Searching and sorting algorithms

An important concept for both searching and sorting algorithms is the algorithm's *running time*. The running time is how we compare one algorithm to another, and determine which algorithm is more efficient. We will also consider *storage requirements* when comparing two algorithms.

Running time = the number of operations needed to complete the search or sort (typically, the number of comparisons we have to make)
How long does sequential search take?
- Best case: find item at the beginning of the array
  - 1 operation
- Worst case: find item in the last slot in the array, or number is not in the array
  - N operations (array has N items)
- Average case: find item somewhere in the middle of the array
  - N/2 operations

Binary search
- Start with a sorted array
- Each time through, we compare the midpoint of the array to the item we're trying to find
- We reset the midpoint depending on whether the item is greater than or less than the midpoint
- We repeat this until we find the item or until there are no more items to be tested

How long does binary search take?
- Best case: item is in the middle of the array
- Worst case: item is at the beginning or end, or on either side of the middle
- Average case: anywhere else
- Q: how many operations are needed?

Binary search running time
- Let $N$ be the size of the array we're searching
- 1st iteration: array size is $N/2$
- 2nd iteration: array size is $N/4$
- 3rd iteration: array size is $N/8$
- $k$th iteration: array size is $N/2^k$
Binary search running time (cont.)

- Size of array is $N = 2^k$
- Solve for $k$:
  $$k = \log_2 N$$
- The number of comparisons is logarithmic!
- Binary search is much faster than sequential search
  - any logarithmic algorithm is a very good algorithm

Order-N notation

- Indicates the scale at which an algorithm completes as a function of the size of the data set (array) on which it operates
- Example: algorithms that take $n$, $n/2$, $4n$, $3.6n$, $100n$ operations all take $n^*(\text{some constant})$ operations to complete
  - these complete in “order-N” time
  - also called linear algorithms
  - $O(n)$

How fast do certain functions grow?

Sorting algorithms

- Selection sort
- Bubble sort
- Quick sort
- Merge sort
Selection sort

- Start at the first element in the array
- Find the minimum element in the array
  - save its index
- Swap the first element with the minimum element in the array
- Repeat for the remaining elements in the array
- In each pass, at least one element will be moved to its proper position in the array

How long does selection sort take to run?

- Number of comparisons:
  - Outer loop executes $N - 1$ times:
    - first time: $N$ comparisons in inner loop
    - second time: $N - 1$ comparisons in inner loop
    - ...
    - $(N-2)th$ time: 2 comparisons in inner loop
  - Number of comparisons = $N + (N-1) + (N-2) + ... + 2$
  - = $N(N+1)/2 + 1$
  - $O(n^2)$

How long does selection sort take to run? (cont)

- Number of swaps:
  - Best case: array is already in order – no swaps
  - Worst case: largest element is at beginning of array, rest of array is ordered
    - swap largest with next element $N-1$ times
    - each swap involves three steps (copy value into temp location, overwrite this location, copy temp into old location)
    - $O(n)$

Bubble sort

- Idea: make more data swaps each pass through the array
  - Q: will this be faster than selection sort?
- During each pass, compare pairs of array elements, and swap if they are out of order
- In each pass, at least one element will be moved to its proper position in the array
Running time of bubble sort

- Very similar to selection sort:
  - $O(n^2)$ comparisons
- Differences:
  - $O(n^2)$ swaps
  - Wu claims: on average, bubble sort will be faster because most items are moved early on in the sort.
    - (do you agree or disagree?)
  - if array is already sorted, bubble sort will only make one pass through the array (in selection sort: $N-1$ passes)
  - worst case for bubble sort: array is in descending order

A diversion into recursion

- Recursive method: a method that calls itself
- Examples:
  - factorial calculation
  - fibonacci numbers
  - exponents
  - Also useful in solving puzzles

Parts of a recursive method

- Test: should recursion stop or continue?
- End case: stopping case for recursion
- Recursive call: call to itself

Gotcha: Easy to get yourself into an infinite loop!

Example: factorial calculation

```java
public int factorial(int n) {
    if (n == 1)
        return 1; // stop case
    else
        return (n * factorial(n-1));
}
```
Why is recursion important now?

- The next 2 sorting algorithms use recursion
- We could write these without recursion (as we can with all recursive methods), but it's easier and more elegant to use recursion

Recursion is not always good!

- Example: computing Fibonacci numbers
  - fib(N) = fib(N-1) + fib(N-2);
  - fib(N) = 1 if N = 0, 1

Recursive fibonacci method

```java
public int fibonacci(int n) {
    if (n==0 || n==1)
        return 1;
    else
        return fibonacci(n-1) + fibonacci(n-2);
}
```

Nonrecursive fibonacci method

```java
public int fibonacci(int n) {
    int f1, f2, fn;
    fn = f1 = f2 = 1;
    for (int i=1; i<n; i++) {
        fn = f1 + f2;
        f1 = f2;
        f2 = fn;
    }
    return fn;
}
```
Nonrecursive vs. recursive

- Recursive is short, elegant BUT it recomputes numbers over and over again
- Nonrecursive is not as elegant BUT it does not repeat calculations

Quicksort

- Divide the array in half (roughly) by choosing a \textit{pivot}
- Divide each subarray in half by choosing another \textit{pivot}
- Repeat until the subarrays contain one cell
- Rebuild the arrays

Running time of quicksort

- Worst case: pivot = smallest (or largest) item in array each time
  - \(O(n^2)\) comparisons
- Best case and average case: pivot splits the array exactly (or nearly) in half
  - Size of array is \(N=2^k\)
  - \(N\) comparisons at each level, \(k\) levels = \(kN\) comparisons
  - Solve for \(k\): \(k = \log_2 N\)
  - so number of comparisons = \(N \log_2 N = O(n \lg n)\)

Merge sort

- Divide array in half
- Divide these subarrays in half
- Repeat until subarrays contain one item
- Merge the two subarrays (combine and sort)
- Repeat until all subarrays have been merged back into a single array
Complexity of merge sort

- Number of swaps per merge: \(2(last - first + 1)\)
- Comparisons:
  - Best case: array is sorted, or all elements in right subarray are greater than all elements in left subarray
    - number of comparisons = \((first + last)/2\)
  - Worst case: last item of one subarray precedes only the last item of the other subarray
    - number of comparisons = \(last - first\)

Merge sort: worst case complexity

- Number of swaps: \(O(n \lg n)\)
- Number of comparisons: \(O(n \lg n)\)
- Good, efficient sorting algorithm BUT need extra storage space for the extra arrays

Sorting algorithms: # comparisons

- Selection sort: \(O(n^2)\)
- Bubble sort: \(O(n^2)\)
- Quicksort: \(O(n \lg n)\)
- Merge sort: \(O(n \lg n)\)

Sorting algorithms: # swaps

- Selection sort: \(O(n)\)
- Bubble sort: \(O(n^2)\)
- Quicksort: \(O(n \lg n)\)
- Merge sort: \(O(n \lg n)\)