Algorithms in Systems Engineering
IE172

Midterm Review

Dr. Ted Ralphs

Textbook Sections Covered on Midterm

Chapters 1-5

Introduction to Algorithms and Object-oriented Programming

Algorithms

- The main theme of the course is the design, implementation, and analysis of algorithms and data structures for solving problems.
- A *problem* specifies the form of the output desired for a given set of inputs.
- An algorithm is a specific procedure for converting input to output.
- An algorithm is said to be *correct* for a given problem if it successfully converts each possible input into an output of the desired form, called a *solution*.
- Ultimately, we are interested in algorithms that are fast.
- We will judge the speed of an algorithm by the number of fundamental operations required to execute it, generally called the *running time*.
- This provides a measure that is independent of hardware.

Object Oriented Programming

- An important underlying philosophy in OO programming is *modularity*.
- A *data structure* is a general scheme for storing and manipulating data, usually independent of a particular programming language.
- An *abstract data type* specifies a particular set of methods for storing and maniupating data within a particular programming language.
- Abstract data types (ADT) extend the set of basic data types available in a language.
  - The *interface* defines how the ADT is used by clients.
  - The *implementation* is the way in which the required operations are implemented internally.
- An ADT can have different implementations, but this is transparent to the client.
- The only difference the client sees is in *performance*.
- The most appropriate implementation for a given application depends on which operations are to be performed most frequently.

Algorithms and Data Structures

- Abstract data types not only use algorithms within their own implementations, but are also building blocks for larger procedures (this is the role of modularity).
  - The list data structure was used in Lab 1 as a building block within a larger algorithm for implementing the Game of Life.
  - A priority queue is a tree-based tree data structure we used to store the event queue in the simulation of Lab 5.
- The analysis of algorithms starts with the analysis of basic data structures and builds up to larger algorithms.
- The goal of analyzing data structures is to understand how they will perform in different applications.

Python

- You should know Python syntax and be able to write simple Python codes out correctly by hand.
- You should know all of the basic Python data structures and functions we've used so far in the labs.
  - Lists
  - Linked Lists
  - Stacks
  - Queues
  - Dictionaries
- You should understand how lists and dictionaries are implemented in the Python language and the basic performance tradeoffs in using them.
- You should understand Python's object-oriented features, such as the class mechanism.
- You should understand how Python's "magic methods" work and how they implement the functionality of objects.

Analysis of Algorithms

Analyzing Algorithms

- The goal of analyzing an algorithm is to determine its performance characteristics.
- The usual measure of performance is "running time," which can be determined either theoretically or empirically.
- Empirical analysis is based on implementing an algorithms on a particular computer in a particular programming language and seeing how it performs.
  - Empricial analysis can be time-consuming and extraneous factors specific to the hardware and language may make it difficult to judge performance.
  - If done correctly, it can be a good indicator of real-world performance.
- Theoretical analysis considers the execution of an abstract version of the algorithm on an abstract model of a computer.
  - To eliminate the effects of hardware and operating environment, we count individual operations.
  - In many cases, it is clear what the dominant operations will be and focus on counting those.

Theoretical Analysis

- A *problem instance* is a specific set of inputs with respect to which we want to solve a give problem.
- The running time of an algorithm is different for different instances.
- This creates difficulties for analysis and leads to two different summary measures of running time.
  - Worst-case
  - Average-case
- Worst-case is almost always easier to compute, but average-case is often a more useful measure.
- We can use either *theoretical* or *empirical* measures to determine running time.

Models of Computation

- In order to analyze the number of steps necessary to execute an algorithm *theoretically*, we have to say what we mean by a "step."
- To define this precisely is tedious and beyond the scope of this course.
- A precise definition depends on the exact hardware being used.
- Our analysis will assume a very simple model of a computer called a *random access machine* (RAM).
- In a RAM, the following operations take one step.
  - arithmetic (addition, subtraction, multiplication, division)
  - data movement (read from memory, store in memory, copy)
  - comparison
  - control (function calls, goto commands)
- This is a very idealized model, but it works in practice.
- We will sometimes need to simplify the model even further.

The Input Size

- The running time of an algorithm generally depends primarily on the number of input values, or the *size of the input*.
- We are interested in how the running time grows generally as the input size grows.
  - Because we are mainly interested in how the running time grows as the instances become larger, we won't need "exact" running times.
  - We will allow some "sloppiness" and ignore constants and low order terms.
  - Because of our many simplifying assumptions, the low order terms may not be accurate anyway.
- The growth rate of algorithms gives us a basis for comparison.
  - Any algorithm can be used to solve a small problem.
  - It is the really large problems that require efficient algorithms.

Growth of Functions

- Consider algorithm A with running time given by f and algorithm B with running time given by g.
- We are interested in knowing

$$L = \lim_{n \to \infty} \frac{f(n)}{g(n)}$$

- There are four possibilities.
  - $L = 0$: g grows faster than f.
  - $L = \infty$: f grows faster than g.
  - $L = c$: f and g grow at the same rate.
  - The limit doesn't exist.

Θ Notation

- We now define the set

$$\Theta(g) = \{f : \exists c_1, c_2, n_0 > 0 \text{ such that } c_1 g(n) \leq f(n) \leq c_2 g(n)\ \forall n \geq n_0\}$$

- If $f \in \Theta(g)$, then we say that f and g *grow at the same rate* or that they are *of the same order*.
- Note that

$$f \in \Theta(g) \Leftrightarrow g \in \Theta(f)$$

- We also know that if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ for some constant c, then $f \in \Theta(g)$.
- If the limit doesn't exist, we don't know.

Big-O Notation

- We now define the set

$$O(g) = \{f : \exists c, n_0 > 0 \text{ such that } 0 \leq f(n) \leq cg(n) \forall n \geq n_0\}$$

- If $f \in O(g)$, then we say that "f is big-O of g" or that g *grows at least as fast as* f.
- Note that if $f \in O(g)$, then either $f \in \Theta(g)$ or $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$
- Some other notation
  - $f \in \Omega(g) \Leftrightarrow g \in O(f)$.
  - $f \in o(g) \Leftrightarrow f \in O(g) \setminus \Theta(g) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.
  - $f \in \omega(g) \Leftrightarrow g \in o(f) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$.

Recursion

- A recursive function is one that calls itself.
- There are two basic types of recursive functions.
  - A *linear recursion* calls itself once.
  - A *branching recursion* calls itself two or more times.
- Generally speaking, recursive algorithms should have the following two properties to be guarantee well-defined termination.
  - They should solve an explicit base case.
  - Each recursive call should be made with a smaller input size.
- The use of recursion makes many algorithms easier to implement, but there are drawbacks.
- Recursive implementations usually use more memory and may not be quite as efficient as their nonrecursive counterparts.
- Every recursive algorithm has a nonrecursive counterpart.

Divide and Conquer

- Many common problems can be solved using a *divide-and-conquer* approach.
- This means breaking a larger problem into pieces that can be solved independently.
- The solutions to the various pieces may then have to recombined in some way.
- Divide-and-conquer algorithms have natural implementations using branching recursions.
  - Divide: Divide the input data into smaller pieces.
  - Conquer: Call the algorithm recursively on each piece.
  - Combine: Combine the results into a solution to the original problem.

Analyzing Recursive Algorithms

- The running times of many divide and conquer algorithms can be analyzed by solving a *recurrence*.
- For an input of size n, let the running times of the divide and combine steps be $f(n)$.
- Suppose that the divide step results in a smaller pieces of size n/b (there may be some overlap).
- If $T(n)$ is the overall running time of the algorithm, then $T(n)$ must satisfy the recurrence

$$T(n) = aT(\frac{n}{b}) + f(n)$$

- The running times of other recursive algorithms also give rise to recurrences.

Analyzing Recurrences

- General methods for analyzing recurrences
  - Make a guess and prove that it's right (usually with induction).
  - Build a recursion tree.
  - Use telescoping (generally used for linear recursions).
  - Use the *Master Theorem* (generally used for branching recursions).
- When we analyze a recurrence, we may not get or need an exact answer.
- We may prove the running time is in $O(f)$ or $\Theta(f)$ for some simpler function f.
- When taking the ratio of two integers, it usually doesn't matter whether we round up or down.

The Master Theorem

- We can use the Master Theorem to analyze a divide and conquer recurrence of the form

$$T(n) = aT(n/b) + f(n)$$

- We have to choose from one of three cases:

  1. If $f(n) \in O(n^{\log_b a - \varepsilon})$, for some constant $\varepsilon > 0$, then $T(n) \in \Theta(n^{\log_b a})$.
  2. If $f(n) \in \Theta(n^{\log_b a})$, then $T(n) \in \Theta(n^{\log_b a} \lg n)$.
  3. If $f(n) \in \Omega(n^{\log_b a + \varepsilon})$, for some constant $\varepsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$, then $T(n) \in \Theta(f(n))$.

- If f is a polynomial, then deciding which case we are in is as simple as comparing the degree of f to $\log_b a$.

Empirical Analysis

- Empirical analysis is based on implementing an algorithms on a particular computer in a particular programming language and seeing how it performs.
- The analysis is largely similar to theoretical analysis except we measure performance using real code on a real computer.
- Performance measures
  - Wall-clock running times
  - Representative operation counts
- Plotting the growth of these measures as a function of input size gives us an idea of the real-world average-case performance.
- The test set chosen heavily influences the results, so it must be chosen wisely.
- Generally speaking, if we choose a "representative" test set, then the empirical results will represent the "avergage case."

Verifying Theory with Empirical Results

- Ideally, we would like to do both theoretical and empirical analyses and see if they agree.
- If we plot both on the same graph, remember that the we must scale the results in some way to account for constants.
- The idea is to see whether the two running time functions are the same, i.e., grow at the same rate.
- If the two functions don't agree, we have one of several cases.
- If the empirical running time function is growing more quickly, then either
  - Our theoretical analysis is incorrect.
  - Our implementation is not efficient.
- If the empirical running time function is growing more slowly, then either
  - Our test set is not exhibiting worst case behavior.
  - Our theoretical analysis is not as tight as it could be.

Bugs and Errors

- Regards of the results of our analyses, we always have to account for the possibility of errors and bugs.
- Even when we get agreement between the empirical and theoretical analyses, it's possible that we have off-setting errors.
- We should always view our results as skeptically as possible.
- We should also be careful to verify the output of our program against a known correct output.
- When debugging, it is important to understand how the program is supposed to operate and to verify at each step that the state is correct.
  - We break the program into components, each of whose operations we can verify independently (this is another advanatge of modularity).
  - We thentry to isolate the component that is faulty by determining the first place in the program we are getting an unexpected result.
- Invariants, which are places in the execution where we know what the state has to be if the program is running correctly, help with this.
- When debugging, we usually start by looking for violations of invariants.

Bottleneck Analysis

- The running time of an algorithm is actually a combinations of the running times of individual components.
- The running time of the algorithm as a whole is determined by the running time of the slowest component, the *bottleneck*.
- By determining the percentage of time or total operations in each component as the size of the input increases, we can determine if there is a bottleneck.
- The bottleneck will eventually consume most of the running time for large enough inputs.
- In practice, this analysis can be done using a profiler.

Basic Data Structures

Data Structures

- Algorithms use *data structures* to store and manipulate data during the course of execution.
- A data structure consists of a specified set of data and a set of algorithms for performing operations on the data.
- In Python, data structures have natural implementations as new *data types* (classes).
- A Python class is composed of
  - data attributes, and
  - method attributes.
- The *data attributes* are the values.
- The *method attributes* are the operations to be performed on these values.
- The most important concept to remember is that of separating the definition (interface) from the implementation.

Lists

- A list is the most basic data structure.
- A list stores a set of elements and supports the following operations.
  - Get the number of elements in the list.
  - Get element j.
  - Set element j.
  - Add an element to the list just before element j.
  - Delete element j from the list.
- There are two basic implementations for lists
  - arrays
  - linked lists

Stacks and Queues

- A *stack* is a special kind of list in which items can only be removed in "last-in, first-out" (LIFO) order.
- A *queue* is a list in which items can only be removed in "first-in, first-out" (FIFO) order.
- The basic operations on a stack are
  - push a new item on the stack.
  - pop the most recently added item off the stack.
- The basic operations on a queue are
  - enqueue a new item.
  - dequeue the most recently added item.
- Stacks and queues can also be implemented using either arrays or linked lists.

Priority Queues

- A priority queue is a data structure for maintaining a list of items that have associated *priorities*.
- The usual operations are
  - construct a queue from a list of items.
  - find the item with the highest priority.
  - insert an item.
  - delete an item.
  - change the priority of an item.
- The most common implementation of priority queues is using a *heap*.
- A heap is a binary tree in which the record stored at each node has a higher priority than either of its children.

Searching

Symbol Tables and Dictionaries

- In the last few lectures, we discussed various methods for sorting a list of items by a specified key.
- We now consider further operations on such lists.
- A *symbol table* is a data structure for storing a list of items, each with a key and satellite data, supporting the following basic operations.
  - construct a symbol table.
  - search for an item (or items) having a specified key.
  - insert an item.
  - remove a specified item.
  - count the number of items.
  - print the list of items.
- Symbol tables are also called *dictionaries* because of the obvious comparison with looking up entries in a dictionary.
- Note that the keys may not have an ordering.

Additional Operations on Symbol Tables

- If the items can be ordered, e.g., have an implementation of `__gt__()` and `__eq__()`, we may support the following additional operations.
  - Sort the items (print them in sorted order).
  - Return the maximum or minimum item.
  - Select the k^{th} item.
  - Return the successor or predecessor of a given item.
- We may also want to be able to join two symbol tables into one.
- These operations may or may not be supported in various implementations.
- The easiest implementation is using an array.

Hash Tables

- *Hash tables* are a data structure for storing a dictionary that supports only the operations
  - insert,
  - delete, and
  - search.
- Most data structures for storing dictionaries depend on using comparison and exchange to order the items.
- This limits the efficiency of certain operations (recall the lower bound on the efficiency of comparison-based sorting).
- A *hash table* is a generalization of an array that takes advantage of our ability to access an arbitrary array element in constant time.
- Using hashing, we determine where to store an item in the table (and how to find it later) without using comparison.
- This allows us to perform all the basic operations extremely efficiently.

Hash Functions

- A *hash function* is a function $h : U \rightarrow 0, \ldots, M-1$ that takes a key and converts it into an array index (called the *hash value*).
- Once we have a hash function, we can use the very efficient array-based implementation to store the table.
- A good hash function minimizes collisions and is easy to compute.
- For a "random" key, we would like the probability of each hash value to be "equally likely."
- A simple method to hash a key x, take $x \mod M$, where M is the size of the hash table (typically a prime number).
- This is called *modular hashing* and is a very popular form of hashing.

Resolving Collisions

- There are two primary of methods of resolving collisions.
- Chaining: Form a linked list of all the elements that hash to the same value.
  - Easy to implement.
  - The table never "fills up" (better for extremely dynamic tables)
  - May use more memory overall.
  - Easy to insert and delete.
- Open Addressing: If the hashed address is already used, use a simple rule to systematically look for an alternate.
  - Very efficient if implemented correctly.
  - When the table is nearly full, basic operations become very expensive.
  - Deleting items can be very difficult, if not impossible.
  - Once the table fills up, no more items can be added until items are deleted or the table is reallocated (expensive).

Sorting

The Sorting Problem

- The sorting problem is fundamental to the study of algorithms.
- Algorithms for sorting are used in a vast number of applications and much is known about them.
- Most often, the items to be sorted are individual *records*, usually consisting of a *key* and related *satellite data*.
- The sorting problem is defined as follows.
  Input: A sequence of n records $a_1, a_2, \ldots, a_n$.
  Output: A reordering $a'_1, a'_2, \ldots, a'_n$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.
- Note that the records can be anything for which a "$\leq$" operator can be defined (usually by comparing the specified key).
- It is known that the running time of any comparison-based sorting algorithm is in $\Omega(n \lg n)$.

Properties of Sorting Algorithms

- In addition to running time, there are a few important properties of sorting algorithm that we may need to consider.
  - A *stable* sorting algorithm is one that leaves duplicate keys in the same relative order that they were in the original list.
  - This is an important property if you want to be able to sort on multiple keys.
  - Another important consideration is whether the algorithm sorts *in place*, i.e., does not have to allocate too much extra memory.
  - Finally, we might consider how well the algorithm performs on arrays that are already sorted, or mostly sorted.
- The sorting algorithm you choose may depend on what you expect the data to look like, e.g., is it "almost sorted."
- The basic operations performed in sorting are comparison and exchange.
- The relative cost of these operations may also help determine the type of sort that is most appropriate.

Basic Sorting Algorithms

- Most straightforward sorting algorithms have a running time in $O(n^2)$.
- Nonadaptive algorithms perform the same sequence of steps for any input and have running times that are very consistent.
  - Selection sort
  - Bubble sort
- Adaptive algorithms can have dramatically different running times for different inputs.
  - Insertion sort: Fast for data that is "almost sorted."
  - Quicksort: Fast for "random" data.

Commonly Used Sorting Algorithms

- Insertion Sort
  - Very efficient for "almost sorted" data.
  - Can be implemented in place and is stable.
  - Slow on average.
- Merge Sort
  - Asymptotically optimal and stable.
  - Cannot be implemented in place.
- Heap Sort
  - Heap sort is asymptotically optimal and can be implemented in place, but it is unstable.
  - Heap sort is based on the construction of a priority queue.
- Quicksort
  - Randomized quicksort has excellent average case performance ($\Theta(n \lg n)$) and can be implemented in place.
  - However, it is unstable and can result in a large call stack and poor performance if not implemented carefully.