Chapter 23

Sorting

Selection Sort

- Strategy (Ascending, Array)
  - Select the smallest element, place in proper position
  - Repeat step 1 with remaining items
- Does not depend on the initial arrangement of the data
- Only appropriate for small n

Selection Sort :: Code

```
public static void selectionSort(int[] list) {
   for (int top = 0; top <= list.length - 2; top++) {
     swap(list, top, findMinIndex(list, top));
   }
}
```

```
private static int findMinIndex(int[] list, int startAt) {
  int minIndex = startAt;
  for (int index = startAt + 1; index <= list.length - 1; index++) {
    if (list[index] < list[minIndex]) {
      minIndex = index;
    }
  }
  return minIndex;
}
```

```
private static void swap(int[] list, int index1, int index2) {
   int tmp = list[index1];
   list[index1] = list[index2];
   list[index2] = tmp;
}
```

Bubble Sort

- Strategy (Ascending, Array)
  - Compare adjacent elements ([i] and [i+1]) and exchange them if they are out of order([i] > [i+1])
  - Will sink the largest element to the end of the array
  - Repeating this process will eventually sort the array into ascending order

Bubble Sort :: Code

```
public static void bubbleSort(int[] list) {
   boolean sorted = false;
   int pass = 1;

   while (pass <= list.length - 1 && !sorted) {
      sorted = true;
      for (int i = list.length - 1; i >= pass; i--) {
         if (list[i] < list[i - 1]) {
            swap(list, i, i - 1);
            sorted = false;
         }
      }
      pass += 1;
   }
}
```

Insertion Sort

- Strategy (Ascending, Array)
  - Place item [i] where i = 1 in its proper location with relation to items [0]...[i-1]
  - Repeat step 1 with rest of items [2]...[n-1]

Insertion Sort :: Code

```
public static void insertionSort(int[] list) {
   int holdItem;
   int j;

   for (int i = 1; i <= list.length – 1; i++ ) {
      holdItem = list[i];
      for (j = i – 1; j >= 0 && list[j] > holdItem; j--) {
         list[j + 1] = list[j];
      }
      list[j + 1] = holdItem;
   }
}
```

MergeSort

- A recursive sorting algorithm
- Gives the same performance, regardless of the initial order of the array items
- Strategy
  - Divide an array into halves
  - Sort each half
  - Merge the sorted halves into one sorted array

MergeSort :: Code

```
public static void mergeSort(int[] list) {

   if (list.length > 1) {
      int n2 = list.length / 2 + 1;
      int[] leftHalf = new int[n2];
      System.arraycopy(list, 0, leftHalf, 0, n2);
      mergeSort(leftHalf);

      int n = list.length - n2;
      int[] rightHalf = new int[n];
      System.arraycopy(list, list.length / 2, rightHalf, 0, n);
      mergeSort(rightHalf);

      merge(leftHalf, rightHalf, list);
   }
}
```

MergeSort :: Code

```
public static void merge(int[] left, int[] right, int[] list) {
  int leftIndex = 0;
  int rightIndex = 0;
  int listIndex = 0;

  while (leftIndex < left.length && rightIndex < right.length) {
    if (left[leftIndex] < right[rightIndex])
      list[listIndex++] = left[leftIndex++];
    else
      list[listIndex++] = right[rightIndex++];
  }

  while (leftIndex < left.length)
    list[listIndex++] = left[leftIndex++];

  while (rightIndex < right.length)
    list[listIndex++] = right[rightIndex++];
```

QuickSort

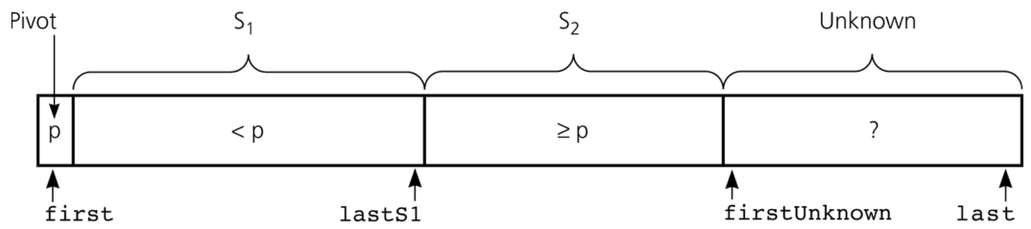
- A divide-and-conquer algorithm
- Strategy
  - Partition an array into items that are less than the pivot and those that are greater than or equal to the pivot
  - Sort the left section
  - Sort the right section

QuickSort: Algorithm

```
QuickSort(array:ArrayType, first:int, last:int) {
   if (first < last) {
      Choose a pivot item p from array[first..last]
      Partition the items of array[first..last] about p

      // the partition is array[first..pivotIndex..last]
      // sort S1
      QuickSort(array, first, pivotIndex -1)

      // sort S2
      QuickSort(array, pivotIndex + 1, last)
   }
```

QuickSort: Partition Invariant

- Use the invariant to develop a partition algorithm
  - The items in region S_1 are all less than the pivot, and those in S_2 are all greater than or equal to the pivot

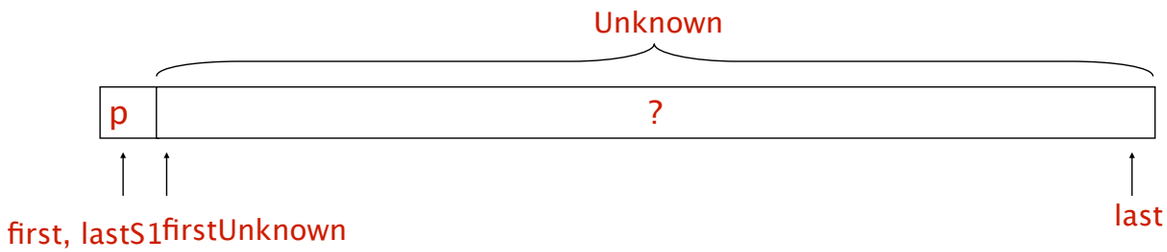


Partition: Algorithm

```
Partition(array:ArrayType, first:int, last:int, pivotIndex:int)
   choose the pivot and swap it with array[first]
   p = array[first]

   lastS1 = first
   firstUnknown = first + 1

   while (firstUnknown <= last) {
      if (array[firstUnknown < p)
         Move array[firstUnknown] into S1
      else
         Move array[firstUnknown] into S2
   }

   swap array[first] with array[lastS1]
   pivotIndex = lastS1
```

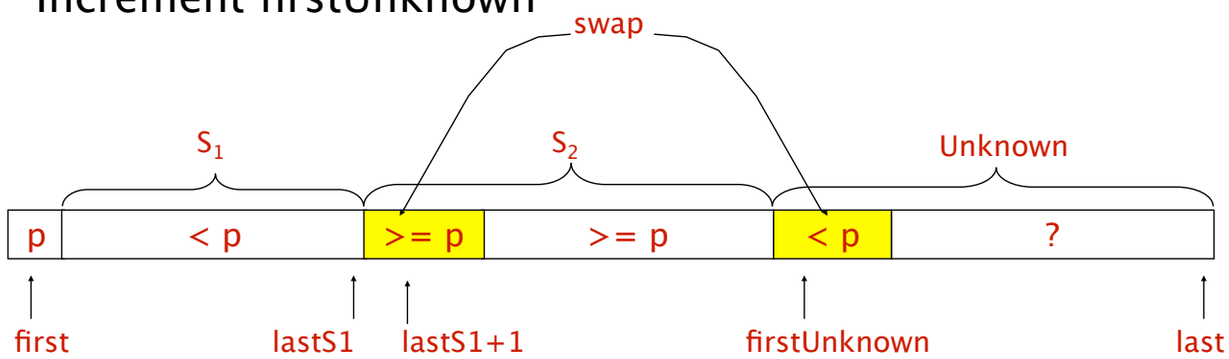
Partition: Initial Array State

```
lastS1 = first
firstUnknown = first + 1
```



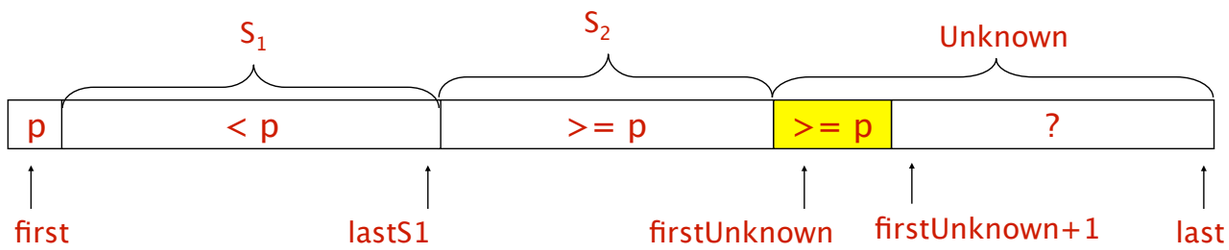
Partition: Move array[firstUnknown] into S_1

swap array[firstUnknown] with array[lastS1 + 1]
Increment lastS1
Increment firstUnknown

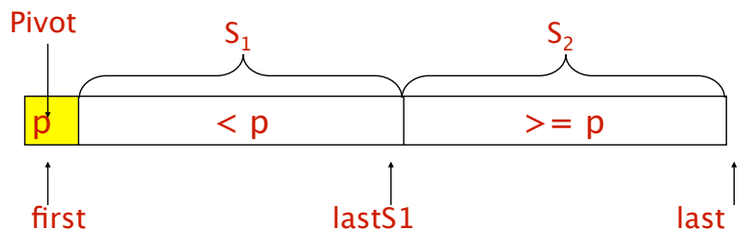


Partition: Move array[firstUnknown] into S_2

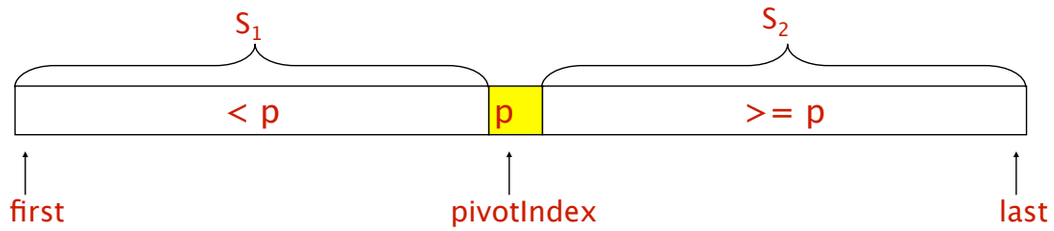
Increment firstUnknown



Partition: Final Task



swap array[first] with array[lastS1]
pivotIndex = lastS1



Heapsort: Strategy

- Transforms the array into a heap
- Removes the heap's root (the largest element) by exchanging it with the heap’s last element
- Transforms the resulting semiheap back into a heap

Heapsort

- Compared to mergesort
  - Both heapsort and mergesort are O(n * log n) in both the worst and average cases
  - However, heapsort does not require a second array

- Compared to quicksort
  - Quicksort is O(n * log n) in the average case
  - It is generally the preferred sorting method, even though it has poor worst-case efficiency : $O(n^2)$

Radix: Traced

0123, 2154, 0222, 0004, 0283, 1560, 1061, 2150	Original integers
(156**0**, 215**0**) (106**1**) (022**2**) (012**3**, 028**3**) (215**4**, 000**4**)	Grouped by fourth digit
1560, 2150, 1061, 0222, 0123, 0283, 2154, 0004	Combined
(00**0**4) (02**2**2, 01**2**3) (21**5**0, 21**5**4) (15**6**0, 10**6**1) (02**8**3)	Grouped by third digit
0004, 0222, 0123, 2150, 2154, 1560, 1061, 0283	Combined
(0**0**04, 1**0**61) (0**1**23, 2**1**50, 2**1**54) (0**2**22, 0**2**83) (1**5**60)	Grouped by second digit
0004, 1061, 0123, 2150, 2154, 0222, 0283, 1560	Combined
(**0**004, **0**123, **0**222, **0**283) (**1**061, **1**560) (**2**150, **2**154)	Grouped by first digit
0004, 0123, 0222, 0283, 1061, 1560, 2150, 2154	Combined (sorted)

Radix Sort: n decimal integers, d digits each

```
RadixSort(array:ArrayType, n:int, d:int)
   for (j = d down to 1) {
      Initialize 10 groups to empty
      Initialize a counter for each group to 0

      for (i = 0 through n-1) {
         k = jth digit of array[i]
         Place array[i] at the end of group k
         Increase kth counter by 1
      }

      Replace the items in array with all items in group 0, followed by all
      the items in group 1 and so on.
   }
```