CS241 – Analysis of Algorithms Spring 2020

- Prerequisites: CMPU102 and CMPU145.
- Lectures: Lectures will be held on M/W 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.
  - ❑ *For this week, read chapters 1-4 of CLRS*

Course Web Page

- All information about the course will be posted on the course web page at
  https://www.cs.vassar.edu/~cs241
  Lecture notes, readings, assignments, videos, etc., will be posted on this site
- 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.

Algorithmic Problems

What is an algorithmic problem?

- An algorithmic *problem* is the complete set of possible input *instances* the algorithm may work on and the desired output from each input instance.

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 $a_1' \leq a_2' \leq ... \leq a_n'$.

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

Observation: Most algorithms we care about run longer on larger inputs. Larger inputs also take more space in memory.

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

Measures of Complexity

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

Time and Space are 1st and 2nd in terms of importance.

For distributed algorithms on, e.g., ad-hoc networks or mobile robots:

Number of messages

Power consumption

Measuring 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 (rather complicated) 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 operation**

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
2. asymptotic analysis of worst-case complexity

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 (usually a function of the input size).

First, we consider iterative algorithms. We will represent the running time of loops (iterative algorithms) using ***summations****.*

Input Effects on Running Time

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

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

Average-case input: Usually close to Worst-case, asymptotically.

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 to n)
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 careful note of whether the algorithms in the book (and those you write or see in class) 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

Simple (non-nested) loops:

```
for k = 1…n
    print(k)
```

$$\sum_{k=1}^{n} 1 = n - 1 + 1 = n$$

Handy Summation Rules for Iterative Algorithms

Summation as i goes from 1 to n of i

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + \ldots + n = (n(n+1))/2$$

Proof of above: Add 1…n to n…1 to get n(n+1). Then divide by 2.

$$\begin{array}{l} (1+2+3+\ldots+n) \\ +\,(n+(n-1)+\ldots+1) \\ \hline (n+1)+(n+1)+\ldots(n+1) \end{array} = \frac{n(n+1)}{2}$$

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
```

NOTE: When a while or for loop is executed, the test in the loop header is executed one more time than the code in the loop body.

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

Expressing loops as summations

Algorithm PrefixAverages-v1(X):
Input: An n-element array X of numbers

Output: An n-element array A of numbers such that A[i] is the average of elements X[1], … , X[i].

```
1. Create an array A such that length[A] = n
2. s = 0
3. for ( i = 1 to n )
4.     s = s + X[i]
5.     A[i] = s / i
6. return A
```

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

Analysis of PrefixAverages-v1

1. Create an array A such that length[A] = length[X] = n
2 . s = 0
3. **for (j** = 1 **to** length[X])
4. **s** = **s** + X[**j**]
5. A[**j**] = s / **j**
6. **return** A

To do in class: Give T(n) and are there b-c & w-c inputs?

1. T(n) = 4n + 3
2. Are there best- and worst-case inputs? No

Expressing loops as summations

Algorithm PrefixAverages-v2(X):
Input: An n-element array X of numbers

Output: An n-element array A of numbers such that A[i] is the average of elements X[1], … , X[i].

1. Create an array A such that length[A] = n
2. **for** (j = 1 to n)
3. a = 0
4. **for** (i = 1 to j)
5. a = a + X[i]
6. A[j] = a / j
7. return A

$$\sum_{j=1}^{n}\sum_{i=1}^{j}1=\sum_{j=1}^{n}j-1+1=\sum_{j=1}^{n}j=\frac{(n^2+n)}{2}$$

Analysis of PrefixAverages-v2

1. Create an array A such that length[A] = n
2. **for** (j = 1 to n)
3. a = 0
4. **for** (i = 1 to j)
5. a = a + X[i]
6. A[j] = a / j
7. return A

$$\sum_{j=1}^{n}\sum_{i=1}^{j}1=\sum_{j=1}^{n}j-1+1=\sum_{j=1}^{n}j=\frac{(n^2+n)}{2}$$

To do in class: Give T(n) and are there b-c & w-c inputs?

1. T(n) = n^2 + 3n + 2

2. Are there best and worst case inputs? No

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 $a_1' \le a_2' \le \ldots \le a_n'$.

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

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 instances of best-case and worst-case inputs.

Best Case input instance: {2 4 6 7 9 10 12} (already sorted)

Worst Case input instance: {12 10 9 8 5 2 1} (reverse sorted)

Analysis of InsertionSort

InsertionSort sorts an array of elements in ascending order.

InsertionSort(A)	times
1. for j = 2 to length[A]	n
2. key = A[j]	n-1
3. i = j - 1	n-1
4. while i > 0 and A[i] > key	$\sum_{j=2}^{n} t_j$
5. A[i + 1] = A[i]	$\sum_{j=2}^{n} (t_j - 1)$
6. i = i - 1	$\sum_{j=2}^{n} (t_j - 1)$
7. A[i + 1] = key	n-1

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 $\{a_1, a_2, \ldots, a_n\}$

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

```
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