Problem-Solving via Search

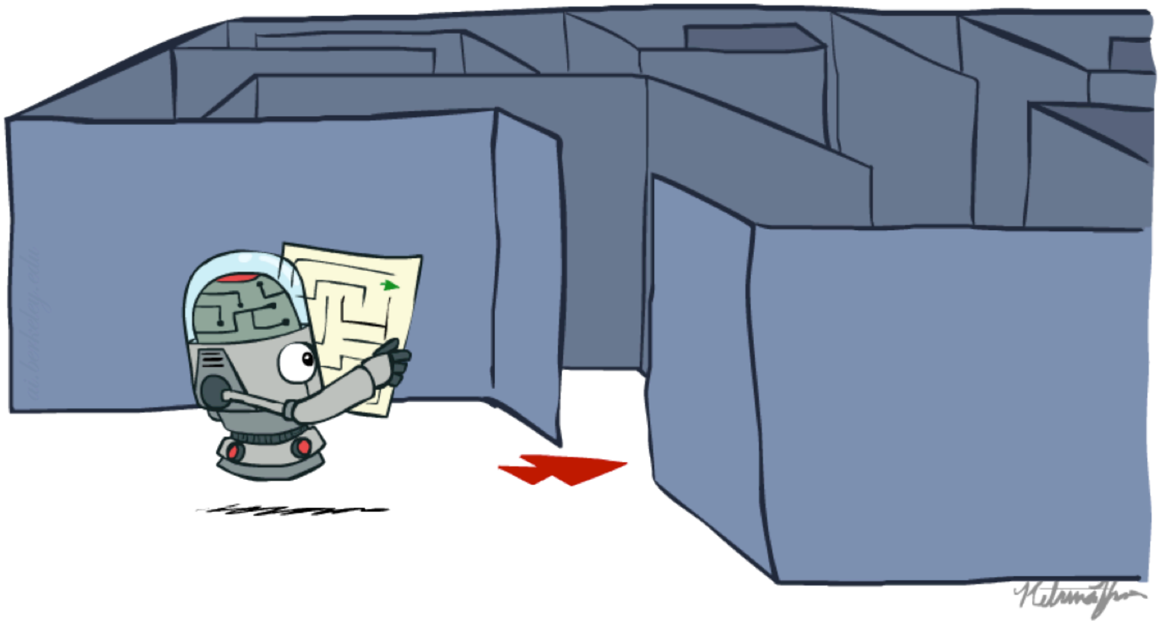
Lecture 3

What is a search problem?

How do search algorithms work and how do we evaluate their performance?

Agenda

- An example problem
- Problem formulation
- Infrastructure for search algorithms
    - Complexity analysis



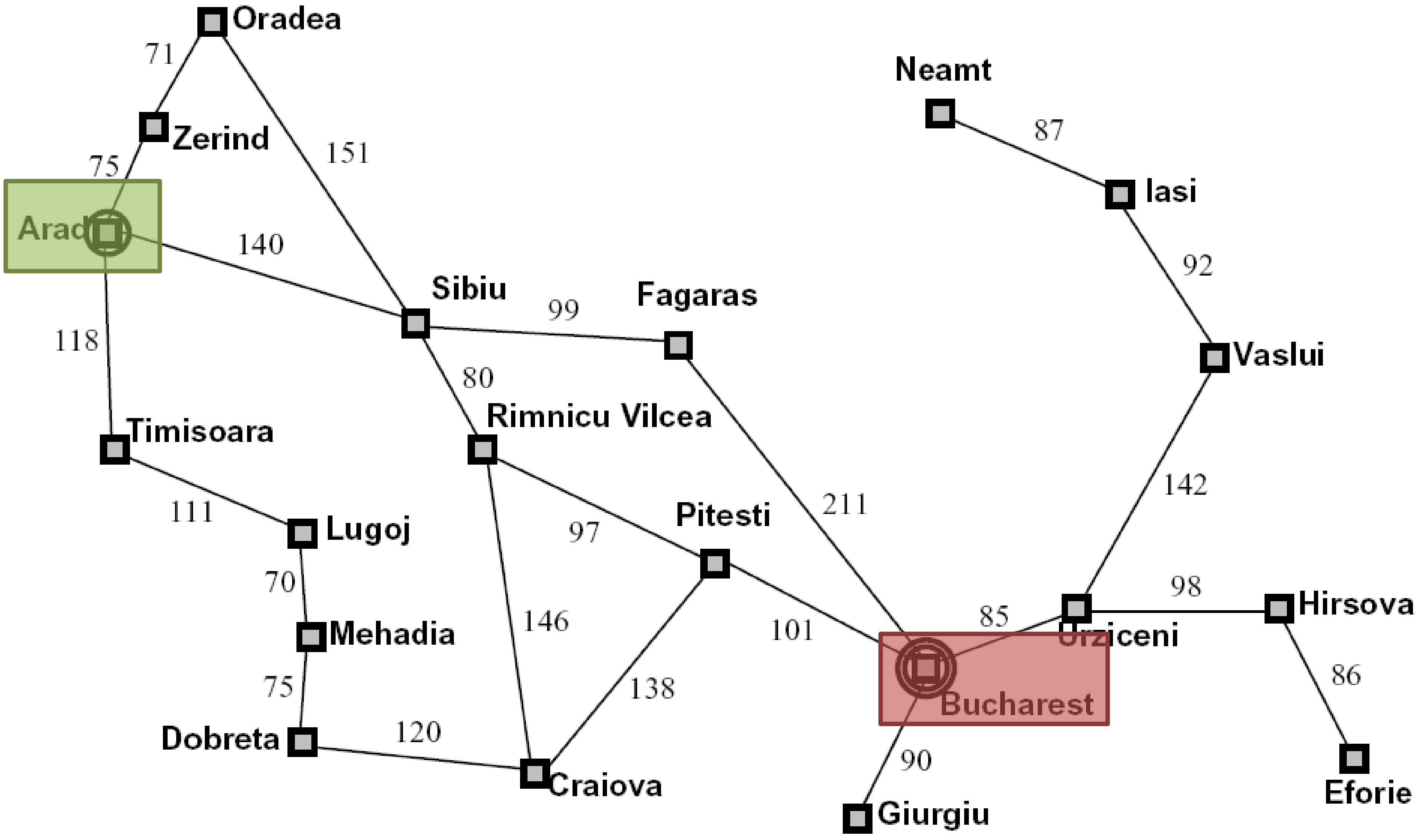
A Motivating Problem

- Start: Arad, Romania
- Goal: Bucharest, Romania
  - Roads leading to Sibiu, Timisoara, Zerind

What is a rational agent to do?

Add Geographical Knowledge

Add Abstraction



Describe the Task

Observability	Full
Certainty	Deterministic
Representation	Discrete
A priori	Known

*Under these conditions we can **search** for a problem **solution**, a fixed sequence of actions*

- *Given a perfect model, can be done **open-loop** (i.e. ignore percepts)*

Search Problem Formalism

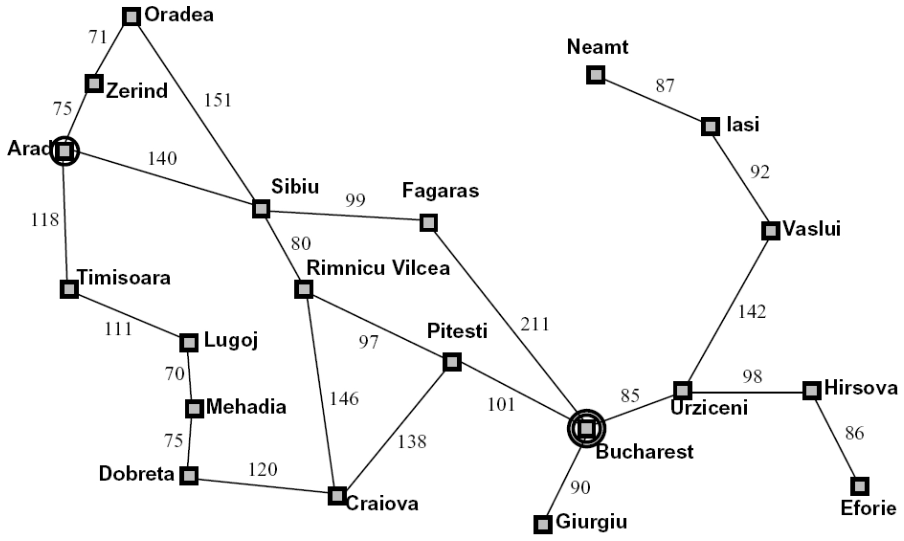
Defined via the following components:

- The **initial state** the agent starts in
- A **successor/transition function**
  S(x) = {action -> state, cost}
- A **goal test**, which returns true if a given state is a goal state
  G(x) = true/false
- A **path cost** that assigns a numeric cost to each path
  - Typically assumed to be sum of action costs

A **solution** is a sequence of actions leading from initial state to a goal state. (**Optimal** = lowest path cost.)

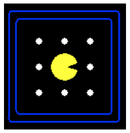
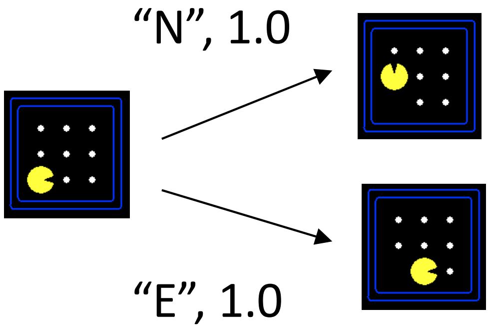
Together the initial state and successor function implicitly define the **state space**, the set of all reachable states

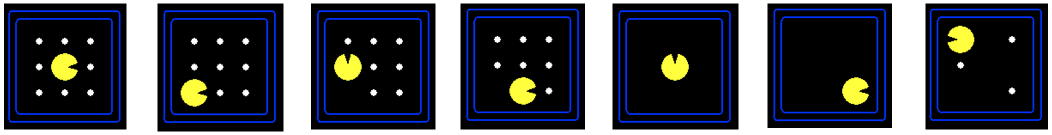
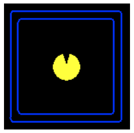
Example: Romanian Travel



- Initial state
  - Arad
- Successor
  - Adjacency, cost=distance
- Goal test
  - City == Bucharest
- State space
  - Cities

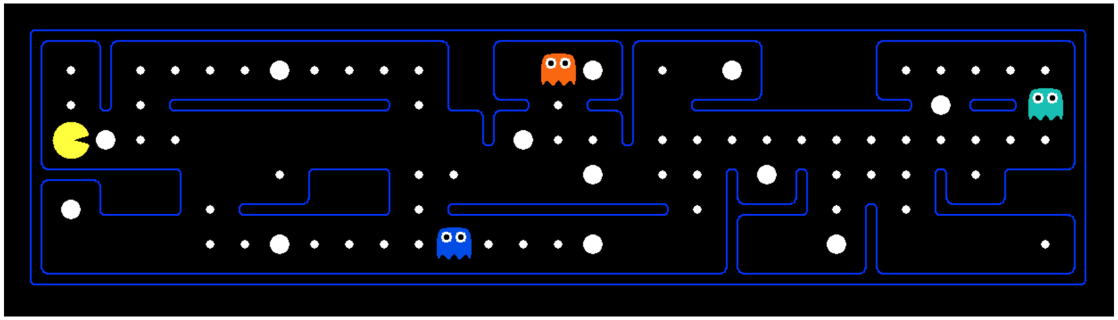
Example: Pacman

- Initial state 
- Successor function 

- State space 
- Goal test: no more food (e.g. )

State Abstraction

- Often world states are absurdly complex



- To solve a particular problem, we abstract the search state to only represent details necessary to solve the problem

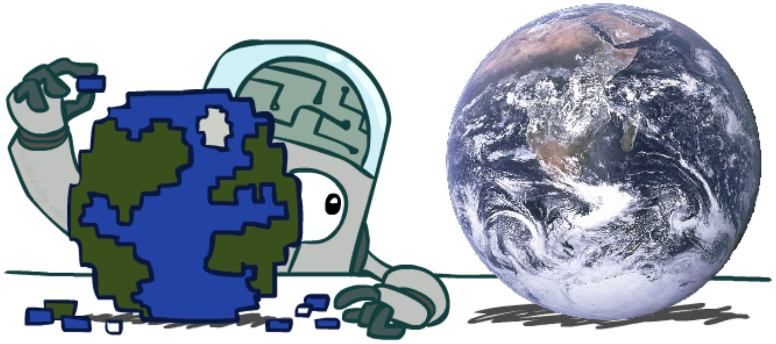
Example Abstractions

Path Planning

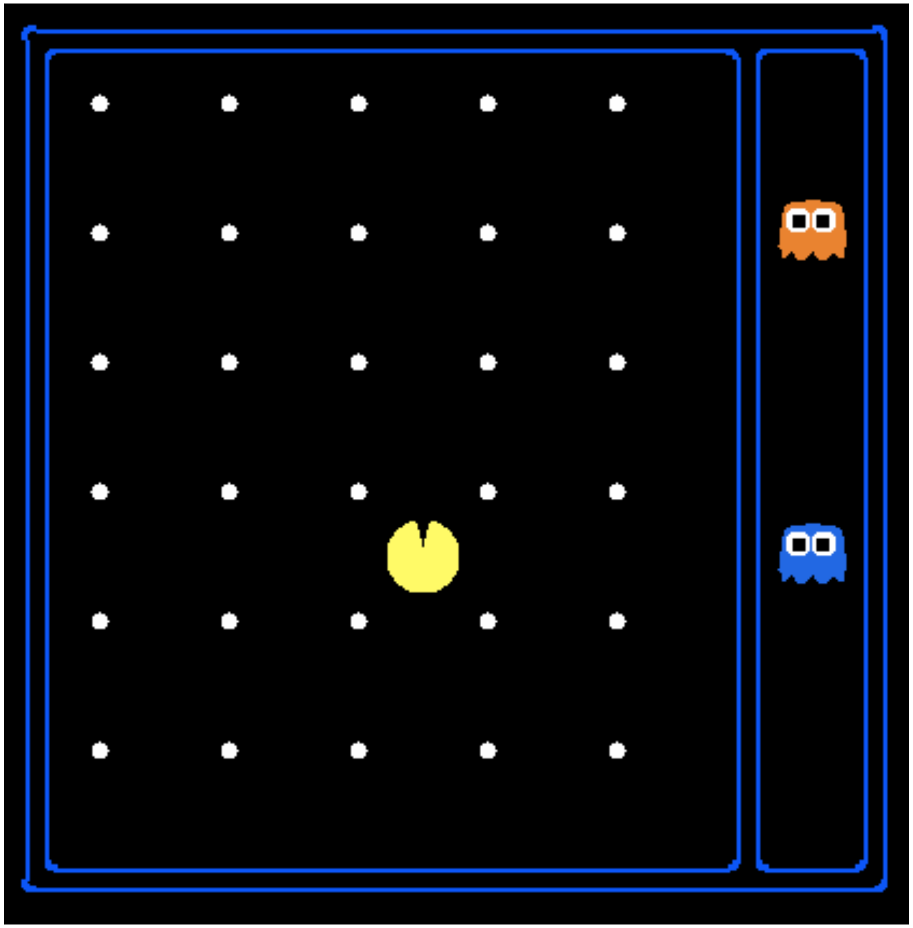
- States: (x,y)
- Actions: NSEW
- Successor: (x’,y’)
- Goal test: (x,y)=END

Eat All the Dots

- States: {(x,y), T/F grid}
- Actions: NSEW
- Successor: (x’,y’), possibly T/F change
- Goal test: grid = all F’s



Abstraction is Necessary



World state

- Agent positions: 120
- Food count: 30
- Ghost positions: 12
- Agent facing: NSEW

How many…

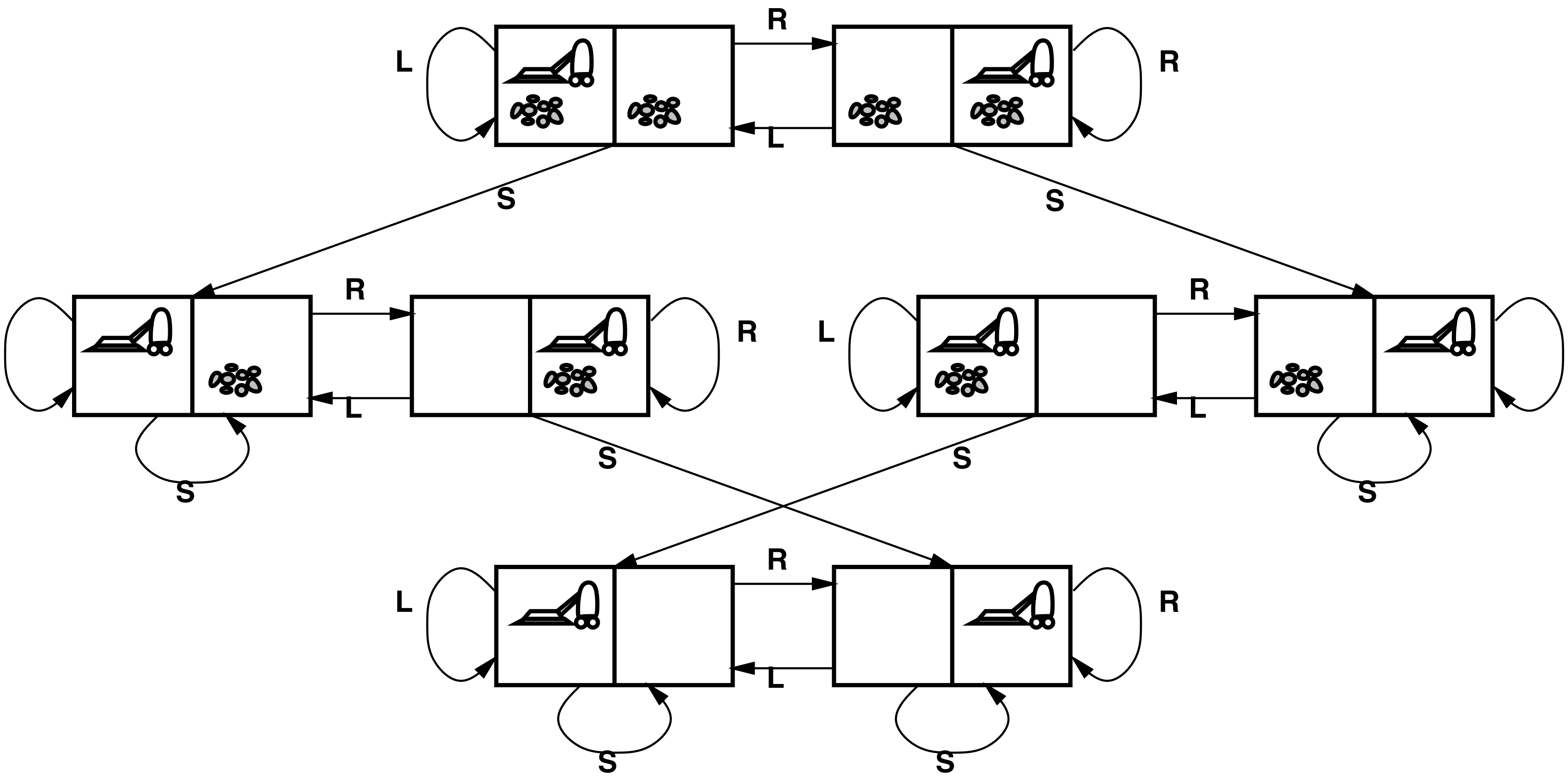
- World states?
  - $120 \times (2^{30}) \times (12^{2}) \times 4$
- States for path planning?
  - 120
- States for eat-all-dots?
  - $120 \times (2^{30})$

Exercise

Describe the `vacuum-cleaner` world search problem:

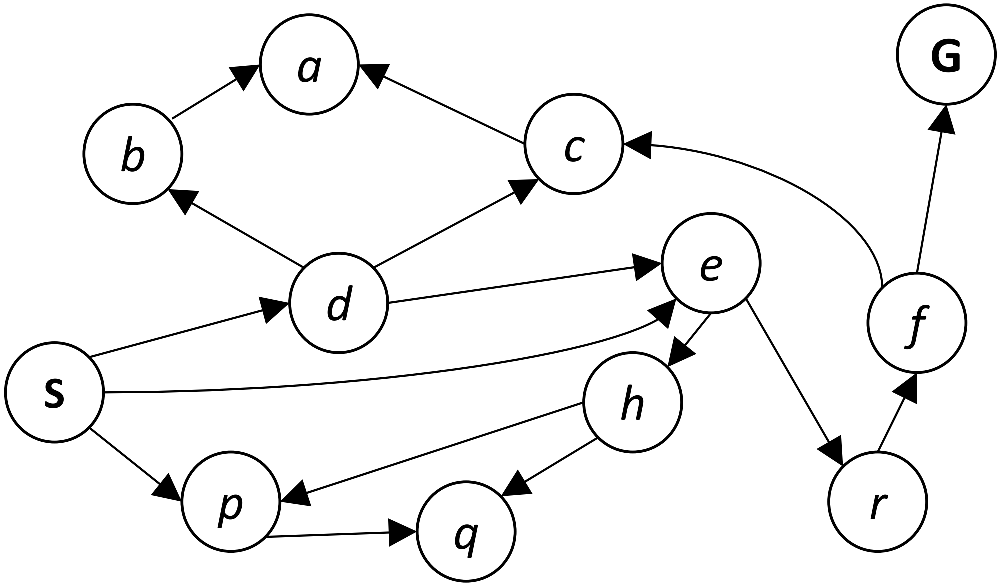
- World state representation
- Search state representation
- Transition model
    - State space
- Goal test

Solution State Space Graph



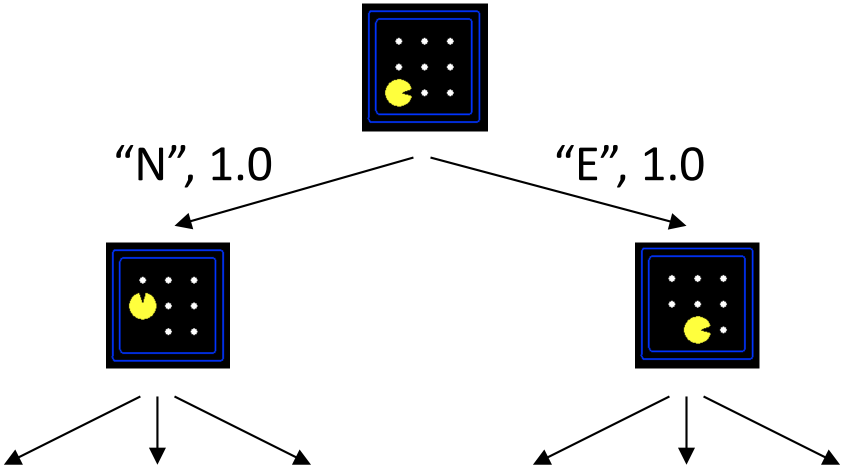
State Space Graph

- State space graph: A mathematical representation of a search problem
  - Nodes are (abstracted) world configurations
  - Arcs represent successors (action results)
  - The goal test is a set of goal node(s)
- In a search graph, each state occurs only once!
- **We can rarely build this full graph in memory (it's too big), but it's a useful idea**

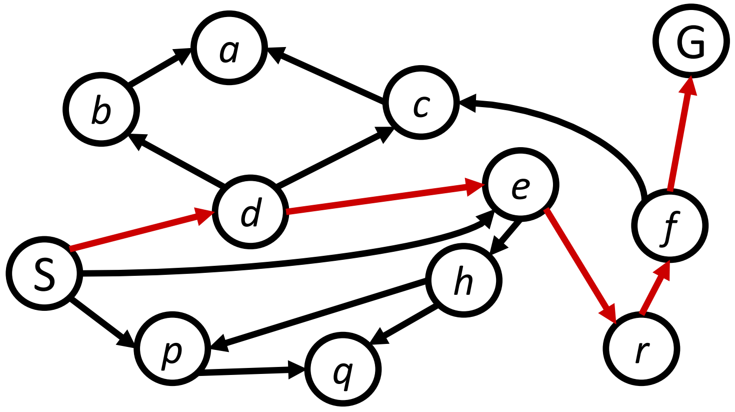


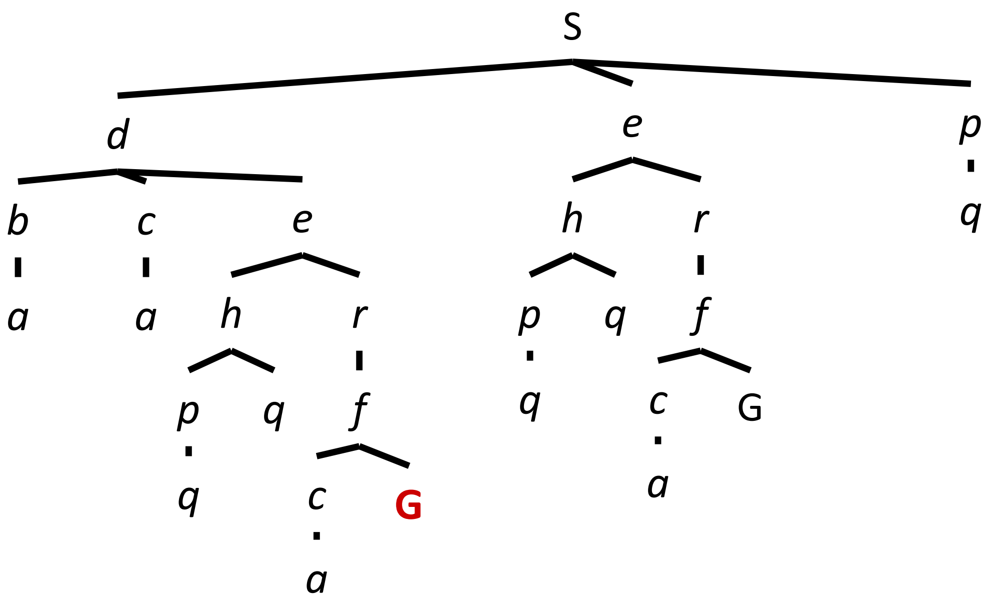
Search Tree

- A "what if" tree of plans and their outcomes
- The start state is the root node
- Children correspond to successors
- Nodes show states, but correspond to PLANS that achieve those states
- **For most problems, we can never actually build the whole tree**



State Space Graph vs. Search Tree



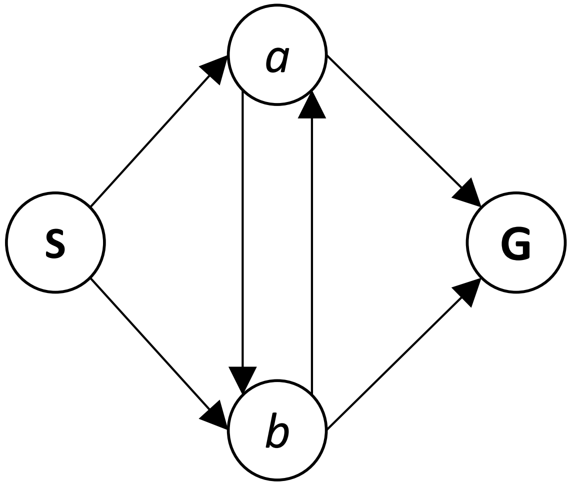


- Each NODE in in the search tree is an entire PATH in the state space graph.

- We construct both on demand – and we construct as little as possible.

Exercise

Consider the following 4-state state space graph…



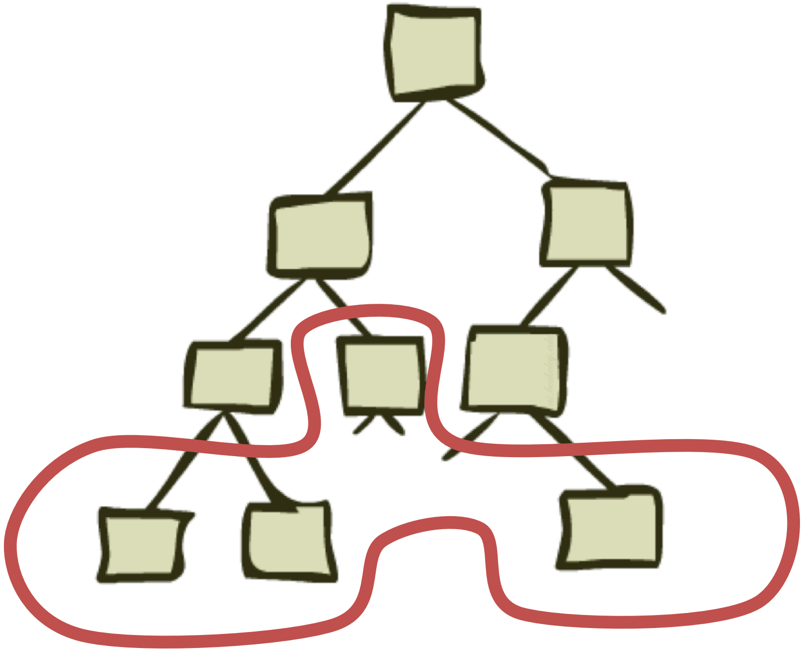
How big is its search tree (from S)?

∞

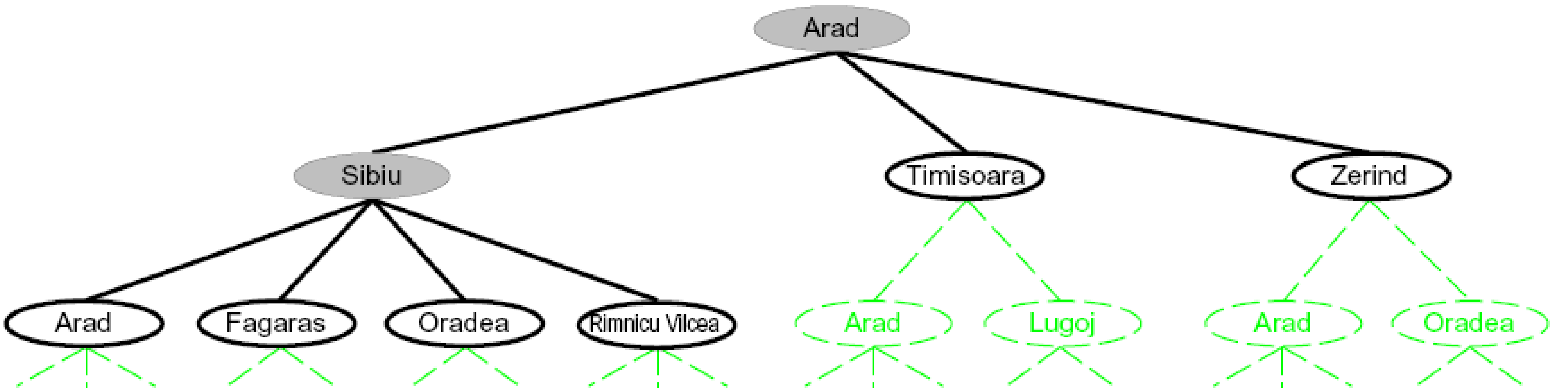
Searching for Solutions

Basic idea: incrementally build a search tree until a goal state is found

- Root = initial state
- Expand via transition function to create new nodes
- Nodes that haven't been expanded are **leaf nodes** and form the **frontier** (**open list**)
- Different **search strategies** (next lecture) choose next node to expand (as few as possible!)
- Use a **closed list** to prevent expanding the same state more than once



General Algorithm



```
function GRAPH-SEARCH(problem, fringe) returns a solution, or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node]) then return node
        if STATE[node] is not in closed then
            add STATE[node] to closed
            fringe ← INSERTALL(EXPAND(node, problem), fringe)
    end
```

Queue (FIFO)
Stack (LIFO)
Priority Queue

Evaluating a Search Strategy

Solution

- **Completeness**: does it always find *a* solution if one exists?
- **Optimality**: does it always find a least-cost solution?

Efficiency

- **Time Complexity**: number of nodes generated/expanded
- **Space Complexity**: maximum number of nodes in memory

Computational Complexity (A.1)

- We are going to be comparing several algorithms
  - How do we tell if one is faster/leaner than another?
- **Benchmarking** involves running the algorithm on a computer and measuring performance (e.g. time in sec, memory in bytes)
  - Unsatisfactory: specific to machine, implementation, compiler, inputs, …
- **Complexity Analysis** is a mathematical approach that abstracts away from these details

Asymptotic Analysis

Basic idea: get a sense of "rate of growth" of an algorithm, which tells us how "bad" it will get as problem size grows

Example

```
def summation(l):
    sum = 0
    for n in l:
        sum += n
    return sum
```

Step 1: Identify Size Parameter

- We need to abstract over the input and just identify what parameter characterizes the size of the input

```
def summation(l):
    sum = 0
    for n in l:
        sum += n
    return sum
```

- For the example what matters is the length of the input list
  - We'll refer to this as *n*

Step 2: Identify Performance Measure

- Again, abstract over the implementation and find a measure that reflects running time (or memory usage), not tied to a particular computer

```
def summation(l):
    sum = 0
    for n in l:
        sum += n
    return sum
```

- In this case it could be lines executed, or operations (additions, assignments) performed
  - Call this f(*n*)

If f(*n*) measures lines executed
f(*n*) = 2*n* + 2

Step 3: Identify Comparison Metric

- It is typically not possible to identify *exactly* the size parameter (i.e. one that perfectly characterizes the performance), and so we settle for a representative metric

- Most common is **worst case**
  - Sometimes best case, average case

Step 4: Approximation

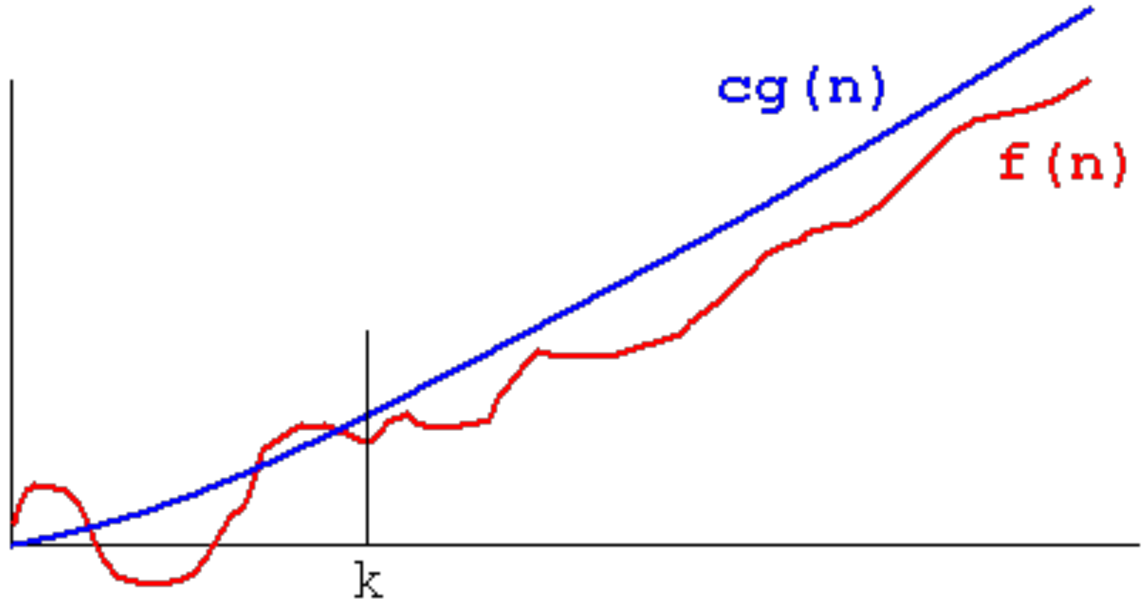
- Typically it is hard to *exactly* compute f(*n*), and so we settle for an approximation
- For worst-case, **Big-O notation**, O(), yields this formal asymptotic analysis…

$$f(n) = \mathcal{O}(g(n)) \text{ as } n \to \infty$$

$$\equiv \exists\, c \in \mathbb{N}, k \in \mathbb{N} \ \text{ s.t.}$$

$$\forall n > k \quad |f(n)| \leq c|g(n)|$$

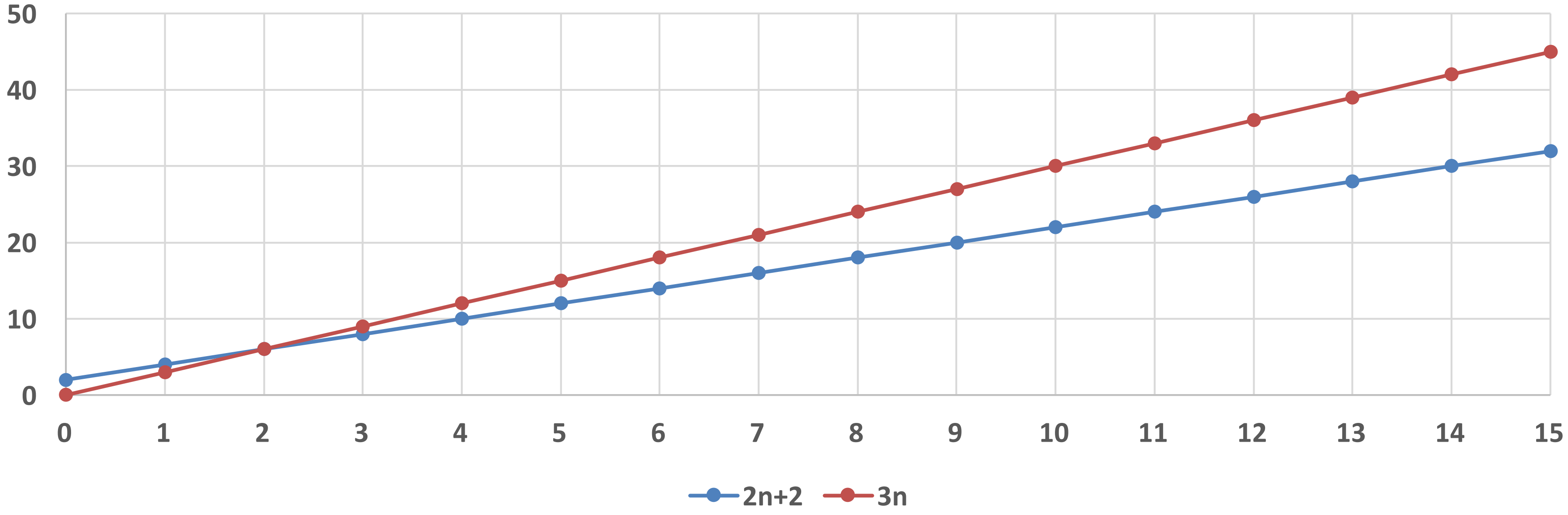
Big-O Definition Visually



Example

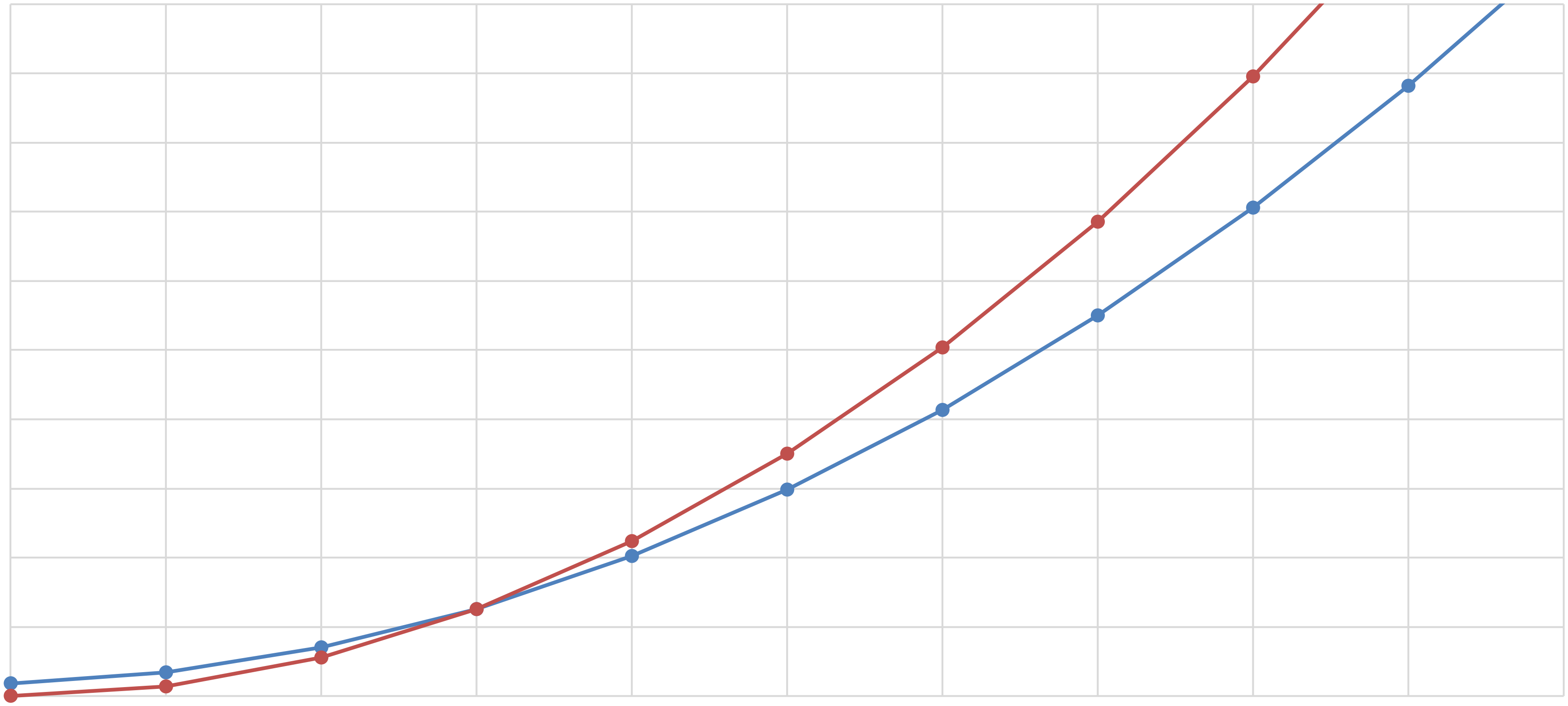
- Since f(n) = 2n + 2, we can show that this function is O(n)
  - c=3, k=2

```
def summation(l):
    sum = 0
    for n in l:
        sum += n
    return sum
```



Exercise

Prove: $5n^2 + 3n + 9 = O(n^2)$



Solution

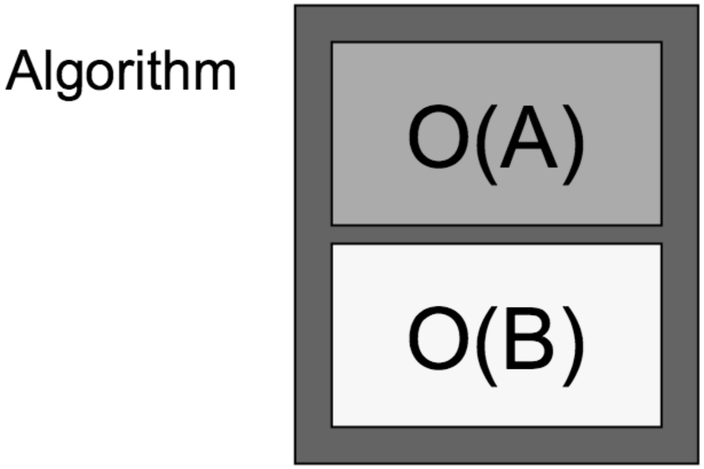
Find c and k such that…

$$\forall n > k \quad cn^2 > 5n^2 + 3n + 9$$

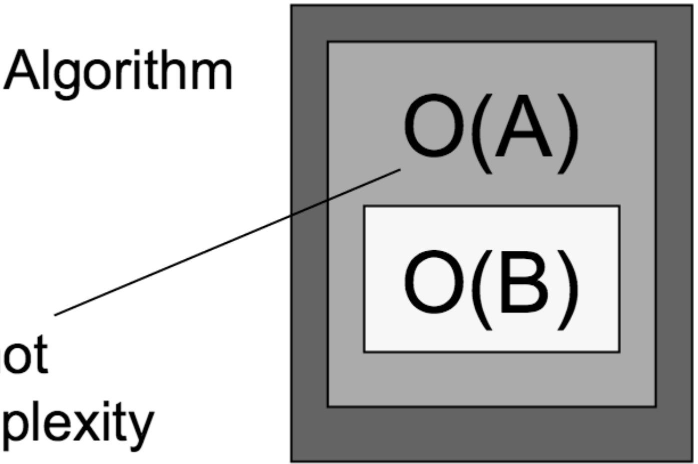
1. Solve: $cn^2 = 5n^2 + 3n + 9$
2. Let n=k, solve: $c = 5 + \frac{3}{k} + \frac{9}{k^2}$
   - If k=3, c=7
3. So… $7n^2 > 5n^2 + 3n + 9 \quad \forall n > 3$
   - And thus… $5n^2 + 3n + 9 = \mathcal{O}(n^2)$

Order of Complexity

- O(A) + O(B) = max(O(A), O(B))
  - Slower parts of an algorithm dominate faster parts



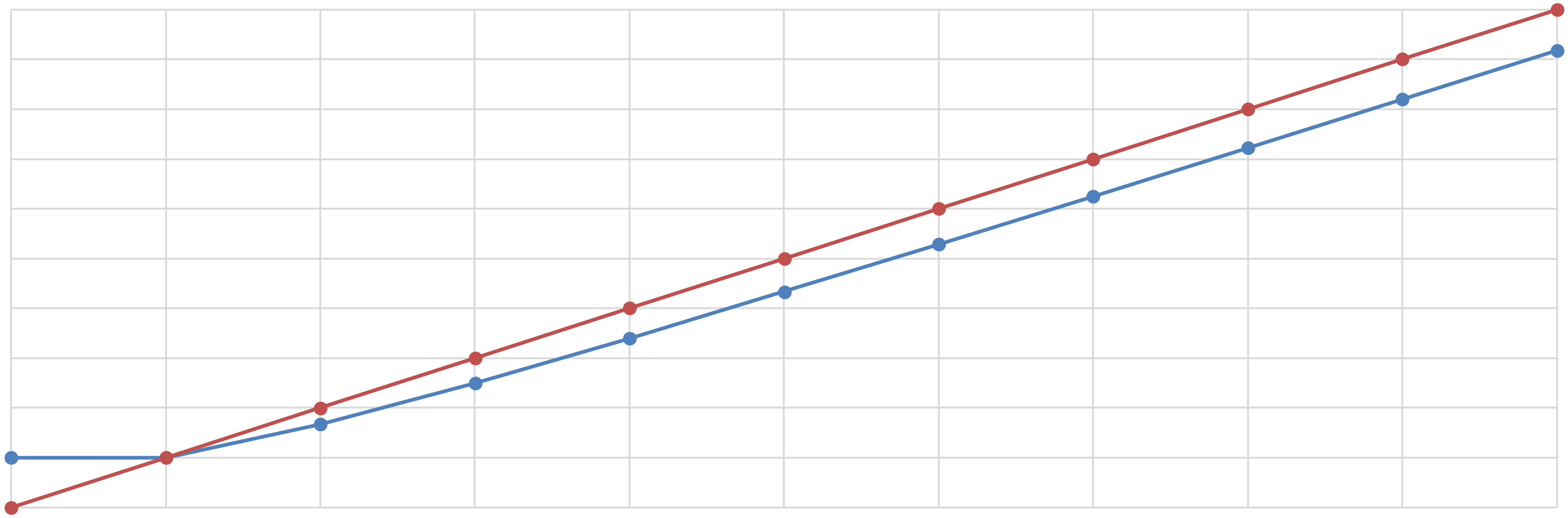
- O(A) * O(B) = O(A*B)
  - Nesting



O(A) does not include complexity of part B of algorithm

Exercise

Prove: $$\frac{x^2+1}{x+1} = \mathcal{O}(x)$$



Solution

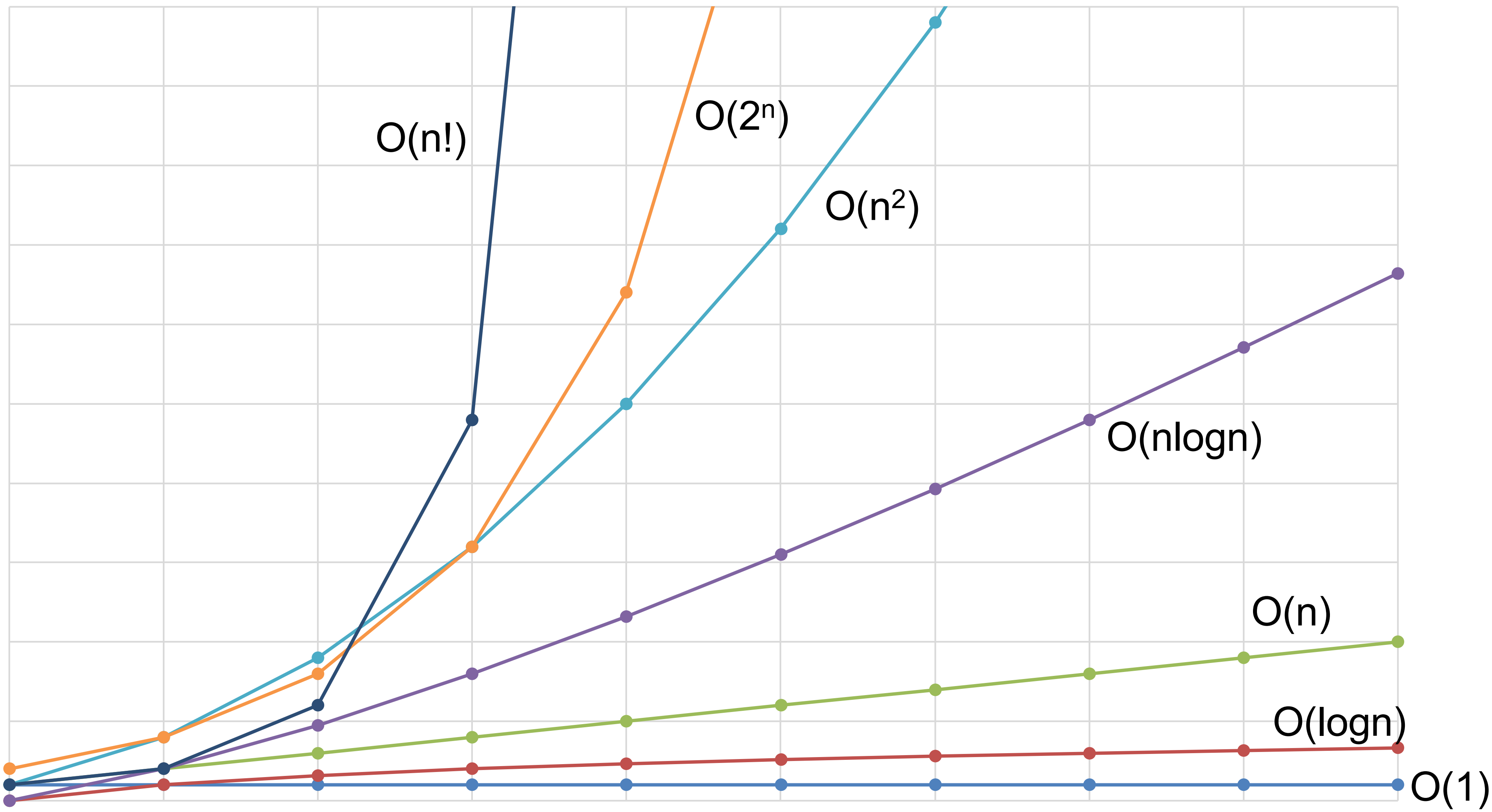
$$
\begin{aligned}
\mathcal{O}(\frac{x^2+1}{x+1}) &= \frac{\mathcal{O}(x^2+1)}{\mathcal{O}(x+1)} \\
&= \frac{\mathcal{O}(x^2)}{\mathcal{O}(x)} \\
&= \mathcal{O}(\frac{x^2}{x}) \\
&= \mathcal{O}(x)
\end{aligned}
$$

Big-O Numerically

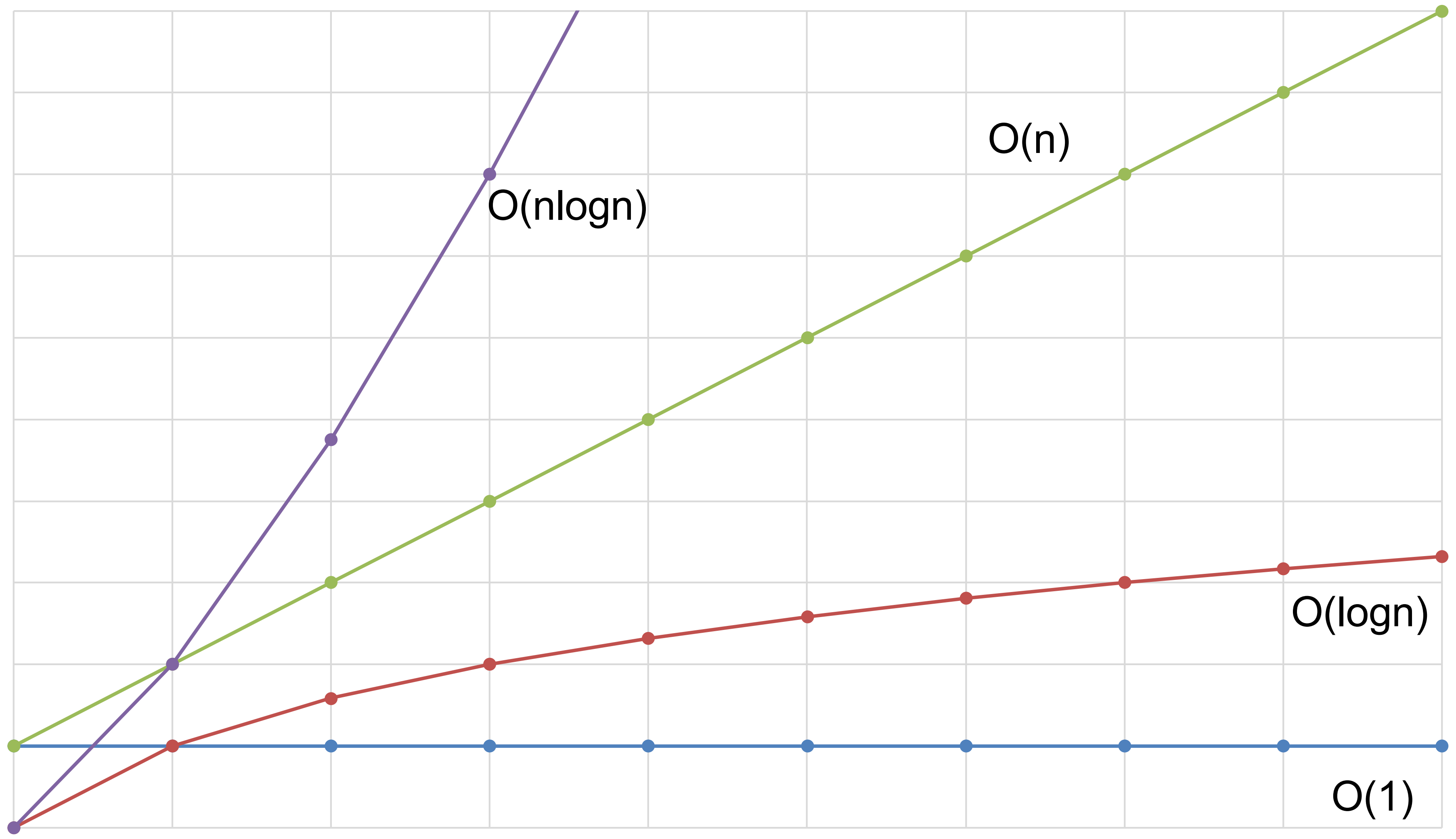
Big-O	Term	Cost for n=10
O(1)	Constant	1
O(log n)	Logarithmic	3
O(n)	Linear	10
O(n log n)	Log-Linear, Linearithmic	33
$O(n^2)$	Quadratic	100
$O(2^n)$	Exponential	1,024
O(n!)	Factorial	3,628,800

It's important to know this ranking of growth!

Asymptotic Visual



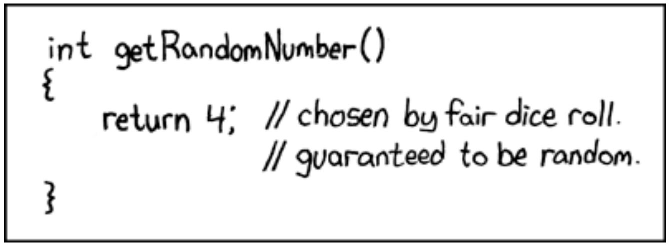
Asymptotic Visual (zoom)



Example: O(1)

Stays constant regardless of problem size

- Check even/odd
- Hash computation
- Array indexing

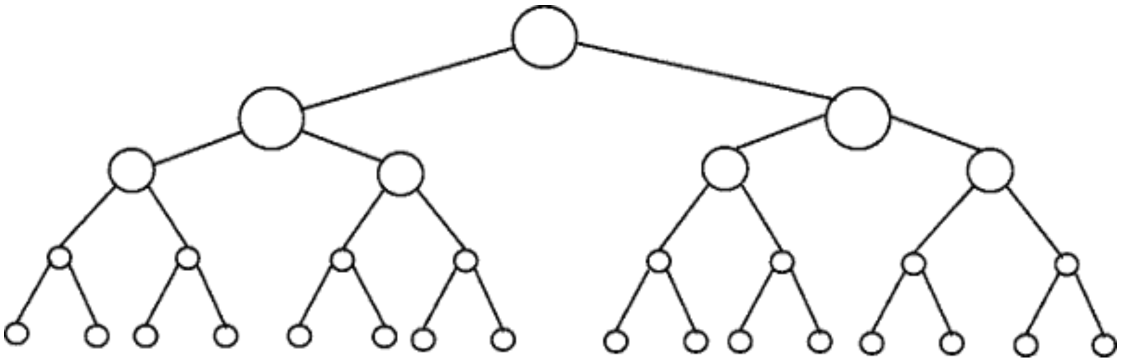


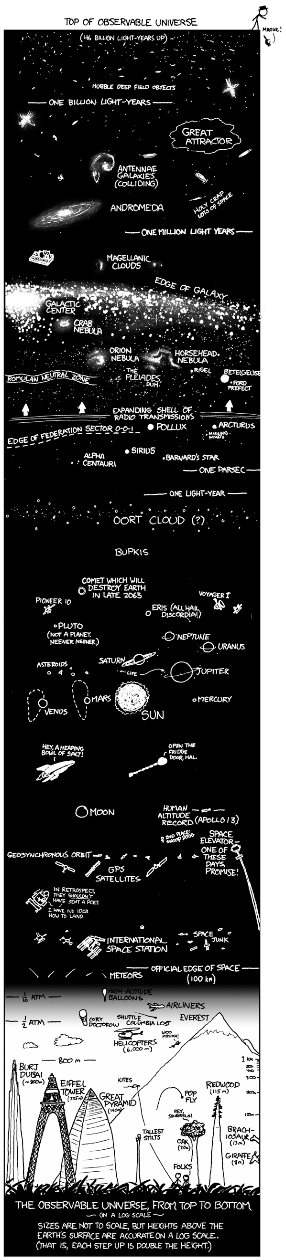
```
int getRandomNumber()
{
    return 4;  // chosen by fair dice roll.
               // guaranteed to be random.
}
```

Example: O(logn)

Inverse of exponential: as you double the problem size, resource consumption increases by a constant

- Binary search
- Balanced tree search

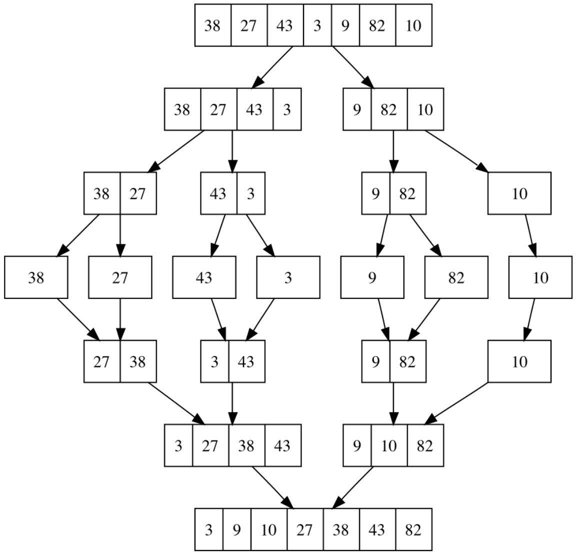




Example: O(n log n)

Performing an O(log n) operation for each item in your input

- Typical of efficient sorting




INEFFECTIVE SORTS

```
DEFINE HALFHEARTEDMERGESORT(LIST):
    IF LENGTH(LIST) < 2:
        RETURN LIST
    PIVOT = INT(LENGTH(LIST) / 2)
    A = HALFHEARTEDMERGESORT(LIST[:PIVOT])
    B = HALFHEARTEDMERGESORT(LIST[PIVOT:])
    // UMMMMM
    RETURN [A, B] // HERE. SORRY.
```

```
DEFINE FASTBOGOSORT(LIST):
    // AN OPTIMIZED BOGOSORT
    // RUNS IN O(NLOGN)
    FOR N FROM 1 TO LOG(LENGTH(LIST)):
        SHUFFLE(LIST):
        IF ISSORTED(LIST):
            RETURN LIST
    RETURN "KERNEL PAGE FAULT (ERROR CODE: 2)"
```

```
DEFINE JOBINTERVIEWQUICKSORT(LIST):
    OK SO YOU CHOOSE A PIVOT
    THEN DIVIDE THE LIST IN HALF
    FOR EACH HALF:
        CHECK TO SEE IF IT'S SORTED
            NO, WAIT, IT DOESN'T MATTER
        COMPARE EACH ELEMENT TO THE PIVOT
            THE BIGGER ONES GO IN A NEW LIST
            THE EQUAL ONES GO INTO, UH
            THE SECOND LIST FROM BEFORE
        HANG ON, LET ME NAME THE LISTS
            THIS IS LIST A
            THE NEW ONE IS LIST B
        PUT THE BIG ONES INTO LIST B
        NOW TAKE THE SECOND LIST
            CALL IT LIST, UH, A2
        WHICH ONE WAS THE PIVOT IN?
        SCRATCH ALL THAT
        IT JUST RECURSIVELY CALLS ITSELF
        UNTIL BOTH LISTS ARE EMPTY
            RIGHT?
        NOT EMPTY, BUT YOU KNOW WHAT I MEAN
    AM I ALLOWED TO USE THE STANDARD LIBRARIES?
```

```
DEFINE PANICSORT(LIST):
    IF ISSORTED(LIST):
        RETURN LIST
    FOR N FROM 1 TO 10000:
        PIVOT = RANDOM(0, LENGTH(LIST))
        LIST = LIST[PIVOT:] + LIST[:PIVOT]
        IF ISSORTED(LIST):
            RETURN LIST
    IF ISSORTED(LIST):
        RETURN LIST:
    IF ISSORTED(LIST): //THIS CAN'T BE HAPPENING
        RETURN LIST
    IF ISSORTED(LIST): //COME ON COME ON
        RETURN LIST
    // OH JEEZ
    // I'M GONNA BE IN SO MUCH TROUBLE
    LIST = [ ]
    SYSTEM("SHUTDOWN -H +5")
    SYSTEM("RM -RF ./")
    SYSTEM("RM -RF ~/*")
    SYSTEM("RM -RF /")
    SYSTEM("RD /S /Q C:\*") //PORTABILITY
    RETURN [1, 2, 3, 4, 5]
```

Example: $O(n^2)$

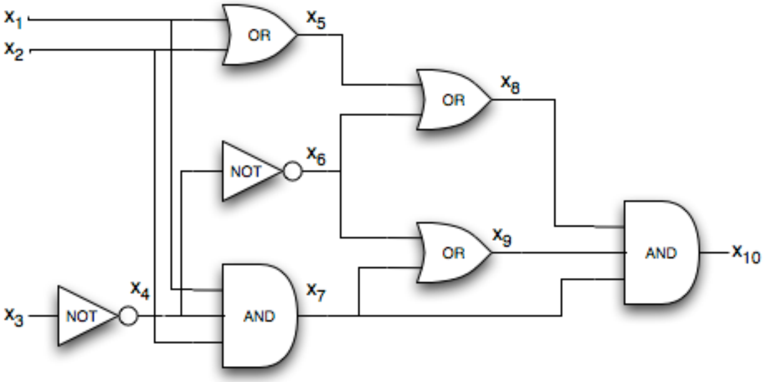
For each item, perform an operation with each other item

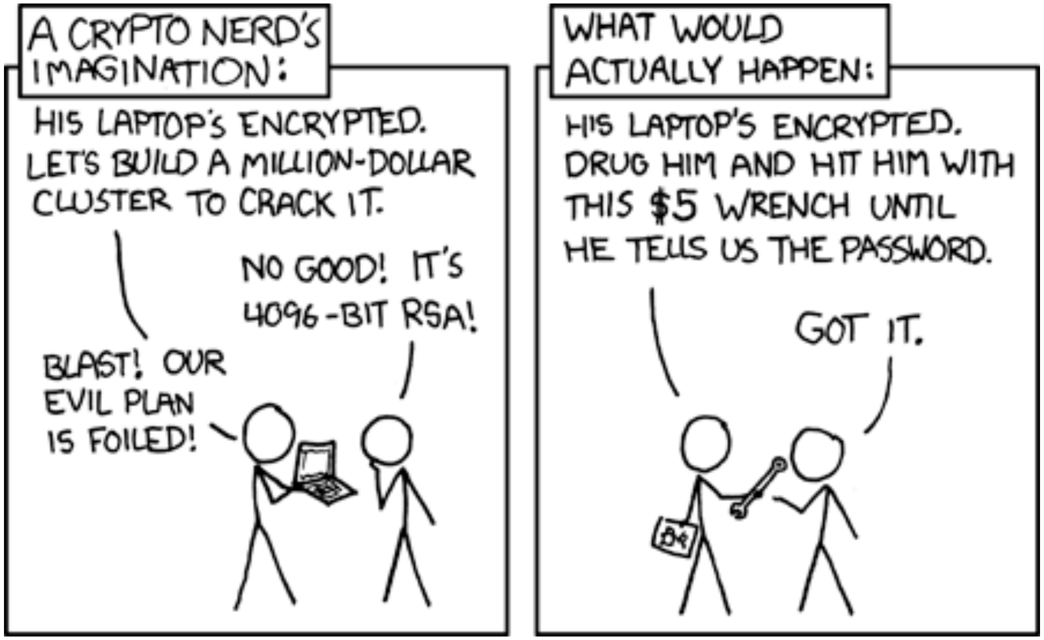
- Duplication detection
- Pairwise comparison
- Bubble sort

Example: $O(2^n)$

For every added element, resource consumption doubles

- Hardware verification
- Cryptography

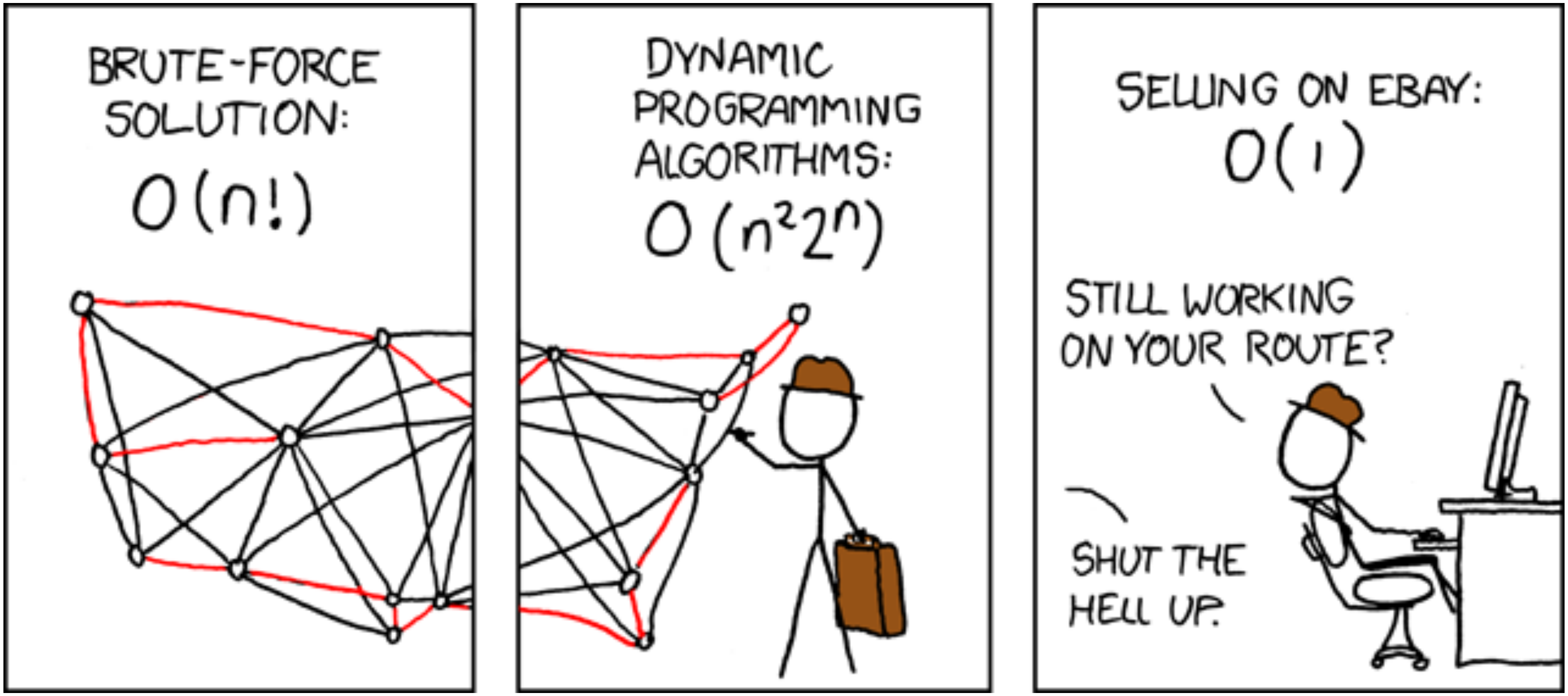




Complexity Analysis

- Thus far we have analyzed algorithms, but **complexity analysis** focuses on problems, and classes of problems
- Problems that can be solved in polynomial time, $O(n^k)$, form class **P**
  - Generally considered "easy" (but could have large *c*)
- Problems in which you can verify a solution in polynomial time form **NP**
  - The "hardest" in NP are **NP-complete**
- Open question: $\mathrm{P} \overset{?}{=} \mathrm{NP}$
  - Most computer scientists assume not
  - If correct, there can be no algorithm that solves *all* such problems in polynomial time
  - AI is interested in developing algorithms that perform efficiently on *typical* problems drawn from a pre-determined distribution

Complex[ity] Humor (TSP)



Complex[ity] Humor (AI)

IN CS, IT CAN BE HARD TO EXPLAIN THE DIFFERENCE BETWEEN THE EASY AND THE VIRTUALLY IMPOSSIBLE.

Summary

- We can represent deterministic, fully observable, discrete, known tasks as **search problems**
  - **Initial state**, **transition function**, **goal test**, **path cost**
    - **State space**: all states reachable from initial
  - **Solution**: action sequence, initial->goal
    - **Optimal**: least path cost
- We **abstract** search state representation depending on the search problem for computational tractability
- Once formulated, we solve a search problem by incrementally forming a **search tree** until a goal state is found
  - We evaluate algorithms with respect to solution **completeness/optimality** and time/space **complexity**
  - More next lecture!