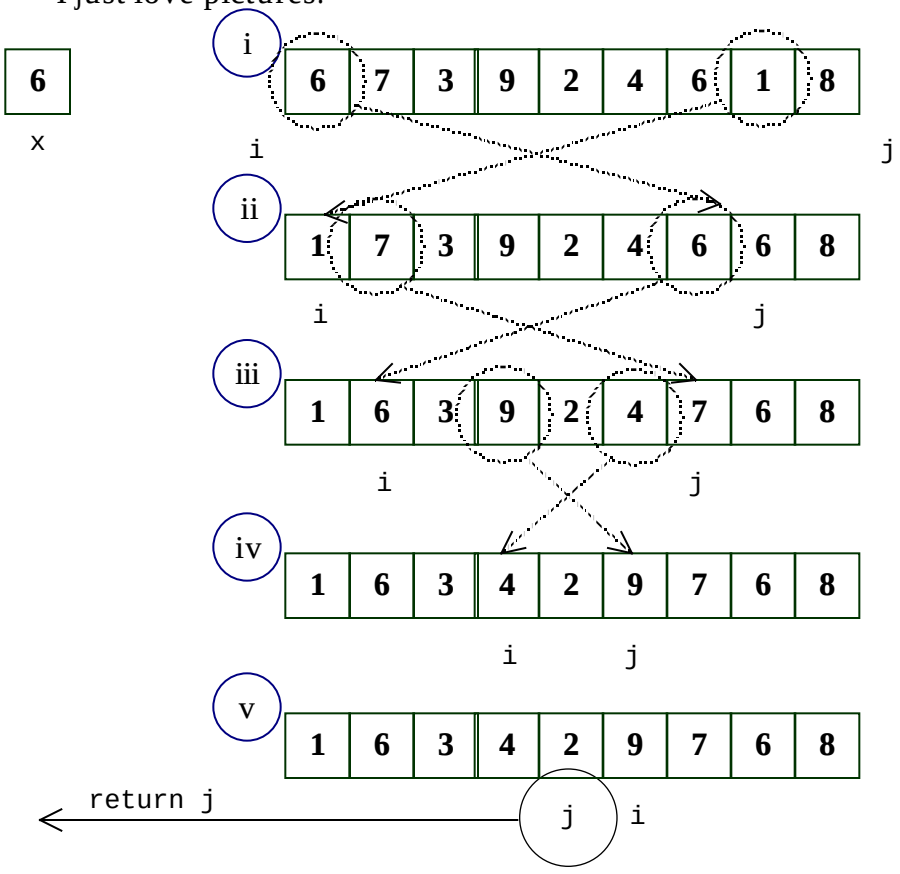
Solution Set 3: Searching and Sorting

Problem 1: Understanding QuickSort (10 points)

- Using Figure 8.1 on CLR, p. 155 as a model, illustrate the operation of `Partition` on the array `A = [6,7,3,9,2,4,6,1,8]`. Rather than use the version of **`Partition`** I used in lecture, simply use the one presented on page 154.

I just love pictures!



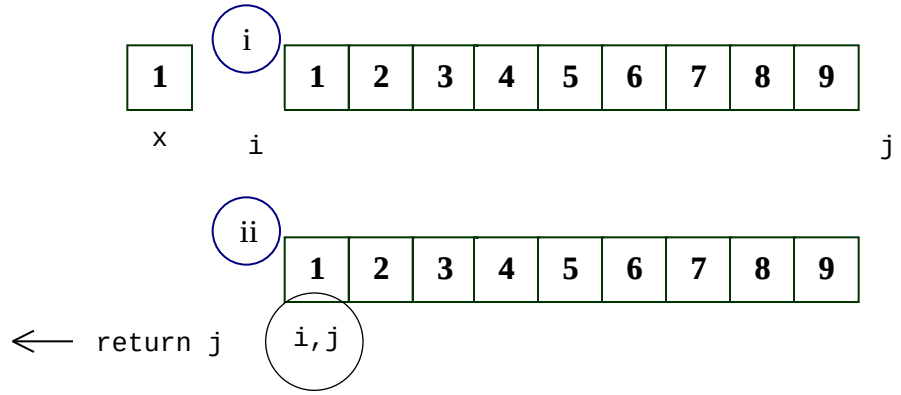
- Show that the running time of **`Quicksort`** is $\Theta(n \lg n)$ when all elements of array `A` are equal. Use the **`Partition`** from the book.

When all of the elements are equal, each iteration of **`Partition`** advance **`i`** and **`j`** toward each other a total of two indices. This marching toward each other is constant for both variables with each iteration, until **`i`** indexes the midpoint of the array and **`j`** indexes either the same element or the one preceding it. Because all of the elements are equal, Partition certainly could not have failed, and because **`j`** indexes the middle element, the return value prompts **`QuickSort`** to be call on two subarrays of either exactly or nearly exactly half the size of the original. The divide-and-conquer strategy of this particular case would be replicated throughout the entire computation, and therefore the would be perfectly modelled by the recurrence and the solution you've all come to know and adore:

$$T(n) = \begin{cases} \Theta(1) & n \le 2 \\ 2T(n/2) + \Theta(n) & \text{otherwise} \end{cases}$$
$$= \Theta(n \lg n)$$

- Show that the running time of **`QuickSort`** is $\Theta(n^2)$ when array `A` is sorted in increasing order. . Again, use the **`Partition`** from the book.

The initial call to **`QuickSort`**—or more specifically, the first call to **`Partition`**—will choose the pivot as what is a minimum element. On the very first iteration, **`i`** advances just one element, but **`j`** proceeds to march the entire length of the array from back to front until it is also blocked by the pivot. No swapping is performed, and instead the value of **`p`** is returned. That becomes an instruction to recursively sort **`A[p]`** through **`A[p]`** (but that's easy), and then recursively sort the already sorted **`A[p+1]`** through **`A[r]`**. The length of this second part is just one shy of the original, and the sorted nature of the subarray is precisely the same.



The $\Theta(n)$ it takes to run **`Partition`** is pretty much the same regardless of what the data looked like going in. If **`Partition`** makes a tragic choice for the pivot, then the full running time is affected dramatically. Even worse, the larger of the two subproblems above displays exactly the same features that foiled the original one—it's sorted. Anti-rage.

The recurrence:

$$T(n) = \begin{cases} \Theta(1) & n \le 2 \\ T(n-1) + \Theta(n) & \text{otherwise} \end{cases}$$
$$= \Theta(n^2)$$

Problem 2: Stack Depth for Quicksort (20 points, courtesy Problem 8-4 on CLR, p. 169)

The **`QuickSort`** algorithm of Section 8.1 contains two recursive calls to itself. After the call to **`Partition`**, the left subarray is recursively sorted and then the right half is recursively sorted. The second recursive call in **`QuickSort`** is not really necessary; it can be avoided by using an iterative control structure. This technique, called tail recursion, is provided automatically by good compilers. Consider the following version of **`QuickSort`**, which simulates tail recursion.

```
Quicksort(A,p,r)
   while p < r
      do q ← Partition(A,p,r)
         Quicksort(A,p,q)
         p ← q + 1
```

· Argue that **`QuickSort`**`(A,1,length[A])` correctly sorts the array A.

Notice that the difference between the above algorithm and that presented in Section 8.2 of the text differ primarily in the conversion of an **`if`** statement to a **`while`** statement and the removal of a recursive call in preference to an update step on the loop invariant. The correctness of the algorithm hinges on the belief that the update of **`p`** and the subsequent return to the top of the while loop body are exactly what are needed to emulate the recursive call. To see that this is the case, simply note that on the very last line **`p`** assumes the value that the second argument of a recursive call would assume if the recursive call were being made.

Compilers usually execute recursive procedures by using a **stack** that contains pertinent information, include the parameter values, for each recursive call. The information for the most recent recursive call is at the top of the stack, and the information for the initial call is at the bottom. When a procedure is invoked, its information is **pushed** onto the

stack; when it terminates, its information is **popped** from the stack. Since we assume that array parameters are actually represented by pointers, the information for each procedure call on the stack requires $\Theta(1)$ stack space. The stack depth is the maximum amount of stack space used at any time during a computation.

- Describe a scenario in which the stack depth of our new **`QuickSort`** is $\Theta(n)$ on an n-element array.

  This worst-case scenario results from the situation where **`Partition`** just happens to always pivot around the maximum element between indices **`p`** and **`r`**, inclusive, and as a result returning and binding **`q`** to one less than the incoming value of **`r`**.

- Modify the code for our new **`QuickSort`** so that the worst-case stack depth is $\Theta(\lg n)$.

  The easiest (and admittedly, very clever) thing to do is to only make the recursive call on the smaller of the two portions to be sorted, and let the while loop contract on the larger portion of the array.

  Behold:

```
Quicksort(A,p,r)
   while p < r
      do q ← Partition(A,p,r)
         if r - q > q - p  // if latter is greater than former
            Quicksort(A,p,q)
            p ← q + 1
         else
            Quicksort(A,q + 1,r)
            r ← q
```

  Notice the introduction of the **`if`** test to determine whether the partition index **`q`** happens to be closer to **`p`** or **`r`**, and based on that which portion should be recursively sorted and which portion sort be sorted iteratively. The maximum number of recursive calls results when **`q`** is categorically returns as the mean of **`p`** and **`r`**. In that case, the number of recursive calls currently active can be as many as the number of times it takes to divide and array of length **`n`** in half before it becomes an array of length 1. We all know from our analysis of the binary search algorithm that that number is $\Theta(\lg n)$.

Problem 3: Finding the majority element (10 points)

Consider an integer array of length n. If a single number occurs more than $\lceil n/2 \rceil$ times, we call that number the **majority** element. If no such number occurs, then the array doesn't have a majority element. Note that an array can have at most one majority element.

- Present a $\Theta(n \lg n)$-algorithm that returns the majority element if such an element exists. If no such element exists, then the algorithm can return anything at all.

  Sort the array using a $\Theta(n \lg n)$-time sort (`HeapSort`, `MergeSort`, `QuickSort` where version of `Partition` chooses median using techniques of Section 10.3), and then return the median element. If a majority element exists, then the median slot must contain it. If you want to confirm that the element is in fact the majority element, you can iterate through the array, count the number of times the alleged majority element exists—if this count is greater than $\lceil n/2 \rceil$, then you have your majority element. The verification can be done in $\Theta(n)$, so that the overall running time is dominated by the sorting itself.

- Present another algorithm that does exactly the same thing, but improves on the running time and runs in $\Theta(n)$ worst-case time.

  Rather than sorting the entire array, just find the median using the Select algorithm as discussed in Section 10.3. Then verify whether or not the median is a majority element or not in exactly the same way you would had you sorted the array to find the median. In this case, both the search and the verification take $\Theta(n)$ time, so the overall algorithm is $\Theta(n)$ as well. Pretty sneaky, huh?

Problem 4: Quick and Dirty Algorithms (10 points)

Consider a set S of $n \geq 2$ distinct numbers given in unsorted order. Each of the following five problem parts asks you to give an algorithm to determine two distinct numbers x and y in the set S that satisfy a stated condition. In as few words as possible, describe your algorithm and justify the running time. To keep your answers brief, use algorithms from lectures and the book as subroutines.

- In $O(n)$ time, determine $x, y \in S$ such that $|x - y| \geq |w - z|$ for all $w, z \in S$.

  In English: determine the two values that are the furthest apart. Take the $O(n)$ time to iterate over all of the elements and keep track of the maximum and minimum values. Section 10.1 talks about this; let x be the maximum and y be the minimum. Ta da!

- In $O(n \lg n)$ time, determine $x, y \in S, x \neq y$ such that $|x - y| \leq |w - z|$ for all $w, z \in S, w \neq z$.

  Sort using **`QuickSort`** (or **`HeapSort`** or **`MergeSort`** if you're worried about the $O(n^2)$ worst-case running time) and then traverse the collection from front to end keeping track of the two neighboring elements closest to each other—that is, after sorting, iterate and track the minimum gap between consecutive elements.

- In $O(n)$ expected time, determine $x, y \in S$ such that $x + y = Z$ where Z is given, or determine that no two such numbers exist. *Hint: use a hashtable, where insertion and lookup take expected constant time.*

  Place all numbers into a hash table of size n. The expected hashing time is $n \cdot O(1) = O(n)$, assuming simple uniform hashing and collisions resolved by chaining, for example. Then, for each $x \in S$, look for $Z - x$ in the hash table. It takes at most n lookups (each running in $O(1)$ expected time) to either find one or to try them all before failing. Thus, the expected total time is $O(n)$.

- (Extra credit: 5 points) In $O(n)$ time, determine $x, y \in S$ such that $|x - y| \leq \frac{1}{n-1}\left(\max_{z \in S} z - \min_{z \in S} z\right)$—that is, determine any two numbers that are at least as close together as the average distance between consecutive numbers in sorted order. *Hint: use divide-and-conquer*.

  Here's the idea: Repeatedly split the set in half by partitioning around the median and recursively look in the half that has the smaller average distance. That half must have a pair of numbers at least as close together as the original average distance, since for any set of numbers (or in this case, distances) at least one number (distance) must be less than or equal to the average. When the recursion gets down to just two numbers, those two are the answer. One subtle point is that the median must be included on both sides of the partition, so that it will be in a subset with each of its neighbors; otherwise, if the correct response includes that median, it might otherwise not be found.

  Algorithm: Input in array A.
  Define: $average\text{-}distance[A] = \frac{1}{n-1}\left(\max_{z \in A} z - \min_{z \in A} z\right)$, where $n = length[A]$. This takes $\Theta(n)$ time (to find the max and min.)

  Our answer is given by **`Find(A)`**, where

```
Find(A)
   if length[A] = 2
      then return A
   Find median x of A in linear time
   Partition A around x into B and C, including
      X in both B and C.
   Calculate average-distance of B and C in linear time
   if average-distance[B] < average-distance[C]
      then return Find[B]
      else return Find[C]
```

The running time is given by our neighborly recurrence relation:

$$T(n) = \begin{cases} \Theta(1) & n \le 2 \\ T(n/2) + \Theta(n) & \text{otherwise} \end{cases}$$
$$= \Theta(n)$$

(solved by the master method, case 3!)