CS/COE 1501
Algorithm Implementation

Course Introduction
Algorithm Analysis
Exhaustive Search

Fall 2018

Sherif Khattab

ksm73@pitt.edu

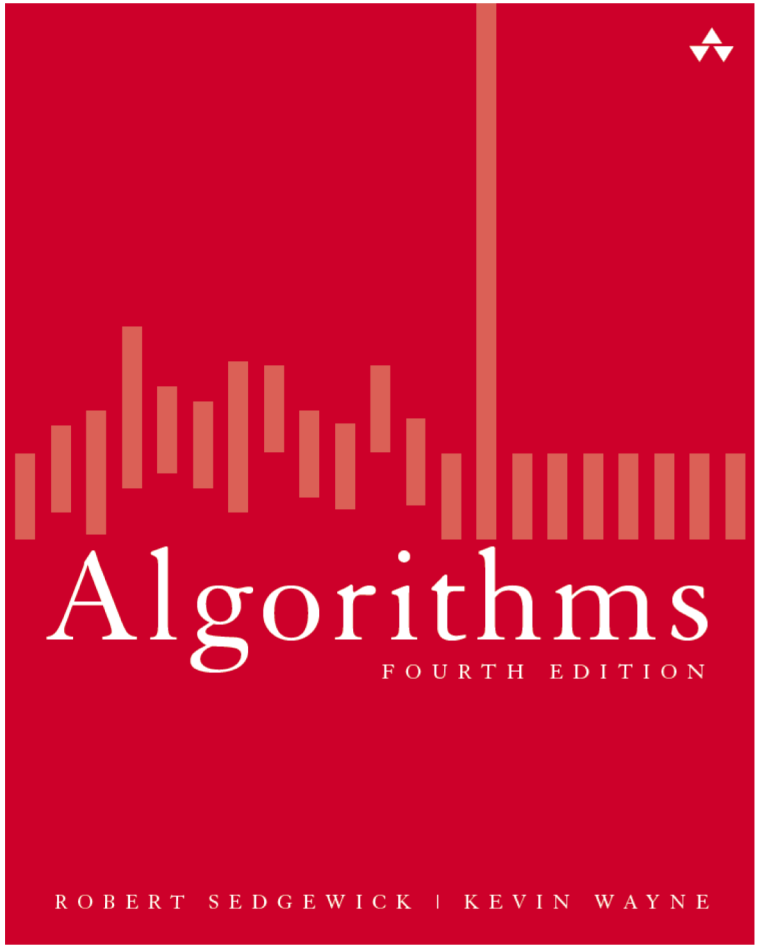
6307 Sennott Square

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Contact Info

- **Course website:** http://www.cs.pitt.edu/~skhattab/cs1501/
- **Instructor:** Sherif Khattab ksm73@pitt.edu
  - OH: MW 10-4pm (other times available by appointment)
    - Please book at http://khattab.youcanbook.me
  - Office: 6307 Sennott Square
- **Teaching Assistants**
  - Ziyu Zhang for Section 1270 and 1280
  - Hanzhong Zhengand Jie Liu (Grader) for Section 1400 and 1410
  - No recitations this week

Textbook



- **Algorithms (4th Edition)**

•Robert Sedgewick and Kevin Wayne

Grades

- 40% exams: 26% on higher grade and 14% on lower
- 40% on best four out of five assignments worth 10% each
  - posted and submitted on **Courseweb**
- 10% recitation participation
- 10% mini-quizzes: on Top Hat during each lecture. The join code for Top Hat is 064642.

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

- 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)

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.

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?

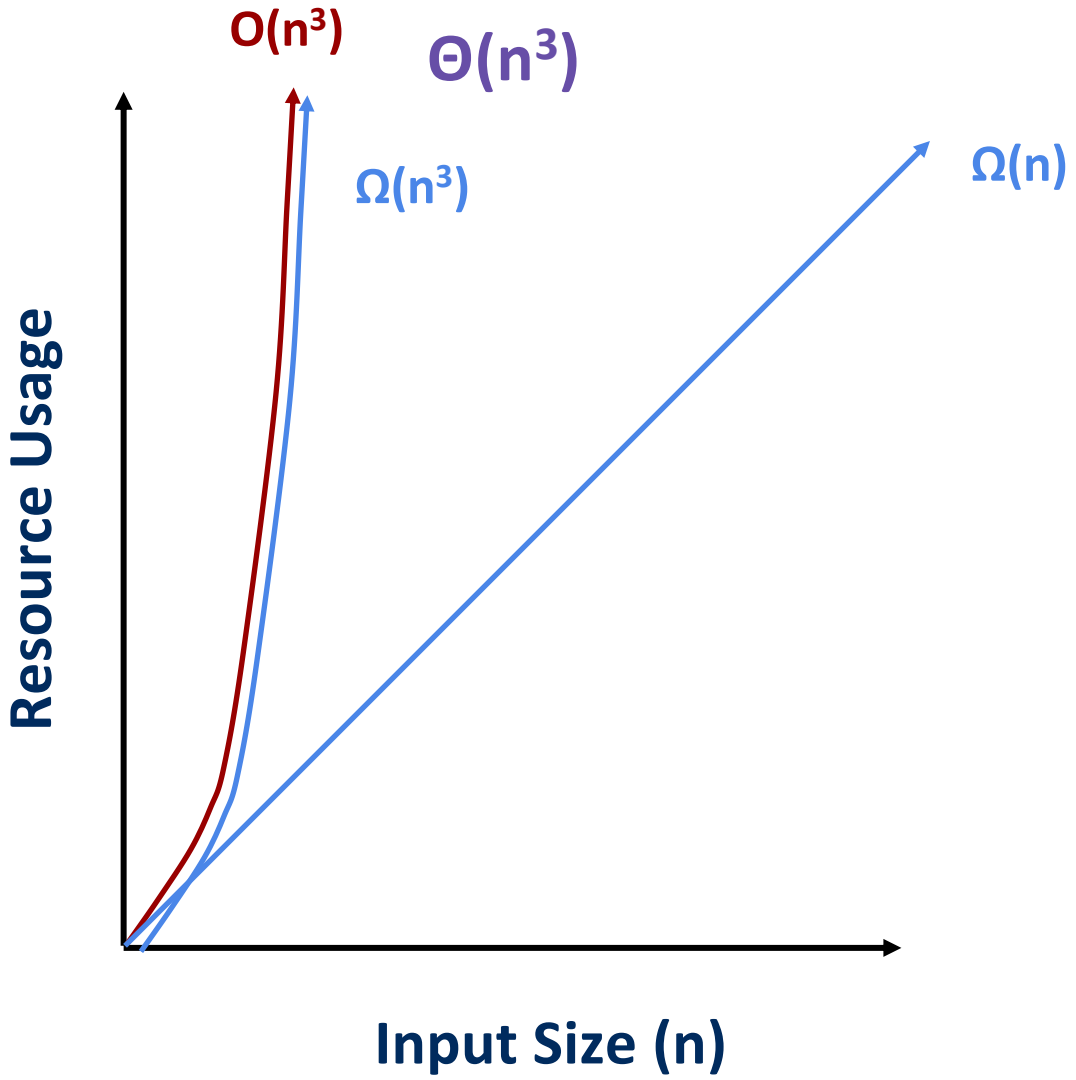
- Big O
  - Upper bound on asymptotic performance
    - As we go to infinity, function representing resource consumption will not exceed specified function
      - E.g., Saying runtime is $O(n^3)$ means that as input size (n) approaches infinity, actual runtime will not exceed n^3

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) ∈ 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;
}
```

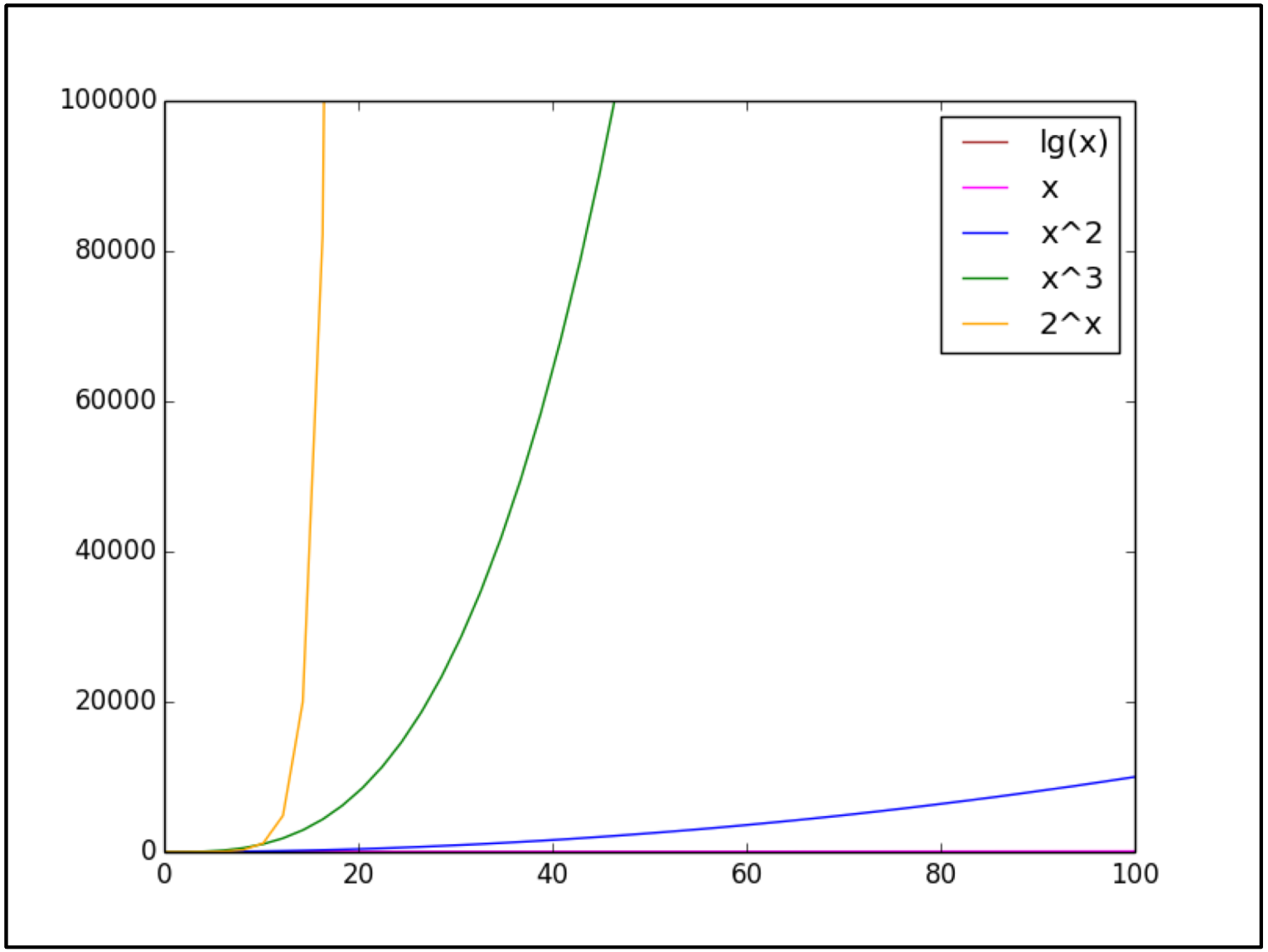
Tilde approximations and Order of Growth

- 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	166,666,667	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

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

Topics for the term

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

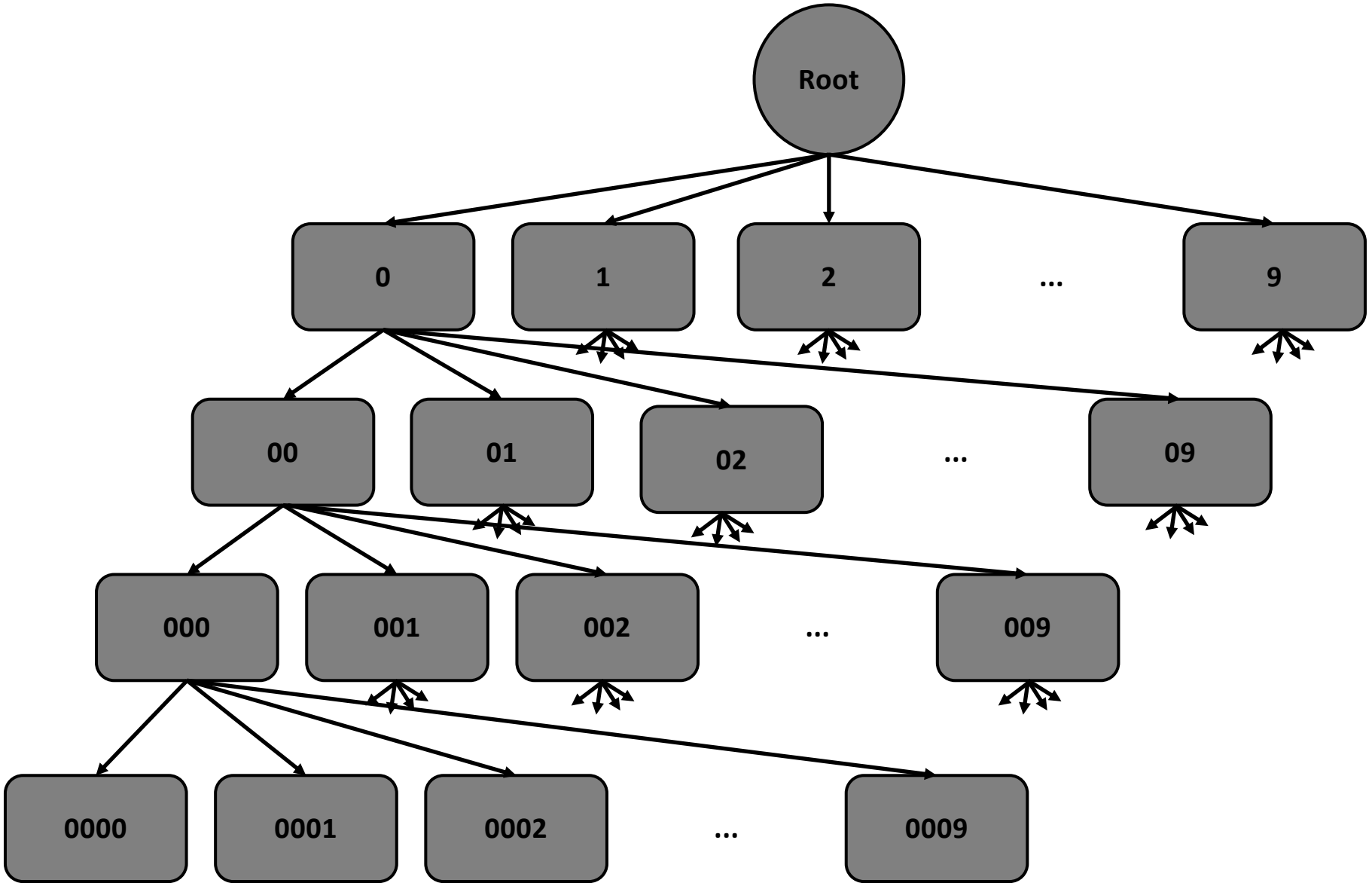
Brute-force (or exhaustive) search

- Find the solution to a problem by considering all potential solutions and selecting the correct one
- Run-time is bounded by the number of potential solutions
  - n^3 potential solutions means cubic run-time
  - 2^n potential solutions means exponential run-time

Password cracking

- Brute force password attacks depend on the length of the password, hence the insecurity of short passwords
- We can view the series of guesses we make as a tree
  - Each path from root to leaf is an attempted solution

PIN cracking example

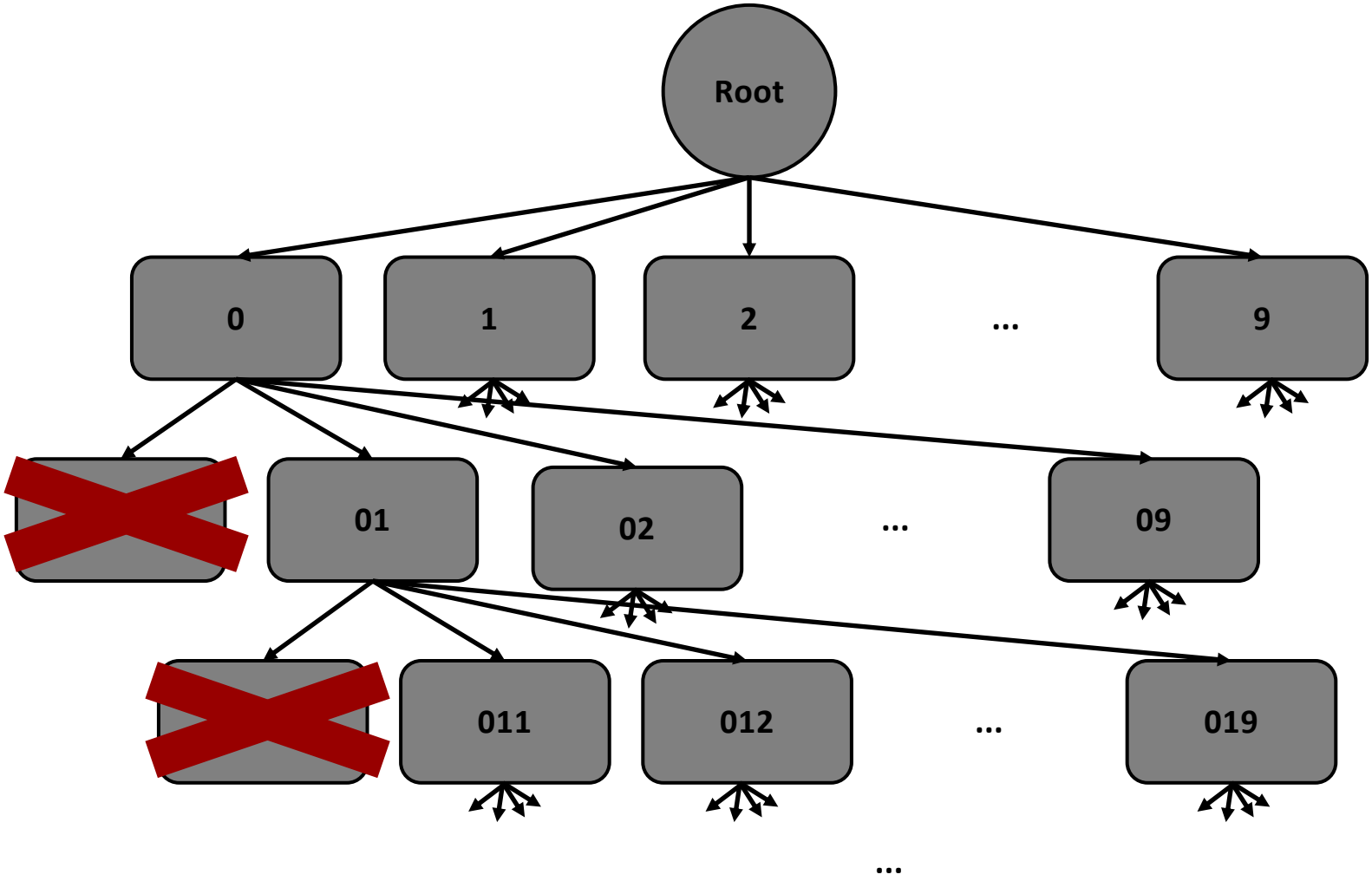


Search space size

- This tree will enumerate 10^n different PINs
  - n is the length of the PIN
  - So for our case 10^4 = 10,000 different PINs
- Note that this is (for a computer) tiny
  - What would be a long password for a computer?
  - Say 128 bits long
    - 2^n different passwords
    - 2^{128} = 340282366920938463463374607431768211456
      - Assuming a supercomputer can check 3386000000000 passwords per second...
      - And we'll on average find the correct password after guessing half the possibilities…
      - We should be able to crack a 128 bit password on our supercomputer in 1.59×10^{17} years using brute force

Back to our PIN cracking example

- What if we have background knowledge that the PIN we're trying to crack doesn't have more than one 0?



Pruning!

- Removes entire subtrees of our search space exploration
- When we can use it, it makes our algorithm practical for much larger values of n
- Does not, however, affect the asymptotic performance of an algorithm
  - Still exponential time requirement for our PIN example

How to enumerate all these possibilities?

- For the PIN example a whole bunch of for loops would do
- In general, exhaustive search trees can be easily traversed via recursion and *backtracking*
  - Recurse until its apparent no solution can be achieved along the current path
  - Undo the path to the point that you can start to move forward again

8 queens problem

- Place 8 queens on a chessboard such that no queen can take another
  - Queens can move horizontally, vertically, and diagonally
- How many ways can you places 8 pieces on a chess board?
  - 64 C 8
  - = 64! / (8! * (64-8)!)
  - = 4,426,165,368
  - Meaning 35,409,322,944 total queen placements
- Do we really need to look through all of these options?

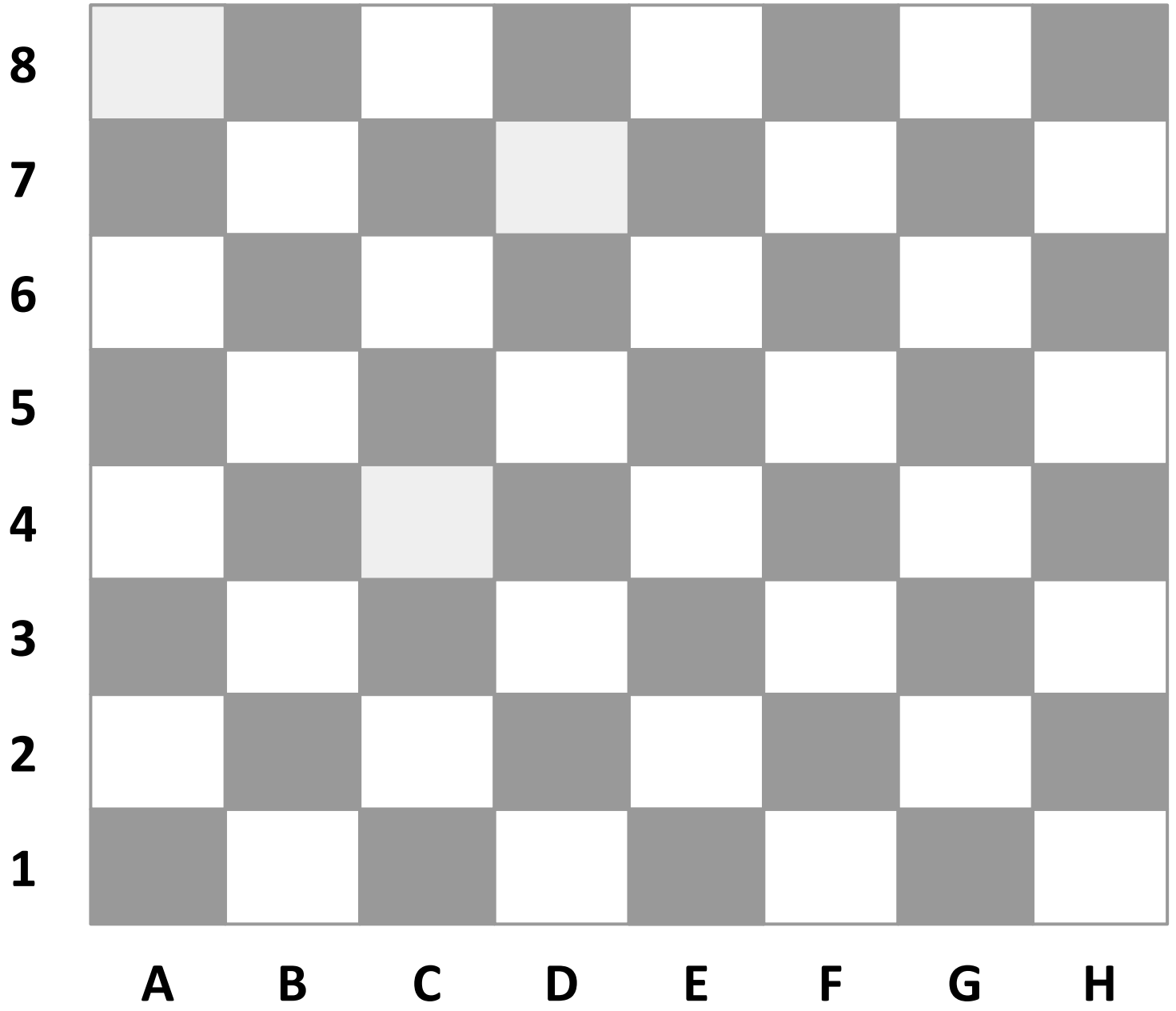
8 queens problem

- Solutions only have one queen per column
  - Still 8^8 = 16,777,216 possible combinations
- Solutions only have one queen per row
  - Combining these two observations, only 8! = 40,320
  - Looking quite feasible!
- Finally, prune subtrees with queens on the same diagonal

Queens.java

- Basic idea:
  - Recurse over columns of the board
  - Each recursive call iterates through the rows of the board
  - Check rows/diagonals
    - Are they currently safe?
      - Place a queen in the current row/col
      - If you are at the end of the board, you've found a solution!
      - Otherwise, try recursive call for the next column
    - If they are not currently safe
      - Continue to the next row in the current recursive call

8 Queens



Another problem: Boggle

- Words at least 3 adjacent letters long must be assembled from a 4x4 grid
- Adjacent letters are horizontally, vertically, or diagonally neighboring
- Any cube in the grid can only be used once per word

Recursing through Boggle letters

- Have 8 different options from each cube
  - From B[i][j]:
    - B[i-1][j-1]
    - B[i-1][j]
    - B[i-1][j+1]
    - B[i][j-1]
    - B[i][j+1]
    - B[i+1][j-1]
    - B[i+1][j]
    - B[i+1][j+1]
- Naively, the runtime here would be 16!
  - = 20,922,789,888,000

Where do we prune?

Implementation concerns with Boggle

- Constructing the words over the course of recursion will mean building up and tearing down strings
  - Moving forward adds a new character to the current word string
  - Backtracking removes the most recent character
  - Basically pushing/popping to/from a string stack
- Push/Pop stack operations are generally $\Theta(1)$
  - Unless you need to resize, but that cost can be amortized
- Java Strings, however, are *immutable*
  - s = new String("Here is a basic string");
  - s = s + " this operation allocates and initializes all over again";
  - Becomes essentially a $\Theta(n)$ operation
    - Where n is the length of the string

StringBuilder to the rescue

- append() and deleteCharAt() can be used to push and pop
  - Back to Θ(1)!
    - Still need to account for resizing, though…
- StringBuffer can also be used for this purpose
  - Differences?