COP 3502: Computer Science I
Spring 2004

– Note Set 12 –
Searching and Sorting – Part 3

Instructor : Mark Llewellyn
markl@cs.ucf.edu
CC1 211, 823-2790
http://www.cs.ucf.edu/courses/cop3502/spr04

School of Electrical Engineering and Computer Science
University of Central Florida

QuickSort

- Quicksort is another recursive sorting algorithm.
- It picks an element from the list and uses it to divide the list into two parts. It then places every element which is smaller than the selected element into the first part and every element which is larger than the selected element is placed into the second part.
- Before we look at Quicksort in any detail, we'll first examine a related problem that will help you understand how and why Quicksort works.

Problem Related to QuickSort: Selection

- We've already examined the problem of searching a list for a specific target element. There is a closely related problem to this called selection.
- The selection problem is similar to the searching problem except where searching is concerned with finding a specific target element in the list, selection is concerned with finding an element which exhibits a certain property, such as the largest, smallest, or median value.
- In general, the selection problem solves the problem of finding the k^{th} largest value in a list.

Problem Related to QuickSort: Selection (cont.)

- One way to solve the selection problem would be to sort the list in decreasing order, and then the *k*th largest value would be in position *k*.
- However, this is a lot more work that we really need to do, because we don't really care about the values that are smaller than the one we want.
- A related technique would be to find the largest value and then move it to the last location in the list. If we again look for the largest value in the list ignoring the value we already found, we get the second largest value, which can be moved to the next to the last location in the list. This process would repeat until we had found the *k*th largest value which would occur on the *k*th pass.
- The algorithm for this is shown on the next page.

FindKthLargest Algorithm

```
FindKthLargest (list, n, k)
   // list      the list of elements to look through
   // n         the number of elements in the list
   // k         the element to select

for (i = 1; i <= k, i++) {
     largest = list[1];
     largestLocation = 1;
     for ( j = 2, j <= n-(i-1); j++){
          if listpj[ > largest then {
               largest = list[j];
               largestLocation = j;
          }
     }
     swap( list[n-(i-1)], list[largestLocation]);
}
end.
```

Complexity of FindKthLargest

- On the first pass a total of *n-1* comparisons would be made. The second pass would total *n-2* comparisons, and so on. On the k^{th} pass a total of *n-k* comparisons would occur.

- This gives: $\sum_{i=1}^{k} n - i = n \times k - \frac{k(k-1)}{2} = O(k \times n)$

- Note that this equation also tells us that if *k* is greater than *n/2*, it would be faster for look for the $n\text{-}k^{th}$ smallest value.

- This algorithm is reasonably efficient for values of *k* that are close to either end of the list, but there is a more efficient way to accomplish this process for values of *k* that are close to the middle of the list. Can you think of a technique that would accomplish this?

Complexity of FindKthLargest (cont.)

- Since we are only interested in the *k*th largest value, we don't really need to know the exact position for the values that are the largest through the *k-1*st largest; we only need to know that they are larger...their position in the list is irrelevant.

- If we choose an element from the list and partition the list into two distinct parts, one containing those elements that are larger than the chosen element and the other containing those elements that are smaller than the chosen element.

- The chosen element will wind up in some position *p* in the list. This will mean that the chosen element is the *p*th largest element in the list.

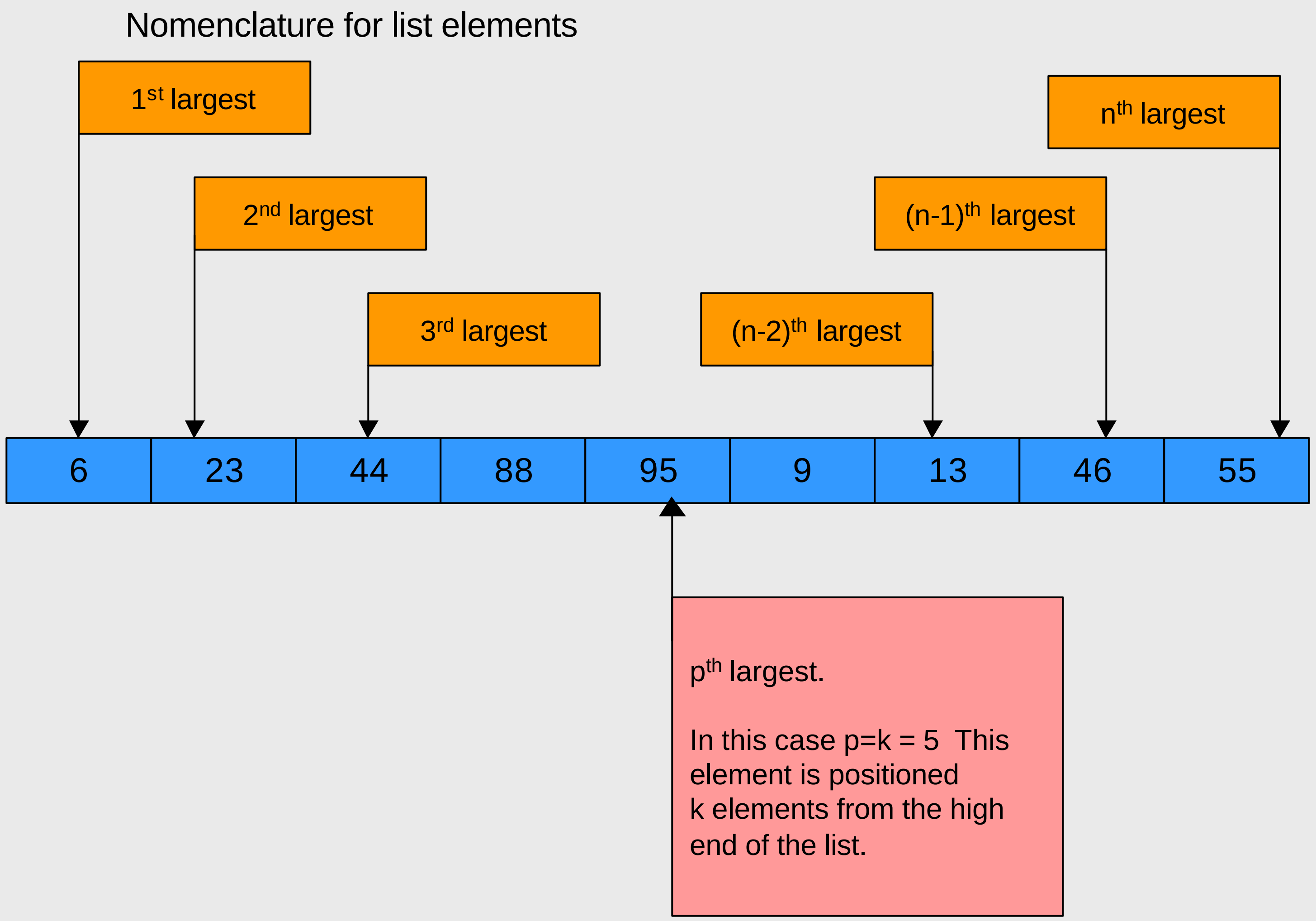
Complexity of FindKthLargest (cont.)

- To accomplish this partitioning, we'll need to compare the chosen element with all of the other elements, requiring *n-1* comparisons.

- If we are lucky and $p = k$, then we're done. If $k < p$, then we need a smaller value to partition on (a smaller *p*) and we'll repeat the process on a second partitioning. If $k > p$, then we need a larger value to partition on (a larger *p*) and we'll use the first partition, but we'll need to reduce the value of *k* by the number of values we've eliminated in the larger partition.

- The following few pages illustrates this technique.

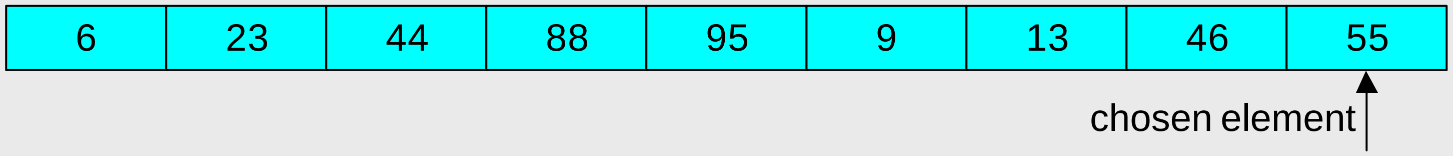
Complexity of FindKthLargest (cont.)


Nomenclature for list elements
1st largest
2nd largest
3rd largest
(n-2)th largest
(n-1)th largest
nth largest
6
23
44
88
95
9
13
46
55
pth largest.
In this case p=k = 5 This element is positioned k elements from the high end of the list.

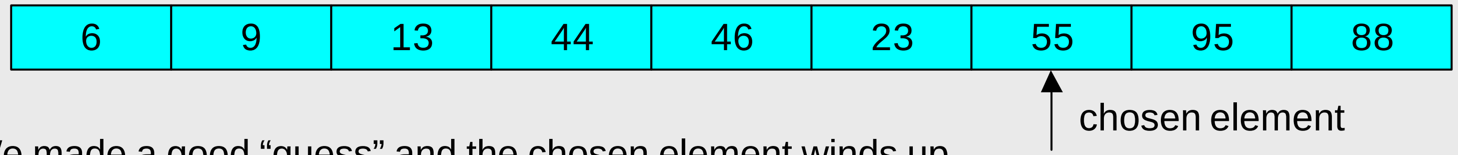
Complexity of FindKthLargest (cont.)

Suppose we are looking for the 7th largest element (k= 7)

Initial List



First partitioning



We made a good "guess" and the chosen element winds up as the 7th largest element which is what we wanted. Element 55 winds up in 7th position, so k = p =7.

Complexity of FindKthLargest (cont.)

Suppose we are looking for the 6th largest element (k= 6)

Initial List

6	23	44	88	95	9	13	46	55

chosen element

First partitioning

6	9	13	23	46	44	55	95	88

chosen element

We made a poor "guess" and the chosen element winds up as the 3rd largest element (p = 3), which is smaller than we wanted. Since k > p, we'll do a second partitioning of the larger numbers.

Second partitioning

6	9	13	23	44	46	55	95	88

ignore these elements this time

chosen element

We made a good "guess" this time and the chosen element winds up as the 6th largest element. Since p=k=6, we are done.

Complexity of FindKthLargest (cont.)

Suppose we are looking for the 2nd largest element

Initial List

6	23	44	88	95	9	13	46	55

chosen element

First partitioning

6	9	23	44	13	46	88	95	55

chosen element

We made a poor "guess" and the chosen element winds up as the 6th largest element (p=6) which is larger than what we wanted. Since k < p, we'll repartition, but select an element smaller than 46.

Second partitioning

6	9	13	44	23	46	88	95	55

new chosen element

ignore these elements this time

We made a good "guess" the second time and the chosen element winds up as the 2nd largest element (p=2) which is what we wanted.

Recursive FindKthLargest Algorithm

```
KthLargestRecursive (list, n, k)
    // list      the list of elements to look through
    // start     the index of the first element to consider
    // end       the index of the last element to consider
    // k         the element of the list we want

if start < end {
       Partition( list, start, end, middle);
       if middle = k
            return list[middle];
       else
            if k < middle
                 return KthLargestRecursive( list, middle+1, end, k);
            else
                 return(KthLargestRecursive( list, start, middle-1, k-middle);
 }
end.
```

Complexity of KthLargestRecursive

- If we assume that on average the partitioning process will divide the list into two roughly equal halves, then approximately:

  $$n + n/2 + n/4 + n/8 + \ldots + 1$$

  will be be necessary.

- This is about 2n comparisons. So the process is linear and independent of *k*.

Quicksort

- Quicksort is another recursive sorting algorithm.
- It picks an element from the list, called the pivot element, and uses it to divide the list into two parts. The first part contains all of the elements that are smaller than the pivot element and the second part of the list contains all of the elements that are larger than the pivot element.
- This process is similar to the one we just saw utilized when searching for the k^{th} largest element in a list. The only difference is that quicksort applies the technique recursively to both parts of the list.
- Quicksort is a very efficient sort on average, although its worst case performance is identical to insertion sort and bubble sort at $O(n^2)$.

Quicksort (cont.)

- Once the pivot element is selected (more on this later) the list is rearranged so that elements smaller than the pivot element are moved before it and elements larger than it are moved after the pivot element.
- The elements in each of the two parts of the list are **not** put in order.
- If the pivot element winds up in location *i*, all we know for certain is that elements in locations 1 through *i-1* are smaller than the pivot element and those in locations *i+1* through *n* are larger than the pivot element.
- Once this is done, quicksort is called recursively on these two parts.

Quicksort (cont.)

- If quicksort is called with a list containing one element, it does nothing (base case) because a one-element list is sorted by default.
- Because the determination of the pivot element and the movement of the elements into the proper section do all the work, the main quicksort algorithm just needs to keep track of the bounds of these two sections of the list.
- Further, because splitting the list into two parts is where the keys are moved around, all the sorting work is done on the way down in the recursive process. **IMPORTANT NOTE:** This is exactly opposite of merge sort, which does its work on the way back up in the recursive process.
- The algorithm for quicksort is shown on the next page.

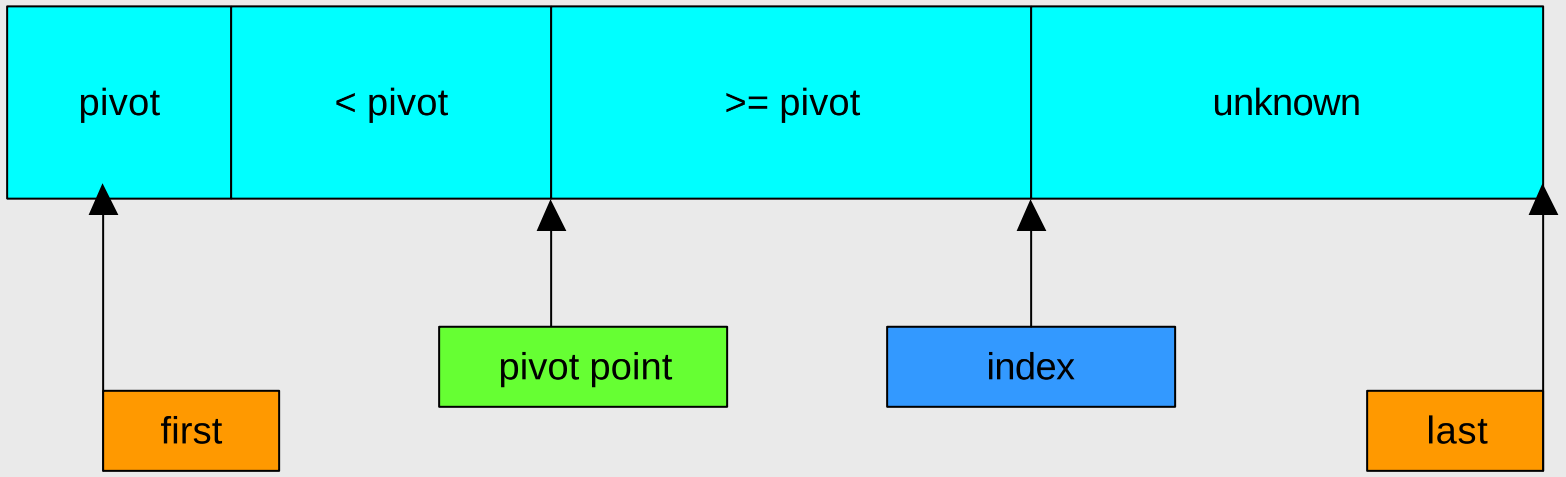
Quicksort Algorithm

```
Quicksort (list, first, last)
   // list        the list of elements to be sorted
   // first       the index of the first element in the part of the list to sort
   // last        the index of the last element to in the part of the list to sort.

if first < last {
      pivot = PivotList( list, first, last);
      Quicksort( list, first, pivot-1);
      Quicksort( list, pivot+1, last);
}
end.
```

PivotList Algorithm

```
PivotList (list, first, last)
    // list        the list of elements to work with
    // first       the index of the first element
    // last        the index of the last element

pivotvalue = list[first];
pivotpoint = first;
for (index = first+1, index <= last, index++) {
      if list[index] < pivotvalue {
           pivotpoint = pivotpoint+1;
           swap( list[pivotpoint+1, list[index]);
      }
}
//move pivot value into correct place in list
swap( list[first], list[pivotpoint]);
return pivotpoint;
end.
```

Relationship between indices and element in Algorithm PivotList



Splitting the List for Quicksort

- There are at least two versions of the PivotList function.
- The version shown on page 19 is easy to program and understand . There is another version which is more complicated to write but is faster than the version on page 19. I'll show you this version at the end of this section of notes, but we'll do our analysis of the quicksort algorithm using the simple partitioning algorithm from page 19.
- The diagram on page 20 illustrates how the PivotList algorithm partitions the list into two distinct sublists. Note that this simple algorithm simply chooses the first element in the list as the pivot element.

Splitting the List for Quicksort (cont.)

- Since the first element in the list is selected as the pivot element, the location of the pivot element is initially set to 1.
- The algorithm then moves through the list comparing this pivot element to the rest of the elements. Whenever it finds an element that is smaller than the pivot element, it will increment the pivot point and then swap this element into the pivot point location.
- After some of the elements have been compared to the pivot inside the loop, there will be four basic parts to the list as shown by the diagram on page 20.

Splitting the List for Quicksort (cont.)

- The first part of the list is simply the pivot element in the first location.
- The second part of the list is from location *first+1* through the pivot location and will consist of all the elements that have been compared which are smaller than the pivot element.
- The third part of the list is from the location after the pivot location through the loop index and will consist of all compared elements which are larger than the pivot element.
- The fourth part of the list consists of all the elements which have not yet been compared.
- These four parts are illustrated in the diagram on page 20.

Worst Case Analysis for Quicksort

- When PivotList is called with a list of *n* elements, it does *n-1* comparisons as it compares the pivot value with every other element in the list.
- Since we know that quicksort is a divide and conquer algorithm, we would be correct in assuming that the best case performance would occur when PivotList creates two equal sized parts to the original list.
- The worst case performance would then logically occur when the two parts of the list are of drastically different sizes. The worst case occurs when one part has no elements and the other part contains *n-1* elements.
  – If the same thing were to occur each time we partitioned the list we would remove only one element from the list on each recursive call (the pivot element).

Worst Case Analysis for Quicksort (cont.)

- Given this worst case scenario, PivotList would do a number of comparisons given by:

$$\text{worst}(n) = \sum_{i=2}^{n} (i-1) = \frac{n(n-1)}{2}$$

- What original ordering of the elements would cause this sort of behavior?
  - If each pass chooses the first element, that element must be the smallest (or largest). A list that is already sorted is one arrangement that would cause this worst case performance.
- All of the other sorts that we have seen, the worst case and average cases have been about the same (equal asymptotically), but as we are about to see, this is not the case for quicksort.

Average Case Analysis for Quicksort

- All of the work of quicksort is done by PivotList, so here is where we need to look for the average case. It is possible for each of the *n* locations in the list to be the location of the pivot element. So we need to see what happens for each of these possibilities and average the results.

- When looking at the worst case, we saw that for a list of *n* elements, *n-1* comparisons were necessary to divide the list. There is no work to put the lists back together. Also notice that when PivotList returns a value of *p*, quicksort is called recursively with lists of *p-1* and *n-p* elements. Our average case needs to look at all *n* possible values for *p*, which gives:

$$\text{average}(n) = (n-1) + \frac{1}{n}\left(\sum_{i=1}^{n}\left[\text{average}(i-1) + \text{average}(n-i)\right]\right) \quad \text{for } n \geq 2$$

$$\text{average}(1) = \text{average}(0) = 0$$

Average Case Analysis for Quicksort (cont.)

- If you look closely at the summation, you will notice the first term is used with values from 0 through *n-1* and the second term is used with values from *n-1* down to *0*. This means that the summation adds up every value of the average from 0 to *n-1* twice. This leads to the following simplification:

$$\text{average}(n) = (n-1) + \frac{1}{n}\left(2\sum_{i=1}^{n}\text{average}(i)\right) \quad \text{for } n \geq 2$$

$$\text{average}(1) = \text{average}(0) = 0$$

Average Case Analysis for Quicksort (cont.)

- This is a very complicated form of a recurrence relation because it depends on not just one smaller value of the average, but rather on every smaller value for the average!

- There are two basic ways to solve such a recurrence relation: (1) come up with an educated guess for the answer and then prove that this answer does satisfy the recurrence relation, or (2) look at the equations for both average(n) and average(n-1). Since these two equations differ by only a few terms: Computing average(n) × n and average(n-1)×(n-1) gets rid of the fractions which gives:

Average Case Analysis for Quicksort (cont.)

$$\text{average}(n)\times n=(n-1)n+2\sum_{i=0}^{n-1}\text{average}(i)=(n-1)n+2\times\text{average}(n-1)+2\sum_{i=0}^{n-2}\text{average}(i)$$

$$\text{average}(n-1)\times(n-1)=(n-2)(n-1)+2\sum_{i=0}^{n-2}\text{average}(i)$$

Subtracting the second equation from the first and simplifying gives:

$$\text{average}(n)\times n-\text{average}(n-1)\times(n-1)=2\times\text{average}(n-1)+(n-1)n-(n-2)(n-1)$$

$$\text{average}(n)\times n-\text{average}(n-1)\times(n-1)=2\times\text{average}(n-1)+n^2-n-\left(n^2-3n+2\right)$$

$$\text{average}(n)\times n-\text{average}(n-1)\times(n-1)=2\times\text{average}(n-1)+2n-2$$

Adding average(n-1)× (n-1) to both sides, we get:

$$\text{average}(n)\times n=2\times\text{average}(n-1)+\text{average}(n-1)\times(n-1)+2n-2$$

$$\text{average}(n)\times n=\text{average}(n-1)\times(2+n-1)+2n-2$$

Average Case Analysis for Quicksort (cont.)

This gives our final recurrence relation:

$$\text{average}(n) = \frac{(n+1)\times \text{average}(n-1) + 2n - 2}{n}$$
$$\text{average}(1) = \text{average}(0) = 0$$

Solving this is not too difficult but does require care because all of the terms on the right-hand side of the equation. If you work through all of the details (we're not going to!), you'll see the final result is:

$$\text{average}(n) \approx 1.4(n+1)\log_2 n$$

Summary of Quicksort

- Quicksort is the fastest known comparison based sorting algorithm.

Quicksort	
best case	$O(n \log_2 n)$
average case	$O(n \log_2 n)$
worst case	$O(n^2)$

Alternative PivotList Function

- A faster alternative for the PivotList function would be to have two indices into the list. The first moves up from the low end of the list and the second moves down from the high end of the list.

- The main loop of the algorithm will advance the lower index until a greater value than the pivot element is found, and the upper index is moved until a value less than the pivot element is found. When this situation occurs these two elements are swapped. The process repeats until the two indices cross over each other.

- These inner loops are very fast because the overhead of checking for the end of the list is eliminated, but one problem is that an extra swap occurs when the indices cross.

PivotList Algorithm

```
FastPivotList (list, first, last)
   // list      the list of elements to work with
   // first     the index of the first element
   // last      the index of the last element

pivotvalue = list[first];
lower = first;
upper = last +1;
do
     do upper = upper–1 until list[upper] <= pivotvalue
     do lower = lower+1 until list[lower] >= pivotvalue
     swap( list[upper], list[lower])
until lower >= upper
//undo the extra exchange
swap( list[upper], list[lower]);
//move pivot point into correct location
swap( list[first], list[upper])
return upper
end.
```

This algorithm requires that the list have one extra location to hold a special sentinel value which is larger than all of the valid key fields in the list.

C Implementation of Quicksort

```
// Code to demonstrate the Partition algorithm.
 #include <stdio.h>
#include <stdlib.h>
#include <time.h>

void create(int *values, int n );
int partition(int *values, int start, int end);
void swap(int *a, int *b);
void print(int *values, int n);

int main() {
      int *nums, mid;

     srand(time(0));
     create(nums, 30);
     mid = partition(nums, 0, 29);

     printf("The index of the partition element is %d\n", mid);
     print(nums, 30);
}
```

C Implementation of Quicksort (cont.)

```
void create(int *values, int n ) {
    int i;
    values = malloc(n*sizeof(int));
    for (int i=0; i<n; i++)
        values[i] = rand()%100;
    }
}

void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

void print(int *values, int n) {
    int i;
    for (i=0; i<n; i++) printf("%d ", values[i]));
    printf("\n");
}
```

C Implementation of Quicksort (cont.)

```
int partition(int *values, int start, int end) {

    // Line up left and right counters.
   int i = start;
   int j = end;

   while (i < j) {
        // Move left counter, then the right counter.
       while (i <= end && values[i] <= values[start])
           i++;
       while (values[j] > values[start])
           j--;
        // Swap out of place values.
       if (i < j)
           swap(values+i, values+j);
   }

    swap(values+start, values+j); // Swap in partition element.
   return j;
}
```

Alternative Pivot Determination Schemes

- Since it's clearly important to get a reasonable "split" when doing a quicksort, there are many different techniques that can be used to ensure a reasonable split of values in the partition step. (We won't look at the implementations, just the ideas. But, you should be able to implement these ideas in code if you ever had to.)

- One technique is to randomly pick three elements in the list to be sorted as candidates for the partition element. Then, choose the middle value of these three elements to be the partition.
  - There is some extra expense here - picking three elements and then doing three comparisons to determine the median of the values, but hopefully, if the array being sorted is large enough, this extra expense will be small enough compared to the gains of a better partition element.

Alternative Pivot Determination Schemes (cont.)

- Clearly, you would not want to do this if you were only sorting 10 or 20 values. In fact, quicksort is most efficient if you implement some simple sort such as insertion sort when you get down to a few elements, say 10 or 20. (This would be your terminating condition in the recursive method.)
- A more precise technique is to pick 5 random elements, determine the median of these five elements and then pick that as the partition element. This can be done in a maximum of 7 comparisons. This will generally give you a better partition element than the median of three technique. Depending on the size of the array being sorted, this extra cost may be worth it.
- The most precise technique would be to determine the median of all elements in the list to be sorted. However, the improvement in run-time rarely justifies the expense of this technique when compared with the median of three or median of five techniques.