Sorting

Exchange Sorts

While all sorting methods operate by exchanging items, the exchange of two items is the dominant characteristic of an exchange sort.

Bubble Sort

```
BubbleSort(a)

do
   swapped = false
   for i = 0 to Length(a) - 2
      if a[i] > a[i + 1]
         temp = a[i]
         a[i] = a[i + 1]
         a[i + 1] = temp
         swapped = true
while swapped
```

Example 1:

0	1	2	3	4	5	6	7	8	9	10	11	12
81	94	11	96	12	35	17	95	28	58	41	75	15
81	11	94	12	35	17	95	28	58	41	75	15	96
11	81	12	35	17	94	28	58	41	75	15	95	
11	12	35	17	81	25	58	41	75	15	94		
		17	35	25	58	41	75	15	81			
			25	35	41	58	15	75				
						15	58					
					15	47						
				15	35							
			15	25								
		15	17									
11	**12**	**15**	**17**	**28**	**35**	**41**	**58**	**75**	**81**	**94**	**95**	**96**

Worst case running time occurs when in put is in reverse order and is $\mathrm{O}(n^2)$. *The best-case running time occurs when input is already sorted and is* $\mathrm{O}(n)$.

Example 2 (worst case running time)

0	1	2	3	4	5
64	51	34	32	21	8
51	34	32	21	8	64
34	32	21	8	51	
32	21	8	34		
21	8	32			
8	21				
8	**21**	**32**	**34**	**51**	**64**

Example 3 (best case running time)

0	1	2	3	4	5
8	21	32	34	51	64
8	**21**	**32**	**34**	**51**	**64**

Example 4

0	1	2	3	4	5
34	8	64	51	32	21
8	34	51	32	21	64
		32	21	51	
	32	21	34		
	21	32			
8	**21**	**32**	**34**	**51**	**64**

One simple improvement can be made on the bubble sort algorithm. Since, after each pass, the largest element in the array will be pushed into position, one can reduce the length of each pass by one each time:

```
BubbleSort (a)

n = Length(a) - 1
do
   n--
   swapped = false
   for i = 0 to n
      if a[i] > a[i + 1]
         temp = a[i]
         a[i] = a[i + 1]
         a[i + 1] = temp
         swapped = true
while swapped
```

This modification cuts the number of comparisons required from, in the worst case, from $n(n-1)$ *to* $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$. *Despite the reduction the algorithm remains in* $\mathrm{O}(n^2)$.

Quicksort

Quicksort is the fastest known sorting algorithm in practice with average running time of $O(n \log n)$.

Worst-case running time is $O(n^2)$.

The basic algorithm to sort an array a consists of four steps:

If the number of elements in a is 0 or 1, then return

Pick any element p in a as the pivot.

Partition $a - \{p\}$ into two disjoint sets

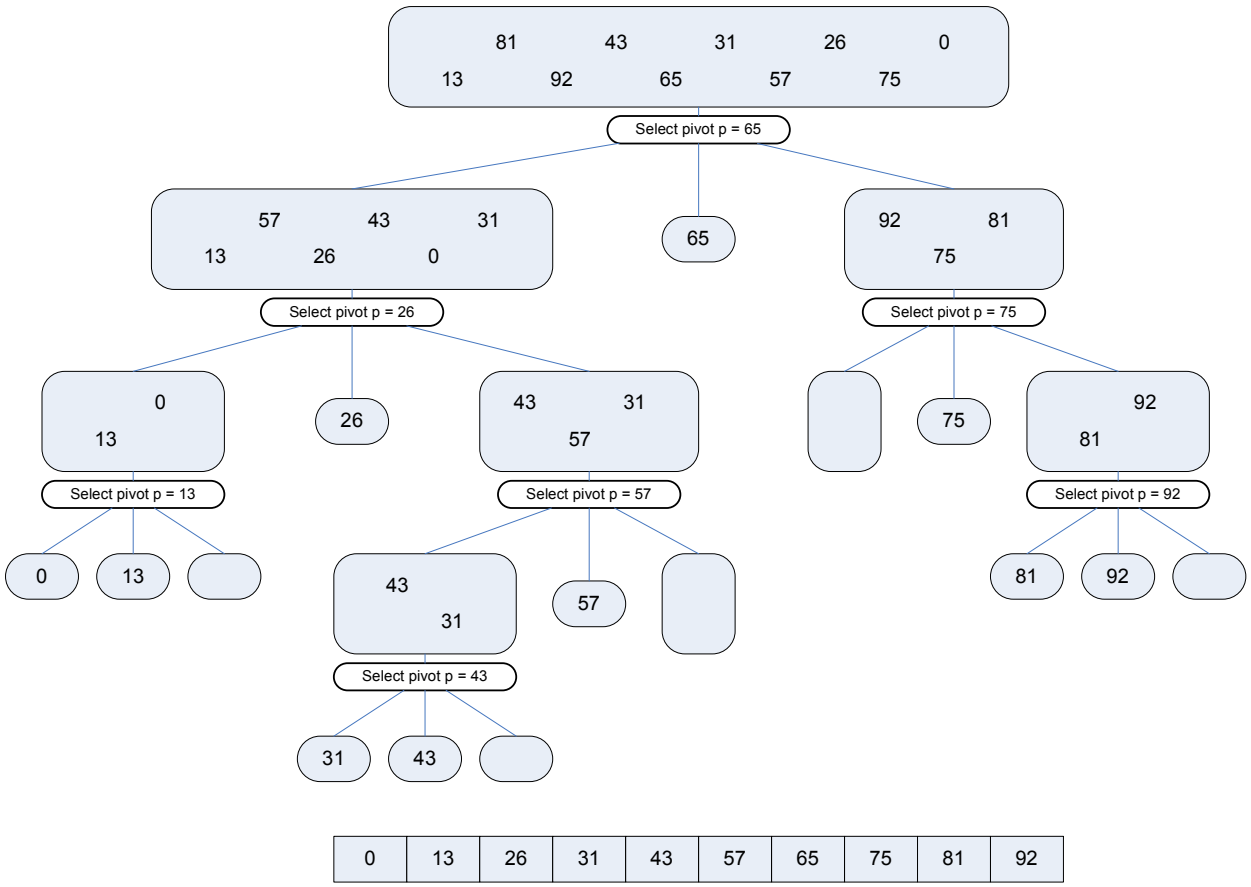
$a_1 = \{x \in a - \{p\} \mid x \leq p\}$

$a_2 = \{x \in a - \{p\} \mid x \geq p\}$

Note: This ambiguously states that items equal to the pivot would go into a_1 and a_2. However, since a_1 and a_2 must be disjoint, such items must go into either one set or the other. Ideally, a good implementation would put approximately half the elements equal to p in a_1, and half in a_2.

Return $\{\text{quicksort}(a_1), p, \text{quicksort}(a_2)\}$

General Example



The quicksort algorithm works regardless of the element chosen as the pivot. However:

Picking the first element as pivot would be easy, but for sorted, nearly sorted, or reverse order inputs, the performance of the algorithm is very poor (i.e. worst-case or near worst-case)

Picking an element randomly has been shown to have no effect on reducing the average running time.

Picking an element using median-of-three reduces the running time by five percent.

Median-of-three Partitioning

The median of a group of numbers is the $\frac{n}{2}^{\text{th}}$ largest number (and would make the best choice for the pivot)

Unfortunately, the median is hard to calculate and would slow down the sort.

Finding the median value of a set itself a $\mathrm{O}(n)$ *problem.*

A good estimate of the median has been found to be the median of the first, middle, and last elements

Example

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

`a[first] = a[0] = 6`

`a[middle] = a[`$\lfloor \frac{1}{2}(0+9) \rfloor$`] = a[`$\lfloor 4.5 \rfloor$`] = a[4] = 8`

`a[last] = a[9] = 0`

`Median = 6`

Partitioning Strategy

Swap the `pivot` with the last element.

`i` starts at the first element

`j` starts at the last element

Move `i` right skipping over elements that are smaller than pivot.

Move `j` right skipping over elements that are larger than pivot.

If `i` is to the left of `j`, swap elements `i` and `j`.

Repeat previous three steps until `i` and `j` cross

Swap elements `pivot` and `i`

Example

0	1	2	3	4	5	6	7	8	9	10	11	12
81	94	11	96	12	35	**58**	95	28	17	41	75	15
						15						58
i											j	
41										81		
	17								94			
			28					96				
						j	i					
							58					95
41	**17**	**11**	**28**	**12**	**35**	**15**	**58**	**96**	**94**	**81**	**75**	**95**

Quicksort

```
QuickSort (a)
QuickSort (a, 0, Length(a) - 1)

QuickSort (a, left, right)
if n < 10
   InsertionSort(&a[left], n)
   return
middle = (left + right) / 2
if a[middle] < a[left]
   Swap(a[middle], a[left])
if a[right] < a[middle]
   Swap(a[right], a[middle])
if a[middle] < a[left]
   Swap(a[middle], a[left])
pivot = a[middle];
Swap(a[middle], a[right - 1])
i = left
j = right - 1
while true
   while a[++i] < pivot
      –
   while pivot < a[--j]
      –
   if i < j
      Swap(a[i], a[j]);
   else
      break;
Swap(a[i], a[right - 1]);
QuickSort(a, left, i - 1);
QuickSort(a, i + 1, right);
```

Insertion sort is used when the number of elements in a partition is less than 10 because the quicksort overhead is too costly (in comparison).

Note that problems can occur in the partitioning depending how it's implemented:

This results in an infinite loop when `a[i] = a[j] = pivot`.

```
while (a[i] < pivot)
   i++
while (pivot < a[j])
   j--
```

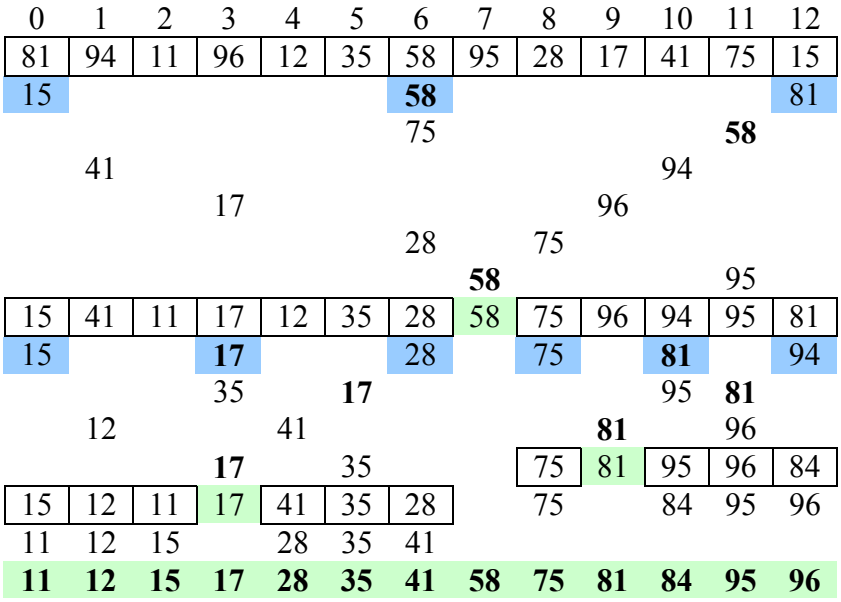
This does extra comparisons.

```
while (a[i] <= pivot)
   i++
while (pivot <= a[j])
   j--
```

This works as we would like

```
while (a[++i] < pivot)
   –
while (pivot < a[--j])
   –
```

Example



For demonstrative purposes, this example does not revert to insertion sort until partition size is three at which point the cost of the insertion sort is no more than the median-of-three portion of the quicksort.