Algorithms Review

- An algorithm is a ~~[illegible]~~ description that provides clear and unambiguous instructions for solving a problem.

- Many times students write down how an algorithm operates on a few special cases or how the algorithm starts and are vague or handwavy about how it works in the general case.

- Please, please, please do not do this.

- If you don't understand how it works in the general case, you aren't finished designing your algorithm.

- The two most important features* of a (deterministic) algorithm are its worst-case run-time and worse-case space (or memory) usage.

*(For theoreticians, anyway)

- We measure how the use of these resources grows as a function of the size of the input

- We use $\mathcal{I}$ to denote the set of instances or inputs to a problem.

Ex: - For the sorting problem,

$$\mathcal{I} = \{ ( ) \} \cup \{ (a_1) \mid a_1 \in \mathbb{R} \} \cup$$

$$\{ (a_1, a_2) \mid a_1, a_2 \in \mathbb{R} \} \cup$$

$$\cdots$$

$$\{ (a_1, a_2, \ldots, a_n) \mid \forall j \;\; a_j \in \mathbb{R} \} \cup \ldots$$

- For problems on graphs,

$$\mathcal{I} = \{ G \mid G \text{ is a graph} \}$$

- ~~[illegible]~~ For an instance $I \in \mathcal{I}$, we use $|I|$ to denote the size of I -- that is, the number of bits needed to store I using a reasonable representation, or some other reasonable notion of size.

Ex: For the sorting problem, we say that the size $|I|$ of an instance $I \in \mathcal{I}$ is the number of elts. we are asked to sort.

$$|(4, -2, 3, 4, 1, 0)| = 6$$

- Let $\mathcal{A}$ be an algorithm which solves a problem with instances $\mathcal{I}$.

- The run-time of $\mathcal{A}$ is the function

$$T(n) = \max_{\substack{I \in \mathcal{I} \\ |I| = n}} \left\{ \begin{array}{c} \text{time used when } \mathcal{A} \text{ runs} \\ \text{on input } I \end{array} \right\}$$

- The Space-usage of $\mathcal{A}$ is the function

$$S(n) = \max_{\substack{I \in \mathcal{I} \\ |I| = n}} \left\{ \begin{array}{c} \text{space used when } \mathcal{A} \\ \text{runs on input } I \end{array} \right\}$$

- The ~~max~~ use of max in these ~~functions~~ definitions causes us to analyze the worst-case behavior of the algorithm.

- What exactly do we mean by an algorithm, or the time/space used by an algorithm on an input $I \in \mathcal{I}$? What are the precise, formal definitions?

- Bad news: the answer depends upon the formal definition/~~specification of~~ model of a computer.

  Eg: Turing Machine, RAM ~~machine~~ model, Lambda calculus

- **Good news**: ~~to any~~ we can translate any algorithm ~~for~~ between ~~models types~~ the various models and only change the time/space usage by a polynomial amount.

- **More good news**: most often,(*) we'll use the **RAM model**, which is closer to our intuition than other models; for example, we can add/multiply/subtract/divide numbers in constant $\Theta(1)$ time.

- **Even more good news**: we won't worry about the details of the formal defn of the **RAM model**: it works the way you think it does.

(*) In complexity theory, the **Turing Machine** is the standard model.

Examples

- Sorting, via BubbleSort:

<u>BubbleSort (A[1..n]):</u>

```
do
  altered ← false
  for i = 1 to n-1
    if (A[i] > A[i+1])
      Swap(A[i], A[i+1])
      altered ← true
until (not altered)
```

- To analyze the algorithm, we must first argue it is <u>correct</u>: that is, no matter what array BubbleSort() is given, it terminates with the contents of A[1..n] in sorted order.
- If the algorithm terminates, the last "for" loop execution ~~once~~ occurs without the altered flag being set to true, so that $A[1] \le A[2] \le \cdots \le A[n]$.

- Therefore to show that the algorithm is correct, it suffices to show it terminates on all inputs.
- So, we complete the <u>proof of correctness</u> and <u>run-time analysis</u> at the same time.
- How long does BubbleSort() take to run?
- On some inputs (eg. $A = (1, 2, 3, \dots, n)$) the algorithm is fast and takes $\Theta(n)$ time, executing the for loop only once.
- Other inputs (e.g. $A = (n, n-1, n-2, \dots, 1)$) take much longer.
- To analyze the run-time, we must look at the <u>worst-case</u>: the array of size n which causes the alg. to run for the longest possible time.

- Exercise: if $A = (n, n-1, \ldots, 1)$ then BubbleSort() executes its "for" loop n times.

  Hence the run-time is at least $n \cdot \Theta(n)$ or $\Omega(n^2)$.

  (To get a lower bound on the run-time, we must prove at least 1 input takes so long.)

- What about an upper-bound?

- Exercise: Let $A[1..n]$ be an array. Show that for each $1 \leq k \leq n$, ~~either the time~~ after the kth execution of the for loop, the largest k elements of A are in their correct, final positions $A[n-k+1], A[n-k+2], \ldots, A[n]$.

  Hint: Induction!

- In particular, the exercise above shows that no matter what $A[1..n]$ is, $A[1..n]$ will be in sorted order after n executions of the for loop; hence, the algorithm executes the for loop at most $n+1$ times and the run-time is <u>at most</u> $(n+1) \cdot \Theta(n)$, or $O(n^2)$.

(To get an upper bound on the run-time, we must ~~[illegible]~~ prove an upper bound on how long the alg. takes on every array of size n.)

- Because we have shown the run-time is $\Omega(n^2)$ and $O(n^2)$, we conclude the run-time of BubbleSort is $\Theta(n^2)$.

- The (worst-case) space usage is just $\Theta(1)$ because the alg. uses constant space for the variables altered and i, and a constant amount of space to support the call and execution of the swap() helper routine.

<u>Remark:</u> There are much better sorting algorithms which ~~run~~ have worst-case $\Theta(n \log n)$ run-times (e.g. Merge-Sort); in fact we can prove that every algorithm which sorts$^{(*)}$ has run-time $\Omega(n \log n)$.

(*) using comparis~~i~~ons