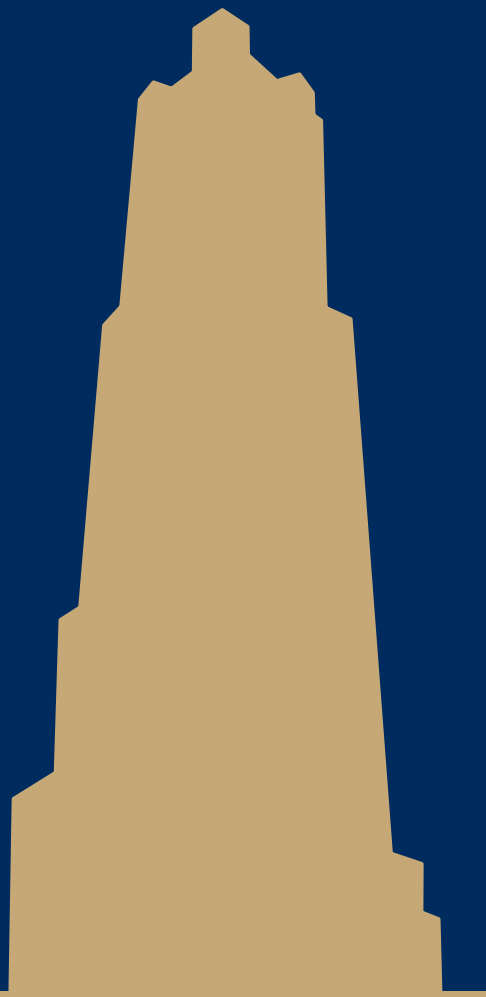
CS/COE 1501

www.cs.pitt.edu/~nlf4/cs1501/

Introduction



Meta-notes

- These notes are intended for use by students in CS1501 at the University of Pittsburgh. They are provided free of charge and may not be sold in any shape or form.
- These notes are NOT a substitute for material covered during course lectures. If you miss a lecture, you should definitely obtain both these notes and notes written by a student who attended the lecture.
- Material from these notes is obtained from various sources, including, but not limited to, the following:
  - Algorithms in C++ by Robert Sedgewick
  - Algorithms, 4th Edition by Robert Sedgewick and Kevin Wayne
  - Introduction to Algorithms, by Cormen, Leiserson and Rivest
  - Various Java and C++ textbooks
  - Various online resources (see notes for specifics)

Instructor Info

- Nicholas Farnan (nlf4@pitt.edu)
  Office: 6313 Sennott Square

NO RECITATIONS THIS WEEK

A note about email

- Prefix all email subjects with [CS1501]
- Address all emails to both the instructor and the TA
- Be sure to mention the section of the class you are in:
  - Day/time
  - Writing/non-writing

Course Info

- Website:
  - www.cs.pitt.edu/~nlf4/cs1501/

- **Review the Course Information and Policies**

- **Assignments will not be accepted after the deadline**
  - No late assignment submissions
  - If you do not submit an assignment by the deadline, you will receive a **0** for that assignment

Up until now, your classes have focused on how you *could* solve a problem. Here, we will start to look at how you *should* solve a problem.

Alright, then, down to business…

First some definitions:

- Offline problem
  - We provide the computer with some input and after some time receive some acceptable output
- Algorithm
  - A step-by-step procedure for solving a problem or accomplishing some end
- Program
  - An algorithm expressed in a language the computer can understand

An algorithm solves a problem if it produces an acceptable output on *every* input

Goals of the course (1)

- To learn to convert non-trivial *algorithms* into *programs*
  - Many seemingly simple algorithms can become much more complicated as they are converted into programs
  - Algorithms can also be very complex to begin with, and their implementation must be considered carefully
  - Various issues will always pop up during *implementation*
    - Such as?...

Example

Input: SPJ query q on relations $R_1, \ldots, R_n$
Output: A query plan for q

```
1:  for i = 1 to n do {
2:      optPlan({R_i}) = accessPlans(R_i)
3:      prunePlans(optPlan({R_i}))
4:  }
5:  for i = 2 to n do {
6:      for all S ⊂ {R_1, ..., R_n} such that |S| = i do {
7:          optPlan(S) = ∅
8:          for all O ⊂ S do {
9:              optPlan(S) = optPlan(S) ∪ joinPlans(optPlan(O), optPlan(S \ O))
10:             prunePlans(optPlan(S))
11:         }
12:     }
13: }
14: finalizePlans(optPlan({R_1, ..., R_n}))
15: prunePlans(optPlan({R_1, ..., R_n}))
16: return optPlan({R_1, ..., R_n})
```

Fig. 1. "Classic" dynamic programming algorithm.

- Pseudocode for dynamic programming algorithm for relational query optimization
- The optimizer portion of the PostgreSQL codebase is over 28,000 lines of code (i.e., not counting blank/comment lines)

Goals of the course (2)

- To see and understand differences in algorithms and how they affect the run-times of the associated programs
  - Different algorithms can be used to solve the same problem
  - Different solutions can be compared using many metrics
    - Run-time is a big one
      - Better run-times can make an algorithm more desirable
      - Better run-times can sometimes make a problem solution feasible where it was not feasible before
    - There are other metrics, though...

How to determine an algorithm's performance

- Implement it and measure performance
  - Any problems with this approach?

- Algorithm Analysis
  - Determine *resource usage* as a function of *input size*
  - Measure *asymptotic* performance
    - Performance as input size increases to infinity

Let's consider ThreeSum example from text

- Problem:
  - Given a set of arbitrary integers (could be negative), find out how many distinct triples sum to exactly zero
- Simple solution: triple for loops!

```
public static int count(int[] a) {
    int n = a.length;
    int cnt = 0;
    for (int i = 0; i < n; i++) {
        for (int j = i+1; j < n; j++) {
            for (int k = j+1; k < n; k++) {
                if (a[i] + a[j] + a[k] == 0) {
                    cnt++;
                }
            }
        }
    }
    return cnt;
}
```

Definition of Big O?

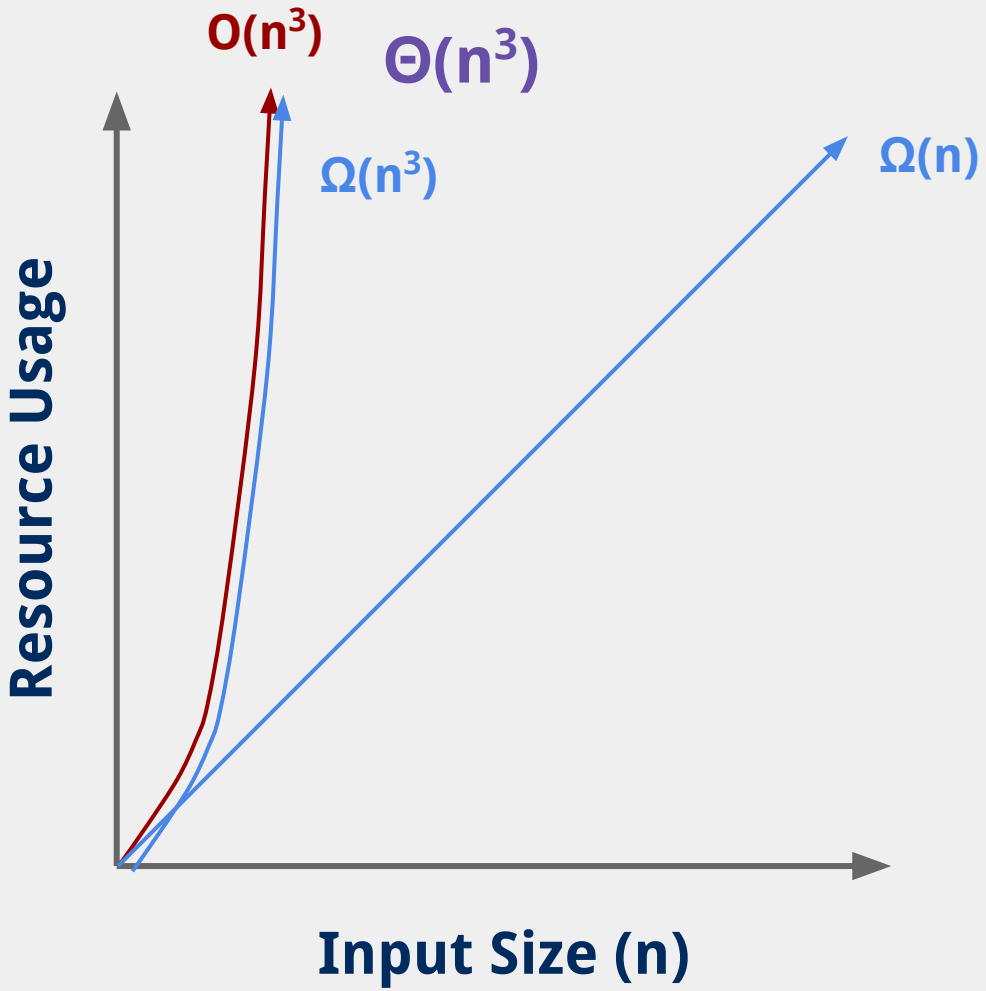
- Big O
  - Upper bound on asymptotic performance
    - As we go to infinity, function representing resource consumption will not exceed specified function
      - E.g., Saying runtime is $O(n^3)$ means that as input size (n) approaches infinity, actual runtime will not exceed n^3

Wait…

- Assuming that definition…
  - Is ThreeSum $O(n^4)$?
  - What about $O(n^5)$?
  - What about $O(3^n)$??
- If all of these are true, why was $O(n^3)$ what we jumped to to start?

Big O isn't the whole story

- Big Omega
  - Lower bound on asymptotic performance
- Theta
  - Upper and Lower bound on asymptotic performance
  - Exact bound


O(n3)
Θ(n3)
Ω(n3)
Ω(n)
Resource Usage
Input Size (n)

Formal definitions

- f(x) is O(g(x)) if constants c and x_0 exist such that:
  - $|f(x)| <= c * |g(x)| \ \forall x > x_0$
- f(x) is Ω(g(x)) if constants c and x_0 exist such that:
  - $|f(x)| >= c * |g(x)| \ \forall x > x_0$
- if f(x) is O(g(x)) *and* Ω(g(x)), then f(x) is Θ(g(x))
  - c_1, c_2, and x_0 exist such that:
    - $c_1 * |g(x)| <= |f(x)| <= c_2 * |g(x)| \ \forall x > x_0$
- May also see $f(x) \in O(g(x))$ or $f(x) = O(g(x))$ used to mean that f(x) is O(g(x))
  - Same for Ω and Θ

Mathematically modelling runtime

- Runtime primarily determined by two factors:
  - Cost of executing each statement
    - Determined by machine used, environment running on the machine
  - Frequency of execution of each statement
    - Determined by program and input

Let's consider ThreeSum example from text

```
public static int count(int[] a) {
    int n = a.length;
    int cnt = 0;
    for (int i = 0; i < n; i++) {
        for (int j = i+1; j < n; j++) {
            for (int k = j+1; k < n; k++) {
                if (a[i] + a[j] + a[k] == 0) {
                    cnt++;
                }
            }
        }
    }
    return cnt;
}
```

Tilde approximations and Order of Growth

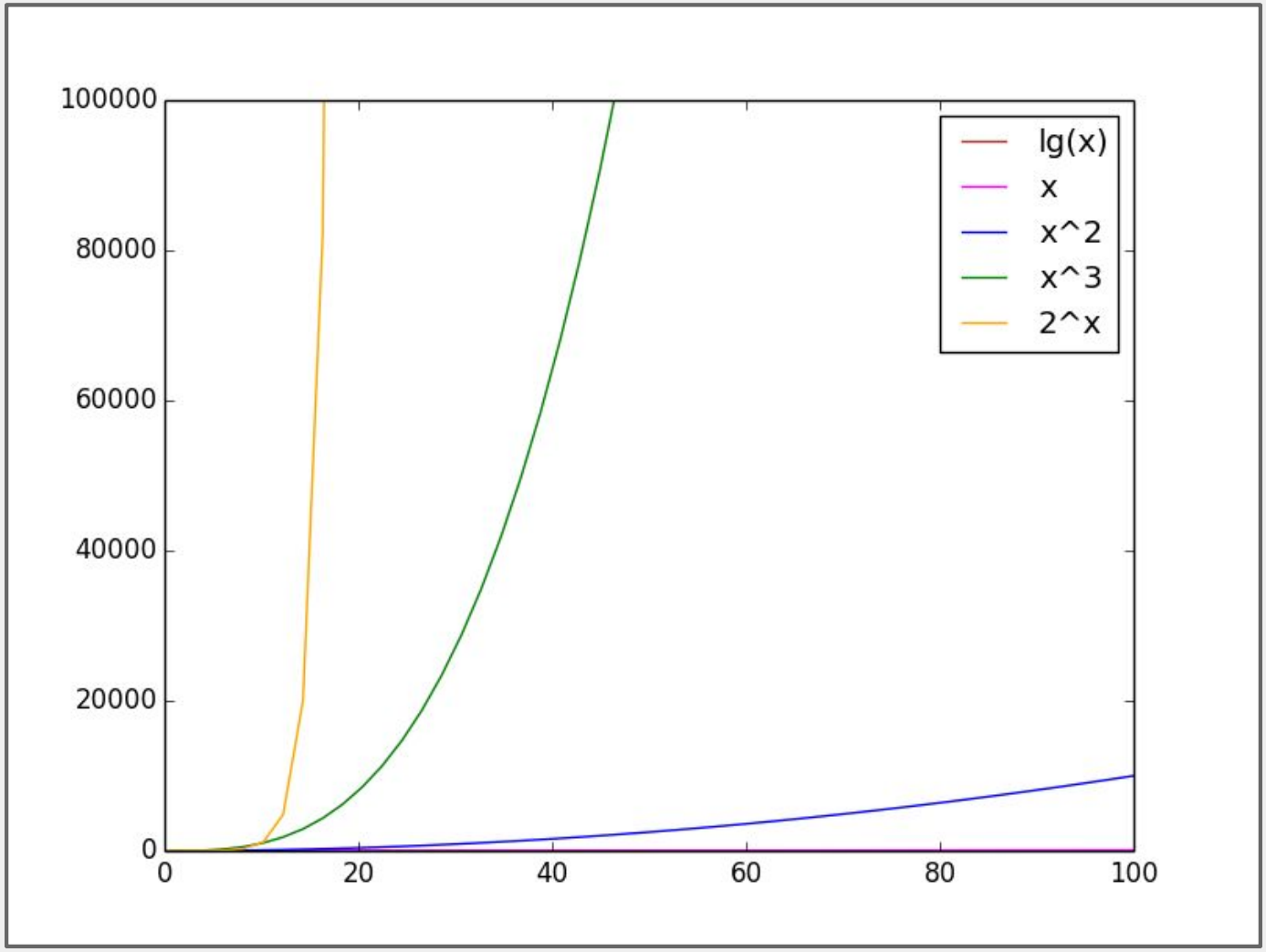
- ThreeSum order of growth:
  - Upper bound: $O(n^3)$
  - Lower bound: $\Omega(n^3)$
  - And hence: $\Theta(n^3)$

- Tilde approximations?
  - Introduced in section 1.4 of the text
  - In this case: $\sim n^3/6$

Common orders of growth

- Constant - 1
- Logarithmic - log n
- Linear - n
- Linearithmic - n log n
- Quadratic - n^2
- Cubic - n^3
- Exponential - 2^n
- Factorial - n!

Graphical orders of growth



How can we ignore lower order terms and multiplicative constants???

- Remember, this is asymptotic analysis

f(n) n =	10	100	1,000	10,000
$n^3/6 - n^2/2 + n/3$	120	161,700	166,167,000	166,616,670,000
$n^3/6$	167	166,667	16,6666,667	166,666,666,667
n^3	1,000	1,000,000	1,000,000,000	1,000,000,000,000

Quick algorithm analysis

- Ignore multiplicative constants and lower terms
- Use standard measures for comparison

Easy to get Theta for ThreeSum

- Why do we need to bother with Big O and Big Omega?

Further thoughts on ThreeSum

- Is there a better way to solve the problem?
- What if we sorted the array first?
  - Pick two numbers, then binary search for the third one that will make a sum of zero
    - a[i] = 10, a[j] = -7, binary search for -3
    - Still have two for loops, but we replace the third with a binary search
      - Runtime now?
    - What if the input data isn't sorted?
- See ThreeSumFast.java

Brief sorting review

- Given a list of *n* items, place the items in a given order
  - Ascending or descending
    - Numerical
    - Alphabetical
    - etc.

Prerequisites

```
boolean less(Comparable v, Comparable w) {
   return (v.compareTo(w) < 0);
}

void exch(Object[] a, int i, int j) {
   Object swap = a[i];
   a[i] = a[j];
   a[j] = swap;
}
```

Bubble sort

- Simply go through the array comparing pairs of items, swap them if they are out of order
  - Repeat until you make it through the array with 0 swaps

```
void bubbleSort(Comparable[] a) {
    boolean swapped;
    do {
        swapped = false;
        for(int j = 1; j < a.length; j++) {
            if (less(a[j], a[j-1]))
                { exch(a, j-1, j); swapped = true; }
        }
    } while(swapped);
}
```

Bubble sort example

SWAPPED!

“Improved” bubble sort

```
void bubbleSort(Comparable[] a) {
   boolean swapped;
   int to_sort = a.length;
   do {
      swapped = false;
      for(int j = 1; j < to_sort; j++) {
         if (less(a[j], a[j-1]))
            { exch(a, j-1, j); swapped = true; }
      }
      to_sort--;
   } while(swapped);
}
```

How bad is it?

- Runtime:
  - $O(n^2)$

"[A]lthough the techniques used in the calculations [to analyze the bubble sort] are instructive, the results are disappointing since they tell us that the bubble sort isn't really very good at all."

Donald Knuth
The Art of Computer Programming

Bubble Sort

What is the most efficient way to sort a million 32-bit integers?

I think the bubble sort would be the wrong way to go.

How can we sort without comparison?

- Consider the following approach:
  - Look at the least-significant digit
  - Group numbers with the same digit
    - Maintain relative order
  - Place groups back in array together
    - I.e., all the 0's, all the 1's, all the 2's, etc.
  - Repeat for increasingly significant digits

Radix sort analysis

- Runtime?
  - n * (length of items in collection)
    - We'll say nk
      - How can we compare this to the n log n runtime that is optimal for comparison-based sorts?
        - Also, why is it called "Radix sort"?

- In-place?
- Stable?

Further thoughts on Eric Schmidt's question...

- 1,000,000 32-bit integers don't take up a whole lot of space
  - 4 MB
- What if we needed to sort 1TB of numbers?
  - Won't all fit in memory…
  - We had been assuming we were performing *internal* sorts
    - Everything in memory
  - We now need to consider *external* sorting
    - Where we need to write to disk

Hybrid merge sort

- Read in amount of data that will fit in memory
- Sort it in place
  - I.e., via quick sort
- Write sorted chunk of data to disk
- Repeat until all data is stored in sorted chunks
- Merge chunks together

External sort considerations

- Should we merge all chunks together at once?
  - Means fewer disk read/writes
    - Each merge pass reads/writes every value
  - But also more disk seeks
- Can we do parallel reads/writes to multiple disks?
- Can we use multiple CPUs/cores to speed up processing

Large scale sorts

- What about when you have 1PB of data?
- In 2008, Google sorted 10 trillion 100 byte records on 4000 computers in 6 hours 2 minutes
- 48,000 hard drives were involved
    - At least 1 disk failed during each run of the sort

Topics for the term

- Searching
- Hashing
- Compression
- Heaps and Priority Queues
- Graph Algorithms
- Large Integer Math
- Cryptography
- P vs NP
- Heuristic Approximation
- Dynamic Programming