CS241 – Analysis of Algorithms Spring 2020

- Prerequisites: CMPU102 and CMPU145.
- Lectures: Lectures will be held on Tuesdays and Thursdays from 1:30 to 2:45 pm in SP 201.
- Textbook: *Introduction to Algorithms (3rd Edition)*, by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.

Course Web Page

- All information about the course will be posted on the course web page at
  https://www.cs.vassar.edu/~cs241
- Check your e-mail frequently for course announcements.

Algorithms

What is an algorithm?

- For the purposes of this class, an *algorithm* is a computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output, and eventually terminates.

Sorting Problem

The algorithmic problem known as *sorting* is defined as follows:

INPUT: An array A[1…n] of *n* totally ordered elements $\{a_1, a_2, ..., a_n\}$
OUTPUT: A permutation of the input array $\{a_1', a_2', ..., a_n'\}$ such that $a_1' \le a_2' \le ... \le a_n'$.

Sorting Problem Input?

Sorting Problem Input

Example instances of input for the sorting problem:
{Mike, Sally, Herbert, Tony, Jill}
{101, 111111, 1111, 100, 1010, 101010}

Example instances of output for the given instances of the sorting problem above:
{Herbert, Jill, Mike, Sally, Tony}
{100, 101, 1010, 1111, 101010, 111111}

Algorithm Efficiency

In our analysis, *time* and *space* efficiency are both considered in the limit as the size of the input grows without bound.

The amount of extra space required is currently of less concern than the running time for many programming applications, because memory capacity is large (and cheap). Extra space includes above what is required to store the input to an algorithm.

We will concentrate mainly on the amount of time an algorithm uses. But for algorithms with equal running time, the space used may be of importance.

Algorithm Time Efficiency

Observation: Most algorithms run longer on larger inputs.

We want to investigate an algorithm's efficiency as a function of some parameter *n* indicating the algorithm's input size.

Analyzing Algorithms

Goal: to predict the number of steps executed by an algorithm in a machine- and language-independent way using:

1. the RAM model of computation: Single processor with *sequential instructions* (no parallel computation)
2. asymptotic analysis of worst-case complexity

Measures of Complexity

What metrics of an algorithm are considered when comparing algorithm complexity?

Time

Space

Number of messages

Power consumption

How to Measure Algorithm Time?

1. Implement algorithm and include a system call to count the number of milliseconds it takes to run.
2. count the exact number of times *each* of the algorithm's operations is executed, assuming each particular line takes a constant amount of time for a data set of size n, and add time of all lines to get a polynomial expression.
3. identify the operations (line or loop) that contribute *most* to the total running time (usually a statement that begins a loop and one or more of the lines internal to the loop) and count the number of times that operation is executed.

    ==> the **basic (aka dominant) operation**

Analysis Fundamental

The data structures used to store information and the algorithms to manipulate the information are generally intertwined. The efficiency of information manipulation is often significantly affected by the data structures used.

RAM Model of Computation

Single-processor machine: instructions are executed sequentially (no concurrent operations.)

The running time *T(n)* of an algorithm on a particular input instance of size *n* is the number of times the basic operation is executed. Expressed in terms of *n*, the input size.

To make the notion of an *algorithm step* as machine-independent as possible, assume:

Each execution of the i^{th} line takes a constant amount of time.

Flow of Control

Loops and their recursive counterparts are the composition of many primitive steps (as you know if you have done any assembly language programming).

Execution time depends on the number of loop iterations or recursive calls (a function of the input size).

We will represent the running time of loops (iterative algorithms) using summations.

Algorithm Efficiency on Different Inputs

Some algorithms take the same amount of time on all input instances of a given size, *n*.

For other algorithms, there are best-case, worst-case, and average-case input instances that depend on other qualities of the input than just the input size.

For algorithm A on input of size *n*:

Worst-case input : The input(s) for which A executes the most steps, considering all possible inputs of size *n*.

Best-case input : The input(s) for which A executes the fewest steps, considering all inputs of size *n*.

Pseudocode

FindMax(A[1...n])

INPUT: An array A of *n* totally ordered items

OUTPUT: The value of the maximum item in the array

```
1. max = A[1]
2. for ( k = 2;  k <=n; k++)
3.       if (A[k] > max)
4.           max = A[k]
5. return max
```

For input of *comparable*, *totally ordered* set of data:
$[23, 53, 5, 34, 42, 18]$ ➔ 53

Book Idiosyncrasy

In our textbook, arrays are usually assumed to be numbered starting at 1, not 0 as we are all used to.

Take note of whether the algorithms in the book (and those you write) use 1- or 0-based indexing on arrays. The index number can be important not only to understand how the algorithm works, but is often essential in proving algorithm correctness.

Handy Summation Rules for Iterative Algorithms

$$\sum_{i=l}^{u} 1 = u - l + 1$$

where l ≤ u & l and u are lower and upper integer limits.

$$\sum_{i=0}^{n} i = \sum_{i=1}^{n} i = 1 + 2 + \ldots + n = \frac{n(n+1)}{2}$$

This type of summation usually applies to loops with one level of nesting (loop inside loop).

Expressing loops as summations

FindMax(A[1...n])

INPUT: An array A of *n* comparable items

OUTPUT: The value of the maximum item in the array

```
1. max = A[1]
2. for ( k = 2;  k <=n; k++ )
3.       if (A[k] > max)
4.           max = A[k]
5. return max
```

$$\sum_{i=2}^{n} 1 = n - 2 + 1$$

Does this algorithm have variable running time on input arrays with different contents or orderings of size *n* ?

Give the line number of the basic operation.

Sorting Algorithms

InsertionSort(A)

INPUT: An array A of *n* items $\{a_1, a_2, ..., a_n\}$

OUTPUT: A permutation of the input array $\{a_1', a_2', ..., a_n'\}$ such that $a_1' \le a_2' \le ... \le a_n'$.

```
1.  for ( j  = 2 to length[A] )
2.      key = A[j]
3.      i = j – 1
4.      while ( i > 0 and A[i] > key)
5.          A[i + 1] = A[i]
6.          i = i - 1
7.      A[i + 1] =  key
```

The while loop may be executed a different number of times for different values of j, so let t_j be the number of times line 4 is executed for each value of j.

Analysis of InsertionSort

```
InsertionSort(A)
1. for j  = 2 to length[A]
2.      key = A[j]
3.      i = j – 1
4.      while i > 0 and A[i] > key
5.          A[i + 1] = A[i]
6.          i = i – 1
7.      A[i + 1] =  key
```

- For insertion sort, does the running time vary for different input instances?

If so, give an instance of best-case and worst-case inputs.

General Plan for Analyzing Time Efficiency of Non-recursive Algorithms

1. Decide on a parameter indicating input size.
2. Identify the algorithm's basic operation.
3. If the number of times the basic operation is executed depends only on the size of the input, give worst-case efficiency. If this number also depends on some additional property, the worst-case and best-case efficiencies should be given separately.
4. If possible, set up a sum expressing the number of times the basic operation is executed.
5. Use standard rules of sum manipulation to find a closed-form solution for the count of operations.

Sorting Algorithm

SelectionSort(A)

INPUT: An array A of *n* items $\{a_1, a_2, ..., a_n\}$

OUTPUT: A permutation of the input array $\{a_1', a_2', ..., a_n'\}$ such that $a_1' \le a_2' \le ... \le a_n'$.

```
1.  for ( i  = 1 to n - 1 )
2.      min = i
3.      for ( j = i+1 to n)
4.          if (A[j] < A[min])
5.              min = j
6.      swap (A[i], A[min])
```

Next Lecture: Chapters 3 and 4
Asymptotic Analysis and Running time of recursive divide-and-conquer algorithms

Assignment 1 is posted.