Quick Sort

**Sorting Quickly**

```
public class SortQuickly {

Θ(1)  public static void swap(String[] array, int i1, int i2) {
        String temp = array[i1];
        array[i1] = array[i2];
        array[i2] = temp;
      }
                                                  N        0        N-1
      public static int partition(String[] array, int low, int high) {
1       int pivotStartIndex = high - 1;
        String pivot = array[pivotStartIndex];
        int smallerBefore = low, largerAfter = high - 2;

N-2     while (smallerBefore <= largerAfter) {
          if (array[smallerBefore].compareTo(pivot) < 0) {
            smallerBefore += 1;
          }
          else {
            swap(array, smallerBefore, largerAfter);
    N-2     largerAfter -= 1;
          }
        }

1       swap(array, smallerBefore, pivotStartIndex);
        return smallerBefore;
      }

      public static void qsort(String[] array, int low, int high) {
        if (high - low <= 1) { return; }
        int splitAt = partition(array, low, high);
        qsort(array, low, splitAt);
        qsort(array, splitAt + 1, high);
      }

      public static void sortD(String[] array) {
        qsort(array, 0, array.length);
      }

      public static void main(String[] args) {
        String[] str = {"f", "b", "a", "e", "d", "c" };
        int[] result = SortQuickly.sortD(str);
        System.out.println(Arrays.deepToString(result));
      }
}
```

**Quicksort: Another magical (recursive) algorithm**

https://www.youtube.com/watch?v=ywWBy6J5gz8

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 14 | 4 | 9 | 12 | 15 | 8 | 19 | 2 |

Select a **pivot** element:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 14 | 4 | 9 | 12 | 15 | 8 | 19 | 2 |

"Partition" the elements in the array (smaller or equal to pivot, larger or equal to pivot)

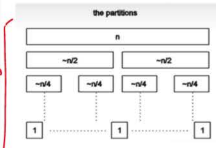
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 4 | 9 | 8 | 15 | **12** | 19 | 14 |

Magically sort the smaller elements and the larger elements (Quicksort)

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 4 | 8 | 9 | 12 | 15 | 19 | 21 |

≤ ≤

Quick Sort: Using a "good" pivot




height (log₂(n))

How many levels will there be if you choose a pivot that divides the list in half?

A. 1
B. log(n)
C. n
D. n*log(n)
E. $n^2$

If the time to partition on each level takes N comparisons, how long does Quicksort take with a good partition?

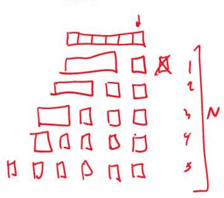
A. O(1)
B. O(log(n))
C. O(n)
D. O(n*log(n))
E. O($n^2$)

Which of these choices would be the *worst* choice for the pivot?

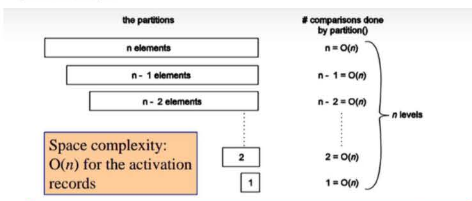
A. The minimum element in the list
B. The last element in the list
C. The first element in the list
D. A random element in the list

best choice
↳ median value

sorted list



Quick sort with a bad pivot




If the pivot always produces one empty partition and one with $n - 1$ elements, there will be $n$ levels, each of which requires $O(n)$ comparisons: $O(n^2)$ time complexity

Which of these choices is a better choice for the pivot?

A. The first element in the list
B. A random element in the list
C. They are about the same

**There are many ways to partition!**

Quick sort – Middle Pivot

1. We always pick the middle location as pivot
2. The data we sort is {2, 3, 1, 5, 4, 6, 7}

After the first split, what is the order of elements in the list that was <= pivot?

A. 1 2 3 4
B. 2 3 1 4
C. 4 3 2 1
D. 3 4 1 2
E. None of the above

2 3 1 5 4 6 7
2 3 1 4 5 6 7

0 8

sort {12, 4, 9, 3, 15, 8, 19, 2}

What is the first pivot?

How would you sort the data around the pivot?

lowIndex = low (0)
highIndex = high - 1 (N-1)
pivotIndex = (high - low) / 2 = (8-0) / 2 = 4

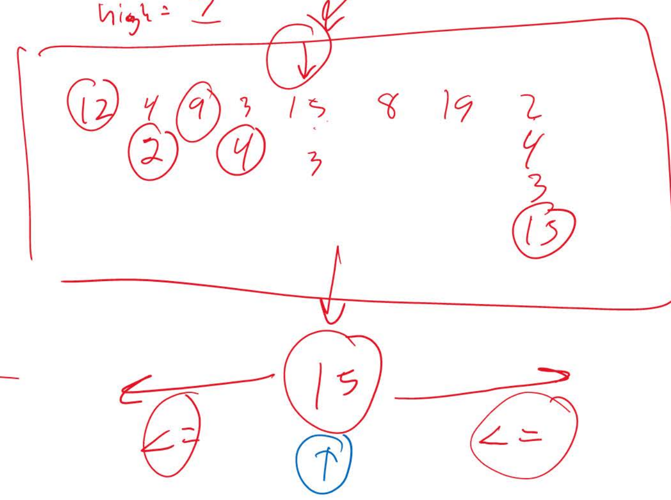
loop
if value[lowIndex] < pivotValue
lowIndex ++
else
swap ( lowIndex, highIndex )
highIndex --

if value[highIndex] > pivot value
highIndex --
else
swap ( lowIndex, highIndex )
lowIndex ++

if ( low >= high )
done = true

swap
chosen index?

low = 0 1 2 3 4
high = 7



```
import java.util.Arrays;
public class Sort {
  static void selectionSort(int[] arr) {
    for(int i = 0; i < arr.length; i += 1) {
      int minIndex = i;
      for(int j = i; j < arr.length; j += 1) {
        if(arr[minIndex] > arr[j]) { minIndex = j; }
      }
      int temp = arr[i];
      arr[i] = arr[minIndex];
      arr[minIndex] = temp;
    }
  }
  static void insertionSort(int[] arr) {
    for(int i = 0; i < arr.length; i += 1) {
      for(int j = i; j > 0; j -= 1) {
        if(arr[j] < arr[j-1]) {
          int temp = arr[j-1];
          arr[j-1] = arr[j];
          arr[j] = temp;
        }
        else { break; } // new! exit inner loop early
      }
    }
  }
}

import java.util.Arrays;
public class SortFaster {

  static int[] combine(int[] p1, int[] p2) {...}
  static int[] mergeSort(int[] arr) {
    int len = arr.length
    if(len <= 1) { return arr; }
    else {
      int[] p1 = Arrays.copyOfRange(arr, 0, len / 2);
      int[] p2= Arrays.copyOfRange(arr, len / 2, len);
      int[] sortedPart1 = mergeSort(p1);
      int[] sortedPart2 = mergeSort(p2);
      int[] sorted = combine(sortedPart1, sortedPart2);
      return sorted;
    }
  }

  static int partition(String[] array, int l, int h) {...}
  static void qsort(String[] array, int low, int high) {
    if(high - low <= 1) { return; }
    int splitAt = partition(array, low, high);
    qsort(array, low, splitAt);
    qsort(array, splitAt + 1, high);
  }
  public static void sort(String[] array) {
    qsort(array, 0, array.length);
  }
}
```

N
O(1)
N
N

Sorted Array → Θ(n)
1 2 3 4 5 6

Reverse sorted array
6 5 4 3 2 1
Θ(n²)
5 6
4 5 6
3 4 5 6
2 3 4 5 6
1 2 3 4 5 6

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

| | Insertion | Selection | Merge | Quick |
|---|---|---|---|---|
| Best case time | Sorted Array<br>Θ(N) | Θ(N²) | Θ(N + log₂(N)) | Median value<br>Θ(N + log₂(N)) |
| Worst case time | Reverse Sorted Array<br>Θ(N²) | Θ(N²) | Θ(N + log₂(N)) | Θ(N²)<br>Average case:<br>Θ(N + log(N)) |
| Key operations | swap(a, j, j-1)<br>(until in the right place) | swap(a, i, indexOfMin)<br>(after finding minimum value) | l = copy(a, 0, len/2)<br>r = copy(a, len/2, len)<br>ls = sort(l)<br>rs = sort(r)<br>merge(ls, rs) | p = partition(a, l, h)<br>sort(a, l, p)<br>sort(a, p + 1, h) |

10 3 9 2 5 7 6 4
3 2 4 6 10 9 8 7
2 4 5 4 7 8 10 9
4 5 9 10
log₂(n)

**Last note about sorting**

Not only do we care about runtime, we also care about

- Space: do we need extra storage?
- Stable: if we have duplicates, do we maintain the same ordering?

| Algorithm | Space | Stable |
|---|---|---|
| Bubble sort | O(1) | Yes |
| Selection sort | O(1) | No |
| Insertion sort | O(1) | Yes |
| Heap sort | O(1) | No |
| Merge sort | O(n) | Yes |
| Quick sort | O(logn) | No |

key value key value
{1, "Greg"} {1, "Derek"}

[ {1, "Greg"}, {1, "Derek"} ]
[ {1, "Derek"}, {1, "Derek"} ]
→ Unstable ordering

Java use for Arrays.sort
↳ Quicksort primitives
↳ Mergesort objects