202: Computer Science II

Northern Virginia Community College

Sorting

Cody Narber, M.S.
December 2, 2017

Problem: Sorting

The sorting problem is defined by its input and its output:

- **Input** - A sequence of comparable data $< d_1, d_2, ..., d_n >$
- **Output** - A permutation (reordering) of the data $< d_i, d_j, ...d_k >$ such that $d_i \leq d_j \leq ... \leq d_k$

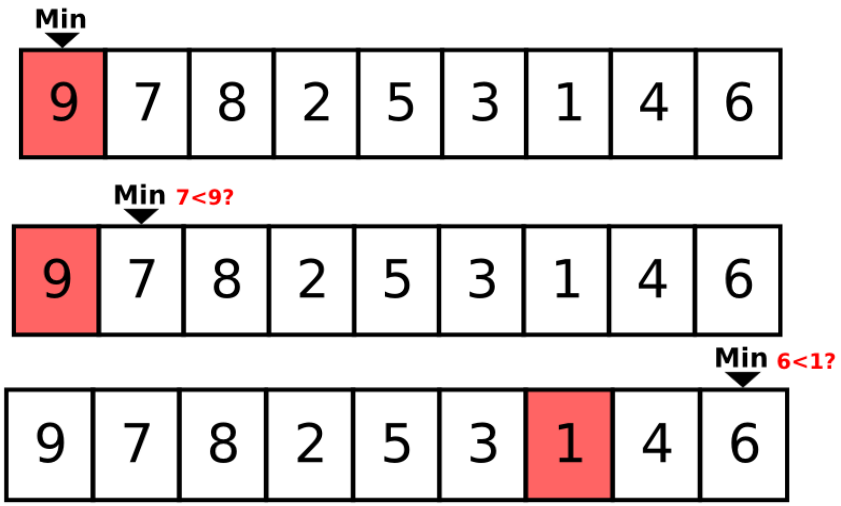
There are several sorting algorithms each with various benefits and drawbacks the few that we will work with are:

- Selection Sort
- Insertion Sort
- Bubble Sort
- Shell Sort
- Heap Sort
- Quick Sort
- Merge Sort

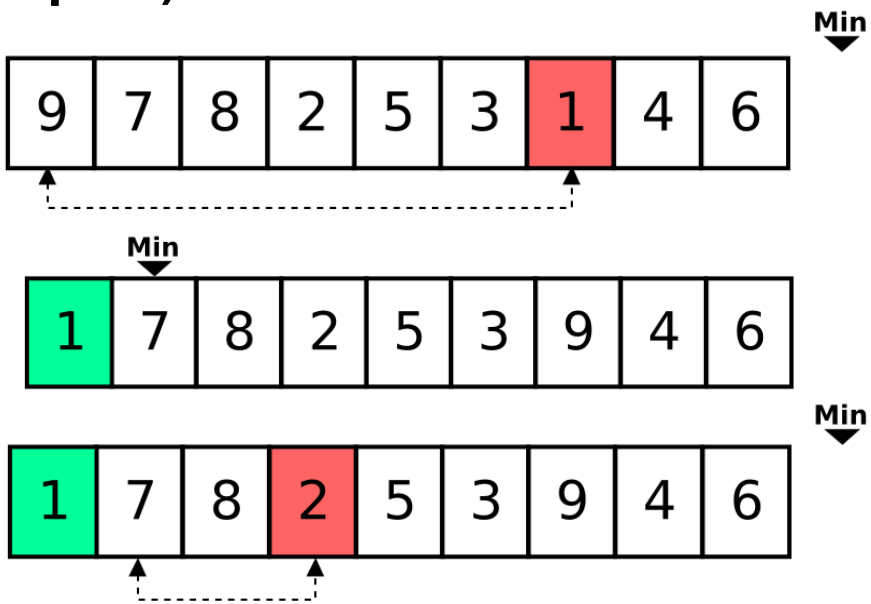
Selection Sort

Selection sort can be thought of as looking for the minimal element in the list and copying it over to a new list repeating with the remaining elements. This could be memory inefficient as it would maintain two lists. Instead we can swap the smallest value with what remains.

Find the Minimum in unsorted portion:

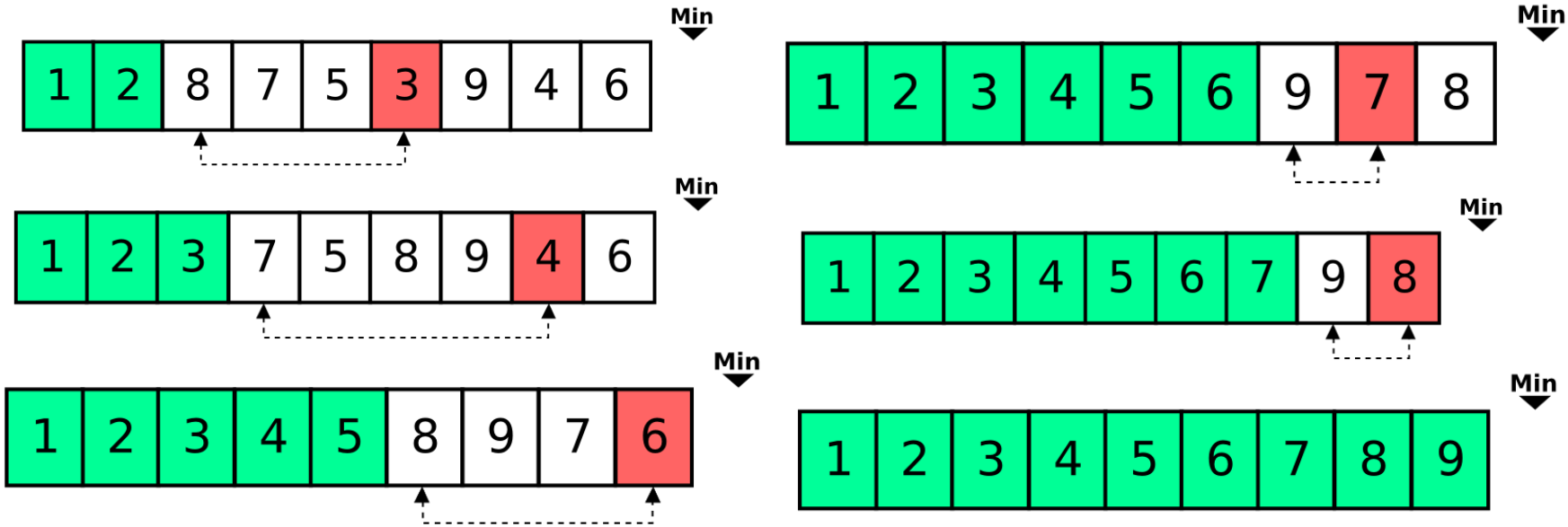


Swap with next location (and repeat):



Selection Sort - 2

Repeat searching for minimal location (denoted as light red color) swapping with back of sorted portion (denoted as cyan color)

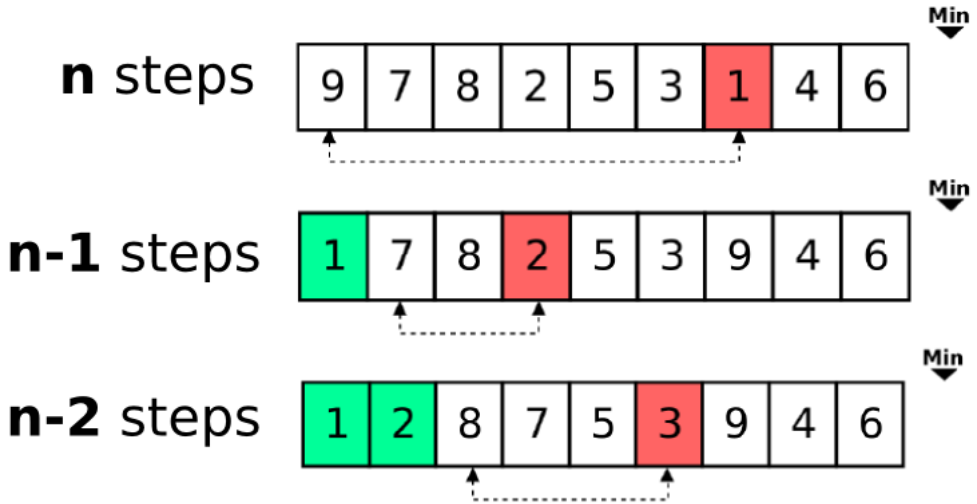


Selection Sort Algorithm

```
1  public void selectSort(int[] toSort){
2        //Precondition: Parameter Array is full
3        //Postcondition: Parameter Array will be sorted
4
5        for(int sortInd = 0; sortInd<toSort.length; sortInd++){
6              int minIndex = sortInd;
7              for(pMI = sortInd; pMI<toSort.length; pMI++){
8                    if(toSort[pMI] < toSort[minIndex])
9                          minIndex = pMI;
10             }
11             //Found the minInd, swap
12             int temp = toSort[sortInd];
13             toSort[sortInd] = toSort[minIndex];
14             toSort[minIndex] = temp;
15       }
16 }
```

Selection Sort Runtime Analysis

Find the Minimum:



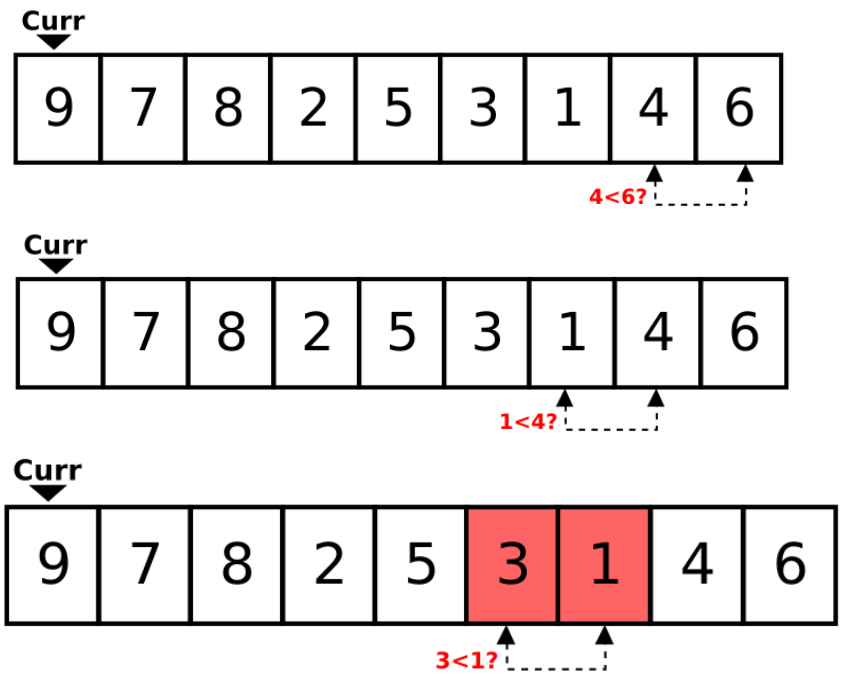
..and so on. thus resulting in the equations:

$$\begin{aligned} \textit{comparisons} &= (n-1)+(n-2)+(\ldots)+3+2+1 \\ &= \sum_{i=1}^{n} i - n = \frac{n(n+1)}{2} - n = \frac{n^2+n}{2} - n \\ &= \frac{n^2}{2} + \frac{n}{2} - \frac{4n}{2} = \frac{n^2}{2} - \frac{n}{2} = O(n^2) \end{aligned}$$

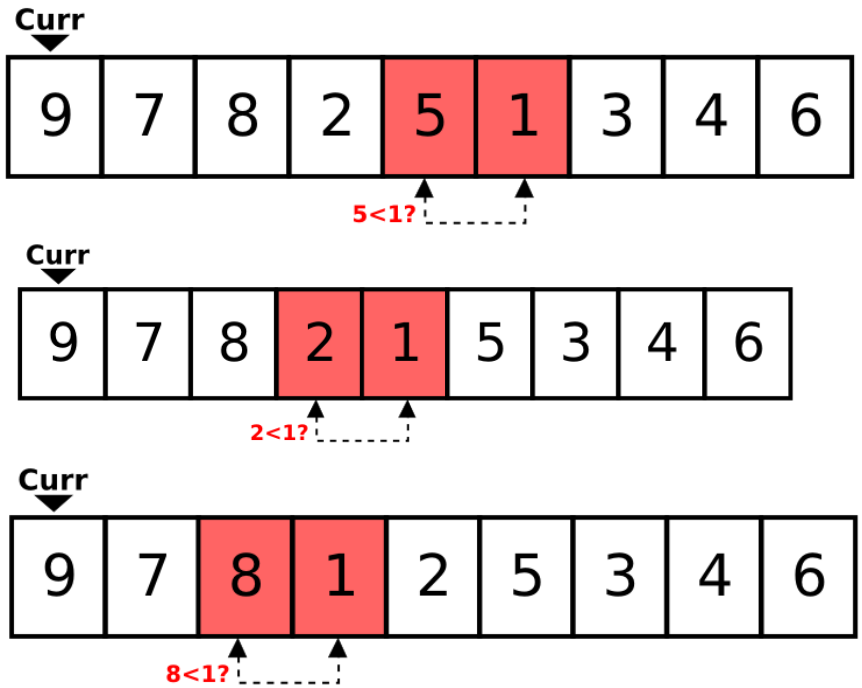
Bubble Sort

Bubble sort can be thought of as bubbling up the smallest value in the array. We will begin at the end of the array and swap as we go to ensure the smaller value comes first.

Check each pair to see if a swap is needed

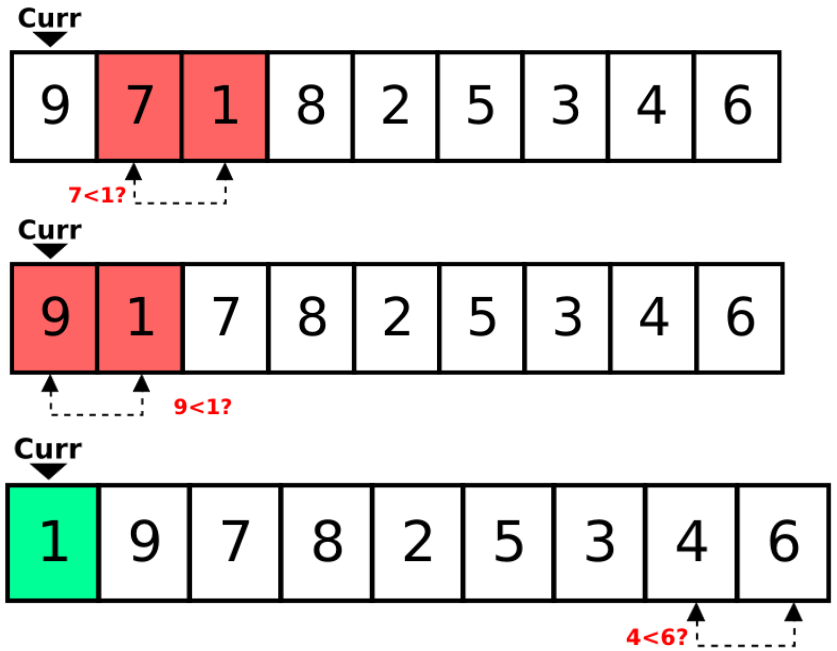


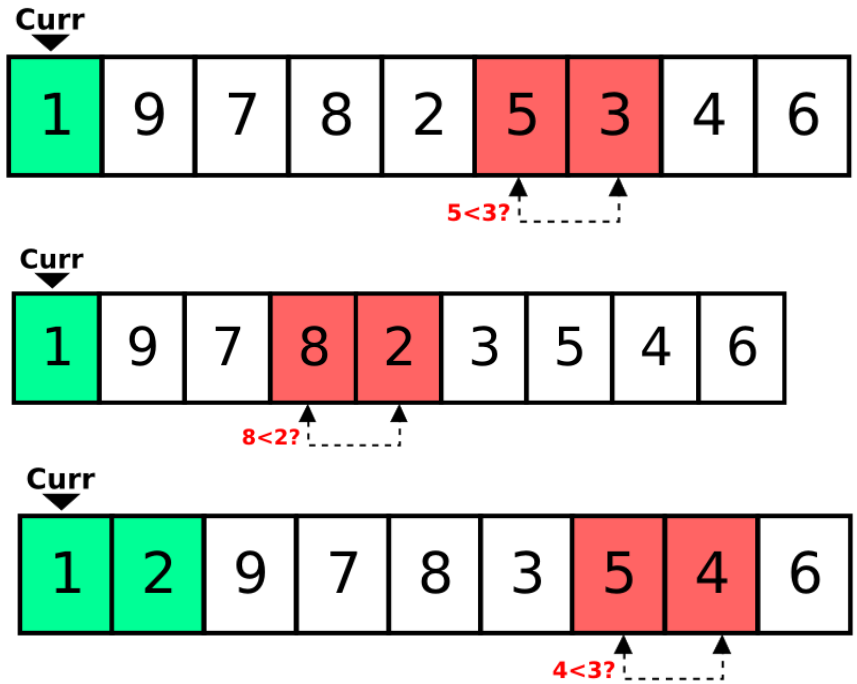
Minimum Found (swap all the way up)



Bubble Sort - 2

Continue swapping to the front to build a sorted part, each minimum will be swapped the whole way when found (other swaps may occur, but it is only the minimum in the unsorted section that really matters.





Bubble Sort Algorithm

```
1  public void bubbleSort(int[] toSort){
2        //Precondition: Parameter Array is full
3        //Postcondition: Parameter Array will be sorted
4
5        for(int sortInd = 0; sortInd<toSort.length; sortInd++){
6              for(cInd = toSort.length-1; cInd>sortInd; cInd--){
7                    if(toSort[cInd] < toSort[cInd-1]){
8                          //Found a smaller entry, swap!
9                          int temp = toSort[cInd];
10                         toSort[cInd] = toSort[cInd-1];
11                         toSort[cInd-1] = temp;
12                   }
13             }//end internal iterator from end
14       } // end sorted sections
15 }
```

Bubble Sort Runtime Analysis

Just as before we are performing:

$$\begin{aligned} \textit{comparisons} &= (n-1) + (n-2) + (\ldots) + 3 + 2 + 1 \\ &= \sum_{i=1}^{n} i - n = \frac{n(n+1)}{2} - n = \frac{n^2+n}{2} - n \\ &= \frac{n^2}{2} + \frac{n}{2} - \frac{4n}{2} = \frac{n^2}{2} - \frac{n}{2} = O(n^2) \end{aligned}$$

but because we are potentially swapping more than once per pass of the outer loop, bubble sort will run slower than selection sort every time....However! If the data is almost sorted, our swaps may take care and we can see if no swaps were needed the data has been sorted and we can end early!

EXAMPLE:

2	1	4	3	6	5	8	9	7

Bubble Sort Algorithm v2

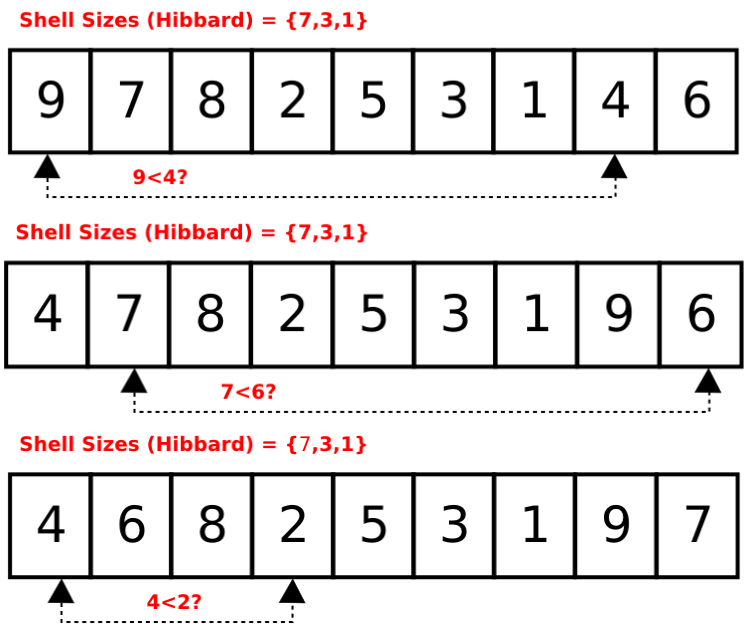
We can just add a flag check to see if we can return early.

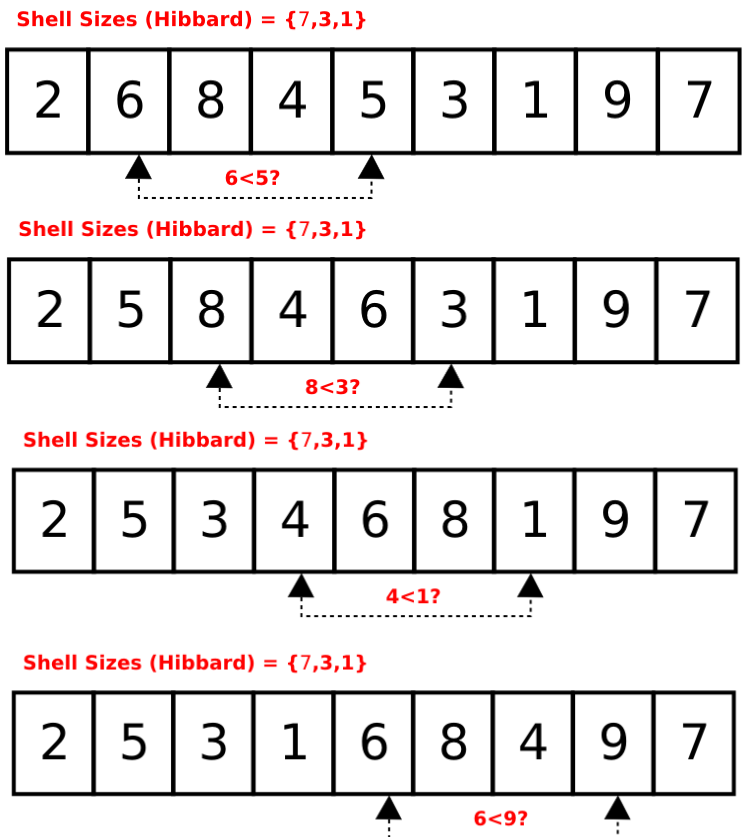
```
1  public void bubbleSort(int[] toSort){
2        //Precondition: Parameter Array is full
3        //Postcondition: Parameter Array will be sorted
4
5        for(int sortInd = 0; sortInd<toSort.length; sortInd++){
6              boolean swapped = false;
7              for(cInd = toSort.length-1; cInd>sortInd; cInd--){
8                    if(toSort[cInd] < toSort[cInd-1]){
9                          //Found a smaller entry, swap!
10                         int temp = toSort[cInd];
11                         toSort[cInd] = toSort[cInd-1];
12                         toSort[cInd-1] = temp;
13                         swapped = true;
14                   }
15             }//end internal iterator from end
16             if(!swapped) return; //No swaps it is sorted!
17       } // end sorted sections
18 }
```

Shell Sort

Shell sort is an extension of Bubble sort in that it uses a set of decreasing gaps (Ex: 31, 15, 7, 3, 1) to use for comparisions/swaps in order to get data on roughly the correct size before settling on pure bubble sort, because as we have seen the closer the data is to sorted the better bubble sort will perform.

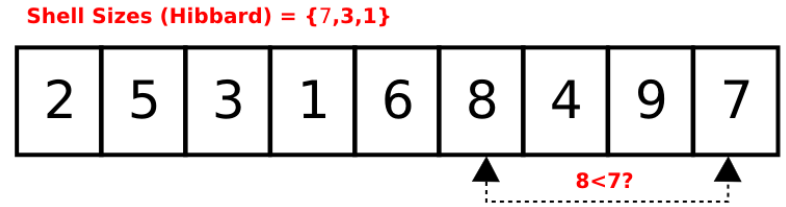
Check each separated pair to see if a swap is needed

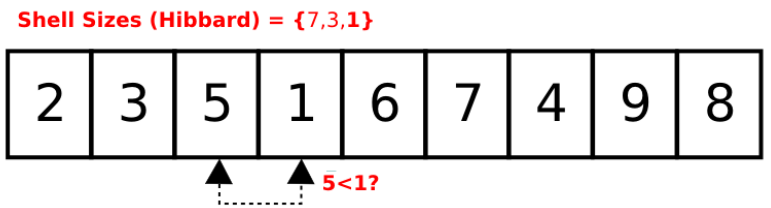


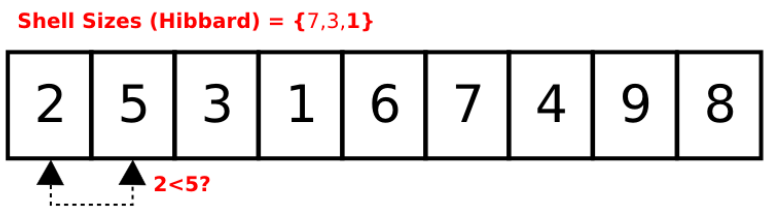


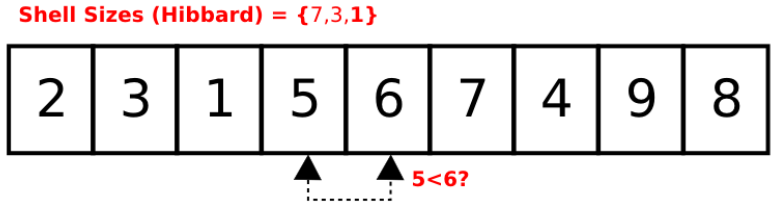
Shell Sort - 2

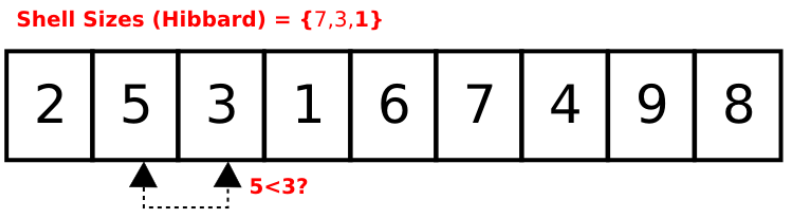
Shell will eventually get down to single swaps, and you will continue just like our revised bubble sort.

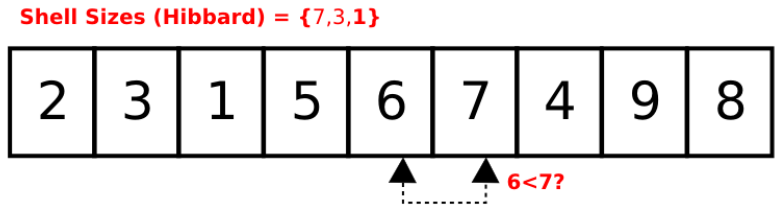












For the most part Shell is better than Bubble as the first gap shifts get things approximately where they need to go esspecially in large set, however, **Bubble v2 can be better than Shell** when the data is perfectly sorted (as shell cannot stop until it gets down to 1 space shifting.

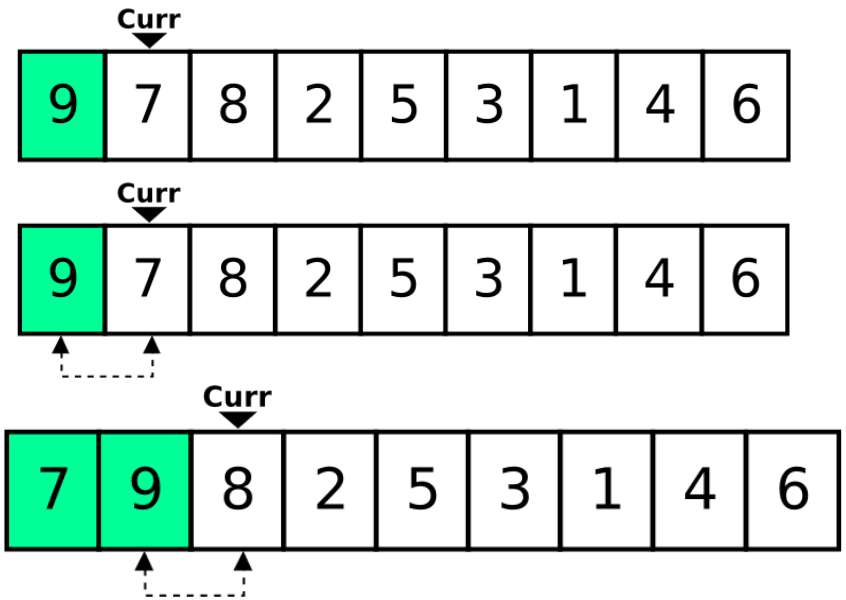
Shell Sort Algorithm

```
1  public void shellSort(int[] toSort){
2        //Precondition: Parameter Array is full
3        //Postcondition: Parameter Array will be sorted
4        int[] gaps = {31, 15, 7, 3};
5        for(int g = 0; g<gaps.length; g++){
6              for(cInd = 0; cInd<toSort.length-gaps[g]; cInd++){
7                    int nInd = cInd+gaps[g];
8                    if(toSort[cInd] < toSort[nInd]){
9                          //Found a smaller entry, swap!
10                         int temp = toSort[cInd];
11                         toSort[cInd] = toSort[nInd];
12                         toSort[nInd] = temp;
13                   }
14             }//end index iteration
15       } // end gap iteration
16
17     bubbleSort(toSort);
18 }
```

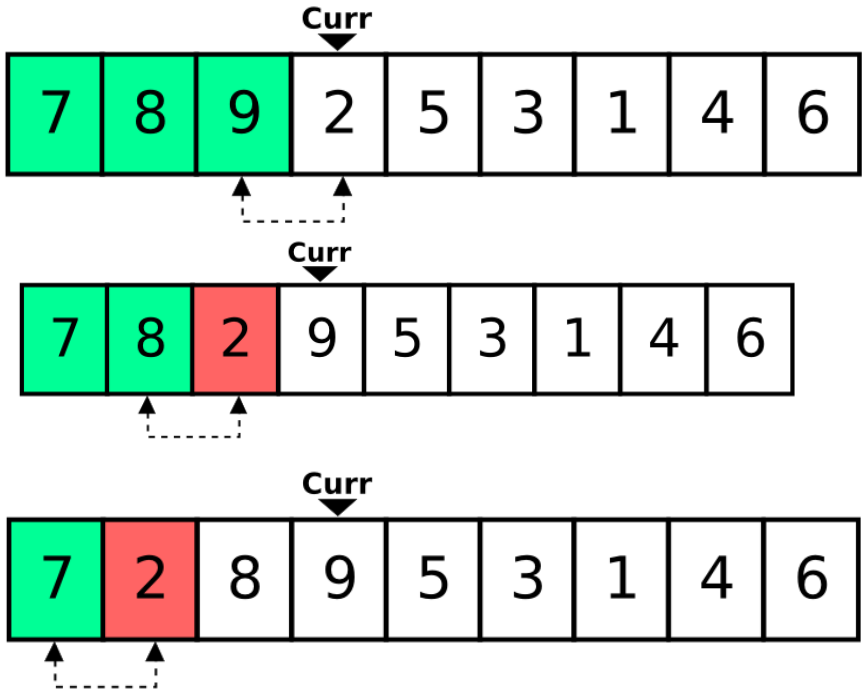
Insertion Sort

Insertion sort works like adding elements to a list shifting elements as needed just like when we created a sorted array-based list.

Check current index with previous in sorted subset

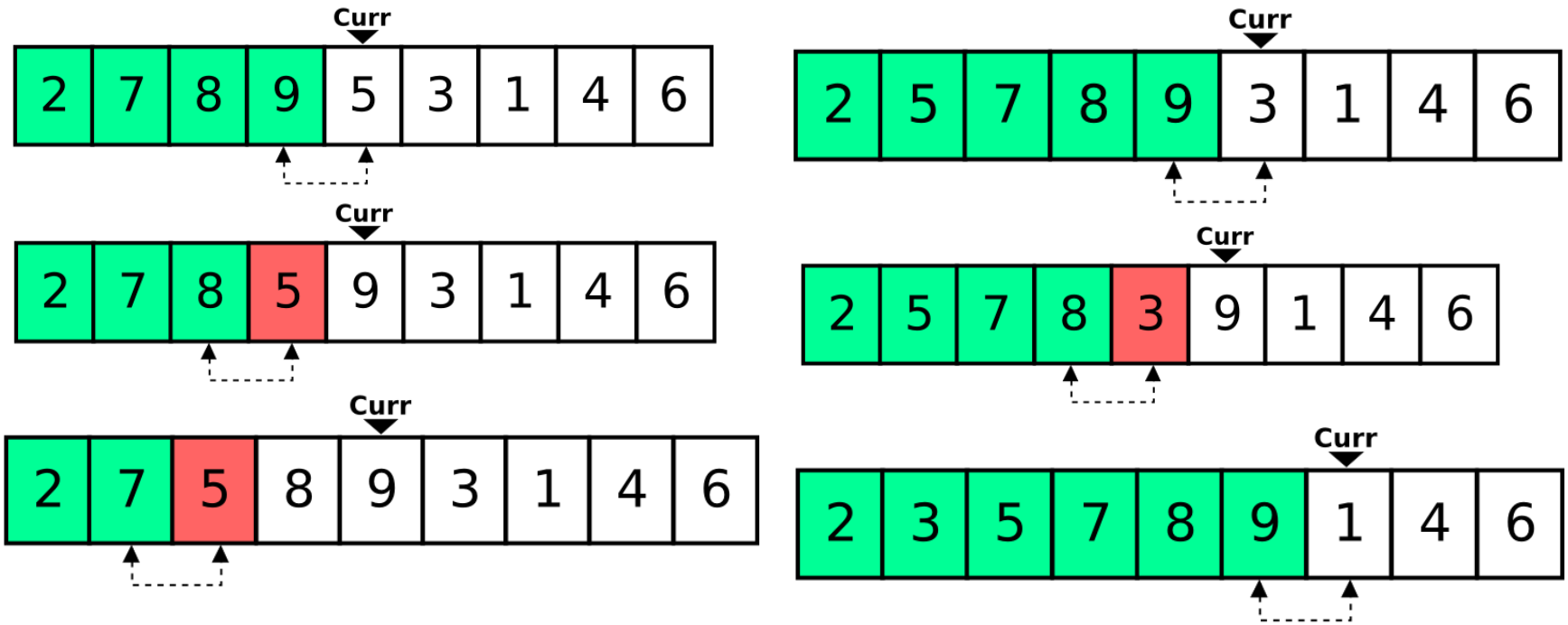


Insert current into proper location (swapping as you go)



Insertion Sort - 2

Continue swapping each new element to be added into sorted set until the previous element is less than



Insertion Sort Algorithm

```
1  public void insertSort(int[] toSort){
2        //Precondition: Parameter Array is full
3        //Postcondition: Parameter Array will be sorted
4
5        for(int sortInd = 1; sortInd<toSort.length; sortInd++){
6             int cInd = sortInd;
7             while(cInd!=0 && toSort[cInd] < toSort[cInd-1]){
8                   //Found a smaller entry, swap!
9                   int temp = toSort[cInd];
10                  toSort[cInd] = toSort[cInd-1];
11                  toSort[--cInd] = temp; //shifting down
12             }//end internal iterator from end
13       } // end sorted sections
14 }
```

Insertion Sort Runtime Analysis

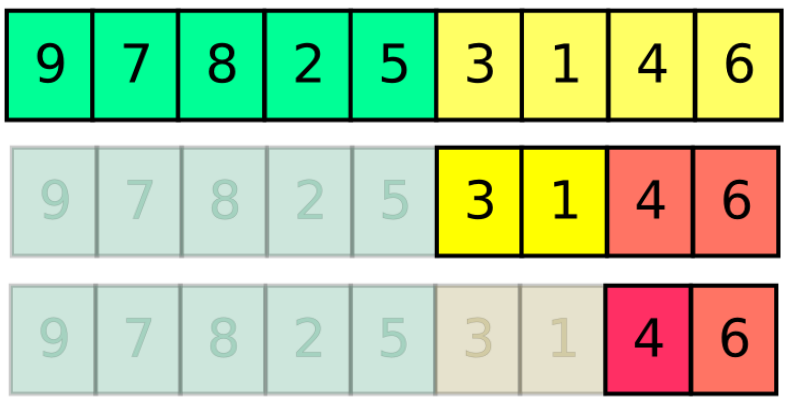
A little different than last time, in that we will be performing at least n comparisons each time we grow our sorted subset, and if the data is already sorted our algorithms will run in N steps...but in the worst case a completely reversed set it will take N*N steps and swaps...so when analyzing average running time we can see that it is $O(n^2)$ just like the rest of the algorithms thus far.

Knowing about the data you are to sort can help you decide which of these: Selections, Bubble, or Insertion is best for your program (they are also very small algorithms to implement...not a lot of programming overhead).

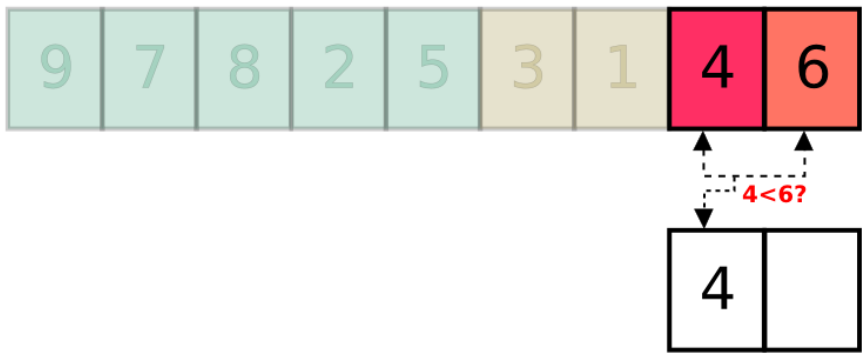
Merge Sort

Merge Sort is a type of algorithm that is known as **divide-and-conquer** – essentially top-down strategy that breaks apart the problem into smaller problems to solve (hmmm...sounds like recursion). Merge sort splits the the array in half and sorts each half, then the results are merged together.

Split the data in half (or almost in half in cases of odd sized data)

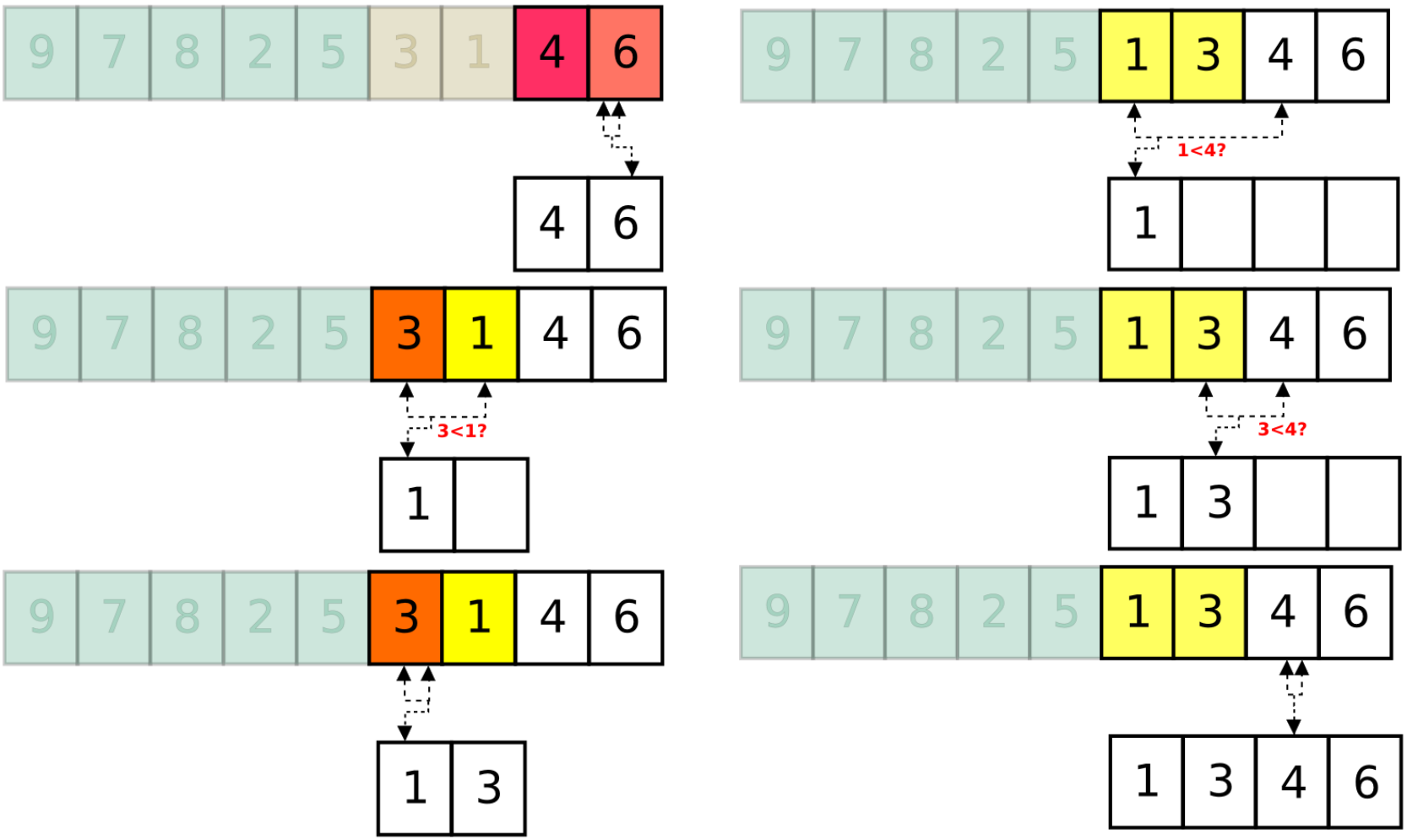


After we know halves are sorted...we can merge, copy into temp array to copy back



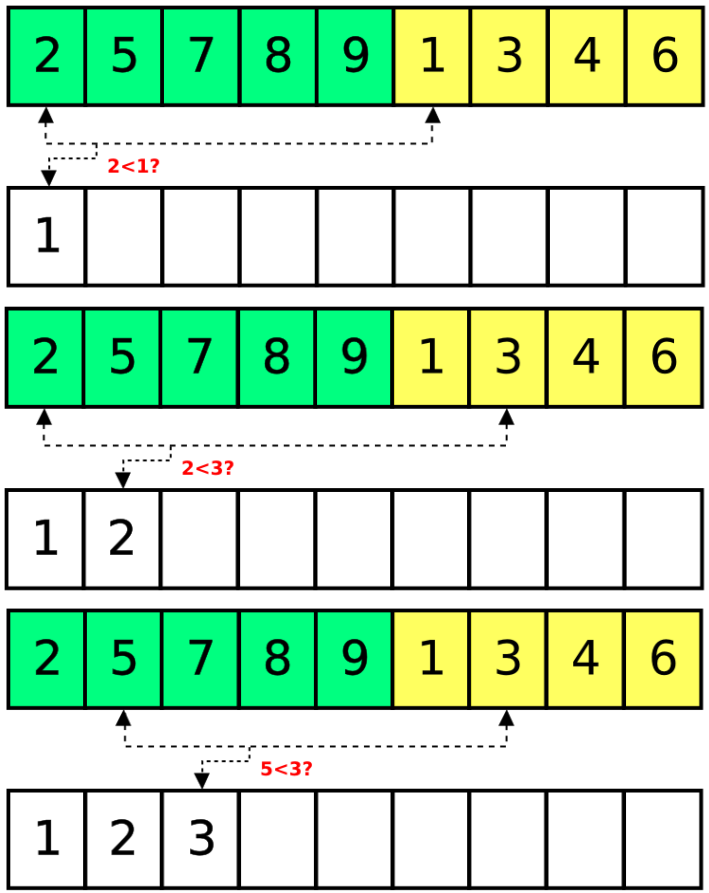
Merge Sort - 2

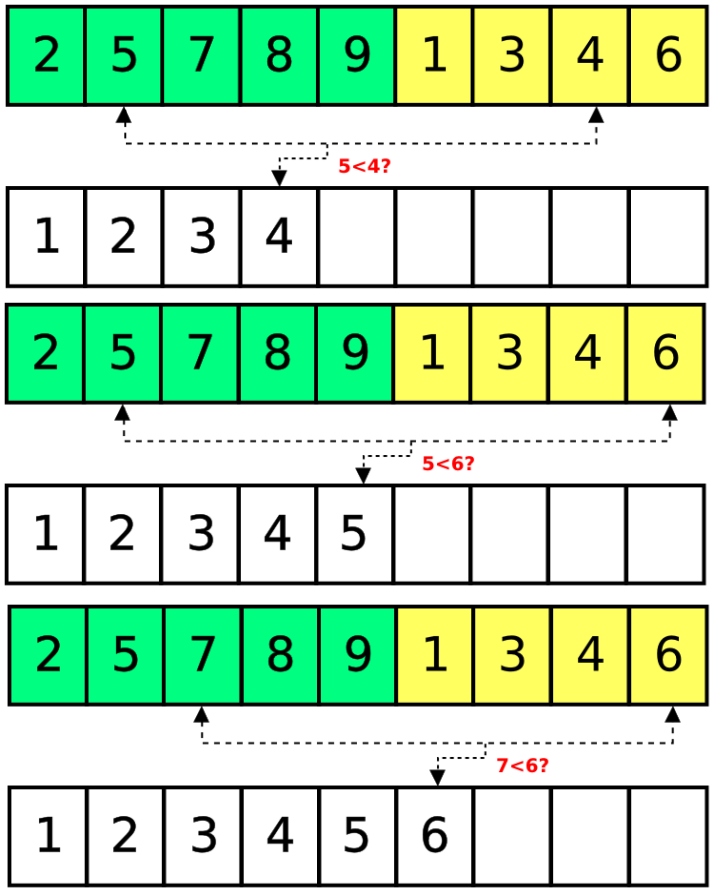
Continue the merge steps...



Merge Sort - 3

Assume we then split and merged the front half...merging the two





Once one half's end is reached we can just copy the remaining values from the other half

Merge Sort Algorithm - 1

```
1  //Precond: Array is full, and sIndex and eIndex are in-bounds
2  //Postcond: Array will be sorted between the specified indices
3  public void mergeSort(int[] toSort, int sIndex, int eIndex){
4       //section of size 0 or 1
5       if(eIndex-sIndex<2) return;
6
7       //recursively merge sort two halves
8       int halfway = (eIndex+sIndex)/2;
9       mergeSort(toSort,sIndex,halfway);
10      mergeSort(toSort,halfway,eIndex);
11
12      //create temporary storage for the sorted data
13      int[] mergeTemp = new int[eIndex-sIndex];
14      int tInd = 0;
15
16      //Iterate two halves with trackers
17      int leftInd = sIndex;
18      int rightInd = halfway;
19
20      //....The merge loop
21 }
```

Merge Sort Algorithm - 2

```
1  //Precond: Array is full, and sIndex and eIndex are in-bounds
2  //Postcond: Array will be sorted between the specified indices
3  public void mergeSort(int[] toSort, int sIndex, int eIndex){
4       //....The dividing and variable allocations
5
6       while(leftInd!=halfway && rightInd!=eIndex){
7            if(toSort[leftInd]<toSort[rightInd])
8                 mergeTemp[tInd++] = toSort[leftInd++];
9            else
10                mergeTemp[tInd++] = toSort[rightInd++];
11      }
12
13      while(leftInd!=halfway)
14           mergeTemp[tInd++] = toSort[leftInd++];
15      while(rightInd!=eIndex)
16           mergeTemp[tInd++] = toSort[rightInd++];
17
18      //copy data back
19      for(int i=0; i<mergeTemp.length; i++)
20           toSort[i+sIndex] = mergeTemp[i];
21 }
```

Merge Sort Runtime Analysis

Since we are looking at dividing a conquering we want to think about each aspect of the algorithm:

- How many times are we dividing?

  DIVIDE: We are splitting in half, and as we have seen with the binary search is $O(log_2 n)$ steps.

- What do we do for each level of division?

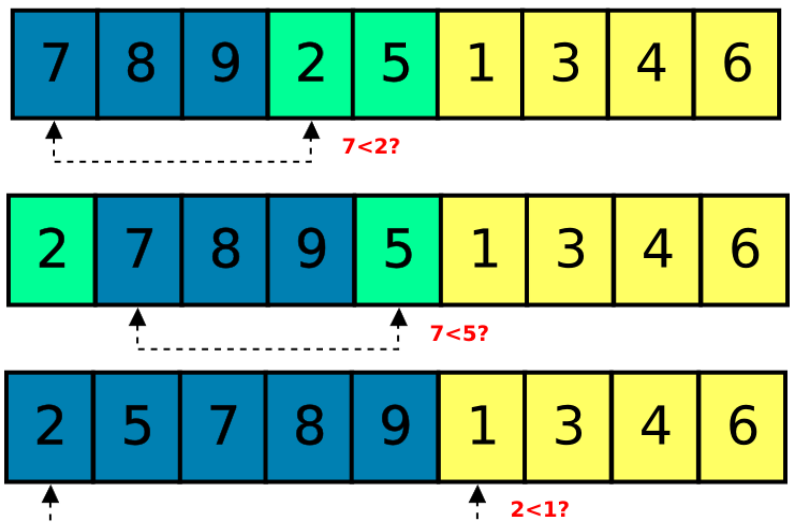
  MERGE: This occurs for each division, and all we are doing is iterating each element of array section copying over into temp storage $O(n)$. The copy back is also $O(n)$ so the whole merge process is also $O(n)$

TOGETHER: Since the merge step occurs after each division, we are looking at the product of the two parts, therefore Merge-Sort takes $O(nlog_2 n)$ time to sort.

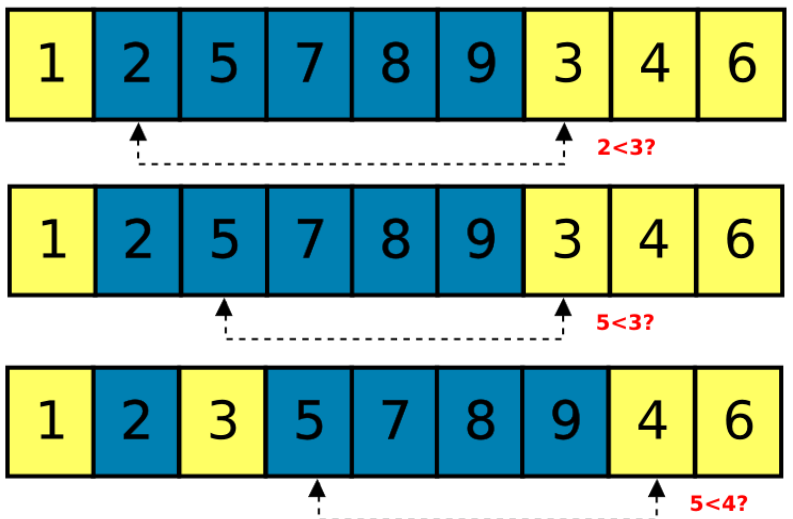
Merge Sort In-Place

You may be thinking that merge sort using temporary storage for large data could eat up a lot of memory, and that is true it can. There is also a way to merge in-place...HOWEVER, we lose something by doing this:

Let's look at the merge steps, much later in execution



We can see that each merge step behaves like a bubble sort

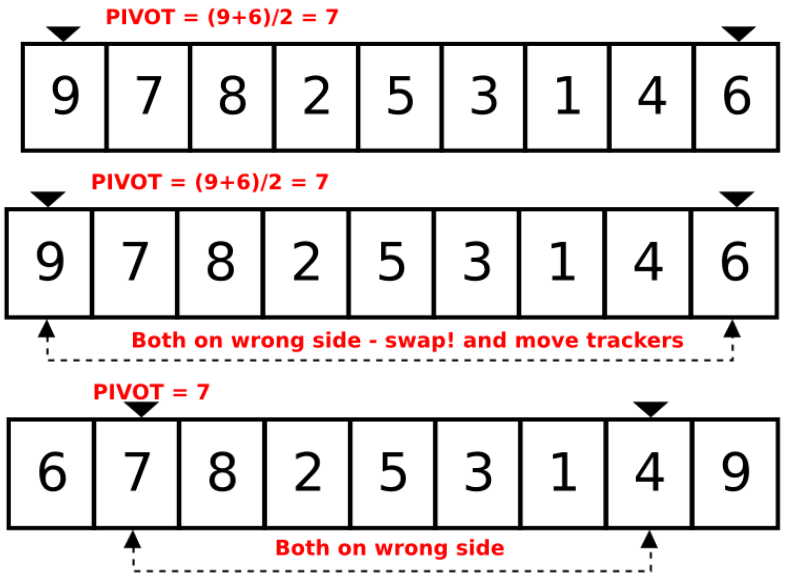


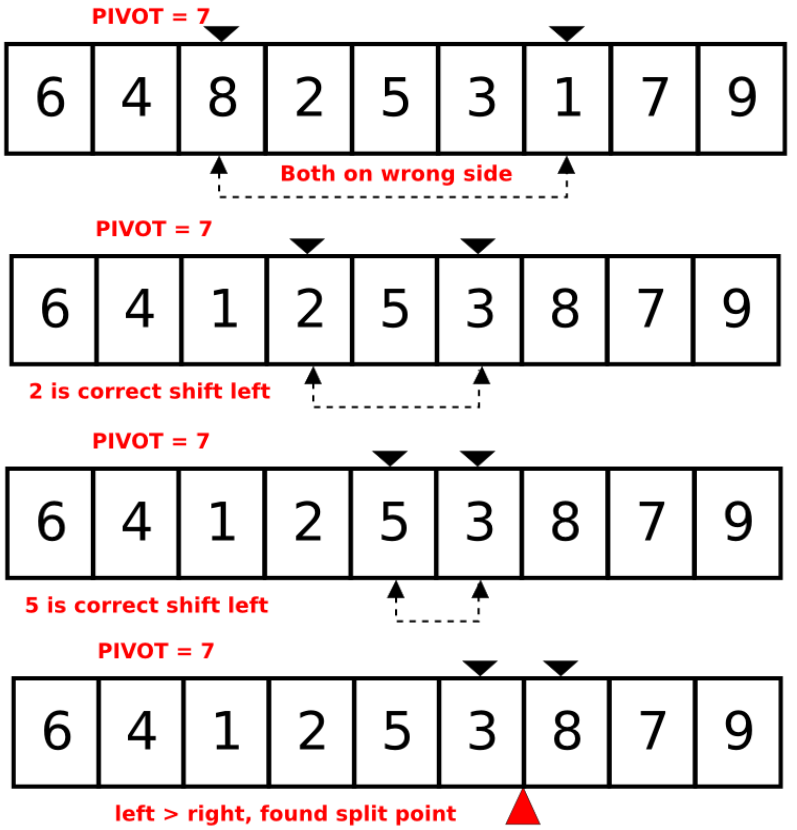
Therefore any savings we had by dividing is lost, and possibly made worse! Memory is cheap!

Quick Sort

Quick Sort is also a **divide-and-conquer** algorithm, but unlike merge sort that splits the data in half on size, we will split on a value we think (estimate to be in the middle) known as the **pivot** value, placing values smaller than the pivot to the left and larger values on the right.

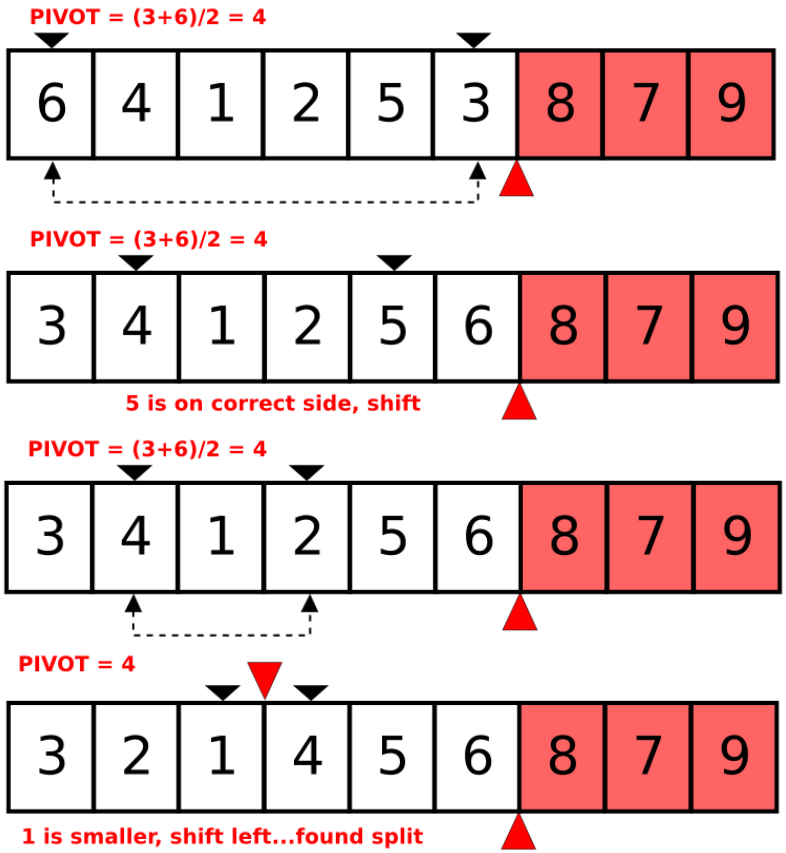
Begin by choosing a pivot (we take midpoint of values at index 0 and length-1)

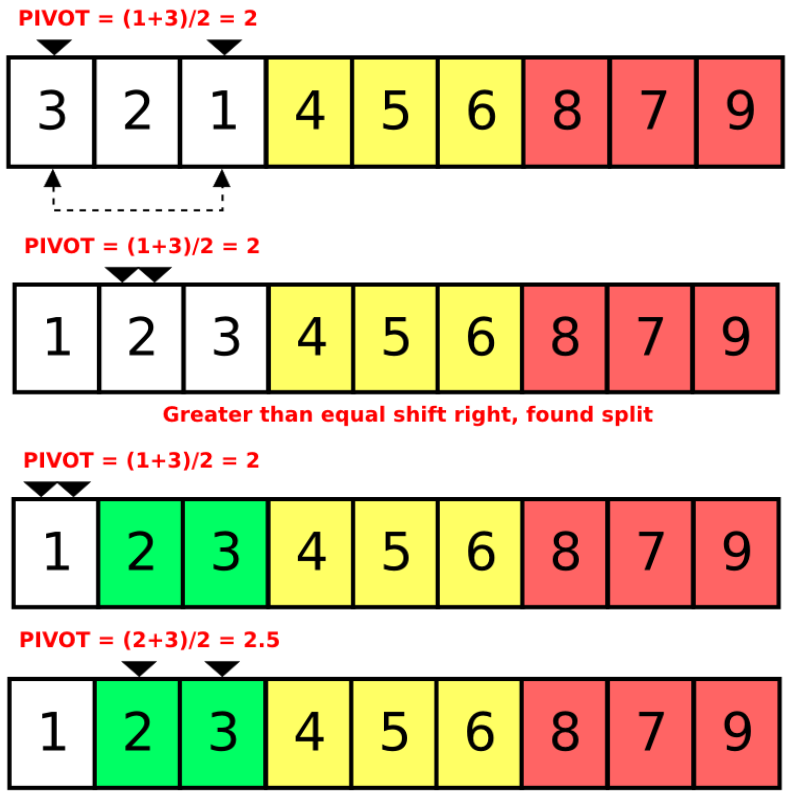




Quick Sort - 2

Continue split and reorganizing...





Continue to process each split that has not gone down to single/two entry sections.

Quick Sort Algorithm - 1

```
1  //Precond: Array is full, and sIndex and eIndex are in-bounds
2  //Postcond: Array will be sorted between the specified indices
3  public void quickSort(int[] toSort, int sIndex, int eIndex){
4       //section of size 0 or 1
5       if(eIndex<=sIndex+1) return;
6       if(eIndex-sIndex==2){ //Swap if needed...no reason to split
7            if(toSort[sIndex]>toSort[eIndex-1]){
8                 int temp = toSort[eIndex-1];
9                 toSort[eIndex-1] = toSort[sIndex];
10                toSort[sIndex] = temp;
11                return;
12           }
13      }
14
15      //find pivot
16      int pivot = (toSort[sIndex]+toSort[eIndex-1])/2;
17
18      //...Find split, by putting items on appropriate sides
19      //and then doing quick sort on split parts...
20 }
```

Quick Sort Algorithm - 2

```
1  //Precond: Array is full, and sIndex and eIndex are in-bounds
2  //Postcond: Array will be sorted between the specified indices
3  public void quickSort(int[] toSort, int sIndex, int eIndex){
4       //...Base Cases and Pivot computation...
5       //Create trackers for both sides
6       int leftInd = sIndex;
7       int rightInd = eIndex-1;
8
9       while(leftInd<=rightInd){
10           if(toSort[leftInd] >= pivot && toSort[rightInd] <
                 pivot){
11               int temp = toSort[leftInd];
12               toSort[leftInd++] = toSort[rightInd];
13               toSort[rightInd--] = temp;
14           }
15           if(toSort[leftInd]< pivot) leftInd++;
16           if(toSort[rightInd] >= pivot) rightInd--;
17      }
18      if(sIndex!=leftInd){
19           quickSort(toSort,sIndex,leftInd);
20           quickSort(toSort,leftInd,eIndex);
21      }
22 }
```

Quick Sort Runtime Analysis

Since we are looking at dividing a conquering we want to think about each aspect of the algorithm:

- How long to find the split?

  SPLIT: We are comparing each element to the pivot so $O(n)$

- How do we divide, when we find the split?

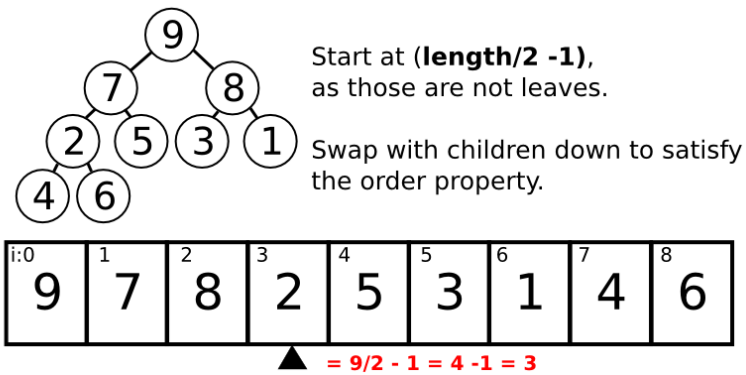
  RECURSION: Like merge sort, quick sort could split the data into two halves which would result in in $O(log_2 n)$ and typically does assuming a good pivot choice. However, it is entirely possible to find a pivot that splits each time into a 1-and-rest split, which would result in $O(n)$

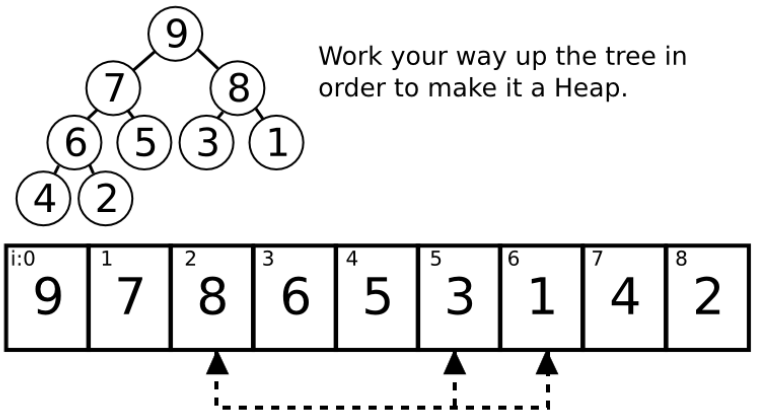
TOGETHER: Like merge sort since we are find the split at each recursion, it would be the product of the above values. Average case: $O(nlog_2 n)$, Worst-case: $O(n^2)$...But...

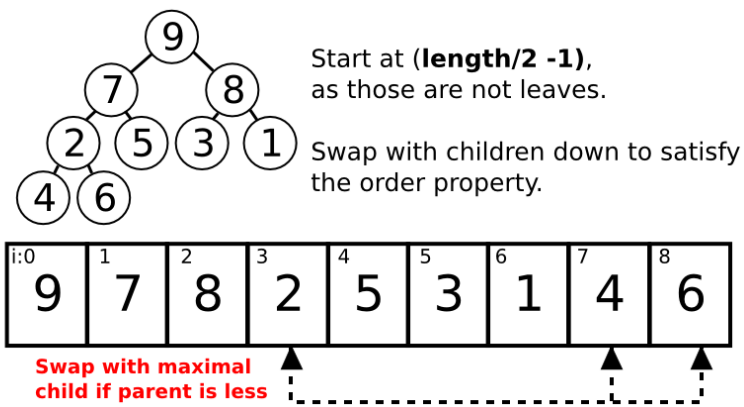
Unlike Merge-sort: no extra space requirements necessary

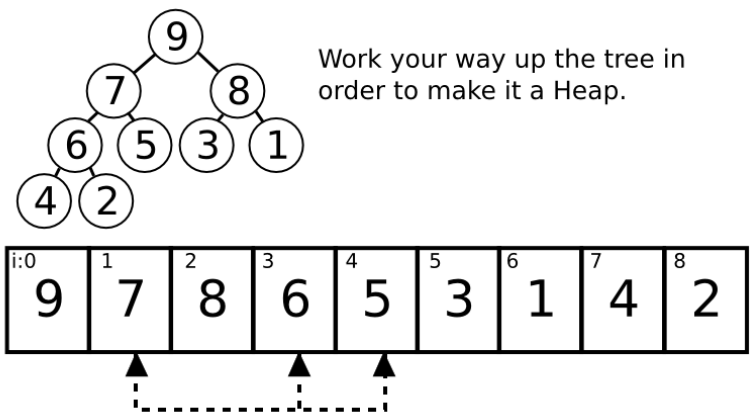
Heap Sort

Heap Sort uses the properties of Heaps to reorder the data by looking at the largest elements one at a time. First we must alter the data to be a heap, starting bottom-up with the subtree.

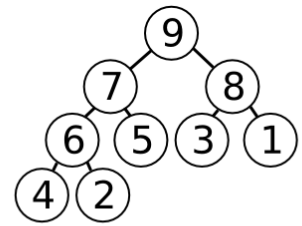




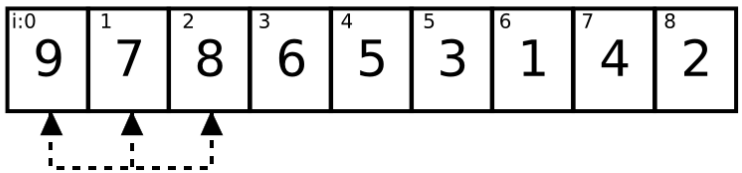


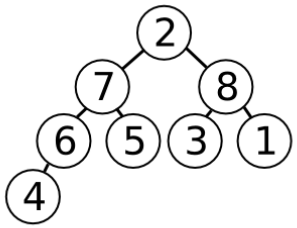


Heap Sort - 2

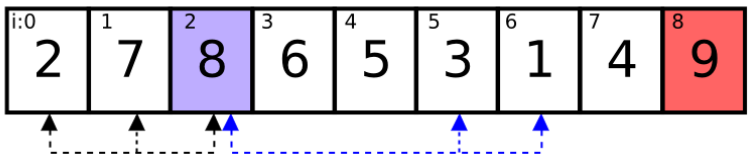


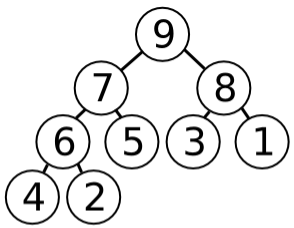
Work your way up the tree in order to make it a Heap.



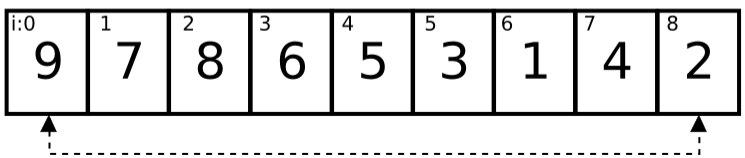


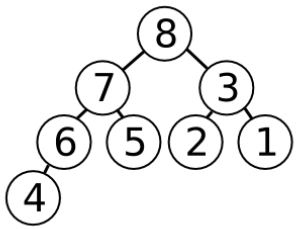
Now that it is a heap we can swap the largest to the end and shrink our heap. **Swapping the new root down our now smaller heap.**



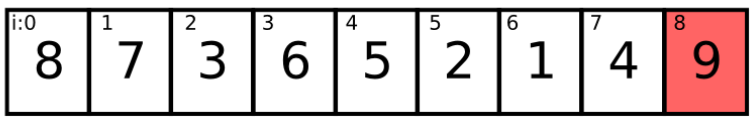


Now that it is a heap we can swap the largest to the end and shrink our heap. Swapping the new root down our smaller heap

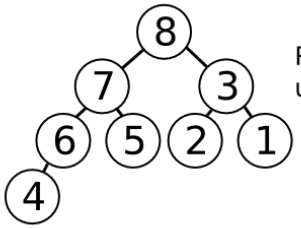




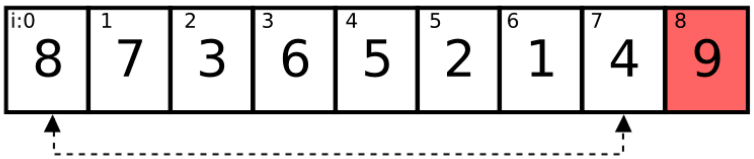
Now that it is a heap we can swap the largest to the end and shrink our heap. **Swapping the new root down our now smaller heap.**

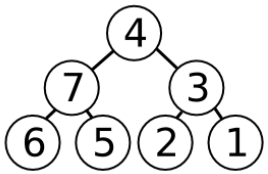


Heap Sort - 3

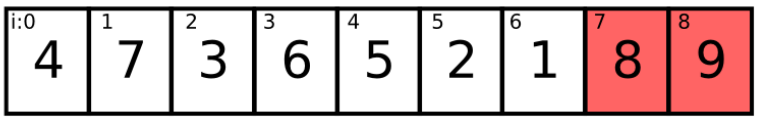


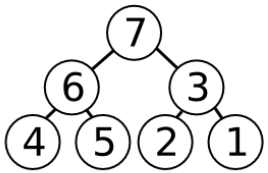
Repeat this process (swap, reheap) until the heap is "empty"



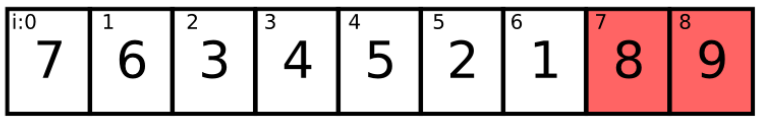


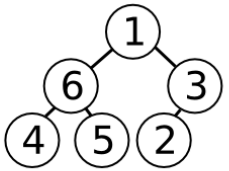
Reheap



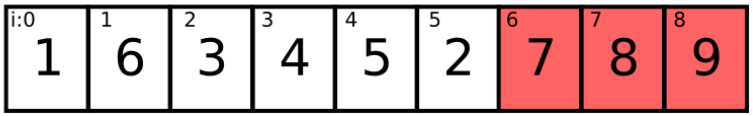


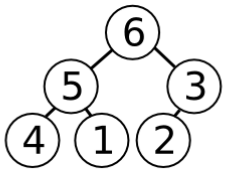
Swap & Shrink



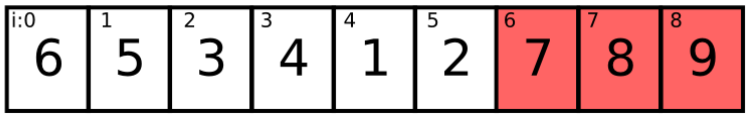


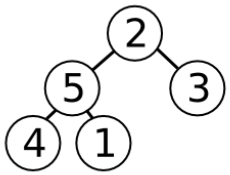
Reheap



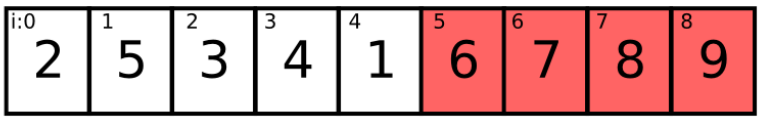


Switch & Shrink

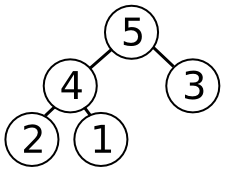




Reheap

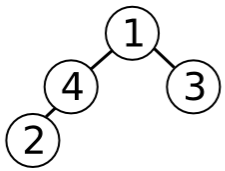


Heap Sort - 4



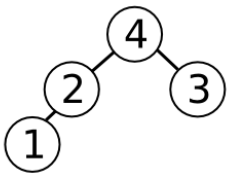
Swap & Shrink

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



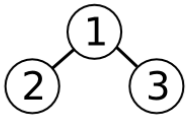
Reheap

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



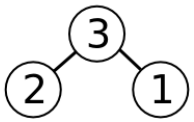
Swap & Shrink

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



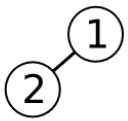
Reheap

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



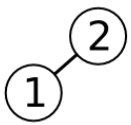
Swap & Shrink

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



Reheap

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



Swap & Shrink

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

Heap Sort Algorithm - 1

```
1  //Precond: Array has values from index to heapEnd
2  //Postcond: Specified subtree will be a heap
3  public void reheapDown(int[] toSort, int index, int heapEnd){
4        int leftInd = 2*index+1;
5        int rightInd = 2*index+2;
6        if(leftInd>=heapEnd) return; //it is a leaf
7        int maxInd = leftInd;
8        if(rightInd<heapEnd && toSort[rightInd]>toSort[leftInd])
9             maxInd = rightInd;
10
11       //got max child
12       if(toSort[index]<toSort[maxInd]){
13            int temp = toSort[index];
14            toSort[index] = toSort[maxInd];
15            toSort[maxInd] = temp;
16            reheapDown(toSort,maxInd,heapEnd);
17       }
18 }
```

Heap Sort Algorithm - 2

```
1  //Precond: Array is full
2  //Postcond: Array will be sorted
3  public void heapSort(int[] toSort){
4       int heapEnd = toSort.length;
5
6       //First make it a heap begining at length/2-1
7       for(int i=heapEnd/2-1; i>=0; i--){
8            reheapDown(toSort,i,heapEnd);
9       }
10
11      //Now we continually swap-shrink, and reheap
12      while(heapEnd>1){
13           int temp = toSort[--heapEnd];
14           toSort[heapEnd] = toSort[0];
15           toSort[0] = temp;
16          reheapDown(toSort,0,heapEnd);
17      }
18 }
```

Heap Sort Runtime Analysis

There are two aspects of this algorithm to examine as well.

- How long does it take to reheap?

  REHEAP: We know from studying heaps in the worst-case it can be swapped from the root to a leaf so $O(log_2 n)$

- How many swap,shrink, reheaps do we do?

  MAIN LOOP: We swap every element to the back of the array so we will loop $O(n)$ times.

TOGETHER: Since the reheap is done within the main loop, it is once again the product. So it will take $O(nlog_2 n)$ time.

INITIAL HEAP: A reheap is done on each of the upper half elements of the array to ensure we have it in heap format $(n/2 * O(log_2 n))$, which results in $O(nlog_2 n)$. And as the building is a separate step of the algorithm (not nested) the total time still takes $O(nlog_2 n)$.

Stability

A **Stable** sort is a sort that maintains the order of equal elements within for example sorting $\{B_1, A_1, A_2, B_2\}$ given $A_1 == A_2$ and $B_1 == B_2$.

A stable sort would result in $\{A_1, A_2, B_1, B_2\}$, whereas an unstable sort would be something like $\{A_1, A_2, B_2, B_1\}$.
Sorting techniques thus far:

- **Selection Sort** - Stable if finding min use $<$ not $<=$
- **Bubble Sort** - Stable if swap only when $<$ not $<=$
- **Shell Sort** - It is NOT stable as larger gaps may change order
- **Insertion Sort** - Stable if when inserting you insert when previous element is $<=$
- **Merge Sort** - Stable if adding left to temporary before right if they are equal
- **Quick Sort** - By definition NOT Stable
- **Heap Sort** - By definition NOT Stable