] has 0 or 1 elements
        return; // nothing to sort
    }
    int m = (s+e)/2; // m is the middle index

    /*
     * Sort the left portion A[s]..A[m-1]
     * Sort the right portion A[m]..A[e-1]
     * Merge the sorted portions
     */
    mergeSort(A, s, m, H);
    mergeSort(A, m, e, H);

    // Merge both sorted arrays
    merge(A, s, m, e, H);
}
```

- ***The merge sort algorithm is not an in-place sorting algorithm, but it is a stable sorting algorithm.***
- Runtime analysis:

- If we want to sort n elements, we can rewrite the algorithm into

  ```
  public static <T extends Comparable<T>> void mergeSort(T[] a, int n, T[] H) {
      if (n <= 1) {
          return;
      }
      mergeSort(a, n/2, H); // running time = T(n/2)
      mergeSort(a, n/2, H); // running time = T(n/2)
      // Final merge
      merge(a, 0, n/2, n, H); // running time = 2n
  }
  ```

  - So, $T(n) = 0$ for $n \leq 1$ and $T(n) = 2T\left(\frac{n}{2}\right) + 2n$ for $n \geq 2$.
  - Now, let's solving the following relationship:

$$
\begin{aligned}
T(1) &= 0 \\
T(n) &= 2T\left(\frac{n}{2}\right) + 2n \\
&= 2\left[2T\left(\frac{n}{4}\right) + 2\left(\frac{n}{2}\right)\right] + 2n \\
&= 2^2T\left(\frac{n}{2^2}\right) + 2(2n) \\
&= 2^2\left[2T\left(\frac{n}{2^3}\right) + 2\left(\frac{n}{2^2}\right)\right] + 2(2n) \\
&= 2^3T\left(\frac{n}{2^3}\right) + 3(2n) \\
&\vdots \\
&= 2^kT\left(\frac{n}{2^k}\right) + k(2n) \\
\text{Set } \frac{n}{2^k} = 1 &\implies 2^k = n \implies k = \log(n) \\
T(n) &= 2^kT\left(\frac{n}{2^k}\right) + k(2n) \\
&= 2^kT(1) + 2kn \\
&= 0 + 2kn = 2n\log(n)
\end{aligned}
$$

  - So, the running time is $T(n) = 2n\log(n)$, which is $\mathcal{O}(n\log(n))$.

- Merge sort is one of few algorithms that has a runtime of $\mathcal{O}(n\log(n))$.
  - $\mathcal{O}(n\log(n))$ is the theoretical lower bound on running time of sorting algorithms.
- Some problems with pure merge sort algorithm:
  - There is a very high overhead to executive a recursive algorithm for tiny sublists.
  - Merge sort will merge even when the 2 halves are sorted.

    - Merge sort is not an in-place algorithm.
        - Merge sort requires an additional array to perform the merge operation.
- To solve the first problem, we can have a cutoff value for the size of the sublist. If the size of the sublist is smaller than the cutoff value, we will use selection sort instead of merge sort. This hybrid algorithm is called **Tim Sort**.

```java
public static <T extends Comparable<T>> void mergeSort(T[] A, int s, int e, T[] H) {
    if (e - s <= CUTOFF) {
        selectionSort(A, s, e);
        return;
    }
    int m = (s + e) / 2;
    mergeSort(A, s, m, H);
    mergeSort(A, m, e, H);
    // Merge both sorted arrays
    merge(A, s, m, e, H);
}
```

- To solve the second problem, we can simply add a comparison of the last element in the left portion and the first element in the right portion. If the last element in the left portion is smaller than the first element in the right portion, we can skip the merge operation.

```java
public static <T extends Comparable<T>> void mergeSort(T[] A, int s, int e, T[] H) {
    if (e - s <= CUTOFF) {
        selectionSort(A, s, e);
        return;
    }
    int m = (s + e) / 2;
    mergeSort(A, s, m, H);
    mergeSort(A, m, e, H);
    // Merge both sorted arrays only when values over lap
    if (A[m-1].compareTo(A[m]) < 0) {
        return;
    }
    merge(A, s, m, e, H);
}
```

Non-recursive Merge Sort

```
public static <T extends Comparable<T>> void mergeSort(T[] A, T[] H) {
    int mergeSize = 1;

    while (mergeSize < A.length) {
        int s, m, e;
        for (s = 0; s < A.length; s += 2*mergeSize) {
            m = s + mergeSize;
            e = Math.min(s + 2*mergeSize, A.length);
            merge(A, s, m, e, H);
        }
        mergeSize *= 2;
    }
}
```

```
A[]: | mergeSize | mergeSize | mergeSize | mergeSize | ...
     s           m          e/s          m           e
```

Quick Sort

- Introduction to Quick Sort
  - Quick sort is the most commonly used sorting algorithm.
  - Quick sort was invented by Antony Hoare in 1959 and was honored as one of the top 10 algorithms in the 20th centry.
  - Some facts on quick sort:
    - It is a divide-and-conquer algorithm. (just like merge sort)
  - Unlike merge sort, quick sort is an **in-place** algorithm. So it does not require an extra array.
- Big idea of quick sort:
  - We select a pivot element in the array.
  - We partition the array into 2 portions:
    - The left portion contains elements that are smaller than the pivot.
    - The right portion contains elements that are larger than the pivot.
  - Insert the pivot to the correct position.

```
public static <T extends Comparable<T>> void quickSort(T[] A, int s, int e) {
    if (e - s <= 1) { // Base case
        return;
    }
    // partition sub array A[s]..A[e-1] using A[s] as pivot
    int pivotLoc = partition(A, s, e);
    // partiion(A, s, e) returns the index (=location) of the pivot element.
    // it is the border of the 2 sub array/groups
    quickSort(A, s, pivotLoc); // sort left portion
    quickSort(A, pivotLoc+1, e); // sort right portion
}
```

The Partition Algorithm for Quick Sort

- A simplified partition algorithm:
    - A partitioning algorithm maintains 2 half arrays: a low half and a high half.
    - The simplified partitioning algorithm only compares the pivot against `A[high]` .

```
           low half                    high half
A[] = [P L L L L L L ? ? ? ? ? ? ? X H H H H H H]
       ^ <---------> ^             ^ <--------->
       |     <P      |             |     >= P
     pivot          low           high
       |                           |
       +---------------------------+
        compares: pivot <==> A[high]
```

- If `A[high] (X) >= pivot` , then `A[high]` belongs to the high partition, and we decrement `high`

```
           low half                    high half
A[] = [P L L L L L L ? ? ? ? ? ? ? X H H H H H H]
       ^ <---------> ^           ^ <----------->
       |     <P      |           |     >= P
     pivot          low         high
       |                         |
       +-------------------------+
        compares: pivot <==> A[high]
```

- If `A[high] (X) < pivot` , then `A[high]` belongs to the low partition: we exchange `X` and `Y` and then increment `low` .

```
                low half                           high half
        A[] = [P L L L L L L Y ? ? ? ? ? ? X H H H H H H]
                 ^ <---------> ^              ^ <---------->
                 |     <P      |              |     >= P
               pivot          low            high

                low half                           high half
        A[] = [P L L L L L L X ? ? ? ? ? ? Y H H H H H H]
                 ^ <-----------> ^            ^ <---------->
                 |     <P        |            |     >= P
               pivot            low          high
```

- We repeat the steps until we get `low > high`. Then, we will put the pivot in its correct location by exchanging `pivot` and `A[high]`.

```java
public static <T extends Comparable<T>> int partition(T[] A, int s, int e) {
    T pivot = A[s];
    int low = s, high = e - 1;

    while (low <= hight) {
        if (A[high].compareTo(pivot) >= 0) {
            high--;
        } else {
            exch(A, low, high);
            low++;
        }
    }
    exch(A, s, high); // A[s] = pivot
    return high;
}
```

- A rather complicated partitioning algorithm:

```java
public static int partition(int[] list, int first, int last) {
    int pivot = list[first]; // choose the first element as the pivot
    int low = first + 1; // index for forward search
    int high = last - 1; // index for backward search

    while (low < high) {
        // Search forward from left
        while (low <= high && list[low] <= pivot) {
            low++;
        }
        // Search backward from right
        while (low <= high && list[high] > pivot) {
            high--;
        }
        // Swap two elements in the list
        if (low < high) {
            int temp = list[high];
            list[high] = list[low];
            list[low] = temp;
        }
    }
    // Adjust high to find the border
    while (high > first && list[high] >= pivot) {
        high--;
    }
    // Swap pivot with list[high]
    if (pivot > list[high]) {
        list[first] = list[high];
        list[high] = pivot;
        return high;
    } else {
        return first; // pivot was the smallest element
    }
}
```

- ***The quick sort algorithm is an in-place algorithm, but it is not stable.***

Runtime Analysis

- Synopsis of the `partition()` algorithm

```java
public statbic <T extends Comparable<T>> int partition(T[] A, int s, int e) {
    T pivot = A[s];
    int low = s+1; high = e-1;
    while (low <= high) {
        doPrimitive(); // either high--; or low++
        // --> loop will run (e-1) - (s+1) times = e-1-s-1=e-s-2 times
    }
    doPrimitive(); // exchange pivot and A[high]
    return hight;
}
```

  - In this case, the running time will be $n-1 \approx n$. Or, $\mathcal{O}(n)$.
- The running time of the quick sort algorithm depends on how `partition()` splits the input array.
  - IMPORTANT: the running time of quick sort depends on how large the 2 array halves are.
  - Suppose the `partition(A, s, e)` algorithm partitions the input as follows:

```
         <-------------- n --------------->
INPUT:   [P .. .. .. .. .. .. .. .. .. .. ..]
RESULT:  [.. .. .. .. .. .. P .. .. .. .. ..]
         <------ k ------>   <--- n-k-1 -->
```

    - The running time $T(n)$ for input size n will have the following recurrence relationship: $T(n) = T(k) + T(n-k-1) + n$.
- The best case running time of quick sort.
  - The best case running time of the quick sort algorithm happens when: the `partition()` algorithm always divides the input array into 2 equal halves.
  - In this case, we have

$$
\begin{aligned}
T(n) &= T(k) + T(n-k-1) + n \\
&= T(n/2) + T(n/2-1) + n \\
&\approx 2T(n/2) + n,
\end{aligned}
$$

    which is similar recurrence relation as the one in merge sort.
    - So, we can solve it with telescoping and the result is $T(n) = n\log(n)$.
- The worst case running time of quick sort:
  - The worst case running time of the quick sort algorithm happens when the `partition()` always divides the input array into (1) array containing the pivot and (2) an array with $n-1$ elements.

- In this case, we have

$$\begin{aligned}
T(n) &= T(0) + T(n-1) + n \\
&= 0 + T(n-1) + n \\
&= T(n-1) + n \\
&= T(n-2) + (n-1) + n \\
&= T(n-3) + (n-2) + (n-1) + n \\
&\vdots \\
&= T(0) + 1 + 2 + \cdots + (n-2) + (n-1) + n \\
&= 1 + 2 + \cdots + (n-2) + (n-1) + n && \text{triangular sum} \\
&= \frac{n(n+1)}{2} - 1 \\
&= \mathcal{O}(n^2)
\end{aligned}$$

- So, the worse case running time is $\mathcal{O}(n^2)$.
- An example that makes quick sort achieve the worse case running time: a sorted array.
  - Because the pivot is always the smallest value, `partition()` will always produce:
    - An array containing the pivot, and
    - An array containing the other $n-1$ elements.
- To prevent the worse case, a commonly used practice when using quick sort is to:
  - Shuffle the input array randomly (To ensure `partition()` will not always find the pivot as the first element)
  - Use quick sort of the shuffled input array.
  - Then, the quick sort will achieve the **average** running time performance with a randomized input array.

- The average running time of quick sort:
  - Recall the recurrence relation for the running time of quick sort is $T(n) = T(k) + T(n-k-1) + n$, where k is the final position of the pivot.
  - Depending on the value of k, the recurrence relation for $T(n)$ are different:

$$\begin{gathered}
T(n) = T(0) + T(n-1) + n \quad \text{if } k = 0 \\
T(n) = T(1) + T(n-2) + n \quad \text{if } k = 1 \\
\cdots \\
T(n) = T(n-1) + T(0) + n \quad \text{if } k = n-1
\end{gathered}$$

  - Since each value of k is equally likely to occur, we can take the average by summing them and dividing the sum by n.

$$\begin{aligned} nT(n) &= (T(0)+\cdots+T(n-1)) + (T(n-1)+\cdots+T(0)) + n^2 \\ nT(n) &= 2T(0)+2T(1)+\cdots+2T(n-1)+n^2 \\ T(n) &= \frac{2}{n}T(0)+\frac{2}{n}T(1)+\cdots+\frac{2}{n}T(n-1)+n \end{aligned}$$

- Now, we want to solve the following recurrence relation:

$$\begin{aligned} T(1) &= 1 \\ T(n) &= \frac{2}{n}T(0)+\frac{2}{n}T(1)+\cdots+\frac{2}{n}T(n-1)+n \\ nT(n) &= 2T(0)+2T(1)+\cdots+2T(n-1)+n^2 \\ (n-1)T(n-1) &= 2T(0)+2T(1)+\cdots+2T(n-2)+(n-1)^2 \\ nT(n)-(n-1)T(n-1) &= 2T(n-1)+n^2-(n-1)^2 \\ &= 2T(n-1)+2n-1 \\ nT(n) &= (n+1)T(n-1)+2n-1 \\ T(n) &= \frac{n+1}{n}T(n-1)+2-\frac{1}{n} \\ \frac{T(n)}{(n+1)} &= \frac{T(n-1)}{n}+\frac{2}{(n+1)}-\frac{1}{n(n+1)} \\ \frac{T(n-1)}{n} &= \frac{T(n-2)}{(n-1)}+\frac{2}{n}-\frac{1}{(n-1)n} \\ \frac{T(n-2)}{(n-1)} &= \frac{T(n-3)}{(n-2)}+\frac{2}{(n-1)}-\frac{1}{(n-2)(n-1)} \\ &\vdots \\ \frac{T(2)}{3} &= \frac{T(1)}{2}+\frac{2}{3}-\frac{1}{2\cdot 3} \\ \frac{T(n)}{(n+1)} &+ \frac{T(n-1)}{n}+\frac{T(n-2)}{(n-1)}+\cdots+\frac{T(2)}{3} \\ &= \frac{T(n-1)}{n}+\frac{T(n-2)}{(n-1)}+\cdots+\frac{T(2)}{3}+\frac{T(1)}{2} \\ &\quad +\frac{2}{(n+1)}+\frac{2}{n}+\frac{2}{(n-1)}+\cdots+\frac{2}{3} \\ &\quad -\frac{1}{n(n+1)}-\frac{1}{(n-1)n}-\cdots-\frac{1}{3\cdot 2} \\ \frac{T(n)}{(n+1)} &\approx 2\left(\frac{1}{(n+1)}+\frac{1}{n}+\frac{1}{(n-1)}+\cdots+\frac{1}{3}+\frac{1}{2}+1\right) \\ &= 2\log(n+1) \end{aligned}$$

    - Therefore, $T(n) = 2(n+1)\log(n+1)$, or $\mathcal{O}(n\log(n))$.

Improvements to Quick Sort

- As mentioned above, one way to improve the quick sort algorithm is to shuffle the input array randomly.

  ```
  public static void main(String[] args) {
      Integer[] A = ...; // Array to be sorted with quick sort

      // shuffle array A
      for (int k = 0; k < N; k++) {
          int i = Math.random() * A.length;
          int j = Math.random() * A.length;
          exch(A, i, j);
      }
      quickSort(A, 0, A.length);
  }
  ```

  - Why shuffle?
    - Make quick sort avoid picking the smallest pivot all the time.
    - Help quick sort achieve the average running time performance with a randomized array.
- Another way to improve the quick sort algorithm is the increase the likelihood to partition the array into 2 equal halves.
  - The best pivot value is the median of the input array.
  - We can improve the likelihood of picking the median by considering 3 (random value) selected from the input array. (Instead of always using the first element as pivot).
  - The median of 3 partition method:

    ```
    A[] = [X .. .. .. .. .. Y .. .. .. .. .. Z]
    Let M = median(X, Y, Z)
        (1) swap M and X
        (2) partition(A, s, e)
    ```

  - Algorithm to find the median of 3 values:

```
public static int medianOf3(int a, int b, int c) {
    if ((b <= a && a <= c) || (c <= a && a<= b)) {
        return a;
    }
    if ((a<= b && b <= c) || (c <= b && b <= a)) {
        return b;
    }
    return c;
}
```

- An improved (faster) algorithm: using the exclusive OR (^) operator.

```
public static int medianOf3(int a, int b, int c) {
    if ((a > b) ^ (a > c)) { // either a > b or a > c
        return a;
    }
    if ((b > a) ^ (b > c)) { // either b > a or b > c
        return b;
    }
    return c;
}
```

Faster Computer VS Better algorithm.

- Consider:
  - A sorting algorithm (e.g. merge sort) with $\mathcal{O}(n\log(n))$ running time.
  - A sorting algorithm (e.g. insertion sort) with $\mathcal{O}(n^2)$ running time running on a faster computer (e.g. 1,000,000 times faster).
- Which one is better?
  - The answer is: it depends.
  - If the input size is small, the faster computer will be better.
  - If the input size is large, the better algorithm will be better.
  - ***A better algorithm will always beat an average algorithm when the input size is sufficiently large.***

Sorting Algorithms in Java's Library

- The `sort()` method (an overloaded method) can be found in:
  - `java.util.Arrays` to sort arrays:

        - `sort(int[] a)` : the sorting algorithm is a Dual-Pivot Quick Sort.
        - `sort(Object[] a)` : This implementation is a stable, adaptive, iterative Merge Sort.
    - `java.util.Collections` to sort collections (e.g. `ArrayList`):
        - According to information on the Internet, they used Merge Sort in Collections.
- Requirement to use `sort()` methods in Java's library:
    - The objects that are sorted must implement the `Comparable<T>` interface.