IE170: Algorithms in Systems Engineering: Lecture 4

Jeff Linderoth

Department of Industrial and Systems Engineering
Lehigh University

January 22, 2007

Everyone Gets an A!

Go Bears!

Taking Stock

Last Time

- Θ, O and Ω
- Recursion. See recursion.
- Analyzing Recurrences

This Time

- Analyzing a simple algorithm
- The impact of data structures

A Canonical Problem

Example: The Sorting Problem

- Input: A sequence of numbers $a_1, a_2, \ldots, a_n$
- Output: A reordering $a'_1, a'_2, \ldots, a'_n$ such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

- Sorting "in place": No new memory is allocated (or at least a constant amount of memory is allocated). (The input is usually overwritten by the output as the algorithm executes.)
- Sorting "out of place": New memory must be allocated

Sample. Reverse-Out-Of-Place

- In this case, we allocate a new array B

```
public static void reverseOP(int A[])
{
  int n = A.length;
  int B[] = new int[n];
  for (int j = 0; j < n; j++) {
    B[n-1-j] = A[j];
  }
  System.arraycopy(B,0,A,0,n);
}
```


Sample. Reverse-In-Place

- Here everything is done directly on A

```
public static void reverseIP(int A[])
{
  int n = A.length;
  for(int j = 0; j < n/2; j++){
    // Swap A[j] and A[n-j-1]
    int t = A[j];
    A[j] = A[n-j-1];
    A[n-j-1] = t;
  }
}
```


Sorting: Some Java Code

```
public static void iSortMe(int A[])
{
  for(int j = 1; j < A.length; j++) {
    int key = A[j];
    int i = j-1;
    while(i >= 0 && A[i] > key) {
      A[i+1] = A[i];
      i = i-1;
    }
    A[i+1] = key;
  }
}
```


Example of How It Works

$j = 1$
11 3 6 1 42 9

$j = 2$
3 11 6 1 42 9

$j = 3$
3 6 11 1 42 9

$j = 4$
1 3 6 11 42 9

$j = 5$
1 3 6 11 42 9

$j = 6$
1 3 6 9 11 42

Is It Correct?!?

- We often use a loop invariant to prove the correctness of an algorithm

---

Insertion Sort Loop Invariant

At the start of each iteration of the outer **for** loop (the loop indexed by j), the subarray $A[0, \ldots, j-1]$ consists of the elements originally in $A[1..j-1]$ but in sorted order

Loop Invariants

It's Like Induction!

- **Base Case:** It is true prior to the first iteration of the loop
- **Maintenance:** If it is true before a loop iteration, it is true after the loop iteration
- **Termination:** Hopefully, the invariant will have a useful property when the loop terminates. In this case, it would "prove" that the array is sorted.

Can We Prove This Works

- **Initialization:** $j = 1$, The subarray $A[0, \ldots, j-1]$ is just $A[0]$ which is in sorted order. **Duh!**
- **Maintenance:** The book (and we) will gloss over this a bit. The loops function is to move $A[j-1], A[j-2], \ldots$ one position to the right until the proper position for item j is found. Thus the subarray $A[0, \ldots, j]$ remains sorted (which becomes $A[0, \ldots, j-1]$ when the loop is incremented
- **Termination:** When loop exits, $j = n$, so the (sub)array $A[0, \ldots, n-1]$ is sorted.

Q.E.D

CountVonCount

```
public static void iSortMe(int A[])
{
  for(int j = 1; j < A.length; j++) {
    int key = A[j];
    int i = j-1;
    while(i >= 0 && A[i] > key) {
      A[i+1] = A[i];
      i = i-1;
    }
    A[i+1] = key;
  }
}
```

Analysis

- To analyze our algorithm, we need to count the number of times each command is done
- $T(n)$: Running time of algorithm if "input size" (array size) is n
- t_j: The number of times the "while" statement is executed for item j

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=1}^{n-1} t_j + c_5 \sum_{j=1}^{n-1}(t_j - 1) + c_6 \sum_{j=1}^{n-1}(t_j - 1) + c_7(n-1)$$

Best Case

- Let's Assume that `A[i]` $\leq$ `key` for each j.
- The array is already sorted!
- The while loop is executed only once each time: $t_j = 1$, so the running time becomes

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

- This is a linear function of n: $T(n) = \Theta(n)$

Worst Case

- We find `A[i]` $>$ `key` for all elements. **while** loop only exits because $i < 0$
- In this case (since must test to see that $i < 0$, $t_j = j$
- In this case running time becomes

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=1}^{n-1} j + c_5 \sum_{j=1}^{n-1}(j-1) + c_6 \sum_{j=1}^{n-1}(j-1) + c_7(n-1)$$

- Aren't you glad Gauss is your friend?
- We will (?) show that this is a quadratic function of n: $T(n) = \Theta(n^2)$

Case Analysis

- We could also perform an average case analysis of this algorithm. (In this case, you would see it $T(n) = \Theta(n^2)$)
- We generally don't do this, because it is hard!

Which Function $T(n)$ Do We Use!?

- Computer scientists are a cautious bunch, so typically we will analyze the worst case behavior.
- It does have some advantages
  - It provides an upper bound
  - For some algorithms it frequently happens

Sorting Exercise

- Insertion Sort
- Merge Sort!
- You will be responsible for knowing how merge sort works (Section 2.3)

What is a Data Structure?

- Computers operate on tables of numbers (the data).
- Within the context of solving a given problem, this data has structure.
- Data structures are schemes for storing and manipulating data that allow us to more easily see the structure of the data.
- Data structures allow us to perform certain operations on the data more easily.
- The data structure that is most appropriate depends on how the algorithm needs to manipulate the data.

Importance of Data Structures

- Specifying an algorithm completely includes specifying the data structures to be used (sometimes this is the hardest part).
- It is possible for the same basic algorithm to have several different implementations with different data structures.
- Which data structure is best depends on what operations have to be performed on the data.

Example

- Consider the two implementations of the list class that you will become intimately familiar with in lab
- An array is a simple data structure that allows us to store a sequence of numbers.
- A linked list does the same thing.
- You should know the difference? (Yes?)

A List Interface

```
public interface MyList
{
  public void add(int index, Object element);
  public boolean contains(Object element);
  public Object get(int index);
  public int indexOf(Object element);
  public Object remove(int index);

}
```


Comparing List Data Structures

- To compare the two data structures, we must analyze the running time of each operation.
- This table compares the running times of the operations.
- Usually list interfaces have other operations
- You will try and implement this stuff in lab

	Array	Linked List
`getNumItems`		
`get`		
`add`		
`remove`		

Next Time

- Back to the Master Theorem – Analyzing Recurrences
- So far, we have covered chapters 1-4 and Appendix A & B