CS4102, Algorithms, Spring 2010

First Principles

- Properties of algorithms
- Counting basic operations
- Time and space complexity
- Worst-case and average-case
- Lower Bounds and Optimality
- …and one slide of summations

Analyzing Algorithms and Problems

- We analyze algorithms with the intention of *improving* them, if possible, and for *choosing* among several available for a problem.

- Correctness
- Simplicity
- Amount of work done, and space used
- Optimality

Correctness can be proved!

- An algorithm consists of sequences of steps (operations, instructions, statements) for *transforming*
  inputs (preconditions) to outputs (postconditions)
- Proving
  - if the preconditions are satisfied,
  - then the postconditions will be true,
  - when the algorithm terminates.
- Good news for you!
  - This course does not <u>emphasize</u> proving correctness.

Simplicity

- Simplicity in an algorithm is a virtue.

- Understandability matters
  - Especially for long-lived software
  - Easier to understand, more difficult to break it when making changes later

We said: Analyzing Algorithms and Problems

- Some terms from page 51 in text about problems:
  - Feasible, tractable problems
  - Intractable problems
    - The class of NP-complete problems
  - Unsolvable problems
    - The Halting Problem
- Is a problem solvable? If so, is it possible to find a reasonably efficient solution?

Levels for Talking about Problem Solving

- **Defining** the problem
- Describing an overall **strategy**
- Describing an **algorithm**:
  - Inputs and outputs
  - Describing the processing steps to transform input to output
- **Analysis**
  - Correctness; Time & Space
  - Is it an optimal algorithm?
- **Implementation** issues
- **Verification**: Is it guaranteed correct?

Example: Search in an unordered array

- Problem:
  - Let *list* be an array containing *n* entries, list[0], …, list[n-1], in no particular order.
  - Find an index of a specified key *target*, if it's in the array;
  - return –1 as the answer if *target* is not in the array.
- Strategy:
  - Compare *target* to each entry in turn until a match is found or the array is exhausted.
  - If *target* is not in the array, the algorithm returns –1 as its answer.

Example: Defining the Algorithm (1)

- Inputs and outputs
  - Input: *list, n, target*, where *list* is an array with *n* entries (indexed 0, …, n-1), and *target* is the item sought. For simplicity, we assume that *target* and the entries of *list* are integers, as is n.
  - Output: Returns *ans*, the location of *target* in *list* (-1 if *target* is not found.)

- Note this description defines the data structure used
  - Very common!

Example: Defining the Algorithm (2)

```
int seqSearch(int[] list, int n, int
  target)
1.   int ans, index;
2.   ans = -1; // Assume failure.
3.   for (index = 0; index < n; index++)
4.        if (target == list [index]) {
5.             ans = index; // Success!
6.             break; // Done!
          }
7.   return ans;
```

Example: Defining the Algorithm (2)

```
def seq_search(list, target):
    ans = -1
    i = 0
    for cur in list:
        if cur == target:
            ans = i
            break
        i = i + 1
    return ans
```

Algorithms: Amount of work done

- We want a measure of work that tells us something about the *efficiency* of the method used by the algorithm
- independent of computer, programming language, programmer, and other implementation details.
- Usually depending on the *size of the input*

- Counting passes through loops
- Basic Operation
  - Identify a particular operation fundamental to the problem
  - the total number of operations performed is roughly proportional to the number of basic operations
- Identifying the properties of the inputs that affect the behavior of the algorithm

Worst-case complexity

- Let D_n be *the set of inputs of size n* for the problem under consideration, and let I be an element of D_n.
- Let t(I) be *the number of basic operations* performed by the algorithm on input I.
- We define the function W(n) by

- $W(n) = \max\{ t(I) \mid I \in D_n \}$
  - called the *worst-case complexity* of the algorithm.
  - W(n) is the maximum number of basic operations performed by the algorithm on any input of size n.
- The input, I, for which an algorithm behaves worst depends on the particular algorithm.

Our example:

- Basic Operation:
  - Comparison of *target* with an array entry
- Worst-Case Analysis:
  - We just said that:
    W(n) is the maximum number of basic operations performed by the algorithm on any input size n.
  - For our example, clearly W(n) = n.
  - What is the worst-case input?
    - *target* is not in the array at all
    - *target* appears only in the last position in the array

Why Measure Worst-Case?

- Are we just pessimists? Give some reasons:
  - (Your ideas here)

- (Some answers)
- We want a upper-bound on behavior
  - Guaranteed no worse than W(n)
- Perhaps the worst-case happens often?
- Average-case is harder to calculate

Average Complexity

- Let P(I) be the *probability* that input I occurs.
- Then the average behavior of the algorithm is defined as:

  $A(n) = \sum_{I \in D_n} P(I)\, t(I).$

  - We determine t(I) by analyzing the algorithm,
  - but P(I) cannot be computed analytically.

Average Complexity (2)

- Sometimes an algorithm succeeds with some known probability:

  $$A(n) = P(succ) \times A_{succ}(n) + P(fail) \times A_{fail}(n)$$

- An element I in D_n may be thought as a set or equivalence class that affect the behavior of the algorithm. (see following e.g. n+1 cases)

Our example: Average-Behavior Analysis

- $A(n) = P(succ) \times A_{succ}(n) + P(fail) \times A_{fail}(n)$
- There are total of n+1 cases of I in D_n
    - Let *target* is in the array be the "succ" cases. There are n cases.
    - Assuming *target* is equally likely found in any of the n location, i.e. $P(I_i \mid succ) = 1/n$
    - for $0 <= i < n$, $t(I_i) = i + 1$
    - $A_{succ}(n) = \sum_{i=0}^{n-1} P(I_i \mid succ)\, t(I_i)$
      $= \sum_{i=0}^{n-1} (1/n)\,(i+1) = (1/n)[n(n+1)/2] = (n+1)/2$
    - Let *target* is not in the array be the "fail" case – just 1 cases, $P(I \mid fail) = 1$
    - Then $A_{fail}(n) = P(I \mid fail)\, t(I) = 1 \times n$
- Let q be the probability for the succ cases
    - $q\,[(n+1)/2] + (1-q)\,n$

Optimality "the best possible"

- Each problem has inherent complexity
  - There is some *minimum* amount of work required to solve it.
- To analyze the complexity of a problem,
  - we choose a class of algorithms, based on which
  - prove theorems that establish a *lower bound* on the number of operations needed to solve the problem.
- Lower bound (for the worst case)

Show whether an algorithm is optimal?

- Analyze the algorithm, call it A, and find the worst-case complexity $W_A(n)$, for input of size n.
- Prove a theorem starting that,
  - for any algorithm in the same class of A…
  - for any input of size n, there is some input for which the algorithm must perform…
  - at least $W_{[A]}(n)$ (lower bound in the worst-case)
- If $W_A(n) = W_{[A]}(n)$
  - then the algorithm A is optimal
- Otherwise, there may be a better algorithm
  - OR there may be a better lower bound.

FindMax example: Optimality

- Problem
  - Finding the largest entry in an (unsorted) array of n numbers
- Algorithm A

```
def find_max(list):
    max = list[0]  # first entry
    for cur in list[1:]: # from 2nd entry on
        if max < cur:
            max = cur
    return max
```

Analyze the algorithm, find $W_A(n)$

- Basic Operation
  - Comparison of an array entry with another array entry or a stored variable.
- Worst-Case Analysis
  - For any input of size n, there are exactly n-1 basic operations
  - $W_A(n) = n-1$

For the class of algorithm [A], find $W_{[A]}(n)$

- Class of Algorithms
  - Algorithms that can compare and copy the numbers, but do no other operations on them.
- Finding (or proving) $W_{[A]}(n)$
  - Assuming the entries in the array are all distinct
    - (permissible for finding lower bound on the worst-case)
  - In an array with n distinct entries, $n - 1$ entries are not the maximum.
  - To conclude that an entry is not the maximum, it must be smaller than at least one other entry. And, one comparison (basic operation) is needed for that.
  - So at least n-1 basic operations must be done.
  - $W_{[A]}(n) = n - 1$
- Since $W_A(n) = W_{[A]}(n)$, algorithm A is optimal.

Our Search Example: Optimality

- Is sequential search optimal? Yes.
- Are there more efficient solutions? Yes.
  - Binary search
  - Hashing
- Is this a contradiction?

- Binary search is optimal
  - W(n) = $\lceil \lg(n+1) \rceil$

Space Usage

- If memory cells used by the algorithms depends on the particular input,
  - then worst-case and average-case analysis can be done.
- Time and Space Tradeoff.

Problems!

(To think about, maybe after studying sorting later)

1. You have 1000's of phone bills and 1000's of checks. Find who didn't pay.

2. You have a list of 30 publishers, and a list of books in library that records publisher. Count how many books published by each of the 30.

3. You have book check-out records for all library users who checked out books in the last year. Count how many distinct users checked out at least one book.

Just one math slide (for now):

Series

- A *series* is the sum of a sequence.
- Arithmetic series
  - The sum of consecutive integers: $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$
- Polynomial Series
  - The sum of squares: $\sum_{i=1}^{n} i^2 = \frac{2n^3 + 3n^2 + n}{6} \approx \frac{n^3}{3}$
  - The general case is: $\sum_{i=1}^{n} i^k \approx \frac{n^{k+1}}{k+1}$
  - Powers of 2: $\sum_{i=0}^{k} 2^i = 2^{k+1} - 1$
- Arithmetic-Geometric Series: $\sum_{i=1}^{k} i2^i = (k-1)2^{k+1} + 2$