

Sorting: Quick Sort



Computer Science Department
University of Central Florida

COP 3502 – Computer Science I



Sorting: Quick Sort

■ Quick Sort

- Most common sort used in practice
- Why?
 - cuz it is usually the quickest in practice!
- Quick Sort uses two main ideas to achieve this efficiency:
 - 1) The idea of making partitions
 - 2) Recursion

- Let's look at the partition concept...



Sorting: Quick Sort

- Quick Sort – Partition
 - **A partition works as follows:**
 - Given an array of n elements
 - You must manually select an element in the array to partition by
 - You must then compare ALL the remaining elements against this element
 - If they are greater,
 - Put them to the “right” of the partition element
 - If they are less,
 - Put them to the “left” of the partition element



Sorting: Quick Sort

■ Quick Sort – Partition

■ A partition works as follows:

- Once the partition is complete, what can we say about the position of the partition element?
- We can say (we KNOW) that **the partition element is in its CORRECTLY sorted location**
- In fact, after you partition the array, you are left with:
 - all the elements to the left of the partition element, in the array, that still need to be sorted
 - all the elements to the right of the partition element, in the array, that still need to be sorted
- **And if you sort those two sides, the entire array will be sorted!**



Sorting: Quick Sort

- Quick Sort

- Partition:

- Essentially breaks down the sorting problem into two smaller sorting problems
 - ...what does that sound like?

- Code for Quick Sort (at a real general level):

- 1) Partition the array with respect to a random element
- 2) Sort the left part of the array using Quick Sort
- 3) Sort the right part of the array using Quick Sort

- Notice there is no “merge” step like in Merge Sort

- at the end, all elements are already in their proper order



Sorting: Quick Sort

■ Quick Sort

■ Code for Quick Sort (at a real general level):

- 1) Partition the array with respect to a random element
- 2) Sort the left part of the array using Quick Sort
- 3) Sort the right part of the array using Quick Sort

■ Quick Sort is a recursive algorithm:

- We need a base case
 - A case that does NOT make recursive calls
- Our base case, or terminating condition, will be when we sort an array with only one element
 - We know the array is already sorted!



Sorting: Quick Sort

■ Quick Sort

■ Let S be the input set.

1. If $|S| = 0$ or $|S| = 1$, then **return**.

2. Pick an element v in S . Call v the **partition element**.

3. Partition $S - \{v\}$ into two disjoint groups:

- $S_1 = \{x \in S - \{v\} \mid x \leq v\}$

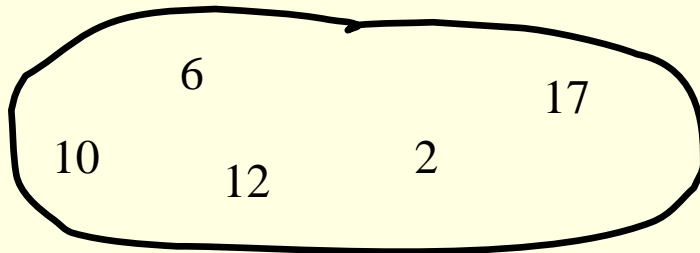
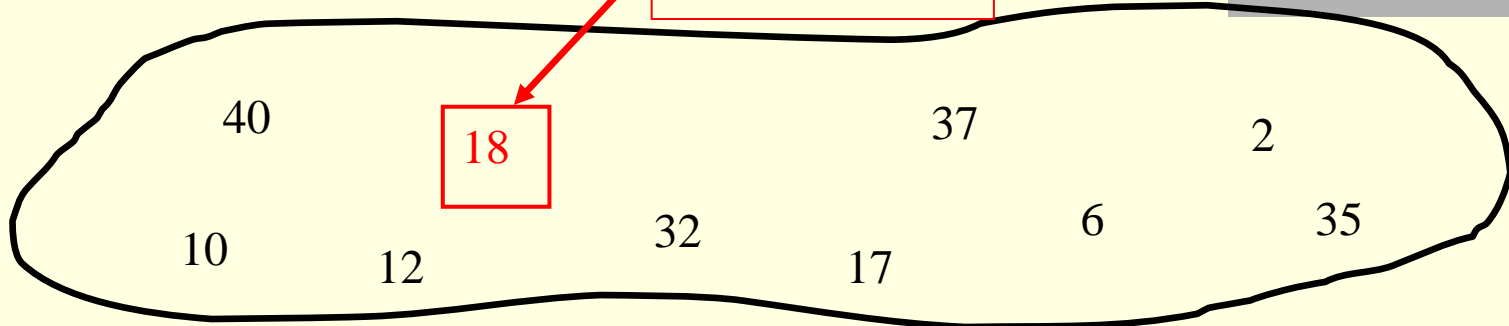
- $S_2 = \{x \in S - \{v\} \mid x \geq v\}$

4. **Return** { quicksort(S_1), v , quicksort(S_2) }

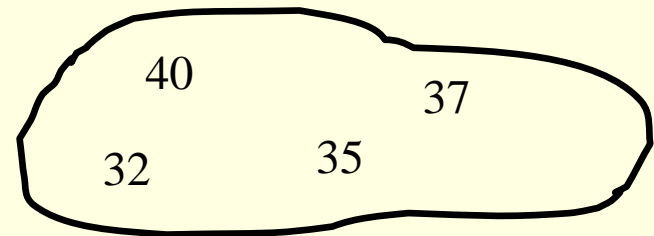
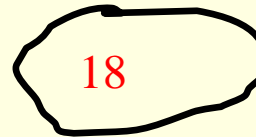


Sorting: Quick Sort

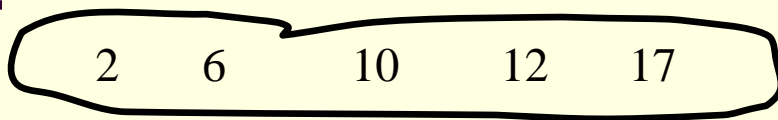
pick a pivot



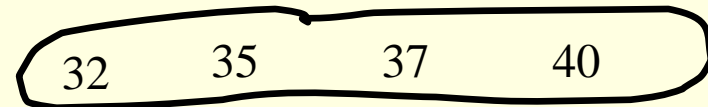
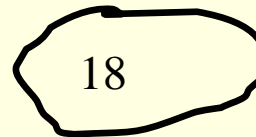
partition



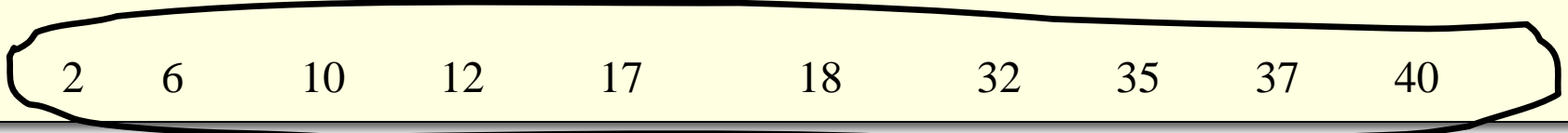
quicksort



quicksort



combine





Sorting: Quick Sort

- The idea of “in place”
 - In Computer Science, an “in-place” algorithm is one where the output usually overwrites the input
 - There is more detail, but for our purposes, we stop with that
 - Example:
 - Say we wanted to reverse an array of n items
 - Here is a simple way to do that:

```
function reverse(a[0..n]) {  
    allocate b[0..n]  
    for i from 0 to n  
        b[n - i] = a[i]  
    return b  
}
```



Sorting: Quick Sort

- The idea of “in place”

- Example:

- Say we wanted to reverse an array of n items
 - Here is a simple way to do that:

```
function reverse(a[0..n]) {  
    allocate b[0..n]  
    for i from 0 to n  
        b[n - i] = a[i]  
    return b  
}
```

- Unfortunately, this method requires $O(n)$ extra space to create the array b
 - And allocation can be a slow operation



Sorting: Quick Sort

- The idea of “in place”

- Example:

- Say we wanted to reverse an array of n items
- If we no longer need the original array a
- We can overwrite it using the following in-place algorithm

```
function reverse-in-place(a[0..n])  
    for i from 0 to floor(n/2)  
        swap(a[i], a[n-i])
```

- Many Sorting algorithms are in-place algorithms
- Quick sort is NOT an in-place algorithm
- BUT, the Partition algorithm can be in-place



Sorting: Quick Sort

- How to Partition “in-place”

- Consider the following list of values that we want to partition

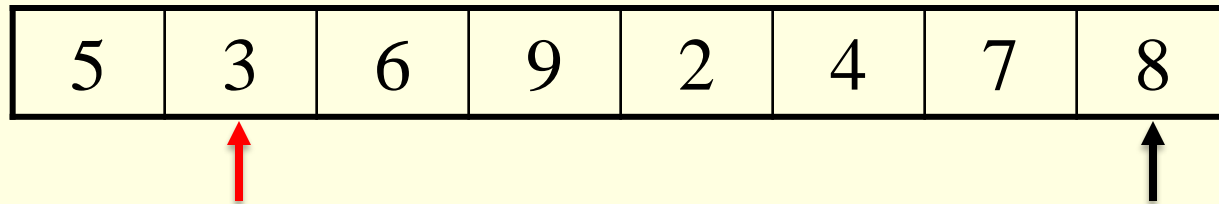
5	3	6	9	2	4	7	8
---	---	---	---	---	---	---	---

- Let us assume for the time being that we will partition based on the first element in the array
- The algorithm will partition these elements “in-place”



Sorting: Quick Sort

■ How to Partition “in-place”



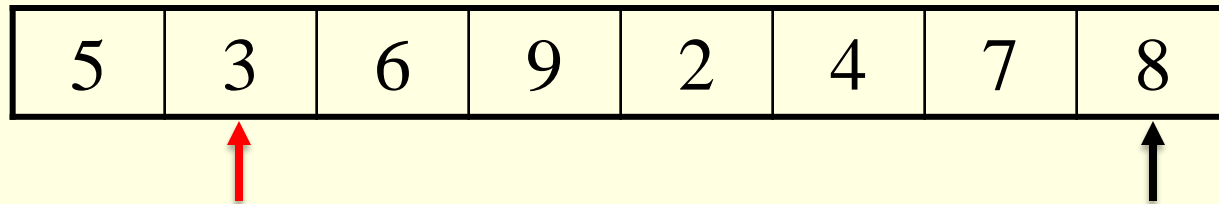
■ Here's how the partition will work:

- Start two counters, one at index one and one at index 7
 - The last element in the array
- Advance the left counter forward until an element greater than the partition element is encountered
- Advance the right counter backwards until a value less than the pivot is encountered

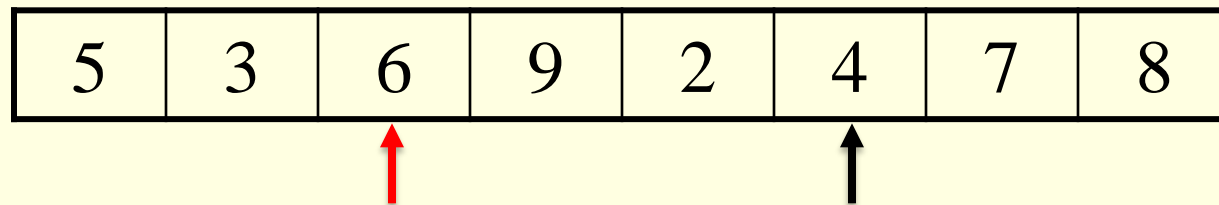


Sorting: Quick Sort

- How to Partition “in-place”



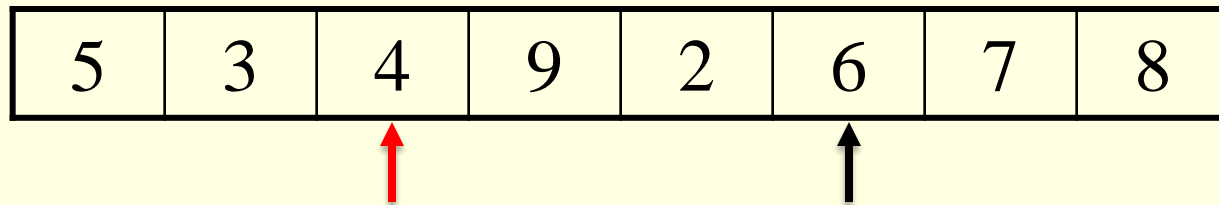
- After these two steps are performed, we have:



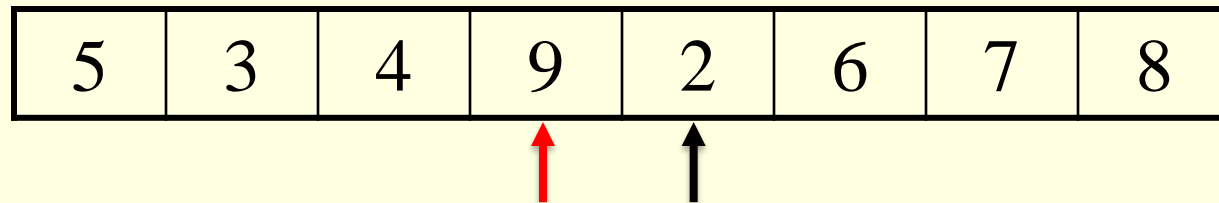


Sorting: Quick Sort

- How to Partition “in-place”



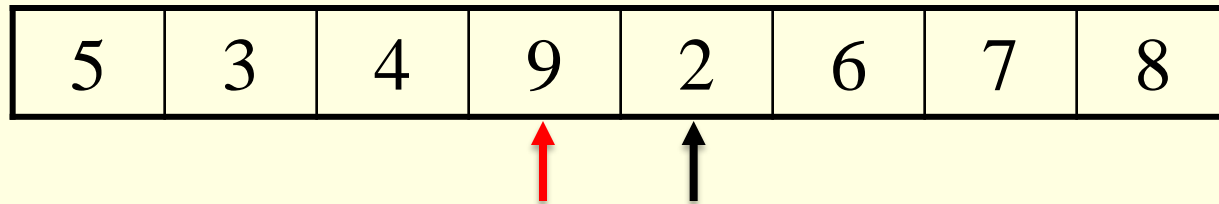
- Now continue to advance the pointers as before



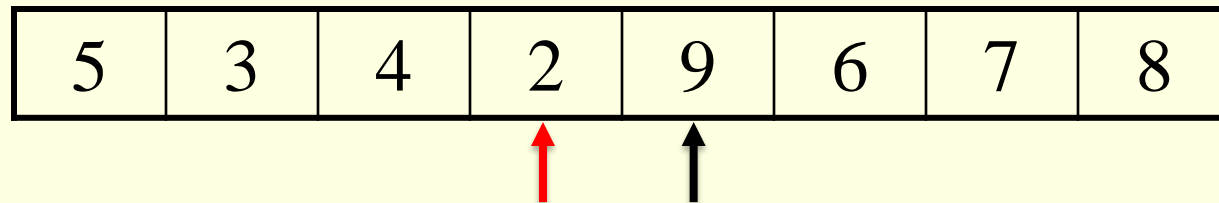


Sorting: Quick Sort

- How to Partition “in-place”



- Then SWAP as before:

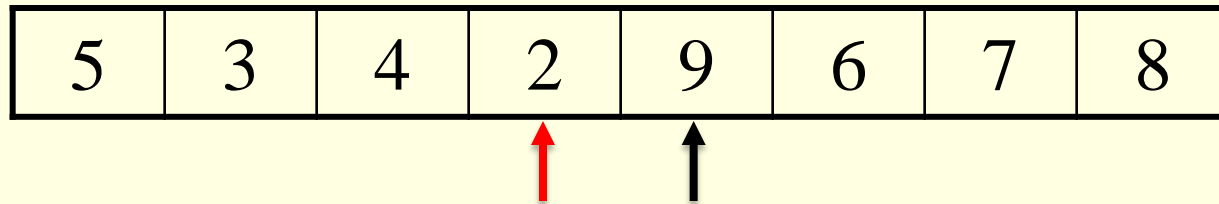


- At some point, the counters will cross over each other

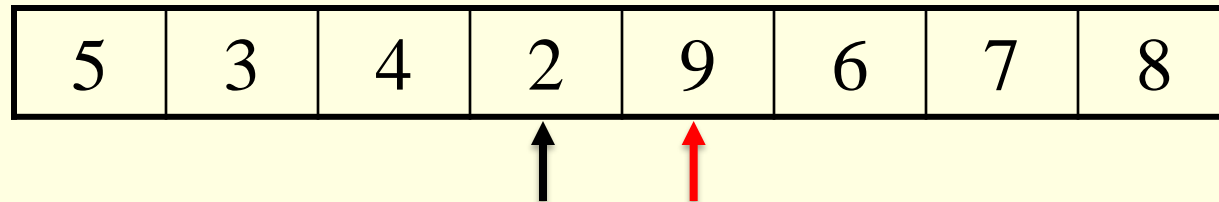


Sorting: Quick Sort

- How to Partition “in-place”



- Again, advance the pointers as before

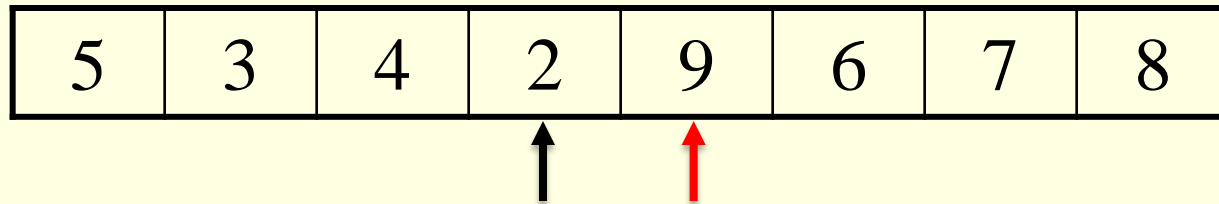


- So we see that the counters crossed over each other

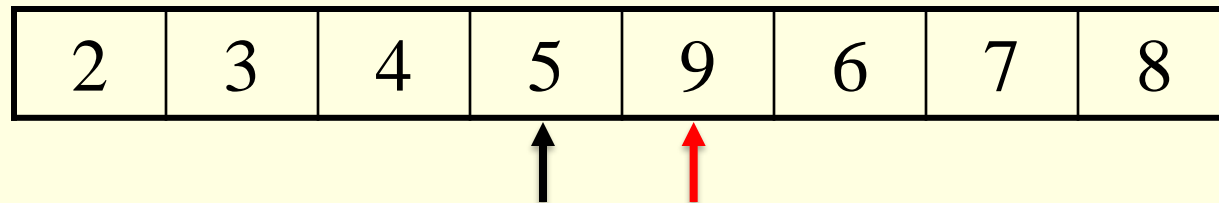


Sorting: Quick Sort

- How to Partition “in-place”



- Now, SWAP the value stored in the original right counter (black arrow) with the partition element



- Finally, RETURN the index the five is stored in (the right pointer) to indicate where the partition element ended up



Sorting: Quick Sort

■ Partition Code

```
int partition(int* vals, int low, int high) {
    int temp;
    int i, lowpos;

    // A base case that should never really occur.
    if (low == high) return low;

    // Pick a random partition element and swap it into index low.
    i = low + rand()%(high-low+1);
    temp = vals[i];
    vals[i] = vals[low];
    vals[low] = temp;

    // Store the index of the partition element.
    lowpos = low;

    // Update our low pointer.
    low++;
}
```



Sorting: Quick Sort

■ Partition Code

```
// Run Partition so long as low and high counters don't cross.
while (low <= high) {
    // Move the low pointer forwards.
    while (low <= high && vals[low] <= vals[lowpos]) low++;

    // Move the high pointer backwards.
    while (high >= low && vals[high] > vals[lowpos]) high--;

    // Now swap the values at those two pointers.
    if (low < high)
        swap(&vals[low], &vals[high]);
}

// Swap the partition element into it's correct location.
swap(&vals[lowpos], &vals[high]);

return high; // Return the index of the partition element.
}
```



Sorting: Quick Sort

■ Quick Sort Code

```
void quicksort(int* numbers, int low, int high) {  
  
    // Only have to sort if we are sorting more than one number  
    if (low < high) {  
  
        // Partition the elements  
        // Partition function returns the index of the  
        // partition element. Saved into "split".  
        int split = partition(numbers,low,high);  
  
        // Recursively Quick Sort the left side  
        quicksort(numbers,low,split-1);  
  
        // Recursively Quick Sort the right side  
        quicksort(numbers,split+1,high);  
  
    }  
}
```



Sorting: Quick Sort

- Choosing a Partition Element
 - For correctness, we can choose any pivot.
 - For efficiency, one of following is best case, the other worst case:
 - pivot partitions the list roughly in half
 - pivot is greatest or least element in list
 - Which case above is best?
 - Clearly, a partition element in the middle is ideal
 - As it splits the list roughly in half
 - But we don't know where that element is
 - So we have a few ways of choosing pivots



Sorting: Quick Sort

■ Choosing a Partition Element

■ first element

- bad if input is sorted or in reverse sorted order
- bad if input is nearly sorted
- variation: particular element (e.g. middle element)

■ random element

- You could get lucky and choose the middle element
- You could be unlucky and choose the smallest or greatest element
 - Resulting in a partition with ZERO elements on one side

■ median of three elements

- choose the median of the left, right, and center elements



Sorting: Quick Sort

- Choosing a Partition Element
 - median of three elements
 - choose the median of the left, right, and center elements
 - There is extra expense with this method
 - Picking three values
 - Doing three comparisons
 - But if the array is large, doing this little extra work will be small compared to the gains of a better partition
 - You could also pick the median of 5 or 7 elements
 - The more you pick the better partition you get



Brief Interlude: FAIL Picture





Sorting: Quick Sort

■ Quick Sort Analysis

- This is more difficult to do than Merge Sort
 - Why?
 - With Merge Sort, we knew that our recursive calls always had equal sized inputs
 - Remember: we would split the array of size n into two arrays of size $n/2$ (so the smaller arrays were always the same size)
- How is Quick Sort different? (more difficult?)
 - Each recursive call of Quick Sort could have a different sized set of numbers to sort
 - Because the size of the sets is based on our partition element
 - If partition element is in the middle, each set has about half
 - Otherwise, one set is large and one is small



Sorting: Quick Sort

■ Quick Sort Analysis

■ Location of partition element determines difficulty

1) If we get lucky

- and the partition element is ALWAYS in the middle:
- Then this is the BEST case
 - As we will always be **halving** the amount of work left

2) If we are unlucky:

- and we ALWAYS choose the first or the last element in the list as our partition
- Then this is the WORST case
 - As we will have not really sorted anything
 - We simply reduced the 2-be-sorted amount by 1



Sorting: Quick Sort

■ Quick Sort Analysis

- Location of partition element determines difficulty

3) If we are neither lucky or unlucky:

- Most likely, we will have some great partitions
- Some bad partitions
- And some okay partitions

- So we need to analyze each case:

- Best case
- Average case
- Worst case

And we **omit** the Average Case due to its difficulty.

*You'll get to see it in CS2.



Sorting: Quick Sort

■ Quick Sort Analysis

■ Analysis of Best Case:

- As mentioned, in the best case, we get a perfect partition every single time
- Meaning, if we have n elements before the partition,
 - we “luckily” pick the middle element as the partition element
 - Then we end up with $n/2 - 1$ elements on each side of the partition
- So if we had 101 unsorted elements
 - we “luckily” pick the 51st element as the partition element
 - Then we end up with 50 elements smaller than this element, on the left
 - And 50 elements, greater than this element, on the right



Sorting: Quick Sort

■ Quick Sort Analysis

■ Analysis of Best Case:

- Again, here are the steps of Quick Sort:
 - 1) Partition the elements
 - 2) Quick Sort the smaller half (recursive)
 - 3) Quick Sort the larger half (recursive)
- So at each recursive step, the input size is **halved**
- Let $T(n)$ be the running time of Quick Sort on n elements
 - And remember that Partition runs on $O(n)$ time
- So we get our recurrence relation for the best case:
 - $T(n) = 2 * T(n/2) + O(n)$
 - This is the same recurrence relation as Merge Sort
 - So in the best case, Quick Sort runs in $O(n \log n)$ time



Sorting: Quick Sort

■ Quick Sort Analysis

■ Analysis of Worst Case:

- Assume that we are horribly unlucky
- And when choosing the partition element, we somehow end up always choosing the greatest value remaining

■ **Now for this worst case:**

- How many times will the Partition function run?
 - Think: when we choose the greatest element (for example)
 - We have the partition element, then ALL other elements are to the left in one partition
 - The “partition” to the right will have ZERO elements
- So Partition will run $n-1$ times
 - The first time results in comparing $n-1$ values, then comparing $n-2$ values the second time, followed by $n-3$, etc.



Sorting: Quick Sort

■ Quick Sort Analysis

■ Analysis of Worst Case:

- How many times will the Partition function run?
 - Partition will run $n-1$ times
 - The first time results in comparing $n-1$ values, then comparing $n-2$ values the second time, followed by $n-3$, etc.

- When we sum the number of compares, we get:

- $1 + 2 + 3 + \dots + (n - 1)$
- You should know what this equals:

$$\frac{(n-1)n}{2}$$

- Thus, the worst case running time is $O(n^2)$



Sorting: Quick Sort

■ Quick Sort Analysis

■ Summary:

- Best Case: $O(n \log n)$
- **Average Case: $O(n \log n)$**
- Worst Case: $O(n^2)$

■ Compare Merge Sort and Quick Sort:

- Merge Sort: guaranteed $O(n \log n)$
- Quick Sort: best and average case is $O(n \log n)$ but worst case is $O(n^2)$

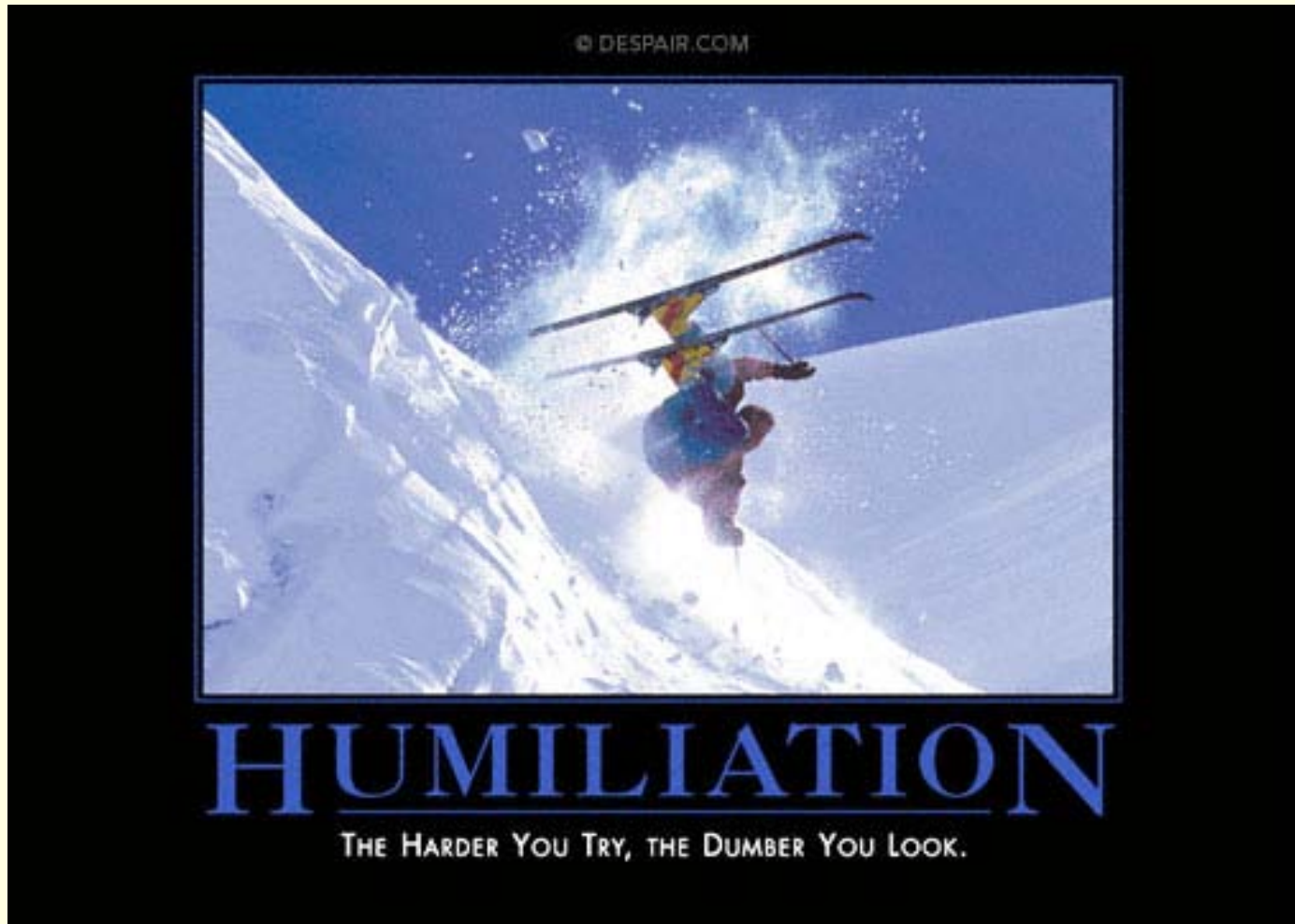


Sorting: Quick Sort

**WASN'T
THAT
THE GREATEST!**



Daily Demotivator



Sorting: Quick Sort



Computer Science Department
University of Central Florida

COP 3502 – Computer Science I