CSE5095: Research Topics in Big Data Analytics **Professor Rajasekaran**
1/21/14 Lecture 1: Introduction to Algorithms note taker: Timothy Becker

There are problems for which we may not be able to devise techniques for solutions. These are known as *undecidable* problems. An example is the halting problem. Problems for which we can develop solution techniques are called *decidable.* An **algorithm** is nothing but a technique that can be used to solve a given problem.

There are decidable problems for which the best known algorithms take a very long time to run. Such problems are known as *intractable.* Decidable problems for which the best known algorithms take a reasonable amount of time are *tractable.* By reasonable amount of time we typically mean a run time that is a polynomial in the input size.

An algorithm is specified in a machine independent and programming language independent manner. A pseudocode can be used to make the description concise. Consider the following example:

[ex 1] Matrix Multiplication:
input: $A_{n\times n}, B_{n\times n}$
output: $C_{n\times n} = AB$

algorithm:

```
for i := 1 to n do:
      for j := 1 to n do:
            C[i,j] := 0
            for k := 1 to n do:
                  C[i,j] := C[i,j]+A[i,k]*B[k,j]
```

Performance Measures are machine and program language independent metrics. Such metrics, for example, can be used to compare different algorithms. Both time and space need to be considered for Big Data Analytics.

[1] time complexity – number of basic operations needed (as a function of the input size)
[2] space complexity – number of memory cells needed (as a function of the input size)
[3] input size – number of memory cells used to define a problem instance.

[ex 2] Matrix Multiplication has:
input size: $n^2 + n^2 = 2n^2$
time complexity: $n^2[n + (n-1)] = 2n^3 - n^2$
space complexity: n^2

Different time complexities
Even if we know the kind of problem we try to solve and the input size, we may not be able to specify the time complexity.

As an example, consider the problem of determining if an element *x* is a member of an array A[1:*n*]. The run time in this case will be dependent on the input instance. If *x* is the first element of the array we make only one comparison. If *x* is the second element (and not the first element) then we make 2 comparisons, and so on.

We can define (at least) three different time complexities, namely, *best case, worst case, and average.*

[ex 3] Array Search: best time: 1
input: $x, A[1:n]$ worst time: n
output: $is\ x \in A[1:n]$? average time: $\sum_{i\in D} p_i t_i$ where $D = the\ set\ of\ all\ inputs, t_i = time\ on\ input\ i$ and p_i is the probability of input *i.*

Average time for array search (assuming that each of the (*n+1*) possibilities is equally likely) = $\frac{1+2+\cdots+n+n}{(n+1)} = \frac{n(n+1)}{2(n+1)} + \frac{n}{(n+1)} = \frac{n}{2} + \frac{n}{(n+1)} \sim \frac{n}{2}$

Asymptotic Functions

[1] Big Oh: we say $f(n) = \mathrm{O}\big(g(n)\big)\ if\ f(n) \le cg(n)\ \forall n \ge n_0$
where $c\ and\ n_0$ are constants.

[2] Little oh: we say $f(n) = \mathrm{o}\big(g(n)\big)\ if\ \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$

[3] Big Omega: we say $f(n) = \Omega\big(g(n)\big)\ iff\ g(n) = O(f(n))$

[4] Big Theta: we say $f(n) = \Theta\big(g(n)\big)\ iff\ f(n) = \mathrm{O}\big(g(n)\big)\ and\ g(n) = \mathrm{O}(f(n))$.

[ex 4] $5n^2 + 7n - 15000 = \mathrm{O}(n^2) = \mathrm{O}(n^3) = \cdots$

[ex 5] $f(n) = n,\ g(n) = n \log\log n$
$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{1}{\log\log n} = 0 \Rightarrow f(n) = \mathrm{o}\big(g(n)\big)$.

Theorem: If $f(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n^1 + a_0 n^0$ then $f(n) = \Theta(n^k)$.

Question: But how is it that we go about designing algorithms that are good?
Answer: There is no magic recipe, and so this is really more of an art than a science.

Randomized Algorithms are algorithms that utilize the outcomes of coin flips in making some of the decisions.

There are two main kinds of Randomized Algorithms:

[1] A Monte Carlo: algorithm always runs for a predetermined amount of time. It provides the correct answer with a high probability. (The probability of an incorrect answer is low).

[2] A Las Vegas: algorithm always provides the correct answer but the run time is a random variable. We normally prefer to prove high probability bounds on this random variable.

Definition: By low probability we mean a probability of $\leq n^{-\alpha}$, where n is the input size and α is a probability parameter (typically assumed to be a constant ≥ 1).

Definition: High probability refers to a probability that is $\geq 1 - n^{-\alpha}$.

[ex 6] $If\ n = 100{,}000 = 10^5,\ and\ \alpha = 100$
$then\ n^{-\alpha} = 10^{-500}$.

[ex 7] Intersection Element Selection:
input: $A, B\ where\ |A| = |B| = n, |A \cap B| = \sqrt{n},\ and\ B\ is\ sorted$
output: $Any\ x \in A \cap B$

A deterministic algorithm:
(1) sort A in $O(n \log n)$ time
(2) merge A and B (return when an element is equal in the merge) in $2n$ time
Total time: $O(n \log n) + 2n = O(n \log n)$

A Las Vegas algorithm:

```
Repeat:
      Pick a random element x of A                        |basic step
      check if x is in B by performing a binary Search    |
      if x is in B: report x and exit                     |
Forever
```

Analysis:
How many times do we have to perform the basic step?

Probability of success in one basic step is
$\frac{\sqrt{n}}{n} = \frac{1}{\sqrt{n}} \Rightarrow$ failure in the first k basic steps is $(1 - 1/\sqrt{n})^k$.

We want this probability to be $\leq n^{-\alpha}$. This probability is $(1 - \frac{1}{\sqrt{n}})^{\sqrt{n}\left(\frac{k}{\sqrt{n}}\right)}$. Using the fact that: $(1 - x)^{\frac{1}{x}} \leq \frac{1}{e} :\ 0 < x < 1$ we require: $e^{-k/\sqrt{n}} \leq n^{-\alpha} \Rightarrow \frac{-k}{\sqrt{n}} \leq -\alpha \log_e n \Rightarrow k \geq \alpha\sqrt{n} \log_e n$. This implies that the time complexity of the algorithm is $\tilde{O}(\sqrt{n} \log^2 n)$.

Definition: We say that the run time of a **Las Vegas** algorithm is $\tilde{\mathbf{O}}(f(n))$ if the runtime is: $\leq c\boldsymbol{\alpha} f(n), \forall\ n \geq n_0$ **with probability** $\geq (\mathbf{1} - \boldsymbol{n^{-\alpha}})$ *where* c *and* n_0 are constants.

(Note:) This definition captures the tradeoff between probability and the resource bounds.

End of Lecture 1.