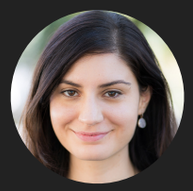
CS062

DATA STRUCTURES AND ADVANCED PROGRAMMING

15: Quicksort



Alexandra Papoutsaki
she/her/hers

Today we will look into quick sort, an algorithm that was named one of the 10 most important algorithms in science and engineering in the 20th century.

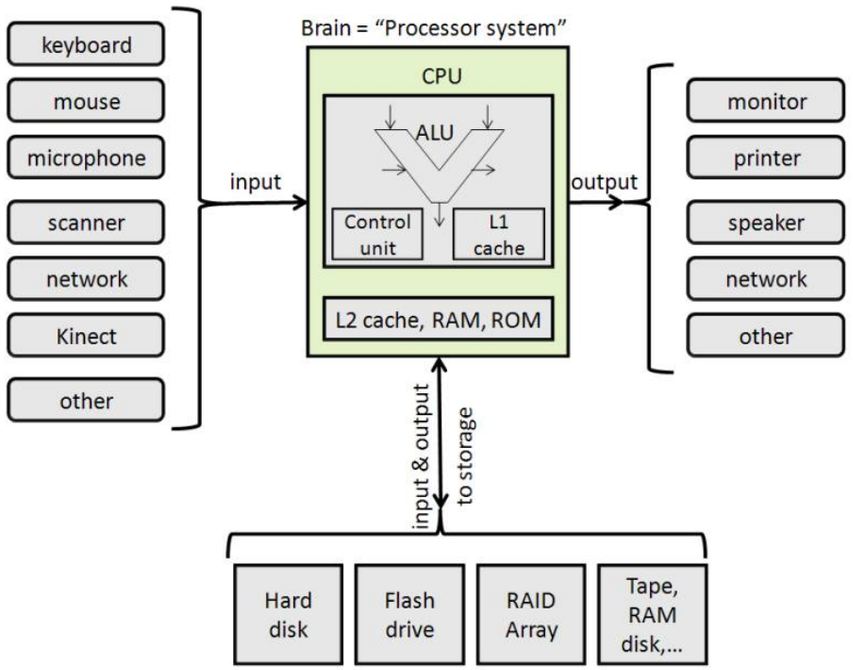
Lecture 13: Quicksort

- This week's assignment
- Quicksort

Some slides adopted from Algorithms 4th Edition or COS226

But before we do so, I wanted to give you an introduction to what this week's assignment will be.

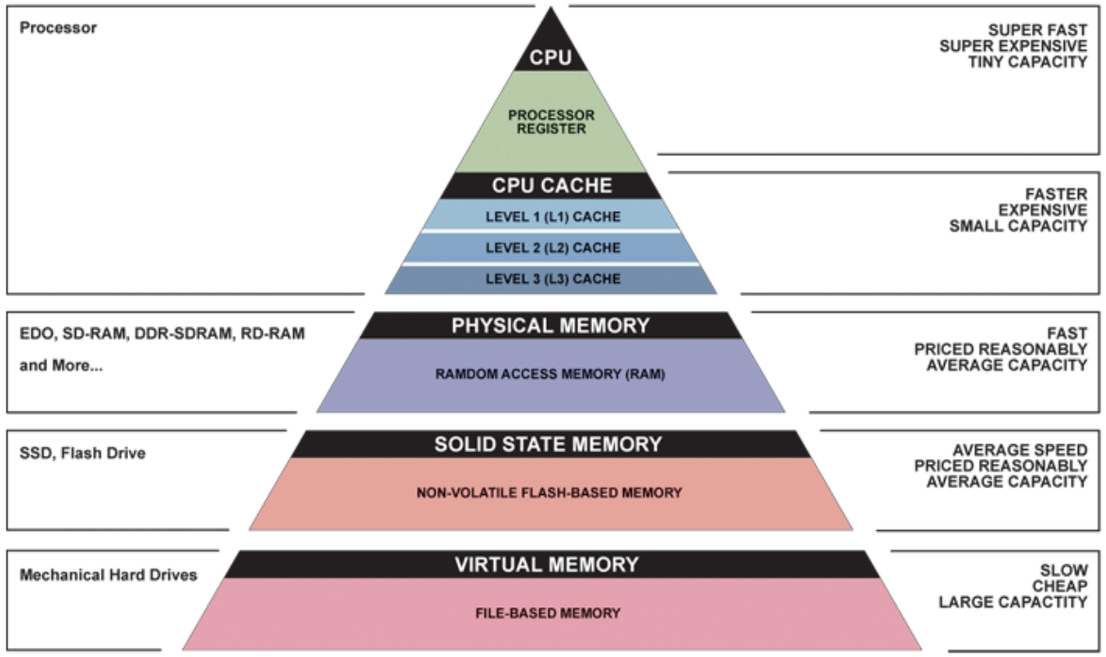
Basic Von Neumann computer architecture



http://cs.sru.edu/~mullins/cpsc100book/module03_internalHardware/module03-05_internalHardware.html

Last time, we learned about merge sort and we mentioned that it was developed by John von Neumann. John von Neumann has made numerous contributions to our field, one of them being the von Neumann architecture. The von Neumann architecture describes an electronic digital computer with these components:
1) a CPU - The CPU acts like the "brain" of a computer system. It contains the circuitry to interpret and execute program instructions. CPU is a processing unit that contains an arithmetic logic unit (which performs arithmetic functions like addition, subtraction, division, etc) and a control unit (performs the task of controlling functions of the computer. It monitors and gives proper instructions to all parts of the computer).
i) A Control Unit (CU) - CU acts like a manager in a computer. CU receives orders from RAM in the form of instruction and decode (break) that instruction down into specific commands for other components inside computer system. It directs the data flow and the operation of the ALU.
ii) An Arithmetic/Logic Unit (ALU) - ALU does all mathematical operations (arithmetic) (+/-/compare) and logical (AND/OR) calculations. Some computers have two ALUs to process two calculations simultaneously. That is called dual core technology. ALUs contain several special storage units called registers.
2) Memory that stores data and instructions and external mass storage
3) Input and output mechanisms
The term "von Neumann architecture" has evolved to mean any stored-program computer in which an instruction fetch and a data operation cannot occur at the same time because they share a common bus. This is referred to as the von Neumann bottleneck and often limits the performance of the system.

Memory Hierarchy (a simplified summarized story)



▲ Simplified Computer Memory Hierarchy
Illustration: Ryan J. Leng

If we were to look into a simplified model of memory hierarchy, at the top we would have memory that would be closest to the processor, that is registers and cache. One step down, we have the physical memory which you might have heard as RAM (although there are other versions of it as well). And then you can have external memory in the form of solid state or flash drives. You can also have virtual memory. For each one of these layers, there is an decreasing support for speed but also a decreasing cost which means that we can have slower but cheaper memory or faster and more expensive.

Registers

- Fastest, most expensive, tiny capacity.
- Run at same speed with CPU's clock
  ~3GHz today (3 billion operations/sec).
- Typically, 16-32 of them per core.
- Can hold 32 or 64 bits based on architecture that
  correspond to data or memory locations.

At the fastest level which is the closest to the CPU, we have registers. A register is a small, very fast storage area inside CPU. It is used to store intermediate values from calculations or instructions that is needed again immediately. For instance, when a ALU is commanded to calculate A*(B+C), ALU needs to calculate B+C first, then ALU need to store the result for a moment and use the result to multiply A. It is faster for ALU to access register than store the data in memory units.
There are only a few registers per core and they hold one instruction that is typically 32 or 64 bit based on architecture. A processor would have around 16 or 32 registers, and these would run at the same speed as the processor. The latest processors run at around 3 Ghz (3 billion operations per second). This would mean that the processor can process 3 billion instructions per second. The registers store data; the processor is the only thing that actually does calculations. These registers are also very very expensive to make, and hence are very very small in size and very few in number.

Cache

- Faster, expensive, small capacity.
- A bit slower than CPU's speed but faster than main memory.
- Typically, 2-3 levels (L1, L2, L3, etc.).
- 32-64 KB for L1, 128-512 KB for L2.

Next, we have the cache. A CPU cache is a hardware cache used by the central processing unit (CPU) of a computer to reduce the average cost (time or energy) to access data from the main memory. A cache is a smaller, faster memory, located closer to a processor core, which stores copies of the data from frequently used main memory locations. Most CPUs have different independent caches, including instruction and data caches, where the data cache is usually organized as a hierarchy of more cache levels (L1, L2, L3, L4, etc.).

Main (physical) memory (RAM)

- Fast, reasonably priced, average capacity.
- Much slower than CPU but significantly faster than disk.
- 8-32 GB.
- All programs and data must fit in memory.
  - If not, virtual memory to the rescue while accessing the disk.

Next comes the main or physical memory which we typically see in the form of RAMs from 8 to 32 GB. They are considerably slower than the CPU but much faster than disk. All programs and data must fit in memory, otherwise virtual memory would need to be employed.

External (secondary or auxiliary) memory (disk)

- Slower speed, reasonably priced, large capacity.
- Most computers have now solid state drives (SSDs) which are faster (but typically more expensive) than mechanical hard disk drives (HDDs).
- Hundreds of GB to a few TB.

At the last level we would have external/secondary/auxilary memory which is the disk. These days most computers have solid state drives but there are also mechanical hard disk drives. Disks are much slower than memory but can support hundreds of GB or even TB.

Assignment 5: On-disk merge sort

- All sorting algorithms we have seen assume that the data to be sorted can fit in main memory (RAM). In the era of big data, this is not always the case.
- For assignment 5, you will work on an external (on-disk) mergesort.
- Data are read from the disk in chunks of maximum size and are individually sorted using regular merge sort and stored back on disk in temporary “chunk" files.
- Temporary files are merged into an increasingly larger temporary file till the entire original data have been sorted and saved back on disk.
  - This is accomplished by iteratively merging the temporary file with a sorted temporary “chunk” file. Two buffered readers allow you to read a line/file at a time, compare them, and choose the “smallest” to be saved into the temporary file that contains all of merged data so far. Essentially, merge method of mergesort!

How do all of these things fit with this upcoming assignment? All sorting algorithms we have seen assume that the data to be sorted can fit in main memory (RAM). In the era of big data, this is not always the case. This week we will work on an external (on-disk) merge sort. Data are read from the disk in chunks of maximum size that can fit in memory and are individually sorted using regular merge sort and stored back on disk in temporary “chunk" files. Temporary files are merged into an increasingly larger temporary file till the entire original data have been sorted and saved back on disk.

BufferedReader

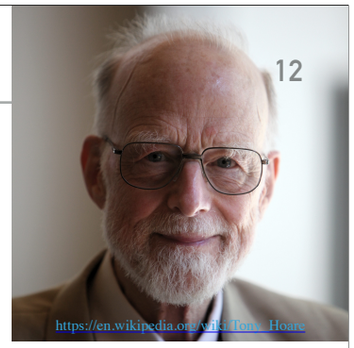
```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class ReadFile {


 public static void main(String[] args) {
  String fileName = "somePath/File.txt";

  try (BufferedReader br = new BufferedReader(new FileReader(fileName))) {

   String strCurrentLine = br.readLine();

   while (strCurrentLine != null) {
        //do something
        strCurrentLine = br.readLine();
   }

  } catch (IOException e) {
   e.printStackTrace();
  }
 }
}
```

So far we have seen the class Scanner when it comes to reading files. There is another class called BufferedReader that we can also use and will be handy because it allows us to read longer streams faster than Scanner. Here is a basic implementation that shows how to open a file and read it line by line.

Lecture 13: Quicksort

- This week's assignment
- Quicksort

With that, we are ready to move on to quick sort.



Basics

- Divide-and-conquer method
- Invented by Sir Tony Hoare in 1959.
  - Wanted to sort Russian words before looking them up in dictionary.
  - Came up with quicksort but did not know how to implement it.
  - Learned Algol 60 and recursion and implemented it.
  - Won the 1980 Turing Award (also invented the concept of null –and regretted it).

Here's the story about quick sort, another divide-and-conquer method, from Wikipedia. The quicksort algorithm was developed in 1959 by Tony Hoare while in the Soviet Union, as a visiting student at Moscow State University. At that time, Hoare worked on a project on machine translation for the National Physical Laboratory. As a part of the translation process, he needed to sort the words in Russian sentences prior to looking them up in a Russian-English dictionary that was already sorted in alphabetic order on magnetic tape. After recognizing that his first idea, insertion sort, would be slow, he quickly came up with a new idea that was Quicksort. He wrote a program in Mercury Autocode for the partition but could not write the program to account for the list of unsorted segments. On return to England, he was asked to write code for Shellsort as part of his new job. Hoare mentioned to his boss that he knew of a faster algorithm and his boss bet sixpence that he did not. His boss ultimately accepted that he had lost the bet. Later, Hoare learned about ALGOL and its ability to do recursion that enabled him to publish the code in Communications of the Association for Computing Machinery, the premier computer science journal of the time. Hoare won the 1980 Turing award for his contributions, one of them being the concept of null which he regretted.

Algorithm sketch

- Partition so that, for some pivot `pivot`:
  - Entry `a[pivot]` is in place.
  - There is no larger entry to the left of `pivot`.
  - No smaller entry to the right of `pivot`.
- Sort each subarray recursively.

The idea of quick sort is that we partition the array so that for some pivot the entry a[pivot] is in place, there is no larger entry to the left of the pivot, and no smaller entry to the right of the pivot. And we recursively sort each subarray. (Get the pivot Friends reference https://www.youtube.com/watch?v=TrQadHVITAI ?)

Quicksort Code

```
// quicksort the subarray from a[lo] to a[hi]
private static <E extends Comparable<E>> void quickSort(E[] a, int lo, int hi) {
    if (lo < hi){
       int pivot = partition(a, lo, hi);
       quickSort(a, lo, pivot-1);
       quickSort(a, pivot+1, hi);
    }
}

/**
* Rearranges the array in ascending order, using the natural order.
* @param a the array to be sorted
*/
public static <E extends Comparable<E>> void quickSort(E[] a) {
   quickSort(a, 0, a.length-1);
}
```

Let's see how we could translate this idea in Java code. You will notice that there is an overloaded method here, quickSort. The public method quickSort is what the client would call by passing an array to be sorted. Internally, that method passes the original array along with 0 and n-1 (a.length-1) to a private method quickSort. This private method is recursive and works by picking a pivot and recursively sorting each subarray.

Lomuto's* partition scheme

- Partition the subarray `a[lo…hi]` so that `a[lo…pivot-1]<=a[pivot]<=a[pivot+1…hi]`
- Start with `pivot` at `hi`, partitioning index `i` at `lo-1`, and pointer `j` at `lo`.
- Repeat the following until pointers `j` moves from `lo` to `hi-1`:
  - If element at `j` is smaller or equal to `pivot` increment `i` by 1 and exchange `a[i]` with `a[j]`.
- Increase `i` by 1 and exchange `a[i]` and `a[hi]`. Return `i` as the new pivot.
- *The recommended textbook follows Hoare's partition scheme which is faster but harder to implement

The key process in quickSort is partition. The target of partitions is, given an array and an element j of array as pivot, put j at its correct position in sorted array and put all smaller elements (smaller than j) before j, and put all greater elements (greater than x) after j. All this should be done in linear time. We will follow Lomuto's scheme which is easier to implement like Hoare's partition scheme (which the textbook follows). In this scheme, we start with the pivot at hi, and have two indices, one i at lo-1 and j at lo. We keep repeating the following until pointers j moves from lo to hi-1:
If element at j is smaller or equal to pivot increment i by 1 and exchange a[i] with a[j].
Once this loop terminates, we increase i by 1 and exchange a[i] and a[hi] and return i as the new pivot.

Partition Code

```
private static <E extends Comparable<E>> int partition(E[] a, int lo, int hi) {
        int i = lo - 1;
        E pivot = a[hi]

        for (int j = lo; j < hi; j++) {
            if (a[j].compareTo(pivot) <= 0) {
                i++;
                E temp = a[i];
                a[i] = a[j];
                a[j] = temp;
            }
        }

        i++;
        E temp = a[i];
        a[i] = a[hi];
        a[hi] = temp;

        return i;
    }
```

This is how we would implement partition based on what we discussed.

Quicksort Example - Sort [S,O,R,T,E,X,A,M,P,L,E]

```
// quicksort the subarray from a[lo] to a[hi]
private static <E extends Comparable<E>> void quickSort(E[] a, int lo, int hi) {
    if (lo < hi){
       int pivot = partition(a, lo, hi);
       quickSort(a, lo, pivot-1);
       quickSort(a, pivot+1, hi);
    }
}

/**
* Rearranges the array in ascending order, using the natural order.
* @param a the array to be sorted
*/
public static <E extends Comparable<E>> void quickSort(E[] a) {
   quickSort(a, 0, a.length-1);
}
```

Call partition with lo=0, hi=10

	S	O	R	T	E	X	A	M	P	L	E
-1	0	1	2	3	4	5	6	7	8	9	10
	lo										hi

Let's assume that someone calls quick sort on the array [S,O,R,T,E,X,A,M,P,L,E]. This will call the private method quick sort which in turn will call partition for the entire array.

Partition Example, lo=0, hi=10

	S	O	R	T	E	X	A	M	P	L	E
-1	0	1	2	3	4	5	6	7	8	9	10
i	lo										hi

```
    // partition the subarray a[lo..hi] so that a[lo..pivot-1] <= a[pivot] <= a[pivot+1..hi] and return
the partitioning index i.
    private static <E extends Comparable<E>> int partition(E[] a, int lo, int hi) {
        int i = lo - 1;
        E pivot = a[hi];
```

First, the call to partition will set i to lo-1, i.e. -1, and pivot the element at the highest location, i.e. E.

Partition Example, lo=0, hi=10

	S	O	R	T	E	X	A	M	P	L	E
-1	0	1	2	3	4	5	6	7	8	9	10
i	lo, j										hi

```
for(int j = lo; j < hi; j++ ){
   if(a[j].compareTo(pivot) <= 0 ) {
      i++;
      E temp = a[i];
      a[i] = a[j];
      a[j] = temp;
   }
}
```

Next we will set j to lo and we will loop doing the following. We will compare the j-th element with the pivot and if it's smaller we will advance i and swap the ith and jth elements. Here, S>E so no swap happens.

Partition Example, lo=0, hi=10

	S	O	R	T	E	X	A	M	P	L	E
-1	0	1	2	3	4	5	6	7	8	9	10
i	lo	j									hi

```
for(int j = lo; j < hi; j++ ){
   if(a[j].compareTo(pivot) <= 0 ) {
      i++;
      E temp = a[i];
      a[i] = a[j];
      a[j] = temp;
   }
}
```

j is now 1. O>E so again we don't satisfy the if statement.

Partition Example, lo=0, hi=10

	S	O	R	T	E	X	A	M	P	L	E
-1	0	1	2	3	4	5	6	7	8	9	10
i	lo		j								hi

```
for(int j = lo; j < hi; j++ ){
   if(a[j].compareTo(pivot) <= 0 ) {
      i++;
      E temp = a[i];
      a[i] = a[j];
      a[j] = temp;
   }
}
```

j is now 2. R>E so again we don't satisfy the if statement.

Partition Example, lo=0, hi=10

	S	O	R	T	E	X	A	M	P	L	E
-1	0	1	2	3	4	5	6	7	8	9	10
i	lo			j							hi

```
for(int j = lo; j < hi; j++ ){
   if(a[j].compareTo(pivot) <= 0 ) {
      i++;
      E temp = a[i];
      a[i] = a[j];
      a[j] = temp;
   }
}
```

j is now 3. T>E so again we don't satisfy the if statement.

Partition Example, lo=0, hi=10

	S	O	R	T	E	X	A	M	P	L	E
-1	0	1	2	3	4	5	6	7	8	9	10
i	lo				j						hi

```
for(int j = lo; j < hi; j++ ){
   if(a[j].compareTo(pivot) <= 0 ) {
      i++;
      E temp = a[i];
      a[i] = a[j];
      a[j] = temp;
   }
}
```

j is now 4. E<=E and we satisfy the if statement.

Partition Example, lo=0, hi=10

	S	O	R	T	E	X	A	M	P	L	E
-1	0	1	2	3	4	5	6	7	8	9	10
	lo, i				j						hi

```
for(int j = lo; j < hi; j++ ){
   if(a[j].compareTo(pivot) <= 0 ) {
      i++;
      E temp = a[i];
      a[i] = a[j];
      a[j] = temp;
   }
}
```

We increase i (to 0)

Partition Example, lo=0, hi=10

	E	O	R	T	S	X	A	M	P	L	E
-1	0	1	2	3	4	5	6	7	8	9	10
	lo, i				j						hi

```
for(int j = lo; j < hi; j++ ){
   if(a[j].compareTo(pivot) <= 0 ) {
      i++;
      E temp = a[i];
      a[i] = a[j];
      a[j] = temp;
   }
}
```

and swap the 0th and 4th elements i.e. E and S.

Partition Example, lo=0, hi=10

	E	O	R	T	S	X	A	M	P	L	E
-1	0	1	2	3	4	5	6	7	8	9	10
	lo, i					j					hi

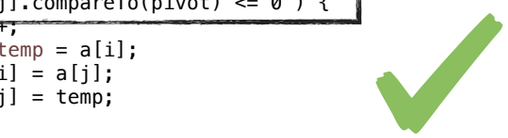
```
for(int j = lo; j < hi; j++ ){
   if(a[j].compareTo(pivot) <= 0 ) {
      i++;
      E temp = a[i];
      a[i] = a[j];
      a[j] = temp;
   }
}
```

j is now 5. X>E so we don't satisfy the if statement.

Partition Example, lo=0, hi=10

	E	O	R	T	S	X	A	M	P	L	E
-1	0	1	2	3	4	5	6	7	8	9	10
	lo, i						j				hi

```
for(int j = lo; j < hi; j++ ){
   if(a[j].compareTo(pivot) <= 0 ) {
      i++;
      E temp = a[i];
      a[i] = a[j];
      a[j] = temp;
   }
}
```



j is now 6. A<=E and we satisfy the if statement.

Partition Example, lo=0, hi=10

	E	O	R	T	S	X	A	M	P	L	E
-1	0	1	2	3	4	5	6	7	8	9	10
	lo	i					j				hi

```
for(int j = lo; j < hi; j++ ){
   if(a[j].compareTo(pivot) <= 0 ) {
      i++;
      E temp = a[i];
      a[i] = a[j];
      a[j] = temp;
   }
}
```

We increase i (to 1)

Partition Example, lo=0, hi=10

	E	A	R	T	S	X	O	M	P	L	E
-1	0	1	2	3	4	5	6	7	8	9	10
	lo	i					j				hi

```
for(int j = lo; j < hi; j++ ){
   if(a[j].compareTo(pivot) <= 0 ) {
      i++;
      E temp = a[i];
      a[i] = a[j];
      a[j] = temp;
   }
}
```

and swap the 1st and 6th elements i.e. O and A.

Partition Example, lo=0, hi=10

	E	A	R	T	S	X	O	M	P	L	E
-1	0	1	2	3	4	5	6	7	8	9	10
	lo	i						j			hi

```
for(int j = lo; j < hi; j++ ){
   if(a[j].compareTo(pivot) <= 0 ) {
      i++;
      E temp = a[i];
      a[i] = a[j];
      a[j] = temp;
   }
}
```

j is now 7. M>E so we don't satisfy the if statement.

Partition Example, lo=0, hi=10

	E	A	R	T	S	X	O	M	P	L	E
-1	0	1	2	3	4	5	6	7	8	9	10
	lo	i							j		hi

```
for(int j = lo; j < hi; j++ ){
   if(a[j].compareTo(pivot) <= 0 ) {
      i++;
      E temp = a[i];
      a[i] = a[j];
      a[j] = temp;
   }
}
```

j is now 8. P>E so we don't satisfy the if statement.

Partition Example, lo=0, hi=10

	E	A	R	T	S	X	O	M	P	L	E
-1	0	1	2	3	4	5	6	7	8	9	10
	lo	i								j	hi

```
for(int j = lo; j < hi; j++ ){
   if(a[j].compareTo(pivot) <= 0 ) {
      i++;
      E temp = a[i];
      a[i] = a[j];
      a[j] = temp;
   }
}
```

j is now 8. L>E so we don't satisfy the if statement.

Partition Example, lo=0, hi=10

	E	A	R	T	S	X	O	M	P	L	E
-1	0	1	2	3	4	5	6	7	8	9	10
	lo		i								j, hi

```
i++;
E temp = a[i];
a[i] = a[hi];
a[hi] = temp;

return i;
```

j reaches hi so we exit the for loop and increase i by 1 to 2.

Partition Example, lo=0, hi=10

	E	A	E	T	S	X	O	M	P	L	R
-1	0	1	2	3	4	5	6	7	8	9	10
	lo		i								j, hi

```
i++;
E temp = a[i];
a[i] = a[hi];
a[hi] = temp;

return i;
```

we swap the i-th and hi-th element, i.e. element at index 2 and 10, i.e. R and E.

Partition Example, lo=0, hi=10

	E	A	E	T	S	X	O	M	P	L	R
-1	0	1	2	3	4	5	6	7	8	9	10
	lo		i								j, hi

```
i++;
E temp = a[i];
a[i] = a[hi];
a[hi] = temp;
```

```
return i;
```

Partition for lo=0, hi=10 is complete. i=2

Call quickSort recursively on left subarray with lo=0, hi =1

and return i, i.e. 2. Now every element on the left of index 2 is smaller or equal to it, E, and every element on the right is larger or equal to it.

Quicksort Example - Sort [E,A]

```
// quicksort the subarray from a[lo] to a[hi]
private static <E extends Comparable<E>> void quickSort(E[] a, int lo, int hi) {
    if (lo < hi){
       int pivot = partition(a, lo, hi);
       quickSort(a, lo, pivot-1);
       quickSort(a, pivot+1, hi);
    }
}
```

Call partition with lo=0, hi=1

	E	A	E	T	S	X	O	M	P	L	R
-1	0	1	2	3	4	5	6	7	8	9	10
	lo	hi									

We next call quickSort recursively on left subarray with lo=0, hi =1

Partition Example, lo=0, hi=1

	E	A	E	T	S	X	O	M	P	L	R
-1	0	1	2	3	4	5	6	7	8	9	10
i	lo	hi									

```
    // partition the subarray a[lo..hi] so that a[lo..pivot-1] <= a[pivot] <= a[pvoti+1..hi] and return
the partitioning index i.
    private static <E extends Comparable<E>> int partition(E[] a, int lo, int hi) {
        int i = lo - 1;
        E pivot = a[hi];
```

We set i to lo-1, i.e. -1, and pivot to element at hi, i.e. A.

Partition Example, lo=0, hi=1

	E	A	E	T	S	X	O	M	P	L	R
-1	0	1	2	3	4	5	6	7	8	9	10
i	lo, j	hi									

```
for(int j = lo; j < hi; j++ ){
   if(a[j].compareTo(pivot) <= 0 ) {
      i++;
      E temp = a[i];
      a[i] = a[j];
      a[j] = temp;
   }
}
```

j starts at 0. E>A so we don't satisfy the if statement.

Partition Example, lo=0, hi=1

	E	A	E	T	S	X	O	M	P	L	R
-1	0	1	2	3	4	5	6	7	8	9	10
	lo, i	j, hi									

```
i++;
E temp = a[i];
a[i] = a[hi];
a[hi] = temp;

return i;
```

j is now equal to hi so we exit the for loop and increase i by 1 to 0.

Partition Example, lo=0, hi=1

	A	E	E	T	S	X	O	M	P	L	R
-1	0	1	2	3	4	5	6	7	8	9	10
	lo, i	j, hi									

```
i++;
E temp = a[i];
a[i] = a[hi];
a[hi] = temp;
return i;
```

we swap the element at index i (0) and hi (1), i,e. swap E and A

Partition Example, lo=0, hi=1

	A	E	E	T	S	X	O	M	P	L	R
-1	0	1	2	3	4	5	6	7	8	9	10

lo, i j, hi

```
i++;
E temp = a[i];
a[i] = a[hi];
a[hi] = temp;
return i;
```

Partition for lo=0, hi=1 is complete. i=0

No partition for subarrays of size 1.

Call quickSort recursively on right subarray with lo=3, hi = 10.

and return i, i.e. Everything on the right of i is larger than A. We don't partition for subarrays of size 1.

Quicksort Example - Sort [T, S, X, O, M, P, L, R]

```
// quicksort the subarray from a[lo] to a[hi]
private static <E extends Comparable<E>> void quickSort(E[] a, int lo, int hi) {
    if (lo < hi){
       int pivot = partition(a, lo, hi);
       quickSort(a, lo, pivot-1);
       quickSort(a, pivot+1, hi);
    }
}
```

Call partition with lo=3, hi=10

	A	E	E	T	S	X	O	M	P	L	R
-1	0	1	2	3	4	5	6	7	8	9	10
				lo							hi

We then call partition to the right subarray with lo=3, and hi = 10.

Partition Example, lo=3, hi=10

	A	E	E	T	S	X	O	M	P	L	R
-1	0	1	2	3	4	5	6	7	8	9	10
			i	lo							hi

```
    // partition the subarray a[lo..hi] so that a[lo..pivot-1] <= a[pivot] <= a[pivot+1..hi] and return
the partitioning index pi.
    private static <E extends Comparable<E>> int partition(E[] a, int lo, int hi) {
        int i = lo - 1;
        E pivot = a[hi];
```

We set i to lo-1, that is 2, and pivot is now R.

Partition Example, lo=3, hi=10

	A	E	E	T	S	X	O	M	P	L	R
-1	0	1	2	3	4	5	6	7	8	9	10
			i	lo, j							hi

```
for(int j = lo; j < hi; j++ ){
   if(a[j].compareTo(pivot) <= 0 ) {
      i++;
      E temp = a[i];
      a[i] = a[j];
      a[j] = temp;
   }
}
```

j starts at 3. T>R so we don't satisfy the if statement.

Partition Example, lo=3, hi=10

	A	E	E	T	S	X	O	M	P	L	R
-1	0	1	2	3	4	5	6	7	8	9	10
			i	lo	j						hi

```
for(int j = lo; j < hi; j++ ){
   if(a[j].compareTo(pivot) <= 0 ) {
      i++;
      E temp = a[i];
      a[i] = a[j];
      a[j] = temp;
   }
}
```

j is now 4. S>R so we don't satisfy the if statement.

Partition Example, lo=3, hi=10

	A	E	E	T	S	X	O	M	P	L	R
-1	0	1	2	3	4	5	6	7	8	9	10
			i	lo		j					hi

```
for(int j = lo; j < hi; j++ ){
   if(a[j].compareTo(pivot) <= 0 ) {
      i++;
      E temp = a[i];
      a[i] = a[j];
      a[j] = temp;
   }
}
```

j is now 5. x>R so we don't satisfy the if statement.

Partition Example, lo=3, hi=10

	A	E	E	T	S	X	O	M	P	L	R
-1	0	1	2	3	4	5	6	7	8	9	10
			i	lo			j				hi

```
for(int j = lo; j < hi; j++ ){
   if(a[j].compareTo(pivot) <= 0 ) {
      i++;
      E temp = a[i];
      a[i] = a[j];
      a[j] = temp;
   }
}
```

j is now 6. O<=R so we satisfy the if statement.

Partition Example, lo=3, hi=10

	A	E	E	T	S	X	O	M	P	L	R
-1	0	1	2	3	4	5	6	7	8	9	10
				lo, i			j				hi

```
for(int j = lo; j < hi; j++ ){
   if(a[j].compareTo(pivot) <= 0 ) {
      i++;
      E temp = a[i];
      a[i] = a[j];
      a[j] = temp;
   }
}
```

We increase i to 3

Partition Example, lo=3, hi=10

	A	E	E	O	S	X	T	M	P	L	R
-1	0	1	2	3	4	5	6	7	8	9	10
				lo, i			j				hi

```
for(int j = lo; j < hi; j++ ){
   if(a[j].compareTo(pivot) <= 0 ) {
      i++;
      E temp = a[i];
      a[i] = a[j];
      a[j] = temp;
   }
}
```

and swap O and T.

Partition Example, lo=3, hi=10

	A	E	E	O	S	X	T	M	P	L	R
-1	0	1	2	3	4	5	6	7	8	9	10
				lo, i				j			hi

```
for(int j = lo; j < hi; j++ ){
   if(a[j].compareTo(pivot) <= 0 ) {
      i++;
      E temp = a[i];
      a[i] = a[j];
      a[j] = temp;
   }
}
```

j is now 7. M<=R so we satisfy the if statement.

Partition Example, lo=3, hi=10

	A	E	E	O	S	X	T	M	P	L	R
-1	0	1	2	3	4	5	6	7	8	9	10
				lo	i			j			hi

```
for(int j = lo; j < hi; j++ ){
   if(a[j].compareTo(pivot) <= 0 ) {
      i++;
      E temp = a[i];
      a[i] = a[j];
      a[j] = temp;
   }
}
```

we increase i to 4.

Partition Example, lo=3, hi=10

	A	E	E	O	M	X	T	S	P	L	R
-1	0	1	2	3	4	5	6	7	8	9	10
				lo	i			j			hi

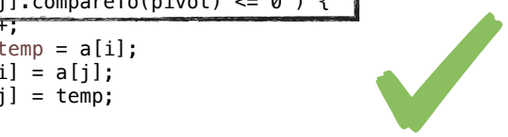
```
for(int j = lo; j < hi; j++ ){
   if(a[j].compareTo(pivot) <= 0 ) {
      i++;
      E temp = a[i];
      a[i] = a[j];
      a[j] = temp;
   }
}
```

and swap M and S

Partition Example, lo=3, hi=10

	A	E	E	O	M	X	T	S	P	L	R
-1	0	1	2	3	4	5	6	7	8	9	10
				lo	i				j		hi

```
for(int j = lo; j < hi; j++ ){
   if(a[j].compareTo(pivot) <= 0 ) {
      i++;
      E temp = a[i];
      a[i] = a[j];
      a[j] = temp;
   }
}
```



j is now 8. P<=R so we satisfy the if statement.

Partition Example, lo=3, hi=10

	A	E	E	O	M	X	T	S	P	L	R
-1	0	1	2	3	4	5	6	7	8	9	10
				lo		i			j		hi

```
for(int j = lo; j < hi; j++ ){
   if(a[j].compareTo(pivot) <= 0 ) {
      i++;
      E temp = a[i];
      a[i] = a[j];
      a[j] = temp;
   }
}
```

we increase i to 5

Partition Example, lo=3, hi=10

	A	E	E	O	M	P	T	S	X	L	R
-1	0	1	2	3	4	5	6	7	8	9	10
				lo		i			j		hi

```
for(int j = lo; j < hi; j++ ){
   if(a[j].compareTo(pivot) <= 0 ) {
      i++;
      E temp = a[i];
      a[i] = a[j];
      a[j] = temp;
   }
}
```

and swap X and P.

Partition Example, lo=3, hi=10

	A	E	E	O	M	P	T	S	X	L	R
-1	0	1	2	3	4	5	6	7	8	9	10
				lo		i				j	hi

```
for(int j = lo; j < hi; j++ ){
   if(a[j].compareTo(pivot) <= 0 ) {
      i++;
      E temp = a[i];
      a[i] = a[j];
      a[j] = temp;
   }
}
```

j is now 8. L<=R so we satisfy the if statement.

Partition Example, lo=3, hi=10

	A	E	E	O	M	P	T	S	X	L	R
-1	0	1	2	3	4	5	6	7	8	9	10
				lo			i			j	hi

```
for(int j = lo; j < hi; j++ ){
   if(a[j].compareTo(pivot) <= 0 ) {
      i++;
      E temp = a[i];
      a[i] = a[j];
      a[j] = temp;
   }
}
```

We increase i to 6

Partition Example, lo=3, hi=10

	A	E	E	O	M	P	L	S	X	T	R
-1	0	1	2	3	4	5	6	7	8	9	10
				lo			i			j	hi

```
for(int j = lo; j < hi; j++ ){
   if(a[j].compareTo(pivot) <= 0 ) {
      i++;
      E temp = a[i];
      a[i] = a[j];
      a[j] = temp;
   }
}
```

and swap T and L.

Partition Example, lo=3, hi=10

	A	E	E	O	M	P	L	S	X	T	R
-1	0	1	2	3	4	5	6	7	8	9	10
				lo				i			j, hi

```
i++;
E temp = a[i];
a[i] = a[hi];
a[hi] = temp;

return i;
```

j is now equal to hi and we exit the for loop to advance i to 7.

Partition Example, lo=3, hi=10

	A	E	E	O	M	P	L	R	X	T	S
-1	0	1	2	3	4	5	6	7	8	9	10
				lo				i			j, hi

```
i++;
E temp = a[i];
a[i] = a[hi];
a[hi] = temp;

return i;
```

we swap the S and R.

Partition Example, lo=3, hi=10

	A	E	E	O	M	P	L	R	X	T	S
-1	0	1	2	3	4	5	6	7	8	9	10
				lo				i			j, hi

```
i++;
E temp = a[i];
a[i] = a[hi];
a[hi] = temp;
```

```
return i;
```

Partition for lo=3, hi=10 is complete. i=7

Call quickSort recursively on left subarray with lo=3, hi = 6.

and return i, 7. Everything on the left of it smaller or equal to R, and everything on the right is larger or equal to R.

Quicksort Example - Sort [O,M,P,L]

```
// quicksort the subarray from a[lo] to a[hi]
private static <E extends Comparable<E>> void quickSort(E[] a, int lo, int hi) {
    if (lo < hi){
       int pivot = partition(a, lo, hi);
       quickSort(a, lo, pivot-1);
       quickSort(a, pivot+1, hi);
    }
}
```

Call partition with lo=3, hi=6

	A	E	E	O	M	P	L	R	X	T	S
-1	0	1	2	3	4	5	6	7	8	9	10
				lo			hi				

We call partition on the left subarray, between 3 and 6.

Partition Example, lo=3, hi=6

	A	E	E	O	M	P	L	R	X	T	S
-1	0	1	2	3	4	5	6	7	8	9	10
			i	lo			hi				

```
    // partition the subarray a[lo..hi] so that a[lo..pivot-1] <= a[pivot] <= a[pivot+1..hi] and return
the partitioning index pi.
    private static <E extends Comparable<E>> int partition(E[] a, int lo, int hi) {
        int i = lo - 1;
        E pivot = a[hi];
```

We set i to lo-1, i.e. 2, and the pivot is now L.

Partition Example, lo=3, hi=6

	A	E	E	O	M	P	L	R	X	T	S
-1	0	1	2	3	4	5	6	7	8	9	10
			i	lo, j			hi				

```
for(int j = lo; j < hi; j++ ){
   if(a[j].compareTo(pivot) <= 0 ) {
      i++;
      E temp = a[i];
      a[i] = a[j];
      a[j] = temp;
   }
}
```

j starts at 3. O>L so we don't satisfy the if statement.

Partition Example, lo=3, hi=6

	A	E	E	O	M	P	L	R	X	T	S
-1	0	1	2	3	4	5	6	7	8	9	10
			i	lo	j		hi				

```
for(int j = lo; j < hi; j++ ){
   if(a[j].compareTo(pivot) <= 0 ) {
      i++;
      E temp = a[i];
      a[i] = a[j];
      a[j] = temp;
   }
}
```

j is now 4. M>L so we don't satisfy the if statement.

Partition Example, lo=3, hi=6

	A	E	E	O	M	P	L	R	X	T	S
-1	0	1	2	3	4	5	6	7	8	9	10
			i	lo		j	hi				

```
for(int j = lo; j < hi; j++ ){
   if(a[j].compareTo(pivot) <= 0 ) {
      i++;
      E temp = a[i];
      a[i] = a[j];
      a[j] = temp;
   }
}
```

j is now 5. M>L so we don't satisfy the if statement.

Partition Example, lo=3, hi=6

	A	E	E	O	M	P	L	R	X	T	S
-1	0	1	2	3	4	5	6	7	8	9	10
				lo, i			j, hi				

```
i++;
E temp = a[i];
a[i] = a[hi];
a[hi] = temp;

return i;
```

j is now equal to hi and we exit the for loop to increase i to 3.

Partition Example, lo=3, hi=6

	A	E	E	L	M	P	O	R	X	T	S
-1	0	1	2	3	4	5	6	7	8	9	10
				lo, i			j, hi				

```
i++;
E temp = a[i];
a[i] = a[hi];
a[hi] = temp;
return i;
```

swap O and L

Partition Example, lo=3, hi=6

	A	E	E	L	M	P	O	R	X	T	S
-1	0	1	2	3	4	5	6	7	8	9	10
				lo, i			j, hi				

```
i++;
E temp = a[i];
a[i] = a[hi];
a[hi] = temp;
```

```
return i;
```

Partition for lo=3, hi=6 is complete. i=3

Call quickSort recursively on right subarray with lo=4, hi = 6.

and return 3. Everything on the left of 3 is smaller or equal to L and everything on the right is larger or equal to L.

Quicksort Example - Sort [M, P, O]

```
// quicksort the subarray from a[lo] to a[hi]
private static <E extends Comparable<E>> void quickSort(E[] a, int lo, int hi) {
    if (lo < hi){
       int pivot = partition(a, lo, hi);
       quickSort(a, lo, pivot-1);
       quickSort(a, pivot+1, hi);
    }
}
```

Call partition with lo=4, hi=6

	A	E	E	L	M	P	O	R	X	T	S
-1	0	1	2	3	4	5	6	7	8	9	10
					lo		hi				

We now call partition on the right subarray between indices 4 and 6.

Partition Example, lo=4, hi=6

	A	E	E	L	M	P	O	R	X	T	S
-1	0	1	2	3	4	5	6	7	8	9	10
				i	lo		hi				

```
    // partition the subarray a[lo..hi] so that a[lo..pivot-1] <= a[pivot] <= a[pivot+1..hi] and return
the partitioning index i.
    private static <E extends Comparable<E>> int partition(E[] a, int lo, int hi) {
        int i = lo - 1;
        E pivot = a[hi];
```

We set i to lo-1, i.e. 3, and pivot is now O.

Partition Example, lo=4, hi=6

	A	E	E	L	M	P	O	R	X	T	S
-1	0	1	2	3	4	5	6	7	8	9	10
				i	lo, j		hi				

```
for(int j = lo; j < hi; j++ ){
   if(a[j].compareTo(pivot) <= 0 ) {
      i++;
      E temp = a[i];
      a[i] = a[j];
      a[j] = temp;
   }
}
```

we start j at 4. M<=O so we satisfy the if statement.

Partition Example, lo=4, hi=6

	A	E	E	L	M	P	O	R	X	T	S
-1	0	1	2	3	4	5	6	7	8	9	10
					lo, j, i		hi				

```
for(int j = lo; j < hi; j++ ){
   if(a[j].compareTo(pivot) <= 0 ) {
      i++;
      E temp = a[i];
      a[i] = a[j];
      a[j] = temp;
   }
}
```

increase i to 4

Partition Example, lo=4, hi=6

	A	E	E	L	M	P	O	R	X	T	S
-1	0	1	2	3	4	5	6	7	8	9	10
					lo, j, i		hi				

```
for(int j = lo; j < hi; j++ ){
   if(a[j].compareTo(pivot) <= 0 ) {
      i++;
      E temp = a[i];
      a[i] = a[j];
      a[j] = temp;
   }
}
```

since i and j are equal, the swap doesn't have an effect.

Partition Example, lo=4, hi=6

	A	E	E	L	M	P	O	R	X	T	S
-1	0	1	2	3	4	5	6	7	8	9	10
					lo, i	j	hi				

```
for(int j = lo; j < hi; j++ ){
   if(a[j].compareTo(pivot) <= 0 ) {
      i++;
      E temp = a[i];
      a[i] = a[j];
      a[j] = temp;
   }
}
```

j is now 5. P>O so we don't satisfy the if statement.

Partition Example, lo=4, hi=6

	A	E	E	L	M	P	O	R	X	T	S
-1	0	1	2	3	4	5	6	7	8	9	10
					lo	i	j, hi				

```
i++;
E temp = a[i];
a[i] = a[hi];
a[hi] = temp;

return i;
```

j is equal to hi so we exit the for loop and increase i to 5.

Partition Example, lo=4, hi=6

	A	E	E	L	M	O	P	R	X	T	S
-1	0	1	2	3	4	5	6	7	8	9	10
					lo	i	j, hi				

```
i++;
E temp = a[i];
a[i] = a[hi];
a[hi] = temp;
return i;
```

swap P and M

Partition Example, lo=4, hi=6

	A	E	E	L	M	O	P	R	X	T	S
-1	0	1	2	3	4	5	6	7	8	9	10
					lo	i	j, hi				

```
i++;
E temp = a[i];
a[i] = a[hi];
a[hi] = temp;
return i;
```

Partition for lo=4, hi=6 is complete. i=5

Call quickSort recursively on right subarray with lo=8, hi = 10.

and return 5. Everything on the left of 5 is smaller or equal than O and everything on the right is larger or equal than O.

Quicksort Example - Sort [X,T,S]

```
// quicksort the subarray from a[lo] to a[hi]
private static <E extends Comparable<E>> void quickSort(E[] a, int lo, int hi) {
    if (lo < hi){
       int pivot = partition(a, lo, hi);
       quickSort(a, lo, pivot-1);
       quickSort(a, pivot+1, hi);
    }
}
```

Call partition with lo=8, hi=10

	A	E	E	L	M	O	P	R	X	T	S
-1	0	1	2	3	4	5	6	7	8	9	10
									lo		hi

We next call partition to right subarray between indices 8 and 10.

Partition Example, lo=8, hi=10

	A	E	E	L	M	O	P	R	X	T	S
-1	0	1	2	3	4	5	6	7	8	9	10
								i	lo		hi

```
    // partition the subarray a[lo..hi] so that a[lo..pivot-1] <= a[pivot] <= a[pivot+1..hi] and return
the partitioning index i.
    private static <E extends Comparable<E>> int partition(E[] a, int lo, int hi) {
        int i = lo - 1;
        E pivot = a[hi];
```

i is set to lo-1, i.e., 7, and pivot is S.

Partition Example, lo=8, hi=10

	A	E	E	L	M	O	P	R	X	T	S
-1	0	1	2	3	4	5	6	7	8	9	10
								i	lo, j		hi

```
for(int j = lo; j < hi; j++ ){
   if(a[j].compareTo(pivot) <= 0 ) {
      i++;
      E temp = a[i];
      a[i] = a[j];
      a[j] = temp;
   }
}
```

j starts at 8. X>S so we don't satisfy the if statement.

Partition Example, lo=8, hi=10

	A	E	E	L	M	O	P	R	X	T	S
-1	0	1	2	3	4	5	6	7	8	9	10
								i	lo	j	hi

```
for(int j = lo; j < hi; j++ ){
   if(a[j].compareTo(pivot) <= 0 ) {
      i++;
      E temp = a[i];
      a[i] = a[j];
      a[j] = temp;
   }
}
```

j is now 9. T>S so we don't satisfy the if statement.

Partition Example, lo=8, hi=10

	A	E	E	L	M	O	P	R	X	T	S
-1	0	1	2	3	4	5	6	7	8	9	10
									lo, i		j, hi

```
i++;
E temp = a[i];
a[i] = a[hi];
a[hi] = temp;

return i;
```

j is equal to hi and we exit the for loop. We increase i to 8.

Partition Example, lo=8, hi=10

	A	E	E	L	M	O	P	R	S	T	X
-1	0	1	2	3	4	5	6	7	8	9	10
									lo, i		j, hi

```
i++;
E temp = a[i];
a[i] = a[hi];
a[hi] = temp;

return i;
```

swap X and S.

Partition Example, lo=8, hi=10

	A	E	E	L	M	O	P	R	S	T	X
-1	0	1	2	3	4	5	6	7	8	9	10
									lo, i		j, hi

```
i++;
E temp = a[i];
a[i] = a[hi];
a[hi] = temp;
return i;
```

Partition for lo=8, hi=10 is complete. i=8

Call quickSort recursively on right subarray with lo=9, hi = 10.

and return 8. Everything on the left of it is smaller or equal to S and everything on the right is larger or equal to S.

Quicksort Example - Sort [T,S]

```
// quicksort the subarray from a[lo] to a[hi]
private static <E extends Comparable<E>> void quickSort(E[] a, int lo, int hi) {
    if (lo < hi){
       int pivot = partition(a, lo, hi);
       quickSort(a, lo, pivot-1);
       quickSort(a, pivot+1, hi);
    }
}
```

Call partition with lo=9, hi=10

	A	E	E	L	M	O	P	R	S	T	X
-1	0	1	2	3	4	5	6	7	8	9	10
										lo	hi

Final call to partition on right subarray indexed between 9 and 10.

Partition Example, lo=9, hi=10

	A	E	E	L	M	O	P	R	S	T	X
-1	0	1	2	3	4	5	6	7	8	9	10
									i	lo	hi

```
    // partition the subarray a[lo..hi] so that a[lo..pivot-1] <= a[pivot] <= a[pivot+1..hi] and return
the partitioning index i.
    private static <E extends Comparable<E>> int partition(E[] a, int lo, int hi) {
        int i = lo - 1;
        E pivot = a[hi];
```

we set i to lo-1, i.e., 8 and pivot is X.

Partition Example, lo=9, hi=10

	A	E	E	L	M	O	P	R	S	T	X
-1	0	1	2	3	4	5	6	7	8	9	10
									i	lo, j	hi

```
for(int j = lo; j < hi; j++ ){
   if(a[j].compareTo(pivot) <= 0 ) {
      i++;
      E temp = a[i];
      a[i] = a[j];
      a[j] = temp;
   }
}
```

j starts at 9. T<=X so we satisfy the if statement.

Partition Example, lo=9, hi=10

	A	E	E	L	M	O	P	R	S	T	X
-1	0	1	2	3	4	5	6	7	8	9	10
										lo, j, i	hi

```
for(int j = lo; j < hi; j++ ){
   if(a[j].compareTo(pivot) <= 0 ) {
      i++;
      E temp = a[i];
      a[i] = a[j];
      a[j] = temp;
   }
}
```

we increase i to 9.

Partition Example, lo=9, hi=10

	A	E	E	L	M	O	P	R	S	T	X
-1	0	1	2	3	4	5	6	7	8	9	10
										lo, j, i	hi

```
for(int j = lo; j < hi; j++ ){
   if(a[j].compareTo(pivot) <= 0 ) {
      i++;
      E temp = a[i];
      a[i] = a[j];
      a[j] = temp;
   }
}
```

but since i and j are the same, swapping doesn't change anything.

Partition Example, lo=9, hi=10

	A	E	E	L	M	O	P	R	S	T	X
-1	0	1	2	3	4	5	6	7	8	9	10
										lo	hi, i, j

```
i++;
E temp = a[i];
a[i] = a[hi];
a[hi] = temp;

return i;
```

j is now equal to hi and we exit the for loop. we increase i by 1 to 10.

Partition Example, lo=9, hi=10

	A	E	E	L	M	O	P	R	S	T	X
-1	0	1	2	3	4	5	6	7	8	9	10
										lo	hi, i, j

~~i++;~~
```
E temp = a[i];
a[i] = a[hi];
a[hi] = temp;
```
return i;

i and hi are the same so no effect from the swap.

Partition Example, lo=9, hi=10

	A	E	E	L	M	O	P	R	S	T	X
-1	0	1	2	3	4	5	6	7	8	9	10
										lo	hi, i, j

```
i++;
E temp = a[i];
a[i] = a[hi];
a[hi] = temp;
```

```
return i;
```

Partition for lo=9, hi=10 is complete.
i=10

Entire array has been sorted (we don't call quickSort on arrays of size 1)

and return 10. Everything on the left of 10 is smaller or equal to X. The entire array is now sorted

Great algorithms are better than good ones

- Your laptop executes 10^8 comparisons per second
- A supercomputer executes 10^{12} comparisons per second

	Insertion sort			Mergesort			Quicksort		
Computer	**Thousand inputs**	**Million inputs**	**Billion inputs**	**Thousand inputs**	**Million inputs**	**Billion inputs**	**Thousand inputs**	**Million inputs**	**Billion inputs**
Home	Instant	2 hours	300 years	instant	1 sec	15 min	Instant	0.5 sec	10 min
Supercomputer	Instant	1 sec	1 week	instant	instant	instant	instant	instant	instant

To put in perspective how quick sort compares to merge sort and insertion sort, quick sort beats them both but it is quite close to merge sort. Again, notice that the algorithm and not the computer is the important piece here.

Quicksort analysis: best case

- Quicksort divides everything exactly in half.
- Similar to merge sort.
- Number of compares is $\sim n \log n$.

Let's talk about the best case scenario for quicksort which will end up with quick sort partitioning everything exactly in half. In that case it would be similar to merge sort and it would have linearithmic performance.

Quicksort analysis: worst case

- Data are already sorted or we pick the smallest or largest key as pivot.
- Number of compares is ~n^2 - quadratic!
- Extremely unlikely (less likely than the probably that your computer is struck by lightning) if we first shuffle and our shuffling is not broken.

In contrast, in the worst case scenario the data are already sorted or we pick a bad pivot and we have to do quadratic comparisons. This is extremely unlikely if we shuffle our array first to make sure that we don't fall in these scenarios.

Things to remember about quick sort

- 39% more compares than merge sort but in practice it is faster because it does not move data much.
  - If good implementation, even in sorted arrays it can be linearithmic. If not, we end up with quadratic.
- $O(n \log n)$ average, $O(n^2)$ worst, in practice faster than mergesort.
- In-place sorting.
- **Not** stable.

The average case, which is extremely likely for any practical application, is going to be about 1.39 n log n. That's compares than Mergesort uses, but Quicksort is much faster, because it doesn't do much corresponding to each compare. It just does the compare and increment a pointer. Whereas, Mergesort has to move the items into and out of the auxiliary array, which is more expensive. It is an in place algorithm but not stable.

Quicksort practical improvements

- Use insertion sort for small subarrays.
- Best choice of pivot is the median of a small sample.

- For years, Java used quicksort for collections of primitives and mergesort for collections of objects due to stability.
  - Has moved to dual-pivot quick sort (Yaroslavskiy, Bentley, and Bloch, 2009) and timsort (Peters, 1993), respectively.

In terms of practical improvements, we can again use insertion sort for small subarrays. The question of finding the median key value and then partitioning on that value (to halve the array and ensure optimal performance) has stumped experts for over a decade. It is possible to find it in linear time but the cost far exceeds the 39 percent savings available from splitting the array into equal parts.
For years, Java used quicksort for collections of primitives and mergesort for collections of objects due to stability. The sorting algorithm is a Dual-Pivot Quicksort by Vladimir Yaroslavskiy, Jon Bentley, and Joshua Bloch. This algorithm offers O(n log(n)) performance on many data sets that cause other quicksorts to degrade to quadratic performance, and is typically faster than traditional (one-pivot) Quicksort implementations.

Sorting: the story so far

Which Sort	In place	Stable	Best	Average	Worst	Remarks
Selection	X		$O(n^2)$	$O(n^2)$	$O(n^2)$	n exchanges
Insertion	X	X	$O(n)$	$O(n^2)$	$O(n^2)$	Use for small arrays or partially ordered
Merge		X	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Guaranteed performance; stable
Quick	X		$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$n \log n$ probabilistic guarantee; fastest in practice

To put things in perspective, quick sort is the fastest sorting algorithm we have seen so far for the average and best case but can degrade to quadratic in the worst case.

Lecture 13: Quicksort

- This week's assignment
- Quicksort

To summarize, today we saw quick sort, another divide and conquer algorithm that partitions the array and ensures that at all elements on the left of the pivot are smaller or equal to it and all the element of it are larger or equal to it. Quicksort might have the same big oh complexity but in practice it's faster than merge sort and does not require extra memory. The only disadvantage is that it can end up being quadratic.

Readings:

- Recommended Textbook:
  - Chapter 2.3 (Pages 288-296)
- Recommended Textbook Website:
  - Quicksort: https://algs4.cs.princeton.edu/23quicksort/
    - We use a different implementation

Code

- Lecture 13 code

Practice Problem

- What would the resulting array for the first call to partition be for the following array: [E,A,S,Y,Q,U,E,S,T,I,O,N].

ANSWER

- What would the resulting array and new pivot index for the first call to partition be for the following array: [E,A,S,Y,Q,U,E,S,T,I,O,N]
- [E, A, E, I, N, U, S, S, T, Y, O, Q] and pivot: at index 4.