csci 210: Data Structures

Program Analysis

Summary

- Summary
  - analysis of algorithms
  - asymptotic analysis and notation
    - big-O
    - big-Omega
    - big-theta
  - commonly used functions
  - discrete math refresher

Analysis of algorithms

- Analysis of algorithms and data structure is the major force that drives the design of solutions.
  - there are many solutions to a problem: pick the one that is the most efficient
  - how to compare various algorithms? Analyze algorithms.

- Algorithm analysis: analyze the cost of the algorithm
  - cost = time: How much time does this algorithm require?
  - The primary efficiency measure for an algorithm is time
    - all concepts that we discuss for time analysis apply also to space analysis
  - cost = space: How much space (i.e. memory) does this algorithm require?
  - cost = space + time
  - cost = bandwidth (amount of data sent over the internet)
  - etc

Analysis of algorithms

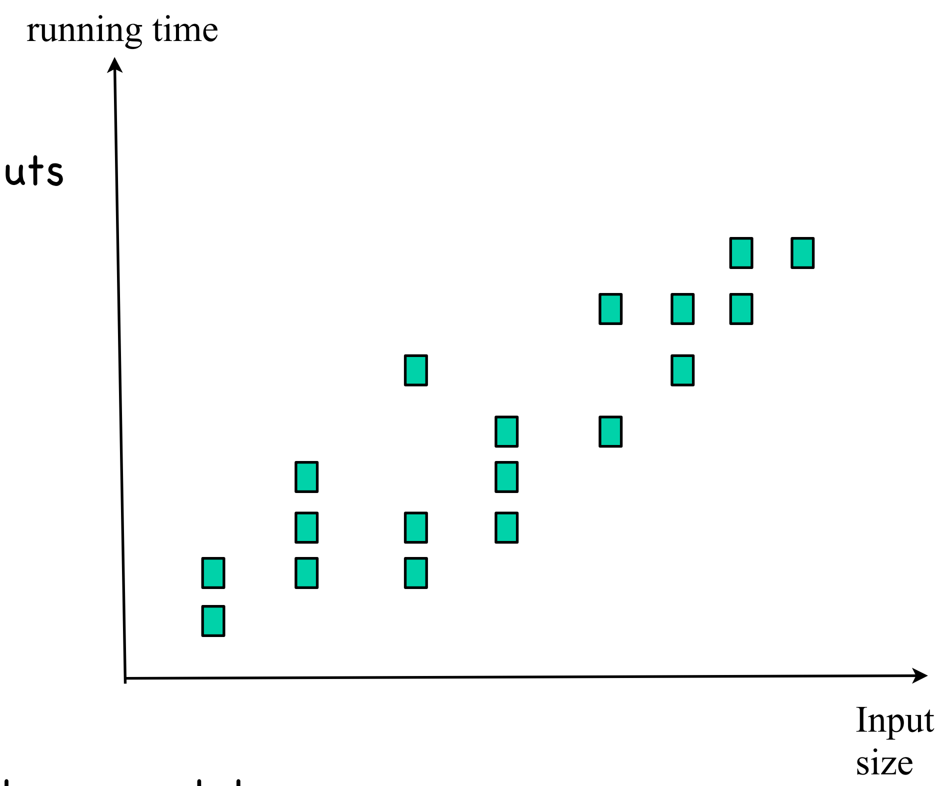
- Running time of an algorithm:
  - it increases with input size
  - on inputs of same size, it can vary from input to input
  - it depends on hardware
    - CPU speed, hard-disk, caches, bus, etc
  - it depends on OS, language, compiler, etc
- Everything else being equal
  - we'd like to compare algorithms
  - we'd like to study the relationship running time vs. size of input

Analysis of algorithms

- How to measure running time of an algorithm?
  - 1. experimental studies
  - 2. theoretical analysis

Analysis of algorithms

- Experimental analysis
  - implement
  - chose various input sizes
  - for each input size, chose various inputs
    - run algorithm
    - time
    - compute average
    - plot



- Limitations
  - need to implement the algorithm
  - need to implement all algorithms that we want to compare
  - need many experiments
  - try several platforms
- Advantages
  - find the best algorithm in practice

Analysis of algorithms

- We would like to analyze algorithms without having to implement them

- Basically, we would like to be able to look at two algorithms flowcharts and decide which one is better

===> theoretical analysis

Theoretical analysis

- RAM model of computation
  - Assume all operations cost the same
  - Assume all data fits in memory
- Running time (efficiency) of an algorithm:
  - the number if operations executed by the algorithm
- Does this reflect actual running time?
  - multiply nb. of instructions by processor speed
    - 1GHz processor ==> 10^9 instructions/second
- Is this accurate?
  - Not all instructions take the same...
  - Various other effects.
  - Overall, it is a very good predictor of running time

Terminology

- Notation: n = size of the input to the problem
- Running time:
  - number of operations/instructions executed on an input of size n
  - expressed as function of n: f(n)

- For an input of size n, running time may be smaller on some inputs than on others
- Best case running time:
  - the smallest number of operations on an input of size n
- Worst-case running time:
  - the largest number of operations on an input of size n
- For any n
  - best-case running time(n) <= running time(n) <= worst-case running time (n)

- Ideally, want to compute average-case running time
  - need to know the distribution of the input
  - often assume uniform distribution (all inputs are equally likely), but this may not be realistic

Examples

- Linear search
- Binary search
- Selection sort
- Insertion sort
- Bubble sort

Linear search

```
//return the position of first occurrence or -1 if not found
int search (double a[], double target)  {
    for (int i=0;  i< a.length; i++)
        if (a[i]  == target)  return i;
    //if we got here, no element matched
    return -1;
}
```

- Analysis
  - best-case: constant
  - worst-case: (order of) n   <------------ linear time

- Other examples (of linear time)
  - doing one pass through an array of n elements, for e.g. finding min/max/average in an array, computing sum in an array

    ```
    int sum = 0
    for (int i=0; i< a.length; i++)
        sum += a[i]
    ```

Binary search

```
//return the index where key is found in a, or -1 if not found
public static int binarySearch(int[] a, int key) {
    int left = 0;
    int right = a.length-1;
    while (left <= right)  {
        int mid = left + (right-left)/2;
        if (key < a[mid]) right = mid-1;
        else if (key > a[mid]) left = mid+1;
        else return mid;
    }
    //not found
    return -1;
}
```

- running time:
  - best case: constant
  - worst-case: lg n   <-------------- logarithmic time

    Why? input size halves at every iteration of the loop

Math refresher

- The arithmetic sum: $1 + 2 + 3 + 4 + ... + (n-2) + (n-1) + n = n(n+1)/2$

- Proof:

Selection sort

```
//selection sort:
for (i=0; i < n-1; i++)
   minIndex = index-of-smallest element in a[i..n-1]
   swap a[i] with a[minIndex]
```

- Analysis
  - index-of-smallest element in a[i..j] takes j-i+1 operations
  - n + (n-1) + (n-2) + (n-3) + ... + 3 + 2 + 1
  - this is n^2 <-------------------- quadratic

  - best case?
  - worst-case?

Bubble sort

```
//assume an array a of n elements:   a[0], ....a[n-1]
for k=1 to n-1
   //do a swap pass
   for i=0 to n-2
      if (a[i] > a[i+1])  then swap a[i], a[i+1]
```

- Analysis

    Best-case?

    Worst-case?

Insertion sort

```
//input: array a[] of size n
for i=1 to n-1
   //invariant: a[0]...a[i-1] is sorted
   shift  a[i] to its correct place so that a[0]...a[i] is sorted
```

- Analysis
  - best case
  - worst-case

Asymptotic analysis

- Focus on the growth of rate of the running time, as a function of n
- That is, ignore the constant factors and the lower-order terms
- Focus on the big-picture
- Example: we'll say that 2n, 3n, 5n, 100n, 3n+10, n + lg n, are all linear

- Why?
  - constants are not accurate anyways
  - operations are not equal
  - capture the dominant part of the running time

- Notations:
  - Big-Oh:
    - express upper-bounds
  - Big-Omega:
    - express lower-bounds
  - Big-Theta:
    - express tight bounds (upper and lower bounds)

Big-Oh

- Definition:
  - f(n), g(n)
  - f is O(g) if exists c > 0 and n_0 such that f(n) <= cg(n) for all n >= n_0
- Big-oh represents an upper bound
- When we say f is O(g) this means that
  - f <= g asymptotically
  - g is an upper bound for f
  - f stays below g as n goes to infinity
- Another way to check is to compute the limit f/g when n goes to infinity
  - if this limit is 0 or a constant ==> f is O(g)
  - if this limit is infinity ==> g is O(f)
- Examples:
  - 2n is O(n), 100n is O(n)
  - 10n + 50 is O(n)
  - 3n + lg n is O(n)
  - lg n is O(log_10 n),
  - lg_10 n is O(lg n)

Exercises

- Mark as true or false:
  - $100n$ is $O(n)$
  - n is $O(n)$
  - $15n+7$ is $O(\lg n)$
  - $15n+7$ is $O(n^2)$
  - $5n^2+4$ is $O(n)$
  - $4n^2+9n+8$ is $O(n^2)$
  - $4n^2+9n+8$ is $O(n^3)$

Big-Omega

- Definition:
  - f(n), g(n)
  - f is Omega(g) if exists c>0 such that f(n) >= cg(n) for all n >= n0
- Big-omega represents a lower bound
- When we say f is Omega(g) this means that
  - f >= g asymptotically
  - g is a lower bound for f
  - f stays above g as n goes to infinity
- Another way to check is to compute the limit f/g when n goes to infinity
  - if this limit is a constant or infinity ==> f is Omega(g)
  - if this limit is 0 ==> g is Omega(f)
- Examples:
  - 3nlgn + 2n is Omega(n)
  - 2n + 3 is Omega(n)
  - $4n^2 + 3n + 5$ is Omega(n)
  - $4n^2 + 3n + 5$ is Omega(n^2)
- O() and Omega() are symmetrical: f is O(g) <====> g is Omega(f)

Exercises

- Mark as true or false:
  - 100n is Omega(n)
  - 2n is Omega(n)
  - 15n+7 is Omega(lg n)
  - 15n+7 is Omega(n^2)
  - 5n^2+4 is Omega(n)
  - 4n^2+9n+8 is Omega(n^2)
  - 4n^2+9n+8 is Omega(n^3)

We want tight bounds

- $2n^2 + n \lg n + n + 10$
  - is $O(n^2)$, $O(n^3)$, $O(n^4)$, $O(n^{10})$...
- $3n + 5$
  - is $O(n)$, $O(n^{10})$, $O(n^2)$, ...
- Let's say you are 2 minutes away from the top and you don't know that. You ask: How much further to the top?
  - Answer 1: at most 3 hours (True, but not that helpful)
  - Answer 2: just a few minutes.
- When finding an upper bound, the goal is to find the best (smallest) one possible.

- $2n^2 + n \lg n + n + 10$
  - is Omega(1), Omega(lg n), Omega(n) , Omega(n lg n)
- $3n + 5$
  - is Omega(1), Omega(lg n) , Omega(n)
- You ask at an interview: How much will my salary be?
  - Answer 1: at least 1 dollar a month (True, but not that helpful)
  - Answer 2: at least 5,000 a month (that's better..)
- When finding a lower bound, the goal is to find the best (largest) one possible.

Big-Theta

- Definition:
  - f is Theta(g) if f is O(g) and f is Omega(g)
  - i.e. there are constants c' and c'' such that c'g(n) <= f(n) <= c''g(n)

- When we say f is Theta(g) this means that
  - f and g have the same order of growth (up to constant factors)

- Another way to compare the order of growth of two functions is to compute their limit f/g as n goes to infinity
  - if the limit is a constant c >0 ==> f = Theta(g)

- Examples:
  - 3n + lg n + 10 is Theta(n)
  - $2n^2$ + n lg n + 5 is Theta(n^2)
  - 3lgn +2 is Theta(lg n)
  - 3n+2, 2n+5, 10n, 1000n are Theta(n)

Using Asymptotic Analysis

- Usually we want to find a theta-bound (i.e. the order of growth) for the worst-case running time
- Examples:
  - worst-case binary search is Theta(lg n)
  - worst-case linear search is Theta(n)
  - worst-case find-min in an array is Theta(n)
  - worst-case insertion sort is Theta(n^2)
  - worst-case bubble-sort is Theta(n^2)

- It is correct to say that worst-case binary search is O(lg n), but a Theta-bound is better

Using Asymptotic Analysis

- best-case running time < running time < worst-case running time
    - Running time is Omega(best-case running time)
    - Running time is O(worst-case running time)

- Examples:
    - binary search is Theta(1) in the best case
    - binary search is Theta(lg n) in the worst case
    - therefore binary search is Omega(1) and O(lg n)

    - worst-case binary search is Theta(lg n)
    - binary search is O(lg n)
    - binary search is Theta(lg n) <---------- NO

Using Asymptotic Analysis

- Suppose we have two algorithms for a problem:
  - Algorithm A has a running time of $O(n)$
  - Algorithm B has a running time of $O(n^2)$
- Which one is better?

Using Asymptotic Analysis

- Suppose we have two algorithms for a problem:
  - Algorithm A has a running time of $O(n)$
  - Algorithm B has a running time of $O(n^2)$
- Which is better?
  - We do not know!!! O() just gives us an upper bound.
  - Scenarios:
    - A is linear, B is quadratic (therefore A is faster)
    - Both are linear (therefore they are equivalent)
    - A is linear, B is logarithmic (therefore B is faster)

Asymptotic Analysis

- Suppose we have two algorithms for a problem:
  - Algorithm A has a running time of Theta(n)
  - Algorithm B has a running time of Theta(n^2)

- Which is better?
  - A is smaller (faster)
  - Theta(n) is better than Theta(n^2), etc

- order classes of functions by their oder of growth
- Theta(1) < Theta(lg n) < Theta(n) < Theta(nlgn) < Theta(n^2) < Theta(n^3) < Theta(2^n)

- Cannot distinguish between algorithms in the same class
  - two algorithms that are Theta(n) worst-case are equivalent theoretically
  - optimization of constants can be done at implementation-time

Order of growth matters

- Example:
  - Say $n = 10^9$ (1 billion elements)
  - 10 MHz computer ==> 1 instruction takes 10^{-7} seconds
  - Binary search would take
    - Theta(lg n) = lg $10^9 \times 10^{-7}$ sec = 30×10^{-7} sec = 3 microsec
  - Sequential search would take
    - Theta(n)= $10^9 \times 10^{-7}$ sec = 100 seconds
  - Finding all pairs of elements would take
    - Theta(n^2) = $(10^9)^2 \times 10^{-7}$ sec = 10^{11} seconds = 3170 years
  - Imagine Theta(n^3)
  - Imagine Theta(2^n)

Order of growth matters

n	lg n	n	n lg n	n^2	n^3	2^n
8	3	8	24	64	512	256
16	4	16	64	256	4,096	65,536
32	5	32	160	1,024	32,768	4,294,967,296
64	6	64	384	4,096	262,144	1.8 x 10^19
128	7	128	896	16,384	2,097,152	3.40 x 10^38
256	8	256	2.048	65,536	16,777,216	1.15 x 10^77
512	9	512	4,608	262,144	134,217,728	1.34 x 10^154
1024	10	1024				
1024^2	20	1,048,576				
10^9						

- Assume we have a 1 GHz computer.
- This means an instruction takes 1 nanosecond (10^{-9} seconds).

- We have 3 algorithms:
- A: $400n$
- B $2n^2$
- C: 2^n

- What is the maximum input size that can be solved with each algorithm in:
  - 1 second
  - 1 minute
  - 1 hour

Running time	1 sec	1 min	1 hour
$400n$			
$2n^2$			
2^n			

Exercise

- We have an array X containing a sequence of numbers. We want to compute another array A such that A[i] represents the average X[0] + X[1] + ... X[i]/ (i+1).
  - `A[0] = X[0]`
  - `A[1] = (X[0] + X[1])/ 2`
  - `A[2] = (X[0] + X[1] + X[2]) / 3`
  - ...

- The first i values of X are referred to as the i-prefix of X.

  X[0] + ... X[i] is called prefix-sum, and A[i] prefix average.

- Application: In Economics. Imagine that X[i] represents the return of a mutual fund in year i. A[i] represents the average return over i years.

- Write a function that creates, computes and returns the prefix averages.

  ```
  double[] computePrefixAverage(double[] X)
  ```

- Analyze your algorithm (worst-case running time).

Asymptotic Analysis: Overview

- Running time = number of instructions in the RAM model of computation
- We want the worst-case running time as a function of input size
- Find the order of growth (a Theta-bound) of the worst-case running time
- Common growth rates

  Theta(1) < Theta(lg n) < Theta(n) < Theta(nlgn) < Theta(n2) < Theta(n3) < Theta(2n)
- At the algorithm design level, we want to find the most efficient algorithm in terms of growth rate
- We can optimize constants at the implementation step

Common running times

- O(lg n)
  - binary search
- O(n)
  - linear search
- O(n-lg-n)
  - performing n binary searches in an ordered array
  - sorting
- $O(n^2)$
  - nested loops
  -
```
for (i=0; i<n; i++)
     for (j=0; j<n; j++)
          //do something
```
  - bubble sort, selection sort, insertion sort
- $O(n^3)$
  - nested loops
  - Enumerate all triples of elements
    - e.g. Imagine cities on a map. Are there 3 cities that no two are not joined by a road?
      - Solution: enumerate all subsets of 3 cities. There are n chose 3 different subsets, which is order n^3.