Comparing Algorithms

Problems, Algorithms, and Programs

- **Problem**: A task to be performed. It is best thought of as a function or a mapping of inputs to outputs
- **Algorithm**: A method or a process followed to solve a problem. An implementation for the function that transforms an input to the corresponding output.
- An algorithm has the following properties:
    1. It must be *correct*
    2. It is composed of a series of *concrete steps*
    3. There can be *no ambiguity* as to which step will be performed next
    4. It must be composed of a *finite* number of steps
    5. It must *terminate* (no infinite loop/recursion)
- **Program**: An instance, or concrete representation, of an algorithm in some programming language.

Comparing Algorithms

- The performance (running time) of an algorithm is an estimate of the number of *basic operations* required by the algorithm to process an input of a certain size.

Comparing Algorithms: example

```
// Return position of largest value in integer array A
static int largest(int[] A) {
  int currlarge = 0;                // Position of largest element seen
  for (int i=1; i<A.length; i++) // For each element
    if (A[currlarge] < A[i])       //   if A[i] is larger
       currlarge = i;              //     remember its position
  return currlarge;                // Return largest position
}
```

- Basic operation: compare an integer's value to that of the largest value seen so far
- We can assume that it takes a fixed amount of time to do one such comparison, regardless of the value of the two integers or their positions in the array
- The most important factor affecting the running time is the size of the array (input)
- For a given input size *n* we often express the time **T** to run the algorithm as a function of *n*, written as **T**(*n*). We will always assume **T**(*n*) is a non-negative value.

Comparing Algorithms

- For a given input size *n,* the time **T** to run the algorithm is expressed as a function of *n*, written as **T**(*n*).
- We will always assume **T**(*n*) is a non-negative value.
- For the function `largest`, `T(n) = c*n.` With `c` the amount of time required to compare two integers
- We say that `largest` runs in O(n) (big-O n)
- What is the runtime function for finding the largest integer in 2D array?

Best, Worst, and Average Cases

- Example
- Best case: rarely happens, too optimistic
- Average case: represents the “typical” behavior of the algorithm on inputs of size *n. Hard to estimate.*
- Worst case: we know for certain that the algorithm must perform at least that well

Comparing algorithms: Sorting

- Insertion Sort

```
public static void insertionSort(Comparable[] A) {
  for (int i=1; i<A.length; i++) // Insert i'th record
    for (int j=i; (j>0) && (A[j].compareTo(A[j-1]) < 0); j--)
      swap(A, j, j-1);
}
```

- Best case: the array is already sorted, we do not enter the inner loop. Runtime O(n)
- Worst case: each iteration of the outer loop does the largest number of comparisons. Runtime $O(n^2)$
- Average: about half of the values out of order. We do not care about the constants. Runtime $O(n^2)$

Comparing algorithms: Sorting

- Bubble Sort

```
public static void bubbleSort(Comparable[] A) {
  for (int i=0; i<A.length-1; i++) // Insert i'th record
    for (int j=1; j<A.length-i; j++)
      if (A[j-1].compareTo(A[j]) > 0)
        swap(A, j-1, j);
}
```

- The number of comparisons made by the inner loop is always the same
- Best case: runtime $O(n^2)$
- Worst case: runtime $O(n^2)$
- Average: runtime $O(n^2)$

Comparing algorithms: Sorting

- Selection Sort

```
public static void selectionSort(Comparable[] A) {
  for (int i=0; i<A.length-1; i++) { // Select i'th biggest record
    int bigindex = 0; // Current biggest index
    for (int j=1; j<A.length-i; j++) {// Find the max value
      if (A[j].compareTo(A[bigindex]) > 0) // Found something bigger
          bigindex = j; // Remember bigger index
    }
    swap(A, bigindex, A.length-i-1); // Put it into place
  }
}
```

- The number of comparisons made by the inner loop is always the same
- Best case: runtime $O(n^2)$
- Worst case: runtime $O(n^2)$
- Average: runtime $O(n^2)$

Comparing algorithms: Sorting

- When the array is nearly sorted, insertion sort is the best algorithm
- Selection sort minimizes the number of swaps