Sorting problem

The sorting problem

- **Input**: a sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$
- **Output**: a permutation $\langle a_1', a_2', \ldots, a_n' \rangle$, s.t. $a_1' \leq a_2' \leq \ldots \leq a_n'$

- An **instance** of a problem:
  - the input needed to compute a solution (e.g.: $\langle 5, 3, 6, 2 \rangle$)

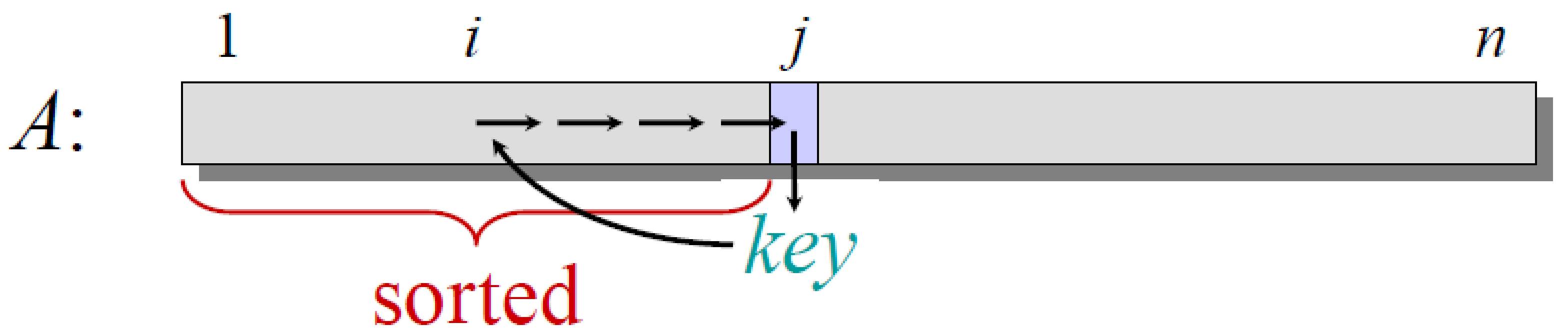
- An algorithm is **correct** if it ends with the correct output in a finite amount of time, on any legitimate input

Insertion Sort (C)

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
           A[j+1] = A[j]
           j = j - 1
      }
      A[j+1] = key
  }
}
```

“pseudocode”

```
INSERTION-SORT (A, n)      ▷ A[1 . . n]
    for j ← 2 to n
        do key ← A[j]
           i ← j – 1
           while i > 0 and A[i] > key
               do A[i+1] ← A[i]
                  i ← i – 1
           A[i+1] = key
```



Correctness Analysis

- Loop invariant:

At the start of each iteration of the **for** loop of lines 1–8, the subarray $A[1 \mathinner{.\,.} j-1]$ consists of the elements originally in $A[1 \mathinner{.\,.} j-1]$, but in sorted order.

- **Initialization:** It is true prior to the first iteration of the loop.
- **Maintenance:** If it is true before an iteration of the loop, it remains true before the next iteration.
- **Termination:** When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

Insertion Sort

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
      }
      A[j+1] = key
  }
}
```

What is the precondition for this loop?

Insertion Sort

```
InsertionSort(A, n) {
  for i = 2 to n {
     key = A[i]
     j = i - 1;
     while (j > 0) and (A[j] > key) {
          A[j+1] = A[j]
          j = j - 1
     }
     A[j+1] = key
  }
}
```

How many times will this loop execute?

Insertion Sort

Statement	Effort
`InsertionSort(A, n) {`	
`  for i = 2 to n {`	$c_1 n$
`    key = A[i]`	$c_2(n-1)$
`    j = i - 1;`	$c_3(n-1)$
`    while (j > 0) and (A[j] > key) {`	$c_4 T$
`      A[j+1] = A[j]`	$c_5(T-(n-1))$
`      j = j - 1`	$c_6(T-(n-1))$
`    }`	0
`    A[j+1] = key`	$c_7(n-1)$
`  }`	0
`}`	

$T = t_2 + t_3 + \ldots + t_n$ where t_i is number of while expression evaluations for the i^{th} for loop iteration

Analyzing Insertion Sort

- $T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 T + c_5(T - (n-1)) + c_6(T - (n-1)) + c_7(n-1)$

  $= c_8 T + c_9 n + c_{10}$
- What can T be?
  - Best case -- inner loop body never executed
    - $t_i = 1 \Rightarrow T(n)$ is a linear function
  - Worst case -- inner loop body executed for all previous elements
    - $t_i = i \Rightarrow T(n)$ is a quadratic function
  - Average case
    - ???

Analysis

- Simplifications
  - Ignore actual and abstract statement costs
  - *Order of growth* is the interesting measure:
    - Highest-order term is what counts

      Remember, we are doing asymptotic analysis

      As the input size grows larger it is the high order term that dominates

Upper Bound Notation

- We say InsertionSort's run time is *$O(n^2)$*
  - Properly we should say run time is *in* $O(n^2)$
  - Read O as "Big-O" (you'll also hear it as "order")
- In general a function
  - f(n) is O(g(n)) if there exist positive constants c and n_0 such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$
- Formally
  - $O(g(n)) = \{ f(n)\text{: } \exists$ positive constants c and n_0 such that $f(n) \leq c \cdot g(n)\ \forall\ n \geq n_0$

Insertion Sort Is $O(n^2)$

- Proof
  - Suppose runtime is $an^2 + bn + c$
    - If any of a, b, and c are less than 0 replace the constant with its absolute value
  - $an^2 + bn + c \le (a + b + c)n^2 + (a + b + c)n + (a + b + c)$
  - $\le 3(a + b + c)n^2$ for $n \ge 1$
  - Let $c' = 3(a + b + c)$ and let $n_0 = 1$
- Question
  - Is InsertionSort $O(n^3)$?
  - Is InsertionSort $O(n)$?

Selection Sort

Idea: walk down the list, and find the smallest (or largest) element, and then swap it to the beginning of the unsorted part of the list.

1. Loop (i) from 0 to the (number of elements to be sorted - 2)

    1.1 Assume the smallest remaining item is at the i^{th} position, call this location smallest.

    1.2 Loop (j) through the remainder of the list to be sorted (i+1 .. size-1).
        1.2.1 Compare the j^{th} & smallest elements in the unsorted list.
        1.2.2 If the j^{th} element is < the smallest element then
            reset the location of the smallest to the j^{th} location.

    1.3 Move the smallest element to the head of the unsorted list,
        (i.e. swap the ith and smallest elements).

```
void SelectionSort(int List[], int Size)
{
   int Begin, SmallSoFar, Check;
   void Swap(int& Elem1, int& Elem2);
   for (Begin = 0; Begin < Size - 1; Begin++) {
      SmallSoFar = Begin;                              // set head of tail
      for (Check = Begin + 1; Check < Size; Check++) { //
         if (List[Check] < List[SmallSoFar])
            SmallSoFar = Check;
      }
      Swap(List[Begin], List[SmallSoFar]);  // put smallest to
front
   }                                     //   of current tail
}

void Swap(int& Elem1, int& Elem2) {
   int tempInt;
   tempInt = Elem1;
   Elem1   = Elem2;
   Elem2   = tempInt; }
```

Selection Sort

12 6	12 6	22	14	8	17

6	12 8	22	14	12 8	17

- Given n numbers to sort:
- Repeat the following n-1 times:
  - Mark the first unsorted number
  - Find the smallest unsorted number
  - Swap the marked and smallest numbers

Selection Sort

6	8	22	14	12	17

6	8	12	14	22	17

- Given n numbers to sort:
- Repeat the following n-1 times:
  - Mark the first unsorted number
  - Find the smallest unsorted number
  - Swap the marked and smallest numbers

Selection Sort

- Given n numbers to sort:
- Repeat the following n-1 times:
  - Mark the first unsorted number
  - Find the smallest unsorted number
  - Swap the marked and smallest numbers

- How efficient is selection sort?
  - In general, given n numbers to sort, it performs n^2 comparisons
- Why might selection sort be a good choice?
  - Simple to write code
  - Intuitive

Selection Sort

- Given n numbers to sort:
- Repeat the following n-1 times:
  - Mark the first unsorted number
  - Find the smallest unsorted number
  - Swap the marked and smallest numbers

Try one!

15	3	11	19	4	7

Bubble Sort

Idea: walk down the list, compare adjacent elements, and swap them if they are in the wrong order.

1. Initialize the size of the list to be sorted to be the actual size of the list.

2. Loop through the list until no element needs to be exchanged with another
   to reach its correct position.

   2.1 Loop (i) from 0 to size of the list to be sorted - 2.

   2.1.1 Compare the i^{th} and $(i + 1)^{st}$ elements in the unsorted list.

   2.1.2 Swap the i^{th} and $(i + 1)^{st}$ elements if not in order (ascending or descending as desired).

   2.2 Decrease the size of the list to be sorted by 1.

```
void BubbleSort(int List[] , int Size) {

   int tempInt;     // temp variable for swapping list elems

   for (int Stop = Size - 1; Stop > 0; Stop--) {

      for (int Check = 0; Check < Stop; Check++) { // make a pass

         if (List[Check] > List[Check + 1]) {  // compare elems

            tempInt          = List[Check];     // swap if in the
            List[Check]      = List[Check + 1]; //   wrong order
            List[Check + 1] = tempInt;
         }
      }
   }
}
```

Bubble Sort

12 6	6 12	22 14	14 22 8	8 22 17	17 22
6	12	14 8	8 14	17	22

- Given n numbers to sort:
- Repeat the following n-1 times:
  - For each pair of adjacent numbers:
    - If the number on the left is greater than the number on the right, swap them.

Bubble Sort

6	12/8	12/8	14	17	22

6	8	12	14	17	22

- Given n numbers to sort:
- Repeat the following n-1 times:
  - For each pair of adjacent numbers:
    - If the number on the left is greater than the number on the right, swap them.

Bubble Sort

- Given n numbers to sort:
- Repeat the following n-1 times:
  - For each pair of adjacent numbers:
    - If the number on the left is greater than the number on the right, swap them

- How efficient is bubble sort?
  - In general, given n numbers to sort, it performs n^2 comparisons
  - The same as selection sort

- Is there a simple way to improve on the basic bubble sort?
  - Yes! Stop after going through without making any swaps
  - This will only help some of the time

Bubble Sort

- Given n numbers to sort:
- Repeat the following n-1 times:
  - For each pair of adjacent numbers:
    - If the number on the left is greater than the number on the right, swap them

Try one!

15	3	11	19	4	7

A Simple Summation

	T(n)	A(n)	In-place?	Stable?
Insertion Bubble	$O(n^2)$	$O(n^2)$	yes	yes
Selection	$O(n^2)$	$O(n^2)$	yes	no
Heap	O(n log n)	O(n log n)	yes	no
Quick	$O(n^2)$	O(n log n)	yes	no

Sorting Algorithms

Comparison-Based Sorting

- Simple: Insertion, Selection, Bubble
- Complex: Merge, Quick, Heap…

Others:

- Counting sort
- Radix sort
- Bucket sort