Algorithmic Analysis and Sorting

Part One

Announcements

- Solutions to warm-up recursion problems have been posted.
- Midterm is next **Monday, July 22** from **7PM – 10PM**.
  - Cubberly Auditorium.
  - Please email Michael and I ASAP if you have a conflict with the exam time.
  - Please email Michael and I in the next couple of days if you need special accomodations.

Midterm

- Close book, close note, close laptop
- No phones or MP3 Players
  - Super lame, but it's been an issue in the past =(
  - If you find the noise of 120 scribbling on paper distracting, then I recommend wearing earplugs
  - If you need to be able to check your phone (e.g. you're an on-call Doctor) then please let me know
- Covers material through this Wednesday

Midterm

- Reference sheet will be provided at the exam
  - Will be posted on the website later today.
  - If you think something is missing that should be there, then please let me know!
- Practice Exam will be posted later today
- **Please do not look at past midterms!**
  - We don't intentionally reuse problems.
  - If you happen to look at a previous midterm by mistake:
    - Don't worry, you're not in trouble, but please let me know just so I can make sure everyone in the class has access to it. I just want things to be fair.

Studying for the Midterm

- Exam tests your understanding of data structures, recursion and algorithmic analysis (this week)
- Studying in CS106B involves:
  - Section handout
  - Practice midterm
  - Problems in class and lecture slides
  - Reading course reader
- **Reading solutions is probably not sufficient!**
  - Study skills handout will be on the website later today. Please read this!
  - Do problems by hand, not on your computer!

What May be on the Midterm

- Data structures:
  - Ability to use them to solve problems
  - Pros and cons of using different data structures
- Recursion:
  - Tower of Hanoi
  - “Divide-and-Conquer” (Random Parking)
  - Exhaustive (Subsets,Permutations)
  - Recursive Backtracking (Shrinkable Words)
- Simple Algorithmic Analysis (Big-O)

What May be on the Midterm

- Mostly coding questions
- Maybe some short answer questions
- Maybe generate a decision tree
- Maybe read some code and tell me what it does

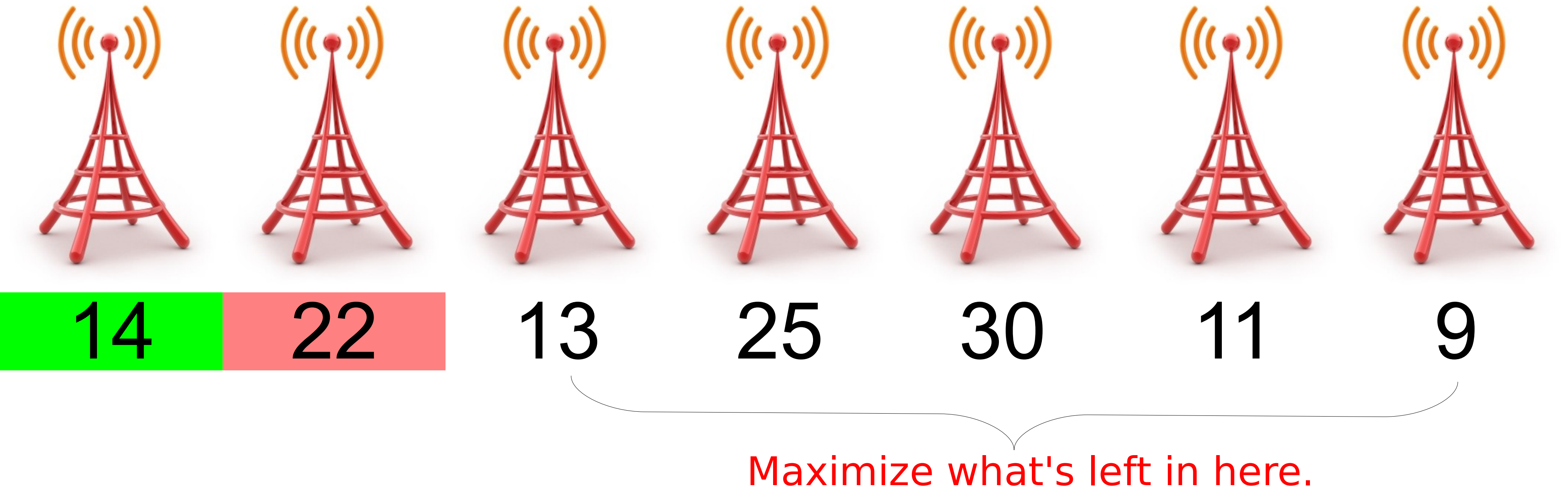
What's **not** on the Midterm

- Name-the-function-call
  - “What's the Stanford C++ method for getting an integer from the user?”
- Specific Algorithms
  - “Implement Shaunting-Yard from memory”

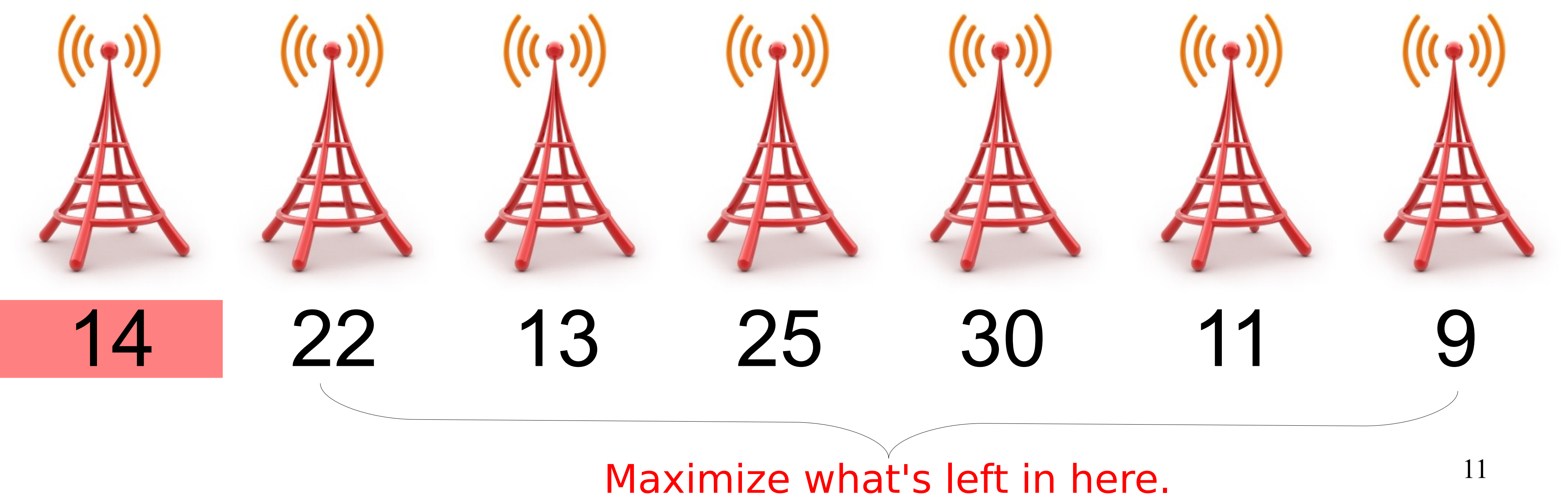
Everything in this class can be understood by anyone through hard work and effective study techniques.

If you would like help studying, please let me know.

Memoization



14
22
13
25
30
11
9
Maximize what's left in here.



14
22
13
25
30
11
9
Maximize what's left in here.

Counting Recursive Calls

- Let n be the number of cities.
- Let $C(n)$ be the number of function calls made.
  - If $n = 0$, there is just one call, so $C(0) = 1$.
  - If $n = 1$, there is just one call, so $C(1) = 1$.
  - If $n \geq 2$, we have the initial function call, plus the two recursive calls. So $C(n) = 1 + C(n - 1) + C(n - 2)$.

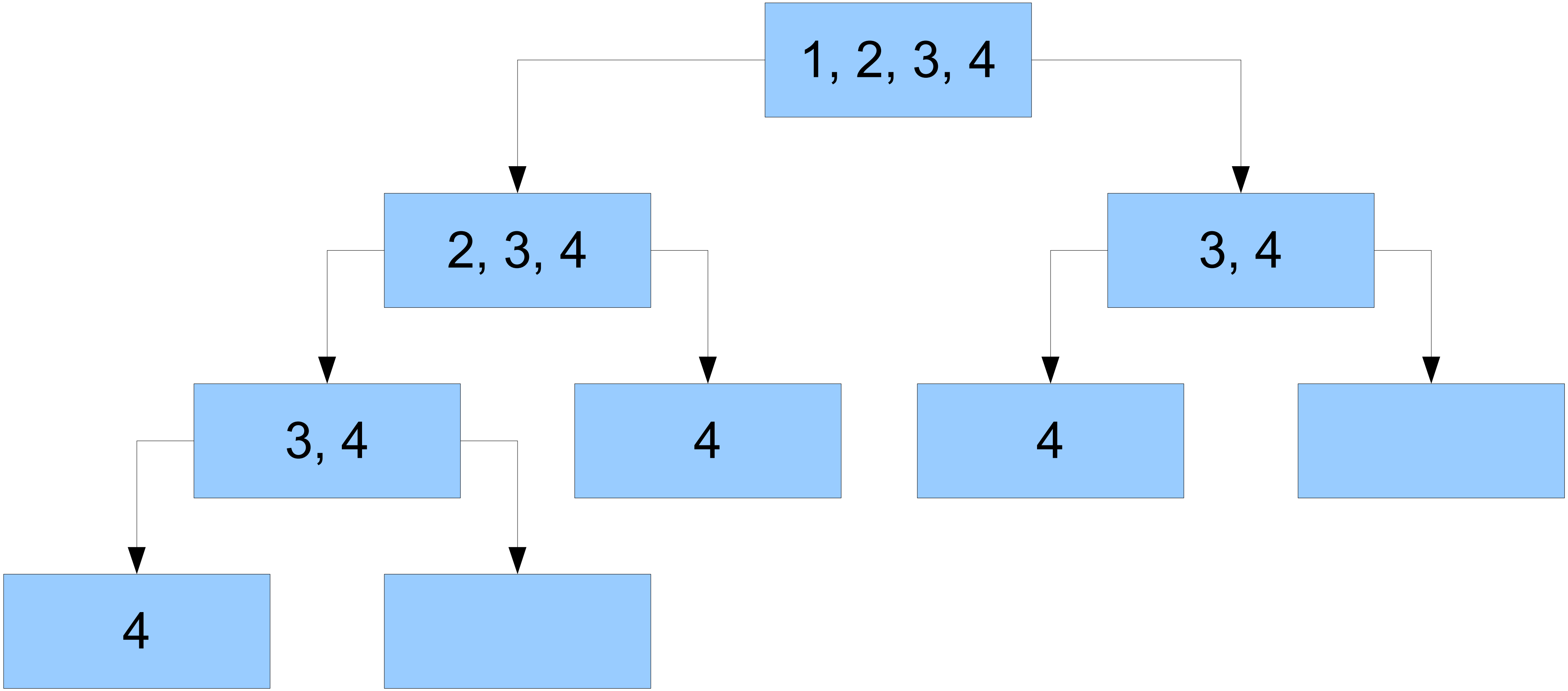
Counting Recursive Calls

- C(0) = C(1) = 1.
- C(n) = 1 + C(n – 1) + C(n – 2)
- This gives the series

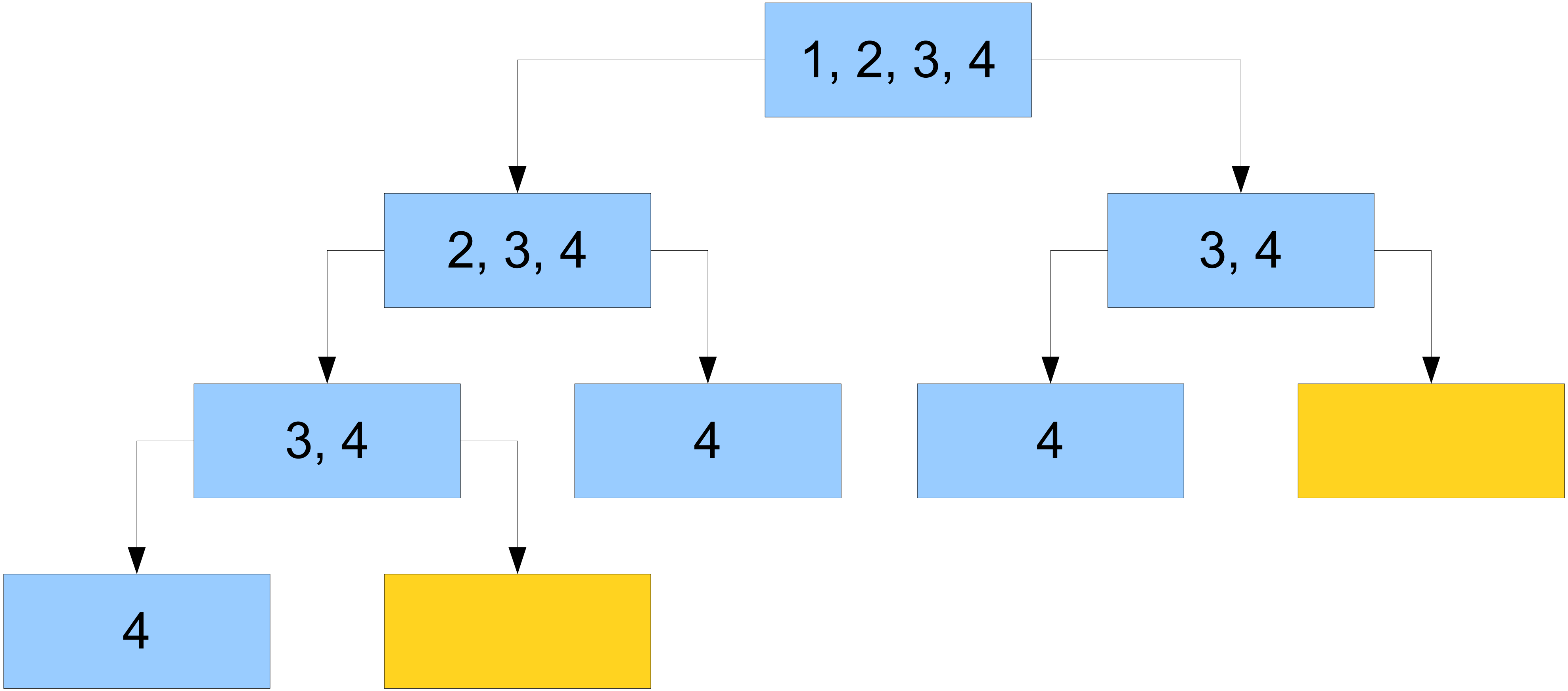
1, 1, 3, 5, 9, 15, 25, 41, 67, 109, 177, 287, 465, 753, 1219, 1973, 3193, 5167, …

- This function grows very quickly, so our solution will scale very poorly.
- Neat mathematical aside – these numbers are called the **Leonardo numbers**.

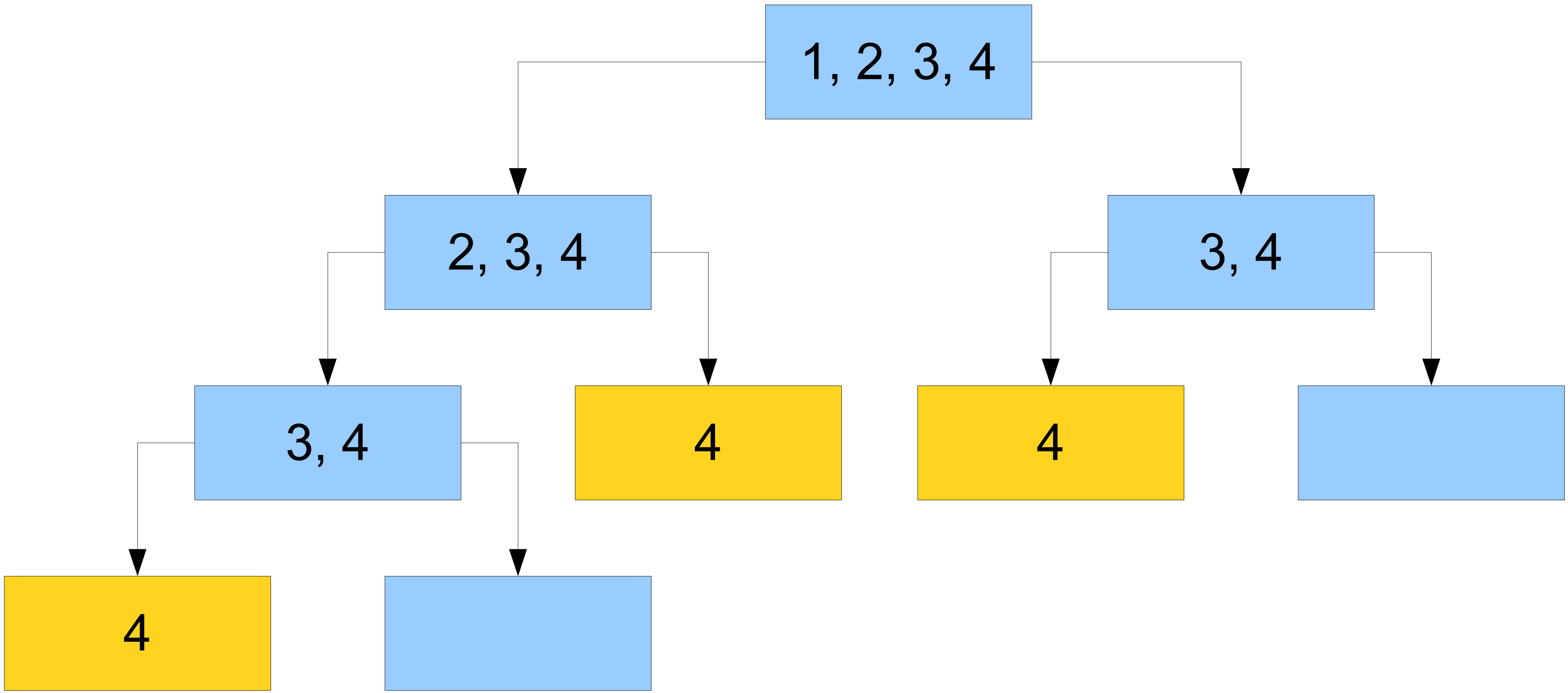
The Call Tree



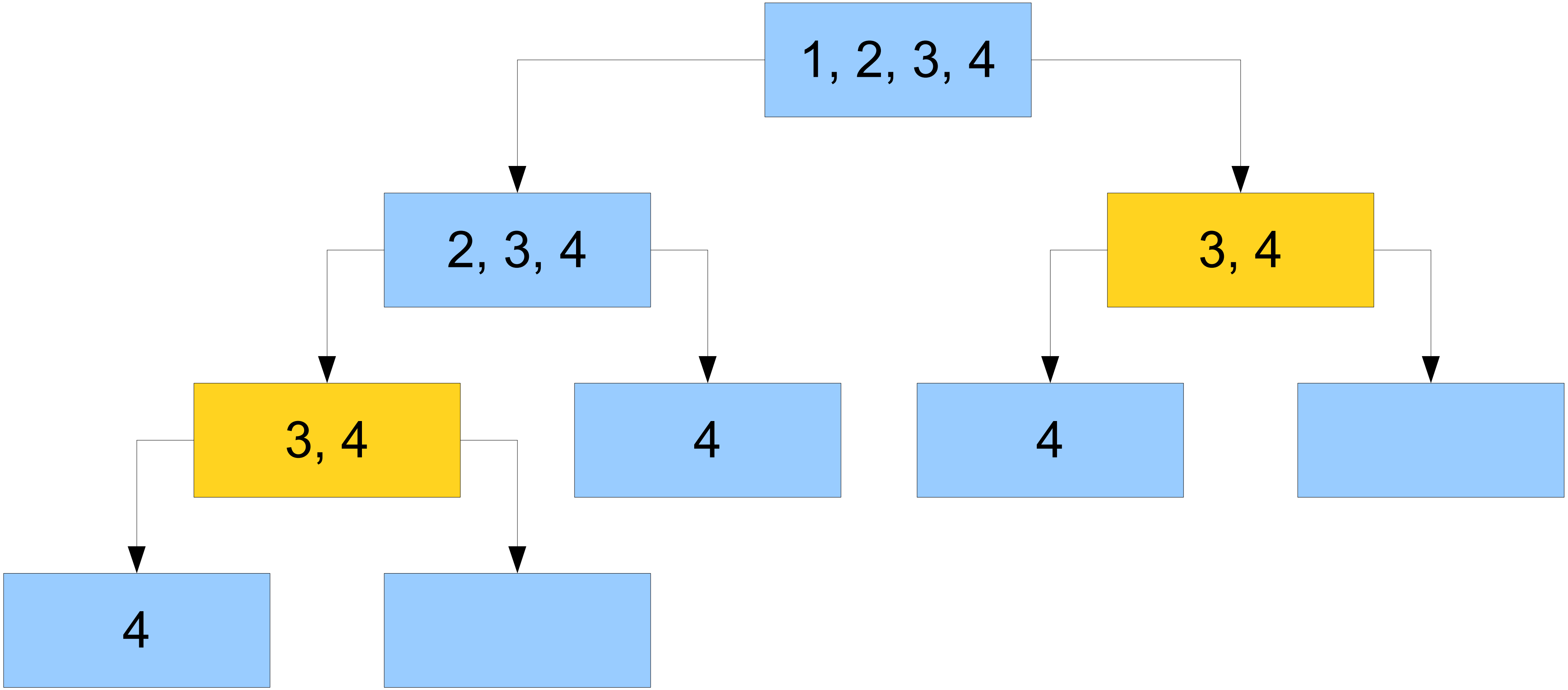
The Call Tree



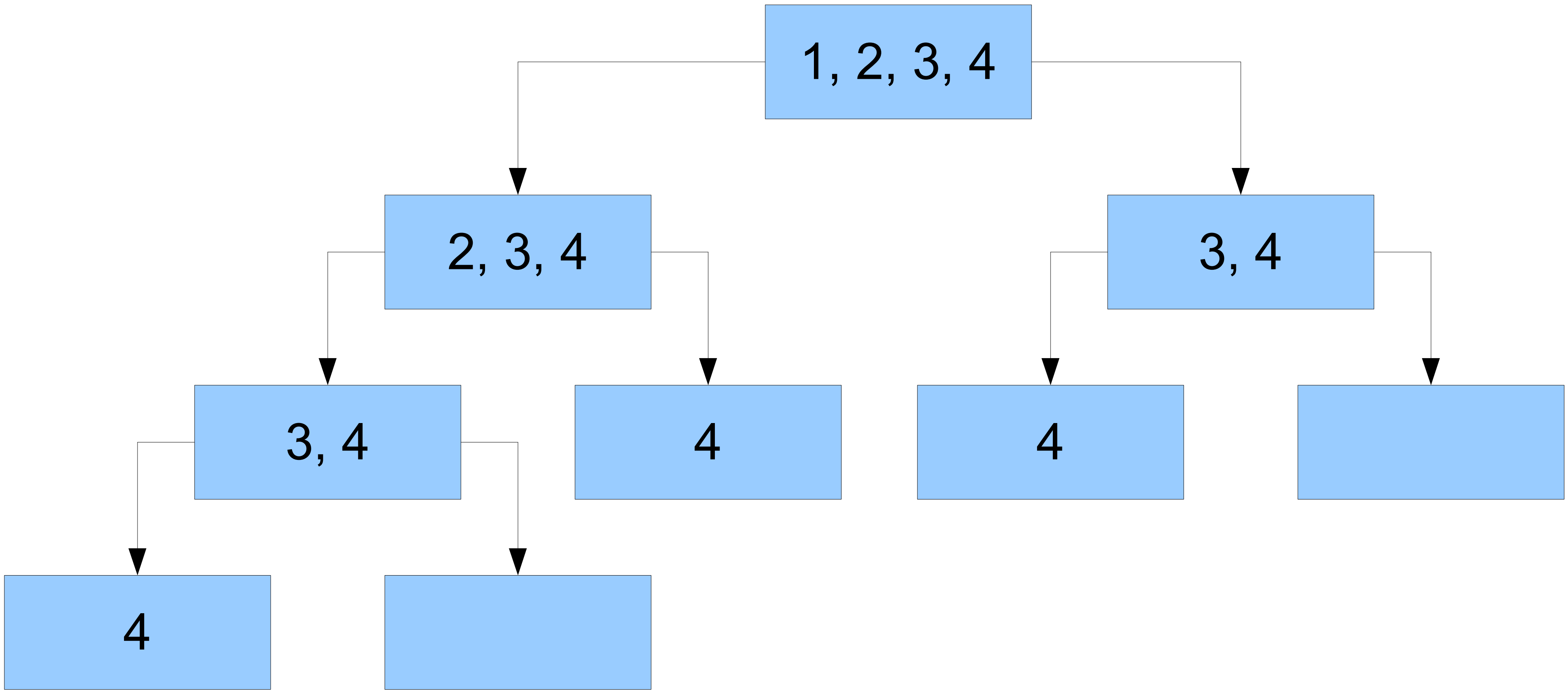
The Call Tree



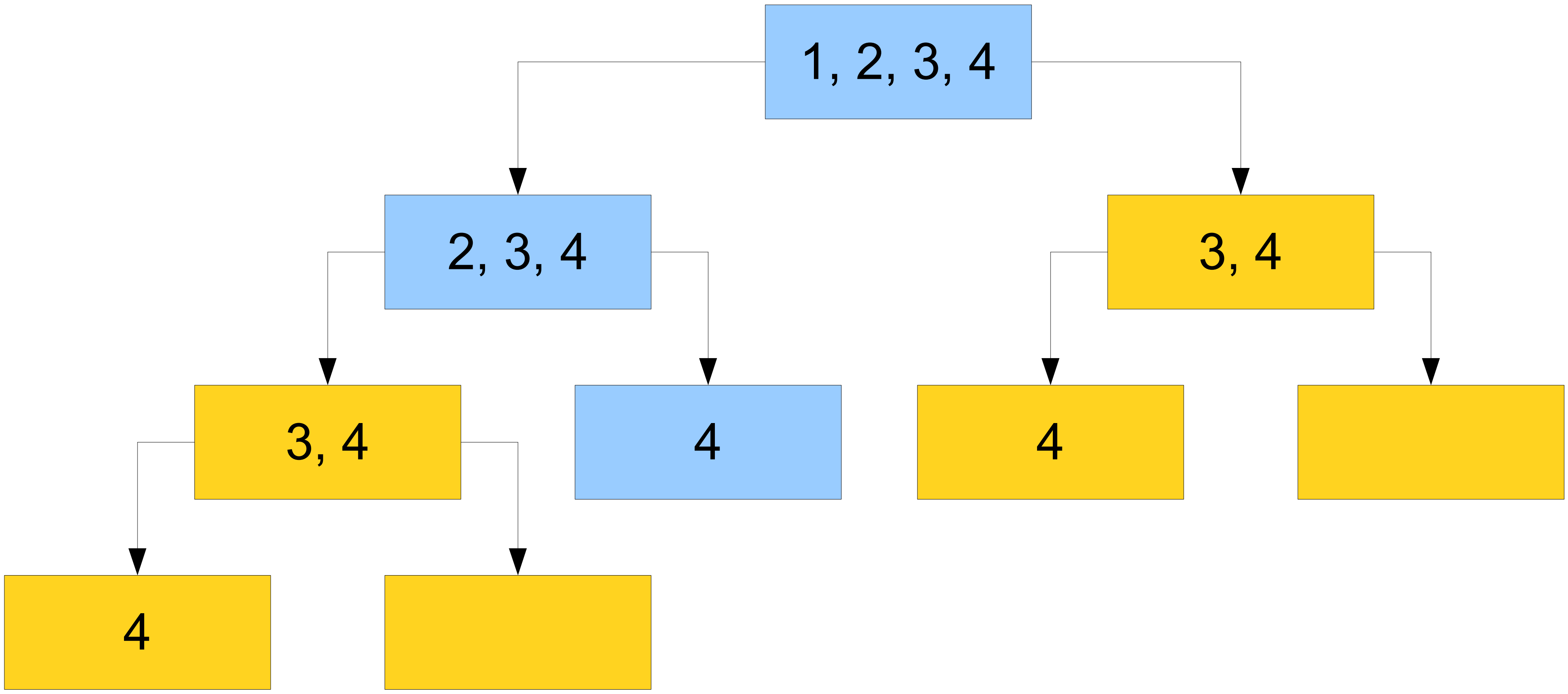
The Call Tree



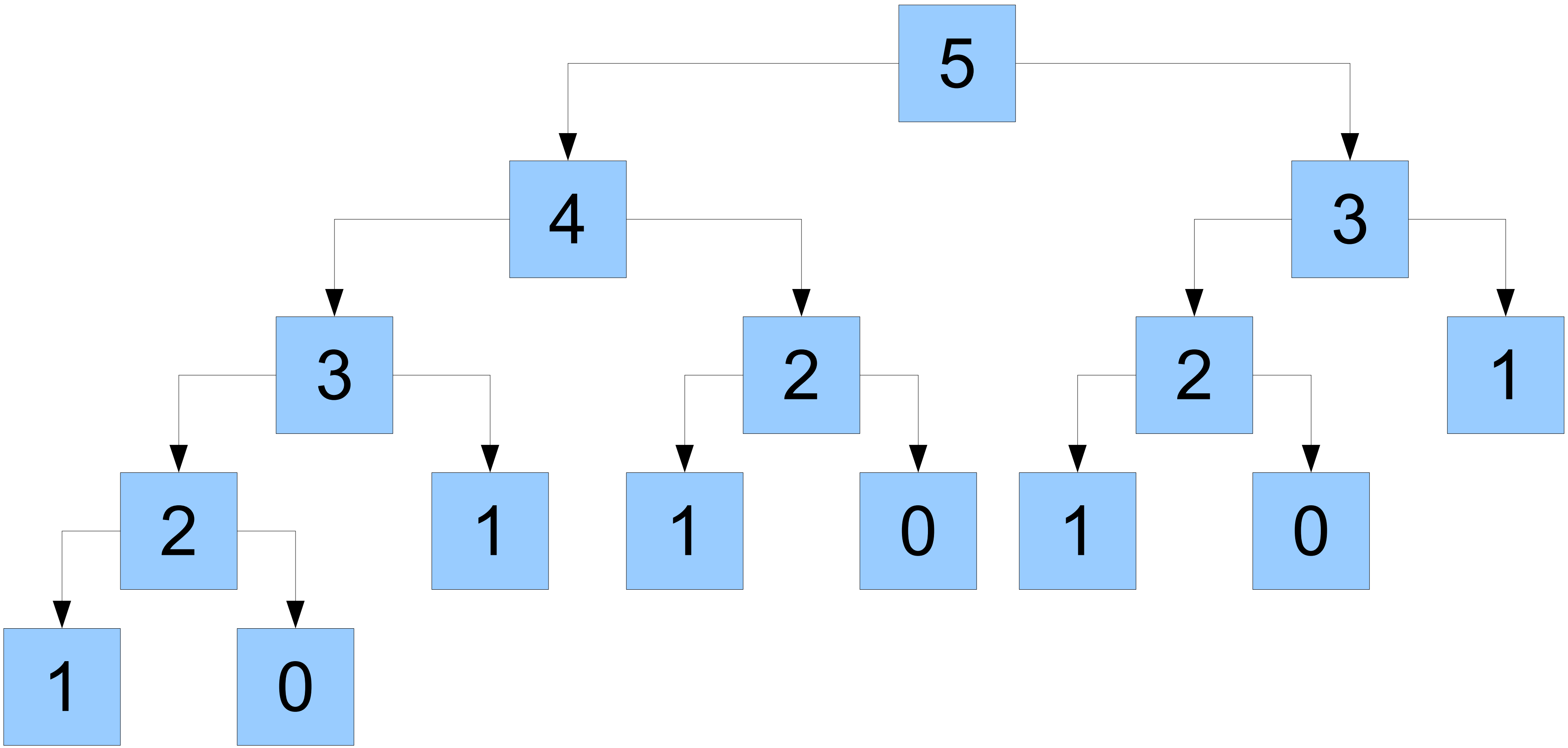
The Call Tree



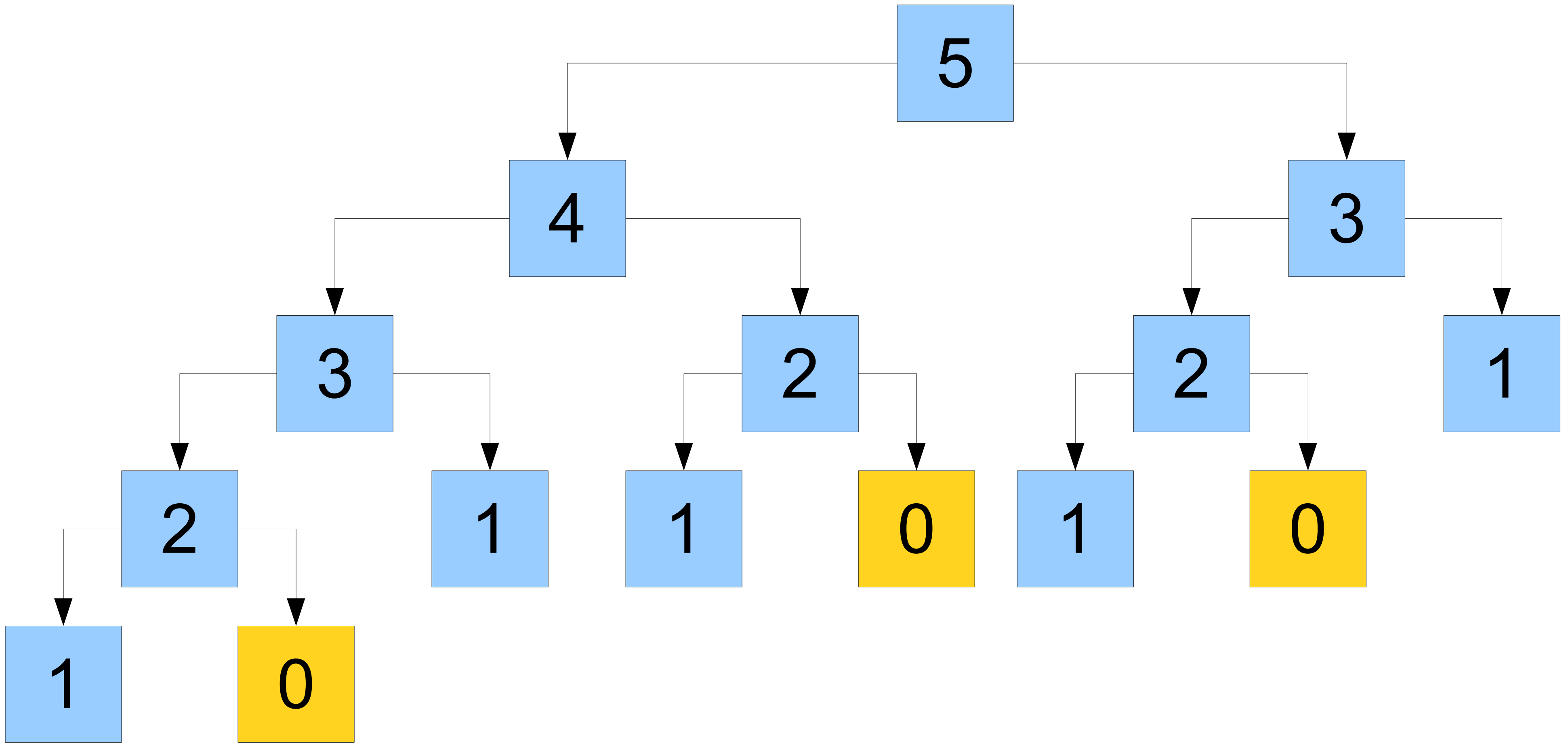
The Call Tree



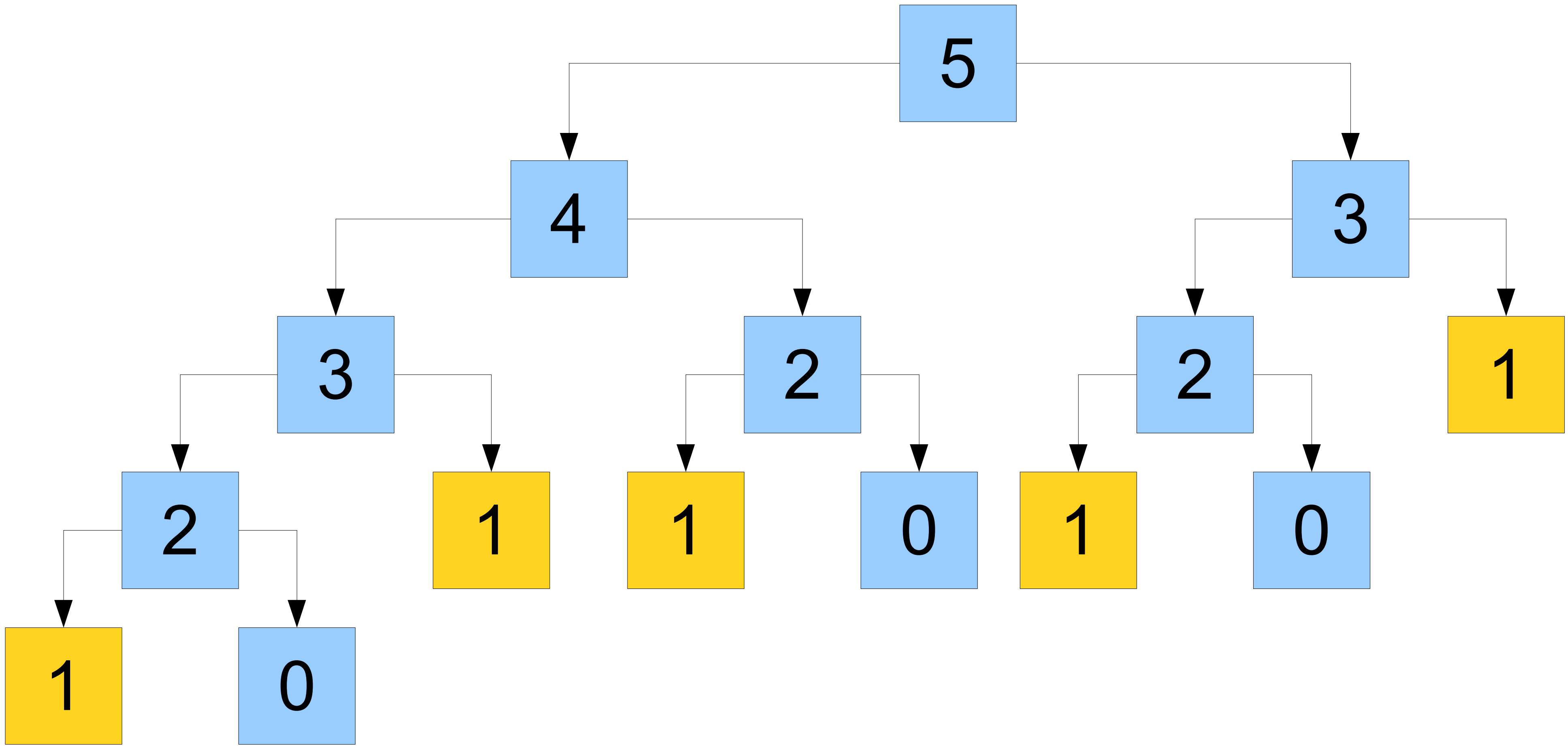
A Bigger Call Tree



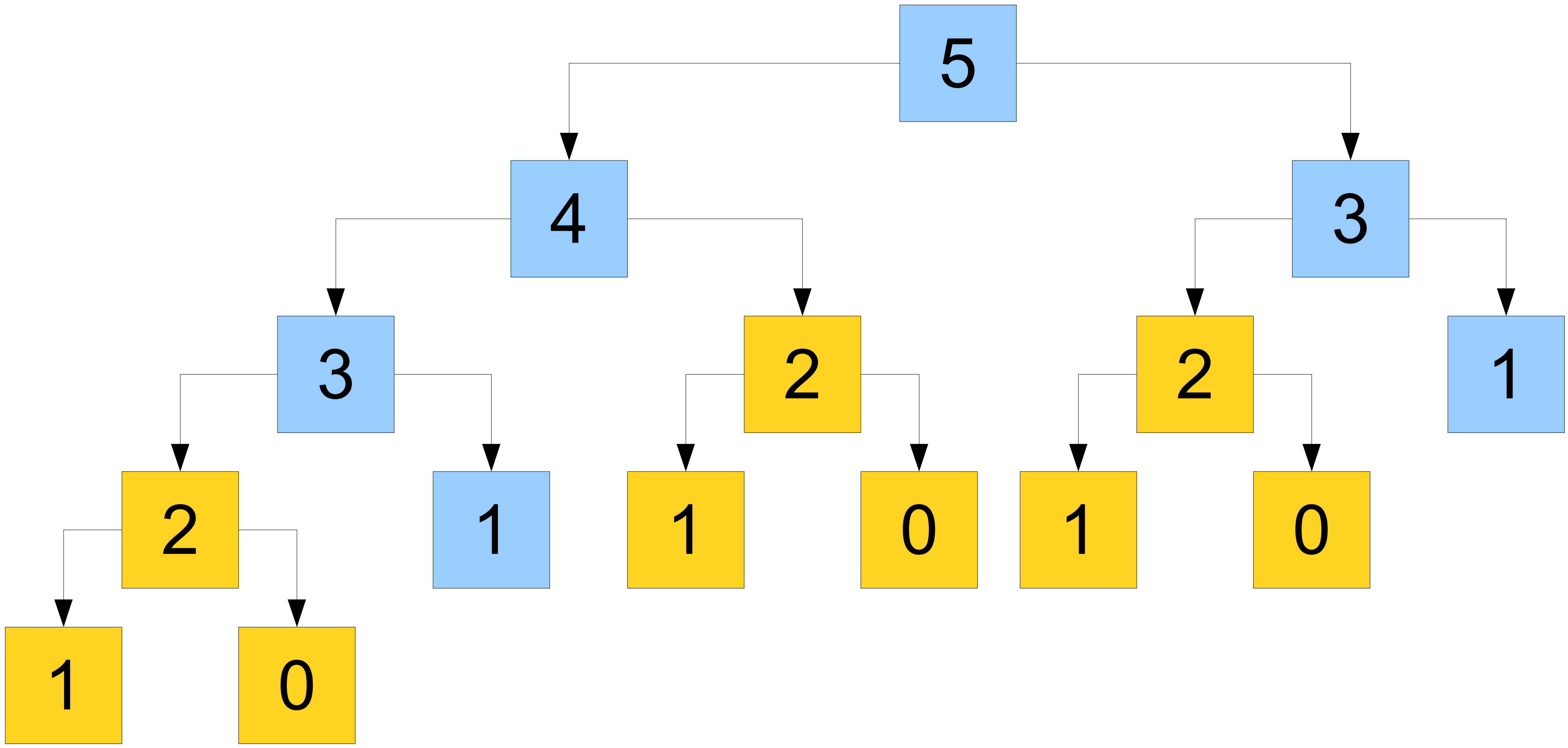
A Bigger Call Tree



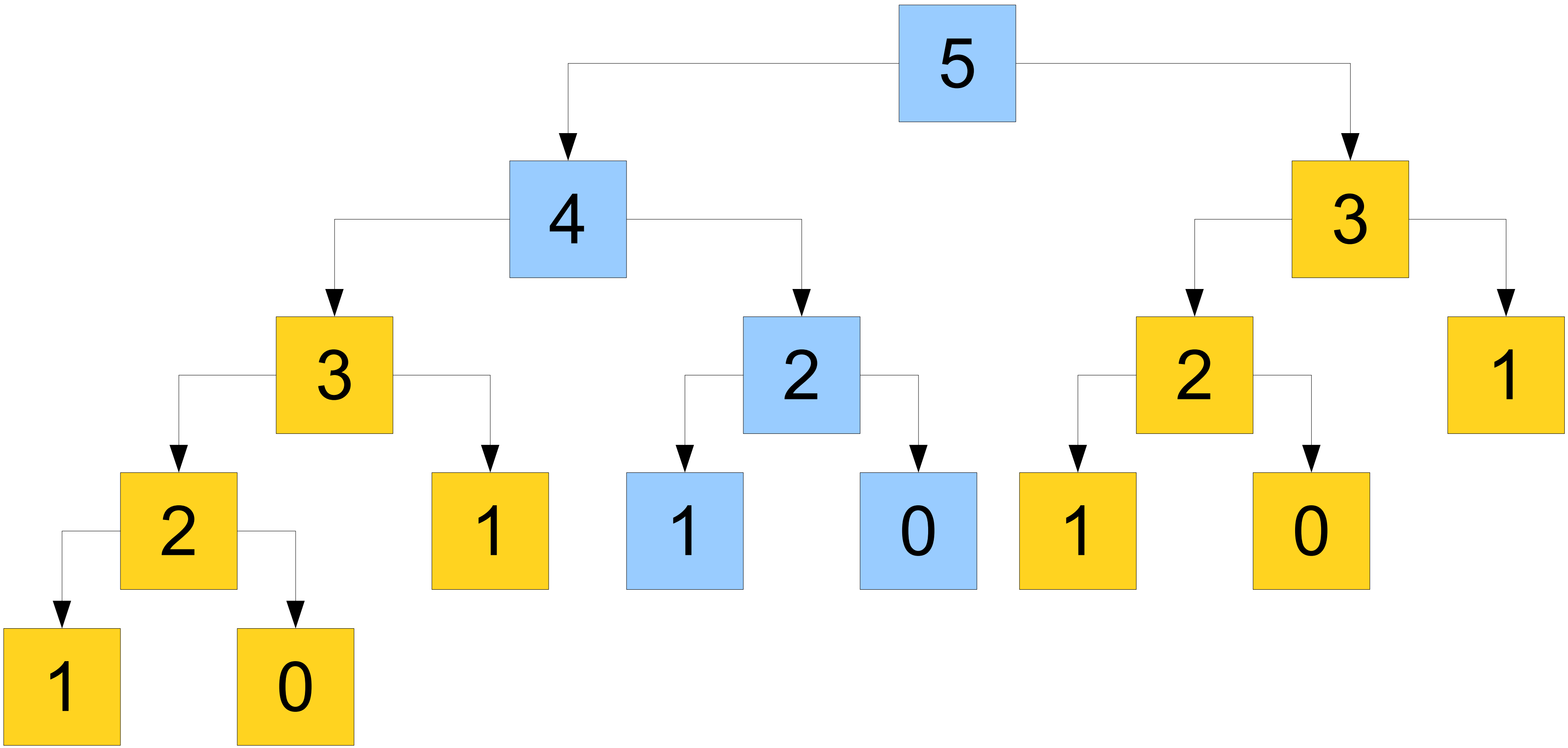
A Bigger Call Tree



A Bigger Call Tree



A Bigger Call Tree



We're doing completely unnecessary work!
Can we do better?

Cell Towers Revisited
(cell-towers.cpp)

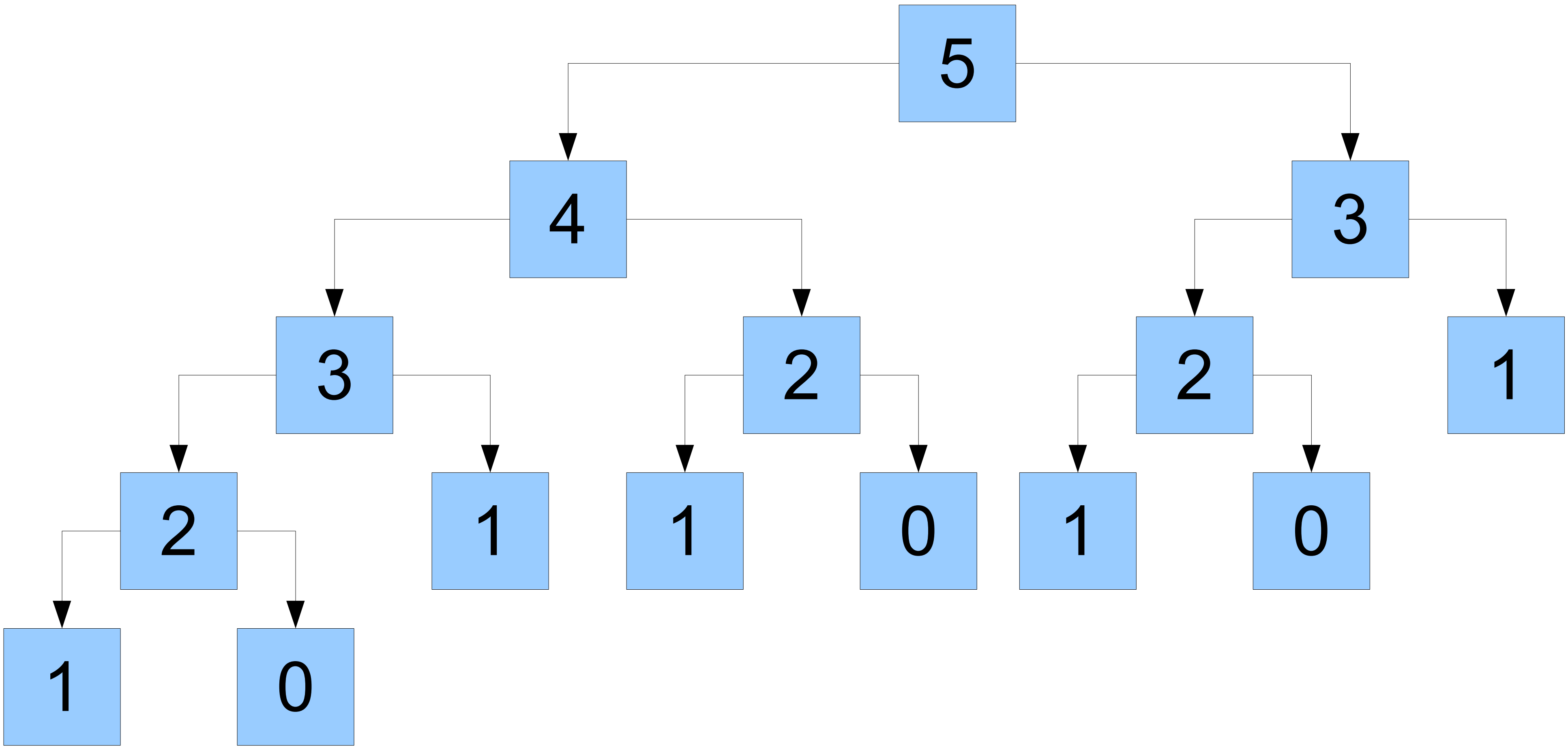
What Just Happened?

- **Remember what values we've computed so far.**
- New base case: If we already computed the answer, we're done.
- When computing a recursive step, record the answer before we return it.
- This is called **memoization**.
  - No, that is not a typo – there's no “r” in memoization.

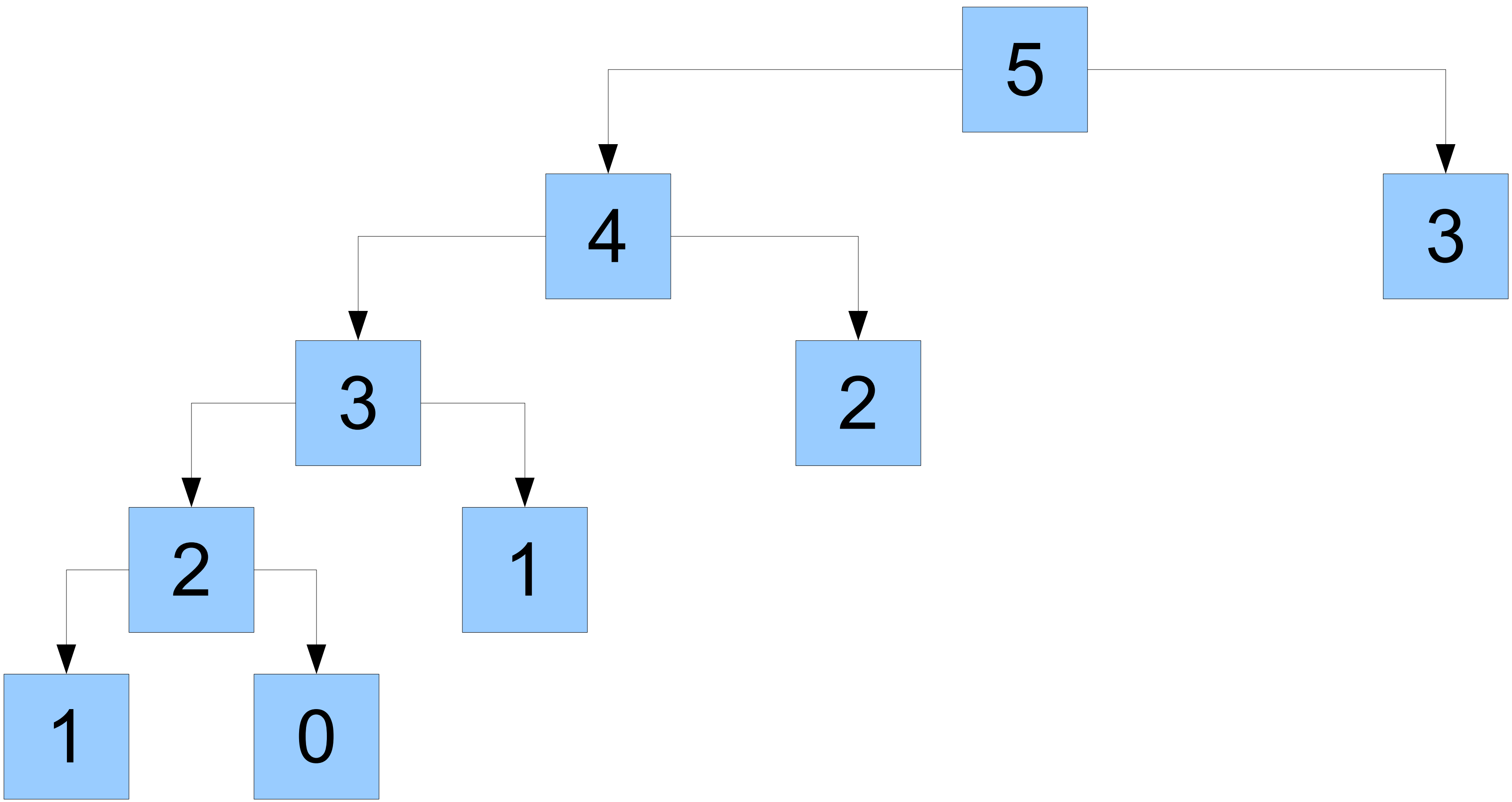
Memoization

- Memoization can be useful if you make redundant recursive calls **and** you don't need to explicitly explore every possible subset/permutation
  - Why wouldn't memoization help in generating permutations/subsets?

Original Call Tree



Memoized Call Tree



Introduction to Algorithmic Analysis

Fundamental Question:

How can we compare solutions to problems?

One Idea: **Runtime**

Why Runtime Isn't a Good Measure

- Fluctuates based on size of input
  - Sorting 2^{10} integers vs 2^{30} integers
- Fluctuates based on computer
  - Sorting integers on a Department of Energy supercomputer vs a personal laptop
- Fluctuates based on difficulty of input
  - Sorting 100 integers that are randomly permuted vs 100 integers that are almost in sorted order

A Better Measure

- Instead of measuring the time it takes for an algorithm to run, measure the amount of “work” it does.
  - Work: Any sort of operation the computer performs (eg. addition, multiplication, checking the condition of an `if` statement)
- Using this as a goal, let's develop a measure that addresses the concerns we outlined earlier

Why Runtime Isn't a Good Measure

- Problem: Fluctuates based on size of input
  - Solution: Let the amount of “work” be a function of the size of its input.

```
double average(Vector<int>& vec) {
   double total = 0.0;
   for (int i = 0; i < vec.size(); i++) {
        total += vec[i];
   }

   return total / vec.size();
}
```

```
double average(Vector<int>& vec) {
    double total = 0.0;
   for (int i = 0; i < vec.size(); i++) {
        total += vec[i];
   }

    return total / vec.size();
}
```

- Let: **`n = vec.size()`**

```
double average(Vector<int>& vec) {
   double total = 0.0;
   for (int i = 0; i < vec.size(); i++) {
       total += vec[i];
   }

   return total / vec.size();
}
```

- Let: **n** = **vec.size()**
  $\mathbf{k}_0$ = work done in each iteration of the for loop

```
double average(Vector<int>& vec) {
   double total = 0.0;
   for (int i = 0; i < vec.size(); i++) {
       total += vec[i];
   }

   return total / vec.size();
}
```

- Let: **n** = **vec.size()**

  $\mathbf{k_0}$ = work done in each iteration of the for loop

```
double average(Vector<int>& vec) {
   double total = 0.0;
   for (int i = 0; i < vec.size(); i++) {
       total += vec[i];
   }

   return total / vec.size();
}
```

- Let: **n** = **vec.size()**
  k_0 = work done in each iteration of the for loop
  k_1 = any other work done in the function (eg: returning a value, initializing **i** to 0)

```
double average(Vector<int>& vec) {
   double total = 0.0;
   for (int i = 0; i < vec.size(); i++) {
       total += vec[i];
   }

   return total / vec.size();
}
```

- Let: **n** = **vec.size()**

  $\mathbf{k_0}$ = work done in each iteration of the for loop

  $\mathbf{k_1}$ = any other work done in the function (eg: returning a value, initializing **i** to 0)

```
double average(Vector<int>& vec) {
    double total = 0.0;
    for (int i = 0; i < vec.size(); i++) {
        total += vec[i];
    }

    return total / vec.size();
}
```

- Let: **n** = **vec.size()**
  $\mathbf{k_0}$ = work done in each iteration of the for loop
  $\mathbf{k_1}$ = any other work done in the function (eg: returning a value, initializing **i** to 0)
- Work = $\mathbf{k_0 n + k_1}$

```
double average(Vector<int>& vec) {
  double total = 0.0;
  for (int i = 0; i < vec.size(); i++) {
      total += vec[i];
}
```

- **k_0n**: component of work done that's *dependent* upon the length of **vec**
- **k_1**: component of work done that's *independent* of the length of **vec**

-

k_0 = work done in each iteration of the for loop

k_1 = any other work done in the function (eg: returning a value, initializing **i** to 0)

- Work = $k_0 n + k_1$

Why Runtime Isn't a Good Measure

- Work = $\mathbf{k_0 n + k_1}$
- How important is the "+ $\mathbf{k_1}$"?
  - As $\mathbf{n}$ becomes large, "$\mathbf{k_0 n + k_1}$" is dominated by the "$\mathbf{k_0 n}$" term, so we can drop the "+ $\mathbf{k_1}$" and still have a good sense of how much work the algorithm does
    - Work = $\mathbf{k_0 n}$

Why Runtime Isn't a Good Measure

- Work = $\mathbf{k_0 n}$
- How important is the "$\mathbf{k_0}$"?
  - "$\mathbf{k_0}$" is a function of how fast a computer can perform basic operations (add, multiply, divide, check boolean value, etc)
    - "$\mathbf{k_0}$" is going to vary from computer to computer
  - Because "$\mathbf{k_0}$" only tells us something about the computer the algorithm is run on, we choose to drop it.
  - Work = $\mathbf{n}$

Big Observations

- Don't need to explicitly compute these constants.
  - Whether runtime is $4n + 10$ or $100n + 137$, runtime is still proportional to input size.
  - Can just plot the runtime to obtain actual values.
- Only the dominant term matters.
  - For both $4n + 1000$ and $n + 137$, for very large n most of the runtime is explained by n.
- Is there a concise way of describing this?

Big-O

Big-O Notation

- Ignore *everything* except the dominant growth term, including constant factors.
- Examples:
  - $4n + 4 = \mathbf{O(}\boldsymbol{n}\mathbf{)}$
  - $137n + 271 = \mathbf{O(}\boldsymbol{n}\mathbf{)}$
  - $n^2 + 1000n + 100000 = \mathbf{O(}\boldsymbol{n}^{\mathbf{2}}\mathbf{)}$
  - $2^n + n^3 = \mathbf{O(2}^{\boldsymbol{n}}\mathbf{)}$

Algorithmic Analysis with Big-O

Algorithmic Analysis with Big-O

```
double average(Vector<int>& vec) {
   double total = 0.0;
   for (int i = 0; i < vec.size(); i++) {
        total += vec[i];
   }

   return total / vec.size();
}
```

Algorithmic Analysis with Big-O

```
double average(Vector<int>& vec) {
    double total = 0.0;
   for (int i = 0; i < vec.size(); i++) {
        total += vec[i];
   }

    return total / vec.size();
}
```

Algorithmic Analysis with Big-O

```
double average(Vector<int>& vec) {
    double total = 0.0;
   for (int i = 0; i < vec.size(); i++) {
        total += vec[i];
   }

    return total / vec.size();
}
```

O(n)

A More Interesting Example

A More Interesting Example

```
bool linearSearch(string& str, char ch) {
   for (int i = 0; i < str.length(); i++) {
      if (str[i] == ch) {
         return true;
      }
   }

   return false;
}
```

A More Interesting Example

```
bool linearSearch(string& str, char ch) {
   for (int i = 0; i < str.length(); i++) {
      if (str[i] == ch) {
         return true;
      }
   }

   return false;
}
```

A More Interesting Example

```
bool linearSearch(string& str, char ch) {
  for (int i = 0; i < str.length(); i++) {
    if (str[i] == ch) {
      return true;
    }
  }

  return false;
}
```

How do we analyze this?

A More Interesting Example

- Say we are performing a linear search for the character 'a' in these two strings:
  - “this is my viola”
  - “actually, that isn't”
- This comes back to one of our original concerns with simply measuring runtime
  - Problem: Runtime fluctuates based on difficulty of input
  - Solution: Make some sort of assumption of the difficulty of the input

Types of Analysis

- Worst-Case Analysis
  - What's the *worst* possible runtime for the algorithm?
  - Useful for "sleeping well at night."
- Best-Case Analysis
  - What's the *best* possible runtime for the algorithm?
  - Useful to see if the algorithm performs well in some cases.
- Average-Case Analysis
  - What's the *average* runtime for the algorithm?
  - Far beyond the scope of this class; take CS109, CS161, CS365, or CS369N for more information!

Types of Analysis

- Worst-Case Analysis
  - What's the *worst* possible runtime for the algorithm?
  - Useful for "sleeping well at night."
- Best-Case Analysis
  - What's the *best* possible runtime for the algorithm?
  - Useful to see if the algorithm performs well in some cases.
- Average-Case Analysis
  - What's the *average* runtime for the algorithm?
  - Far beyond the scope of this class; take CS109, CS161, CS365, or CS369N for more information!

Worst Case Analysis

```
bool LinearSearch(string& str, char ch) {
  for (int i = 0; i < str.length(); i++)
    if (str[i] == ch)
      return true;

  return false;
}
```

- Assume that "ch" is the *worst* possible location for this algorithm
  - In this case, "ch" is not in str

O(n)

Determining if a Character is a Letter

Determining if a Character is a Letter

```
bool isAlpha(char ch) {
    return (ch >= 'A' && ch <= 'Z') ||
           (ch >= 'a' && ch <= 'z');
}
```

Determining if a Character is a Letter

```
bool isAlpha(char ch) {
   return (ch >= 'A' && ch <= 'Z') ||
          (ch >= 'a' && ch <= 'z');
}
```

O(1)

What Can Big-O Tell Us?

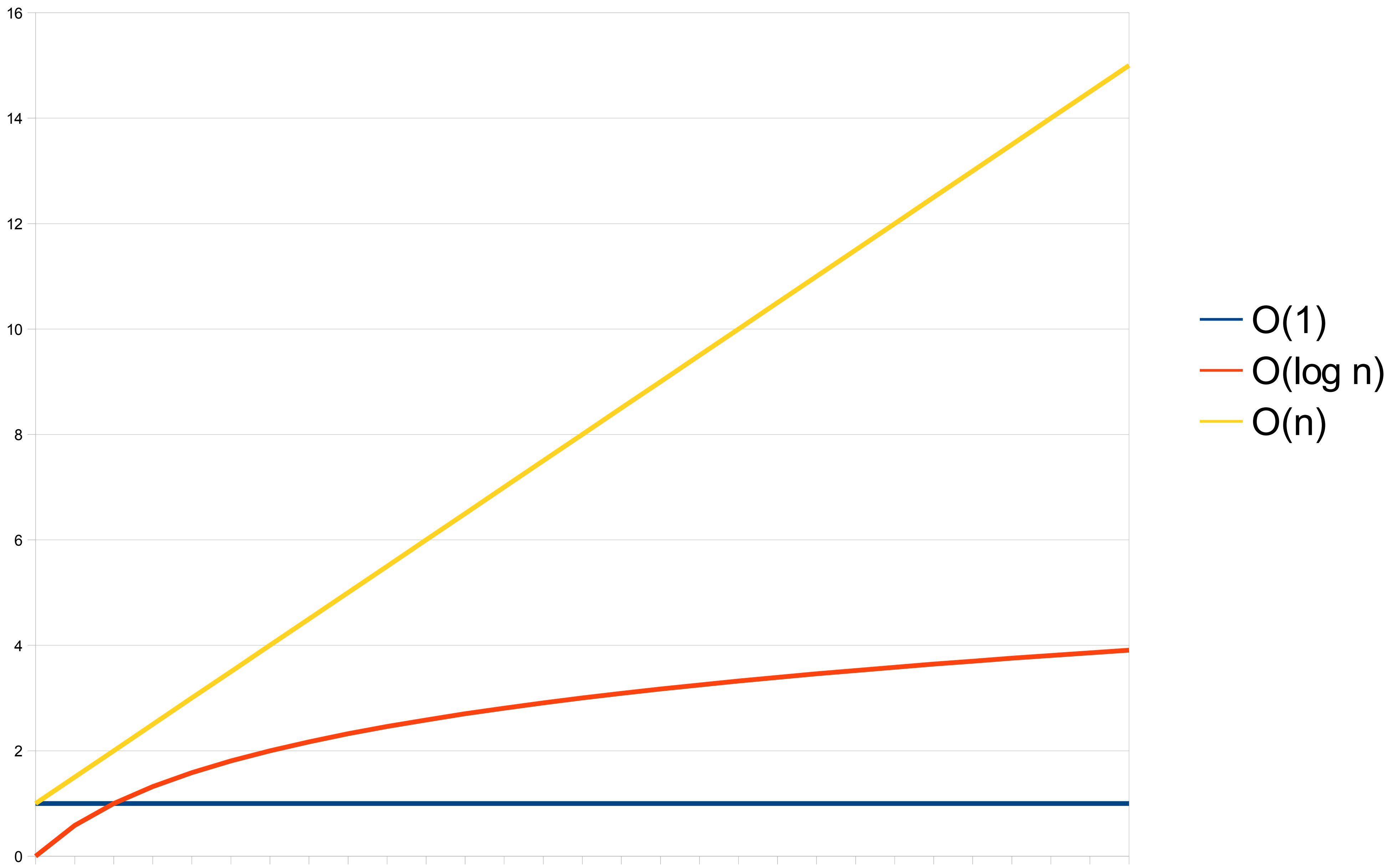
- Long-term behavior of a function.
  - If algorithm A is O(n) and algorithm B is $O(n^2)$, **for large inputs** algorithm A will always be faster.
  - If algorithm A is O(n), **for large inputs**, doubling the size of the input roughly doubles the runtime.
    - In other words, Big-O tells us how the running time of an algorithm grows as the size of its input grows

What “large” means on the terms we dropped!

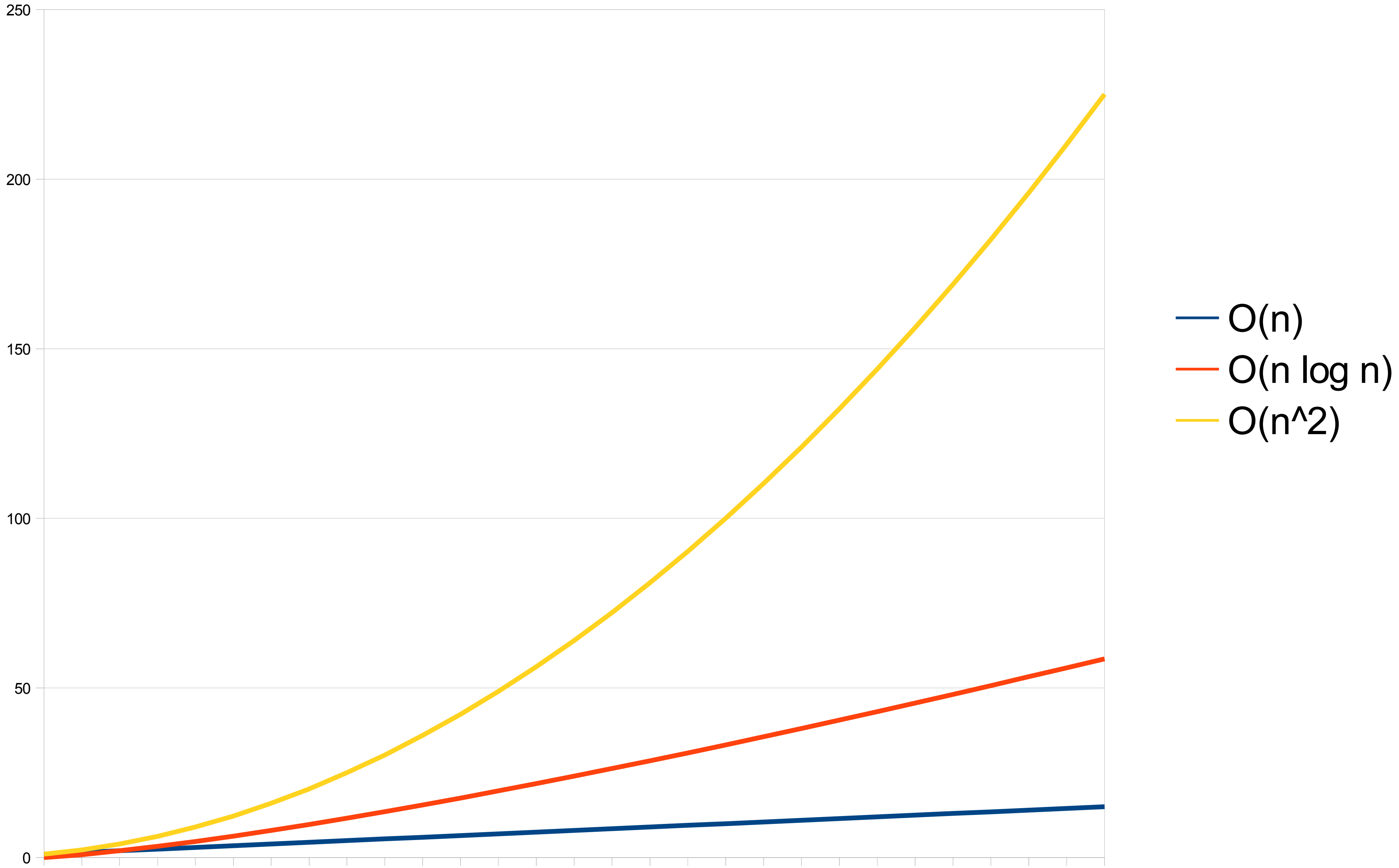
What *Can't* Big-O Tell Us?

- The actual runtime of a function.
  - $10^{100}n = O(n)$
  - $10^{-100}n = O(n)$
- How a function behaves on small inputs.
  - $n^3 = O(n^3)$
  - $10^6 = O(1)$

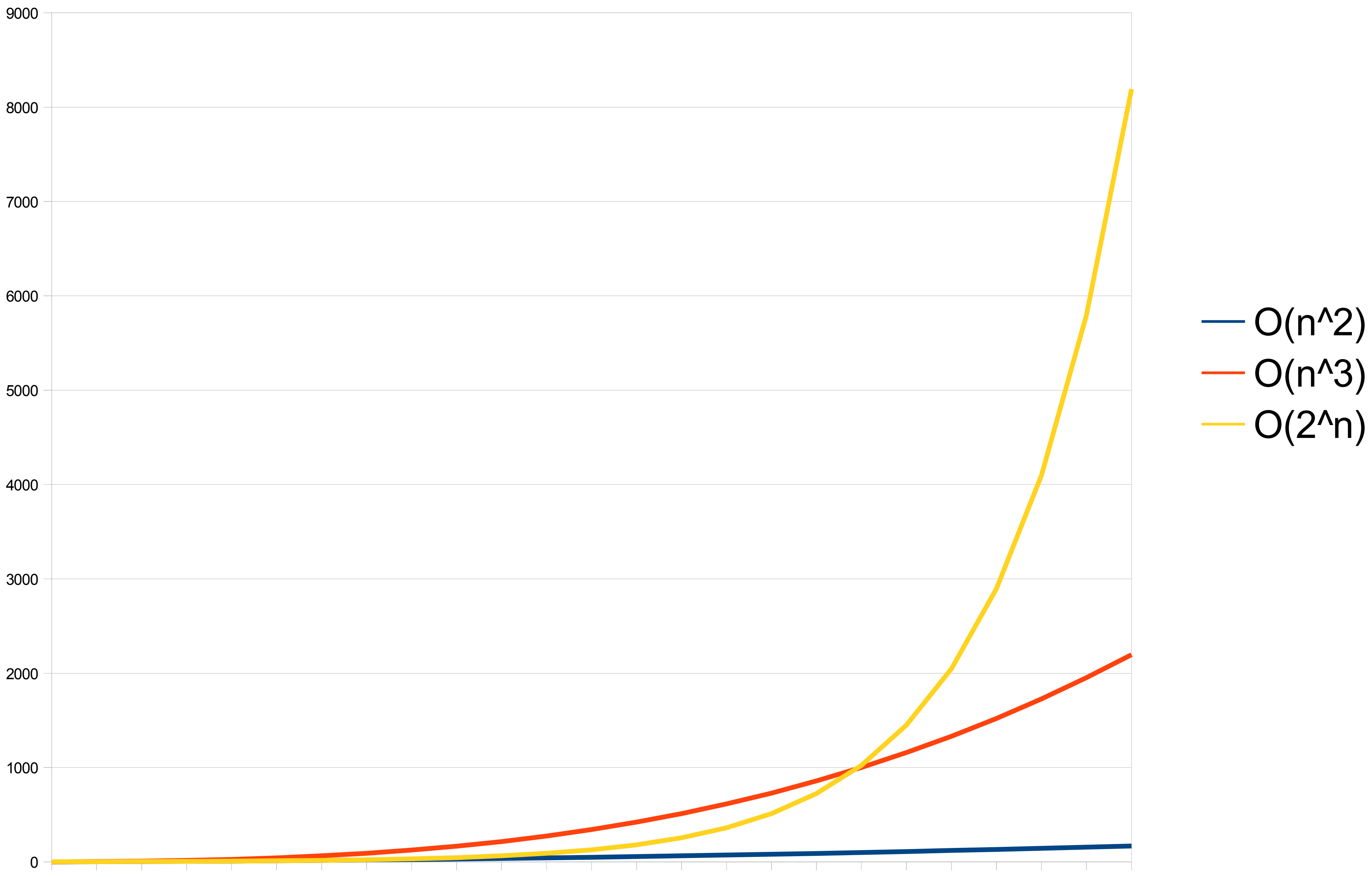
Growth Rates, Part One



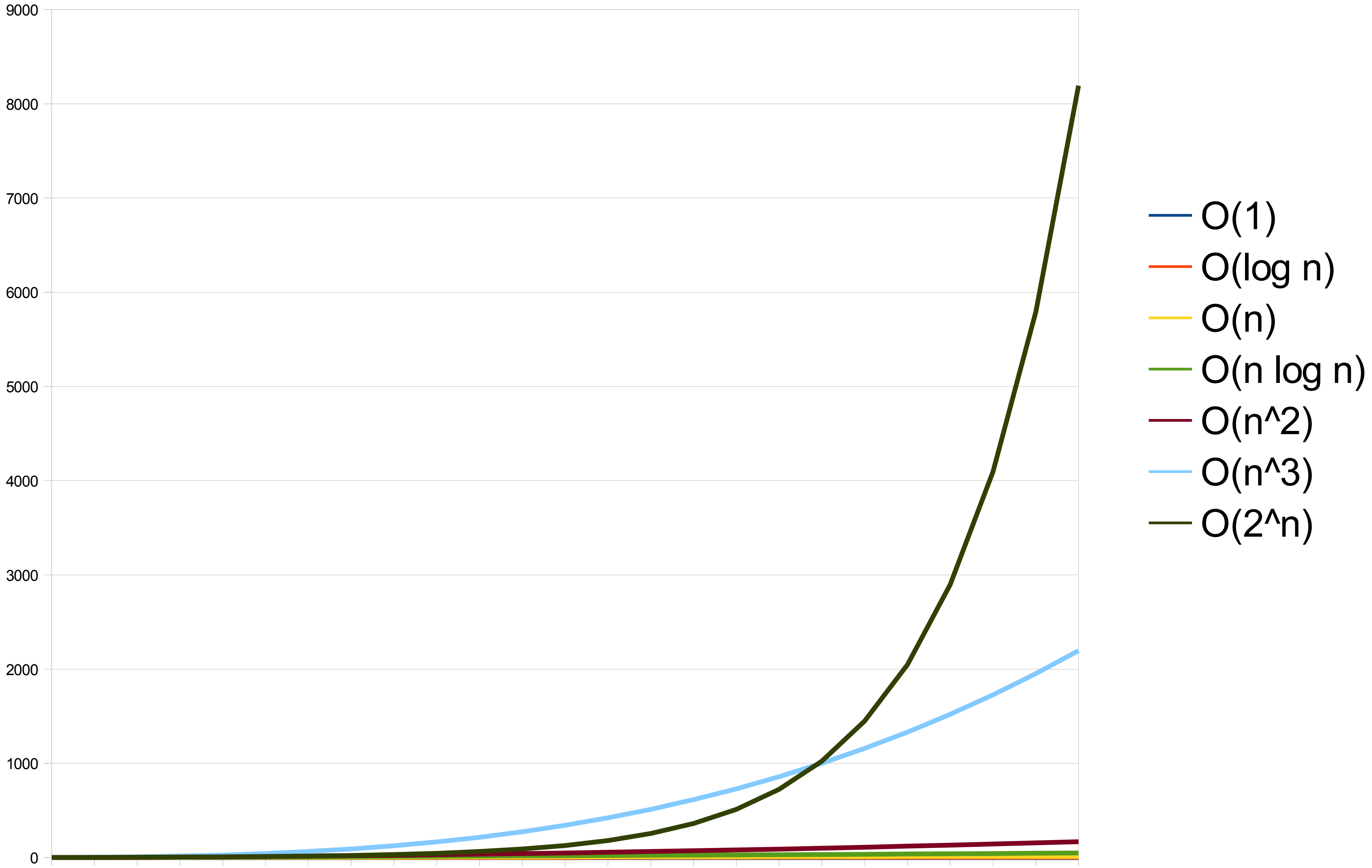
Growth Rates, Part Two



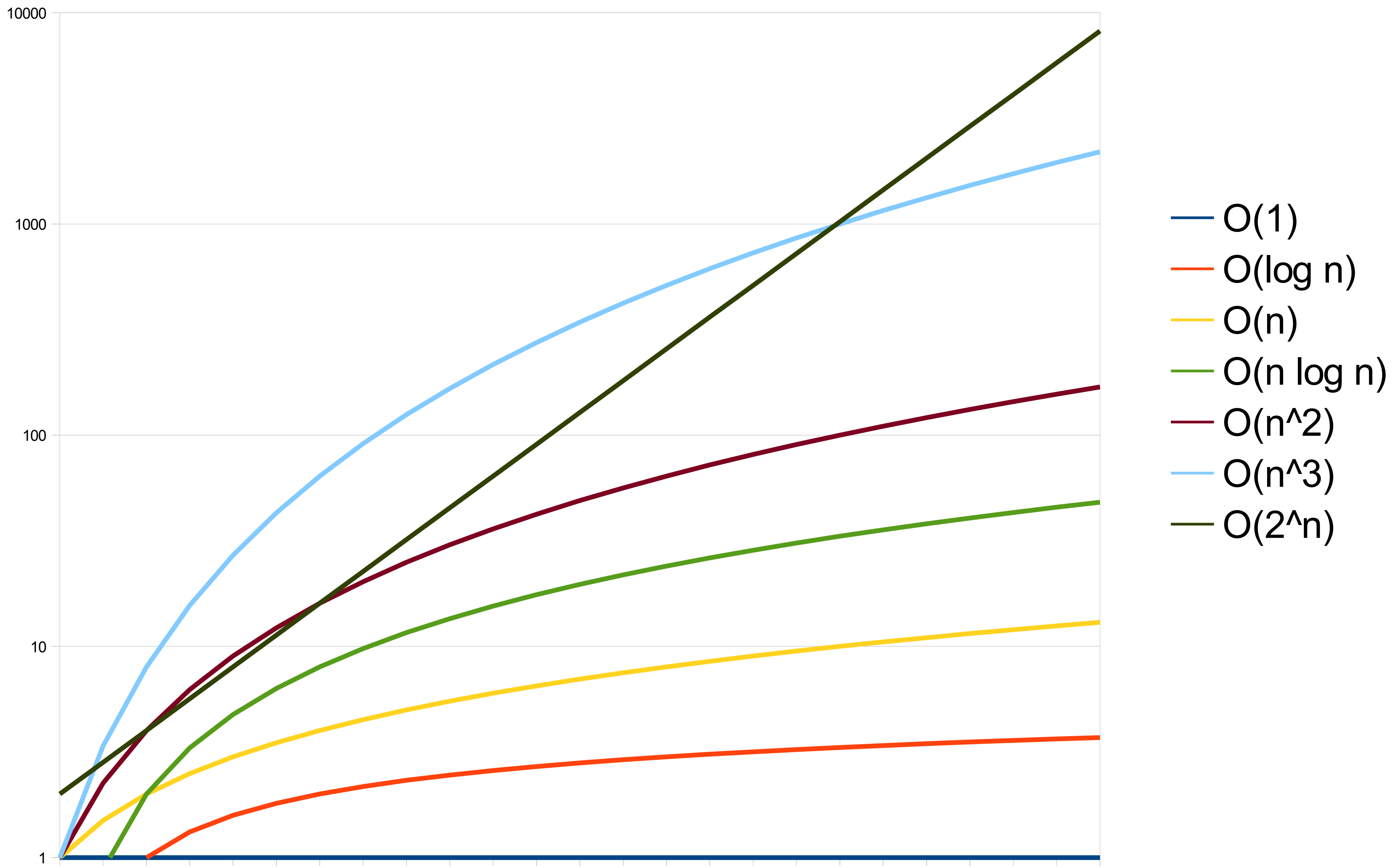
Growth Rates, Part Three



To Give You A Better Sense...



Once More with Logarithms



Comparison of Runtimes

(1 operation = 1 microsecond)

Size	1	lg n	n	n log n	n^2	n^3
100	1μs	7μs	100μs	0.7ms	10ms	<1min
200	1μs	8μs	200μs	1.5ms	40ms	<1min
300	1μs	8μs	300μs	2.5ms	90ms	1min
400	1μs	9μs	400μs	3.5ms	160ms	2min
500	1μs	9μs	500μs	4.5ms	250ms	4min
600	1μs	9μs	600μs	5.5ms	360ms	6min
700	1μs	9μs	700μs	6.6ms	490ms	9min
800	1μs	10μs	800μs	7.7ms	640ms	12min
900	1μs	10μs	900μs	8.8ms	810ms	17min
1000	1μs	10μs	1000μs	10ms	1000ms	22min
1100	1μs	10μs	1100μs	11ms	1200ms	29min
1200	1μs	10μs	1200μs	12ms	1400ms	37min
1300	1μs	10μs	1300μs	13ms	1700ms	45min
1400	1μs	10μs	1400μs	15ms	2000ms	56min

Summary of Big-O

- A means of describing the growth rate of a function.
- Ignores all but the leading term.
- Ignores constants.
- Allows for quantitative ranking of algorithms.
- Allows for quantiative reasoning about algorithms.

Sorting Algorithms

The Sorting Problem

- Given a list of elements, sort those elements in ascending order.
- There are ***many*** ways to solve this problem.
- What is the ***best*** way to solve this problem?
- We'll use big-O to find out!

The Sorting Problem

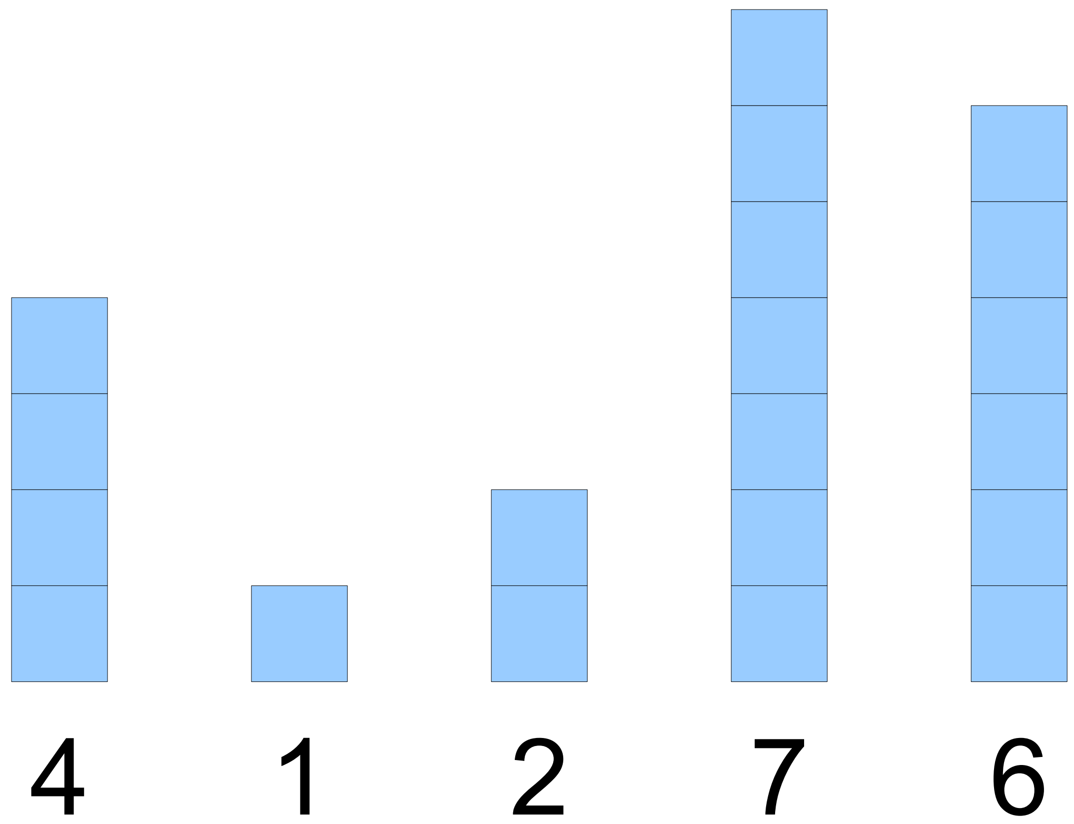
- Sorting is *extremely* important in Computer Science.
  - Searching through sorted data is much faster than searching through unsorted data due to **Binary Search**
    - It's okay if you haven't heard of Binary Search before, we'll cover it soon.
  - Many data structures in Computer Science are simply fancy ways of storing data in sorted order

The Sorting Problem

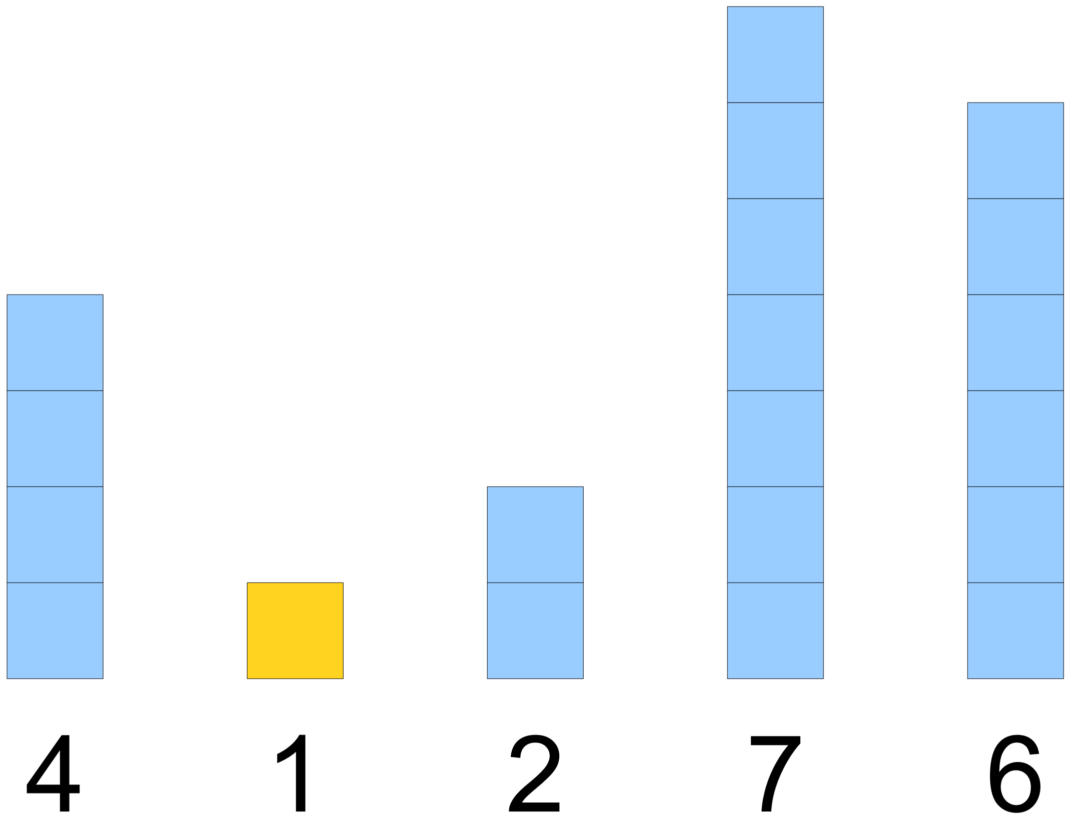
- Graphics: “Which objects can you see in a scene?”
- Scientific Simulation: “What particles are close enough to each other to exert some sort of force?”
- Machine Learning: “What training instance is this test instance most similar to?”

An Initial Idea: **Selection Sort**

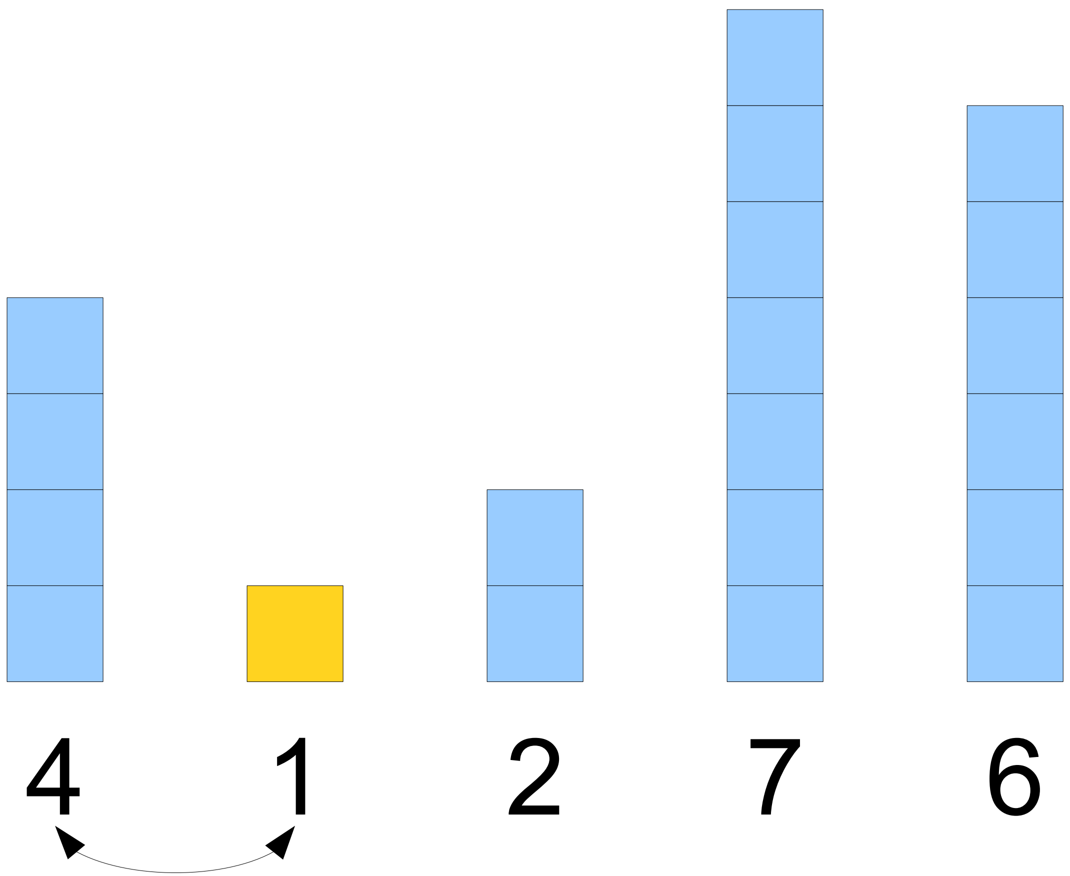
An Initial Idea: **Selection Sort**



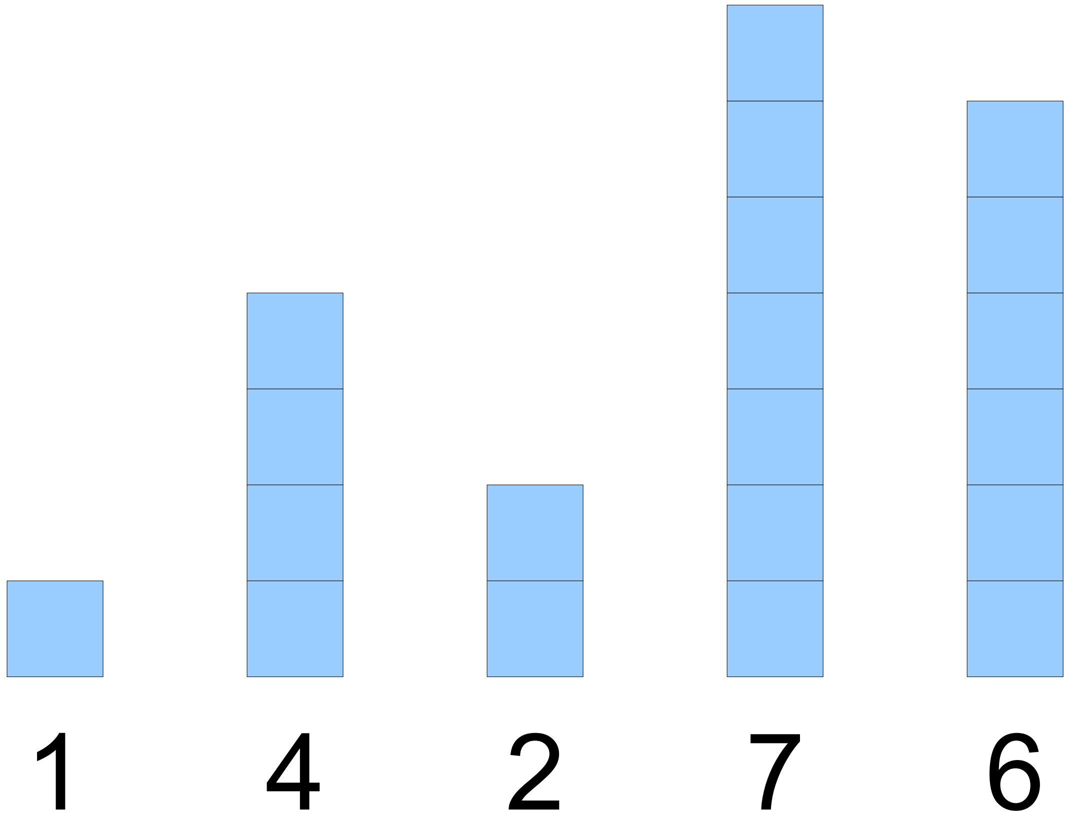
An Initial Idea: **Selection Sort**



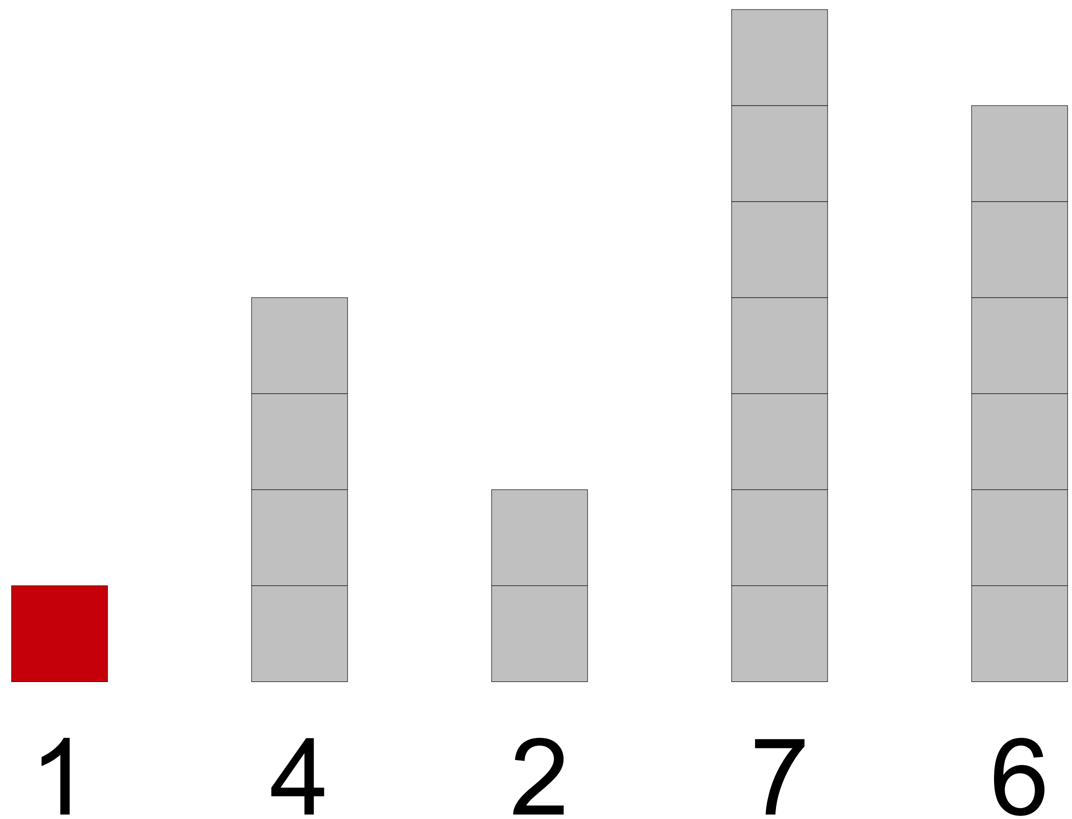
An Initial Idea: **Selection Sort**



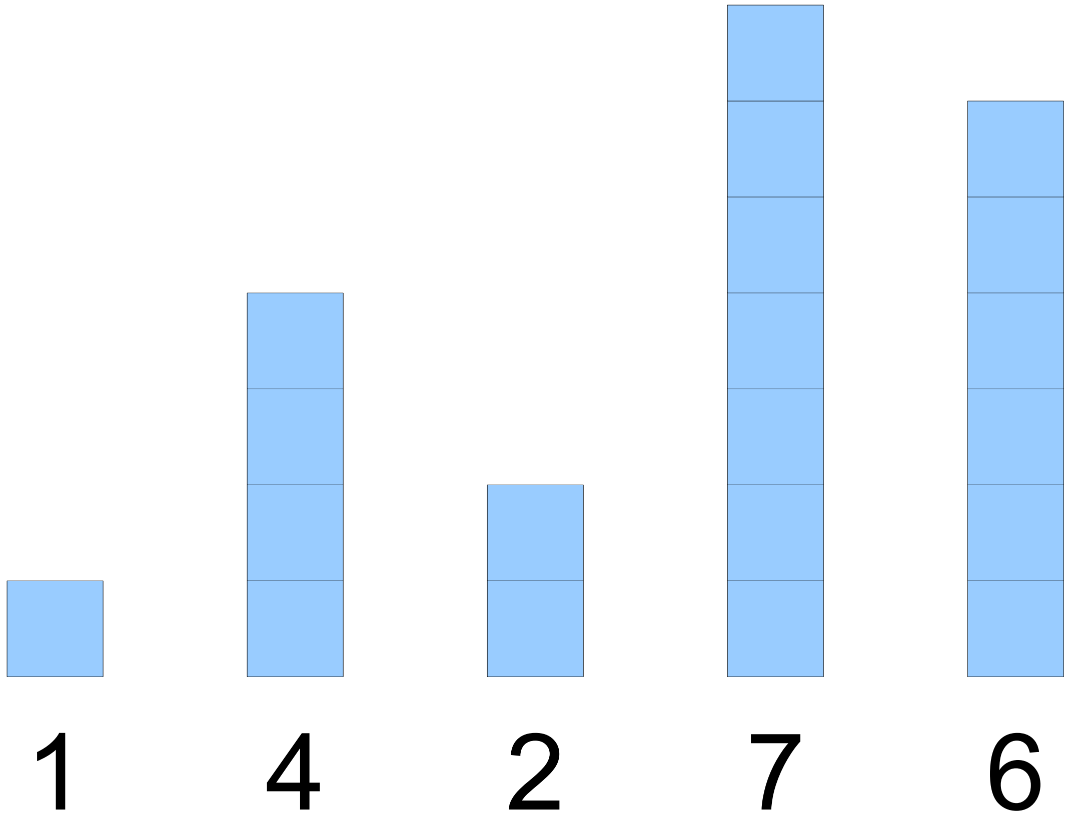
An Initial Idea: **Selection Sort**



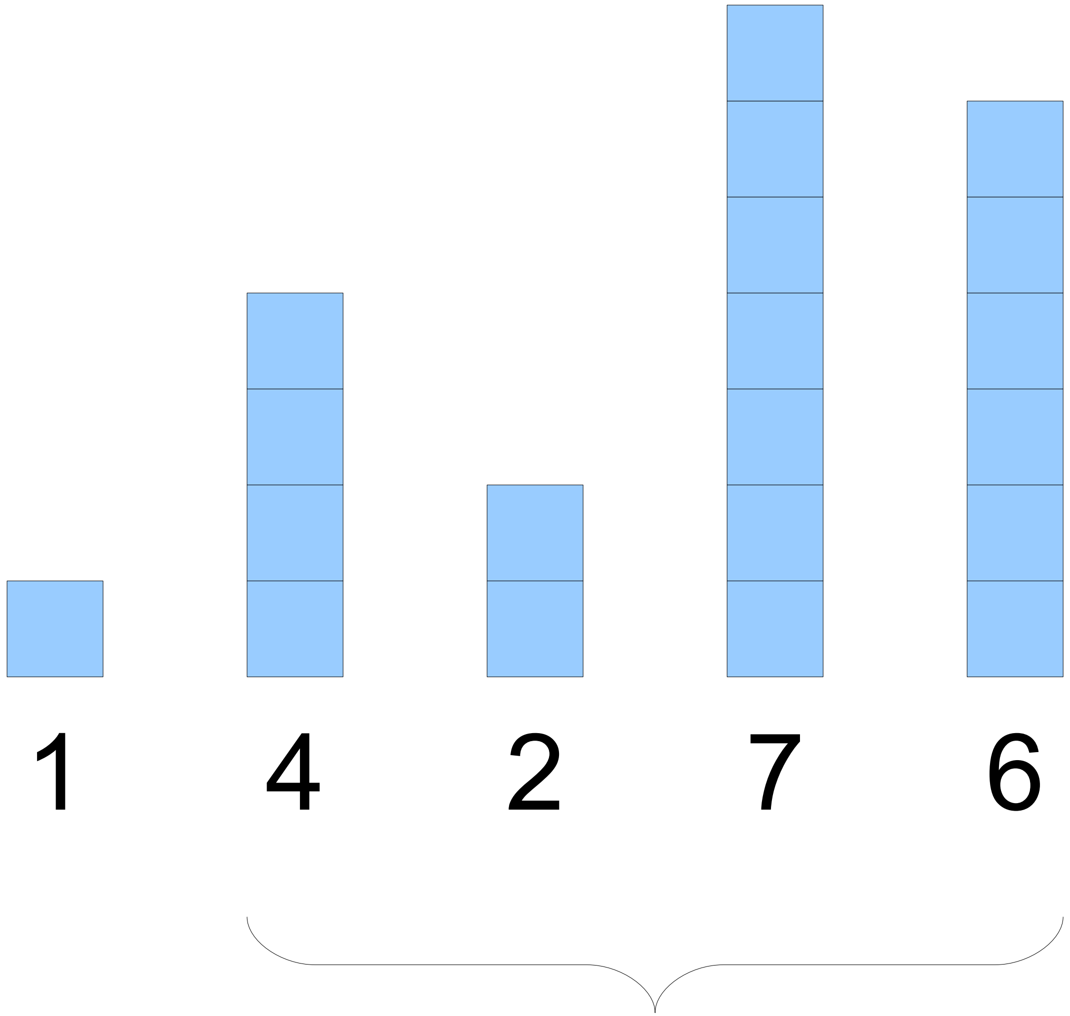
An Initial Idea: **Selection Sort**



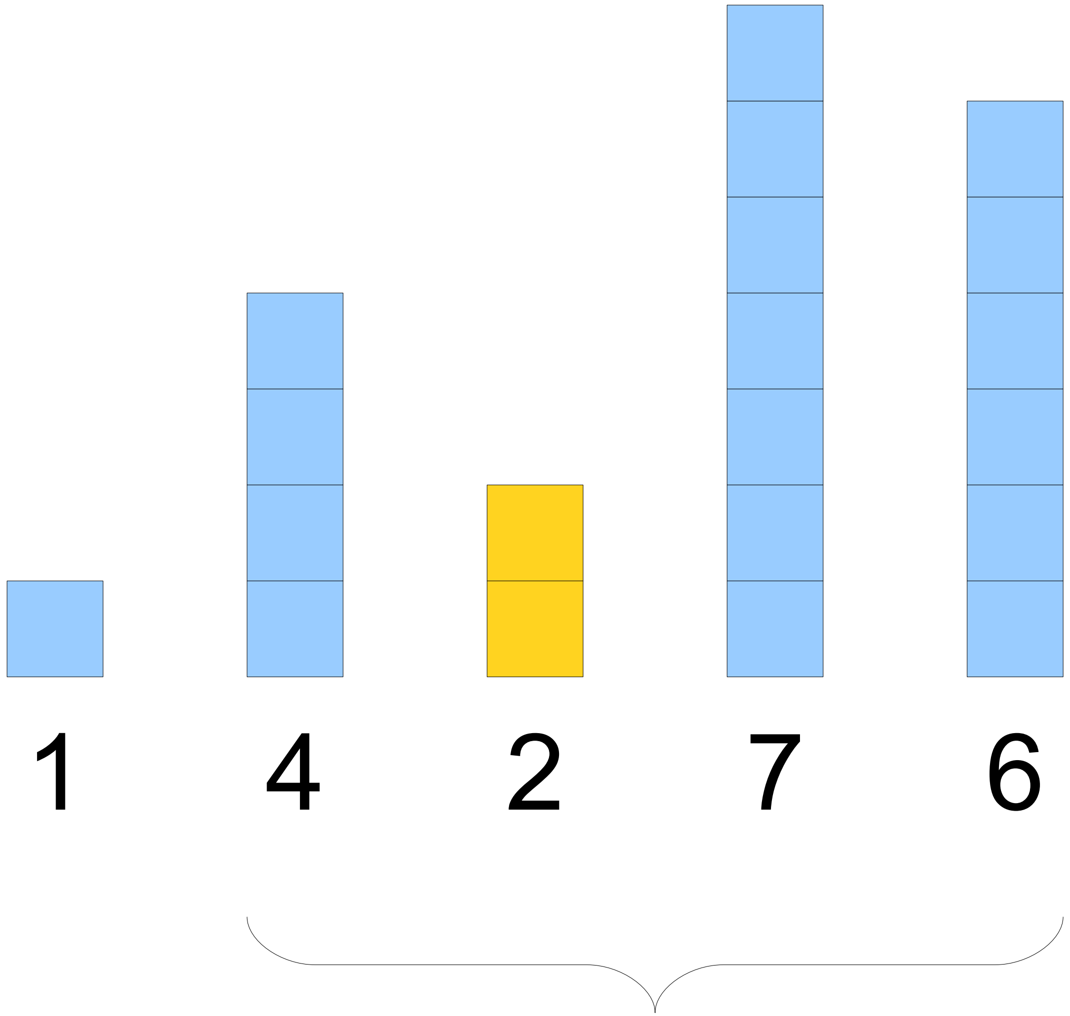
An Initial Idea: **Selection Sort**



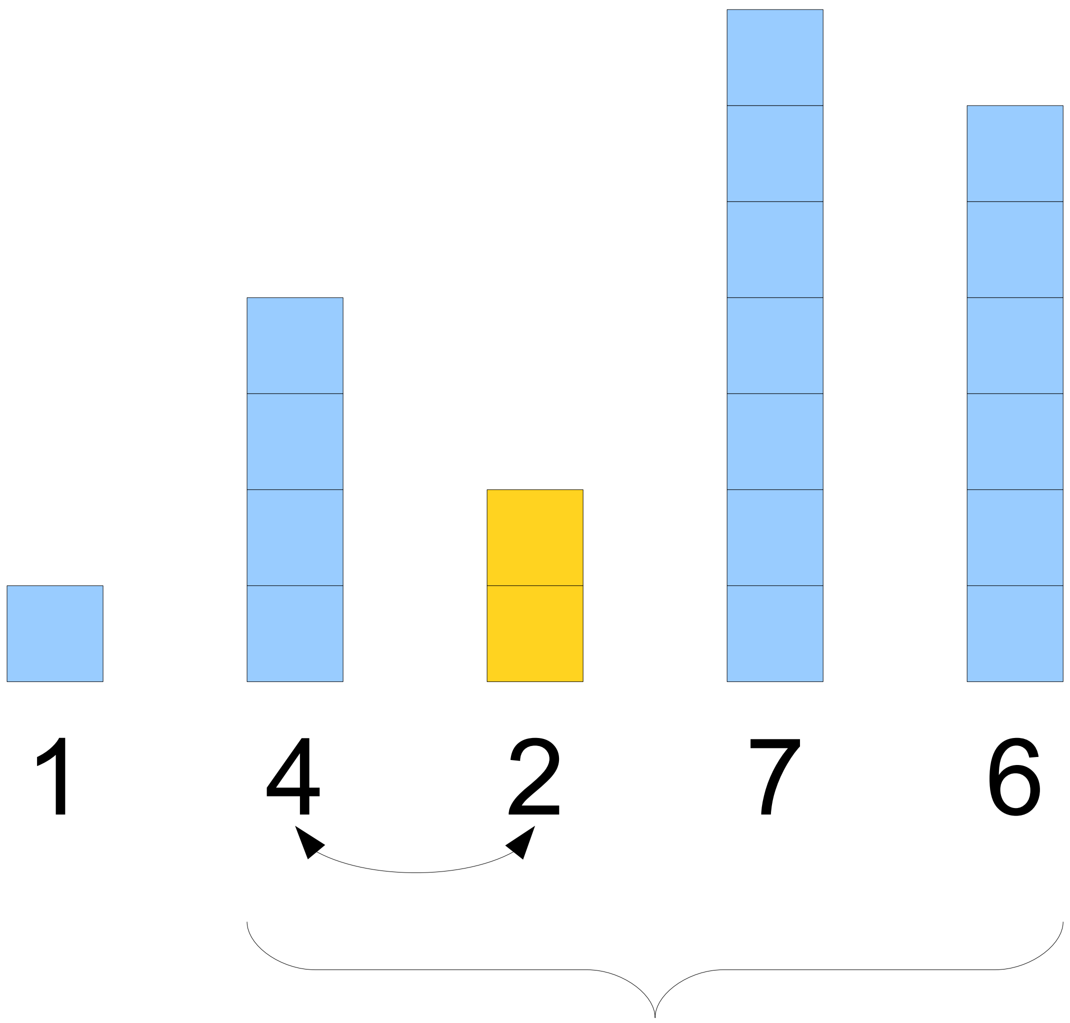
An Initial Idea: **Selection Sort**



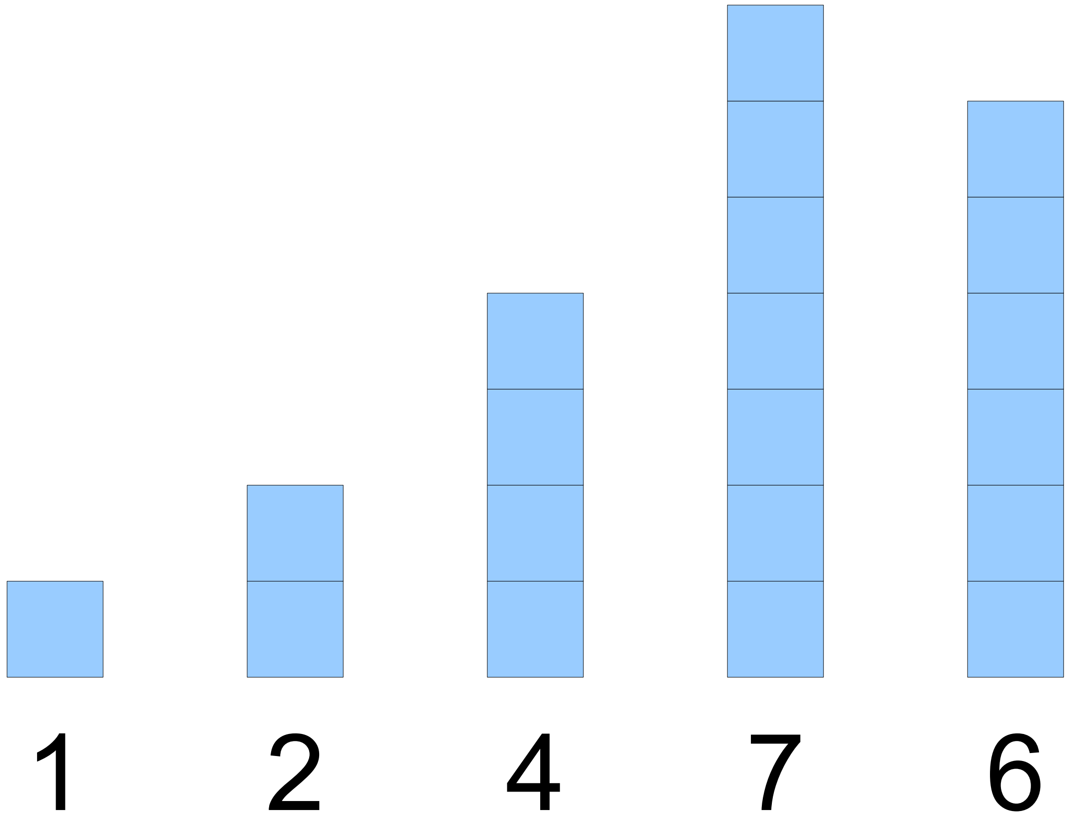
An Initial Idea: **Selection Sort**



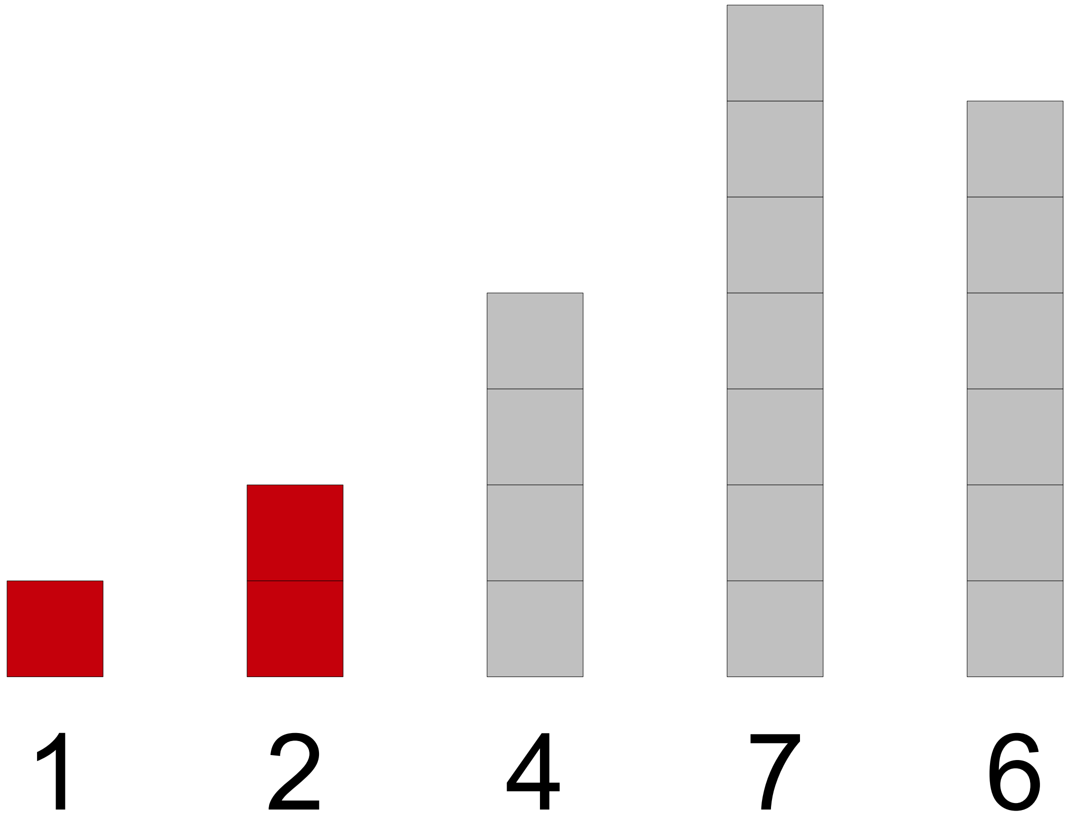
An Initial Idea: **Selection Sort**



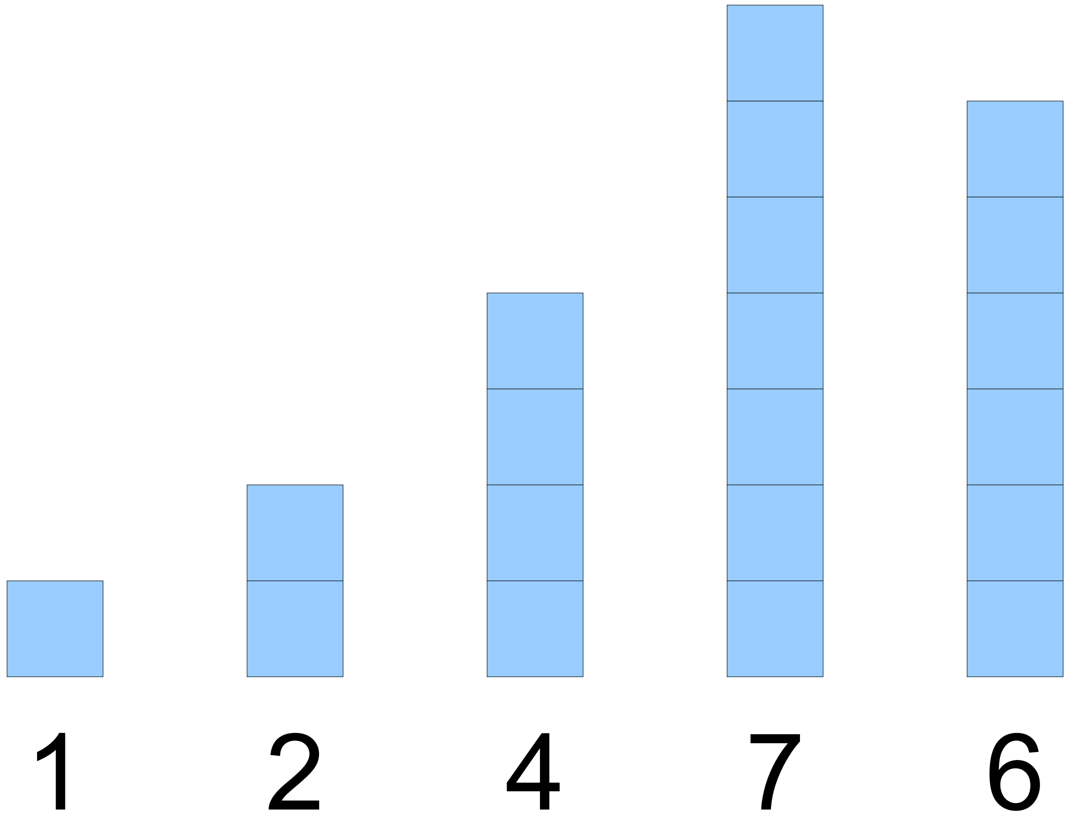
An Initial Idea: **Selection Sort**



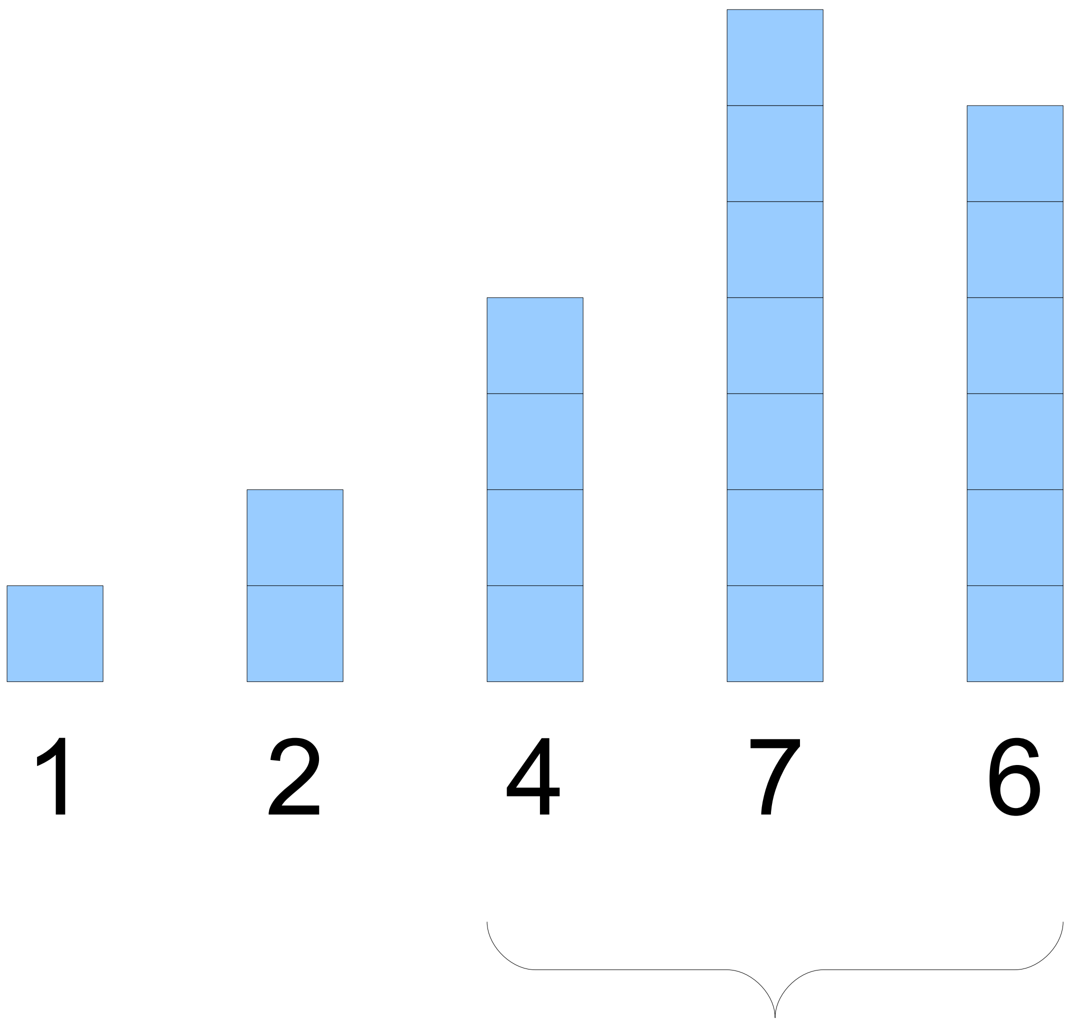
An Initial Idea: **Selection Sort**



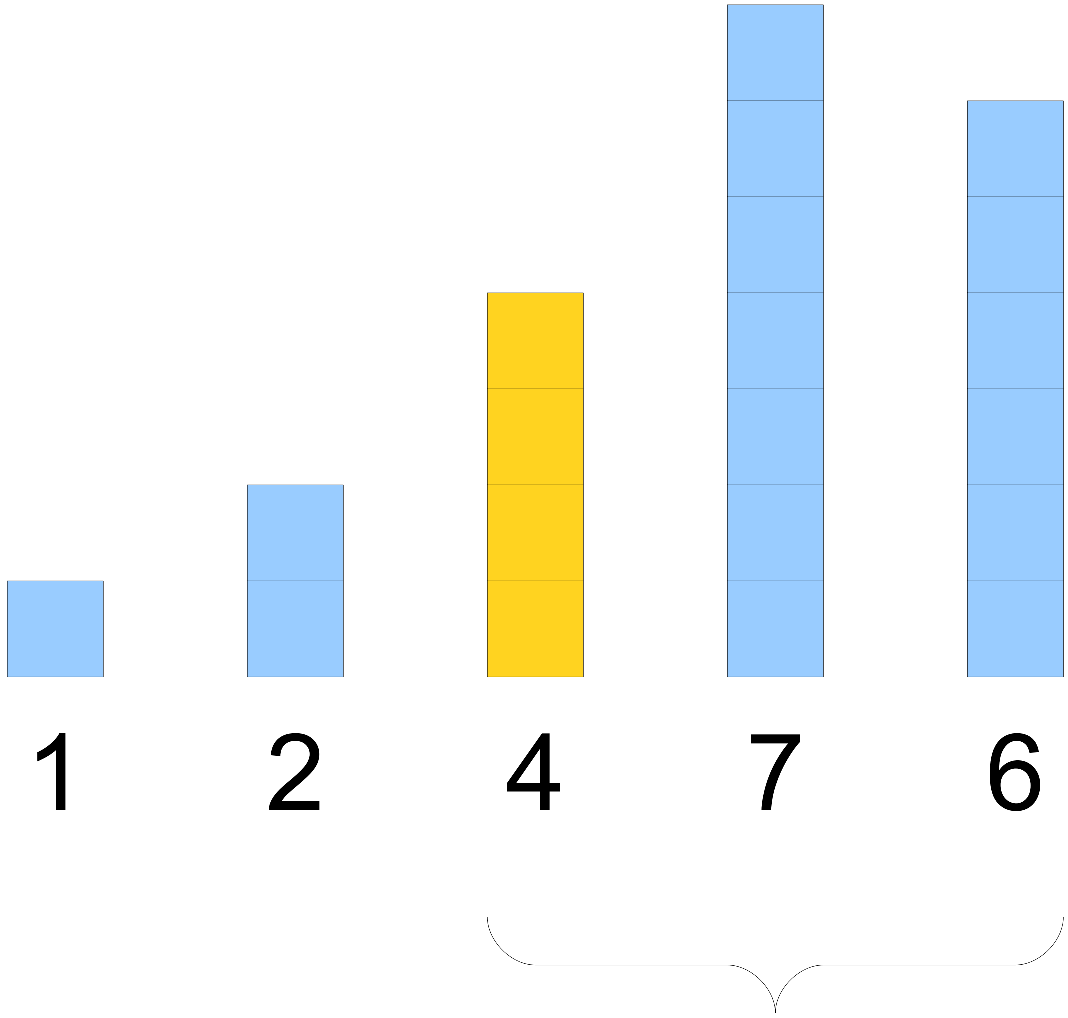
An Initial Idea: **Selection Sort**



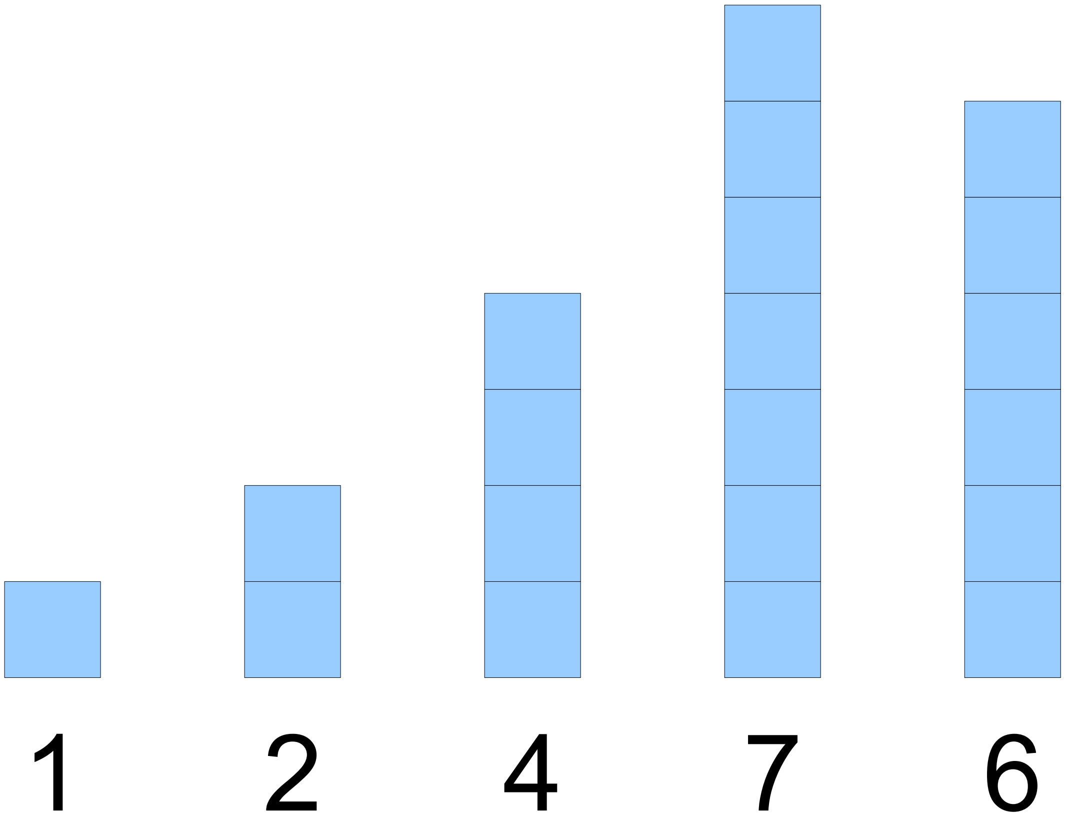
An Initial Idea: **Selection Sort**



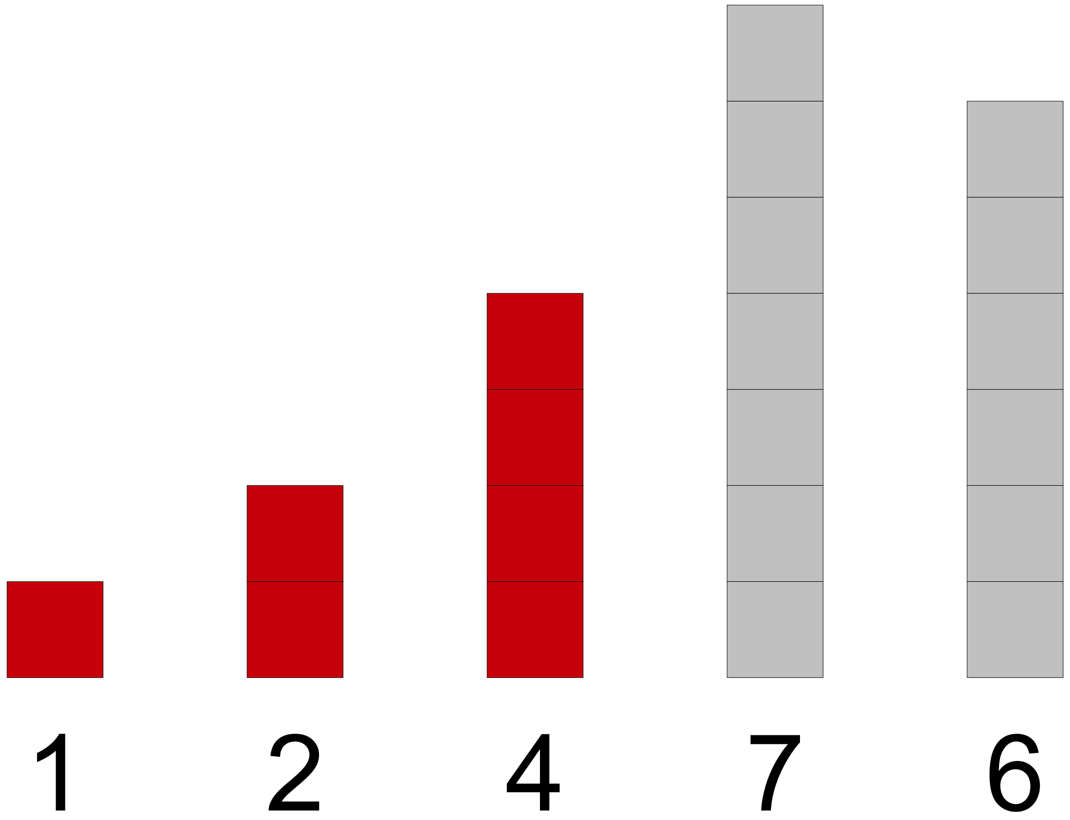
An Initial Idea: **Selection Sort**



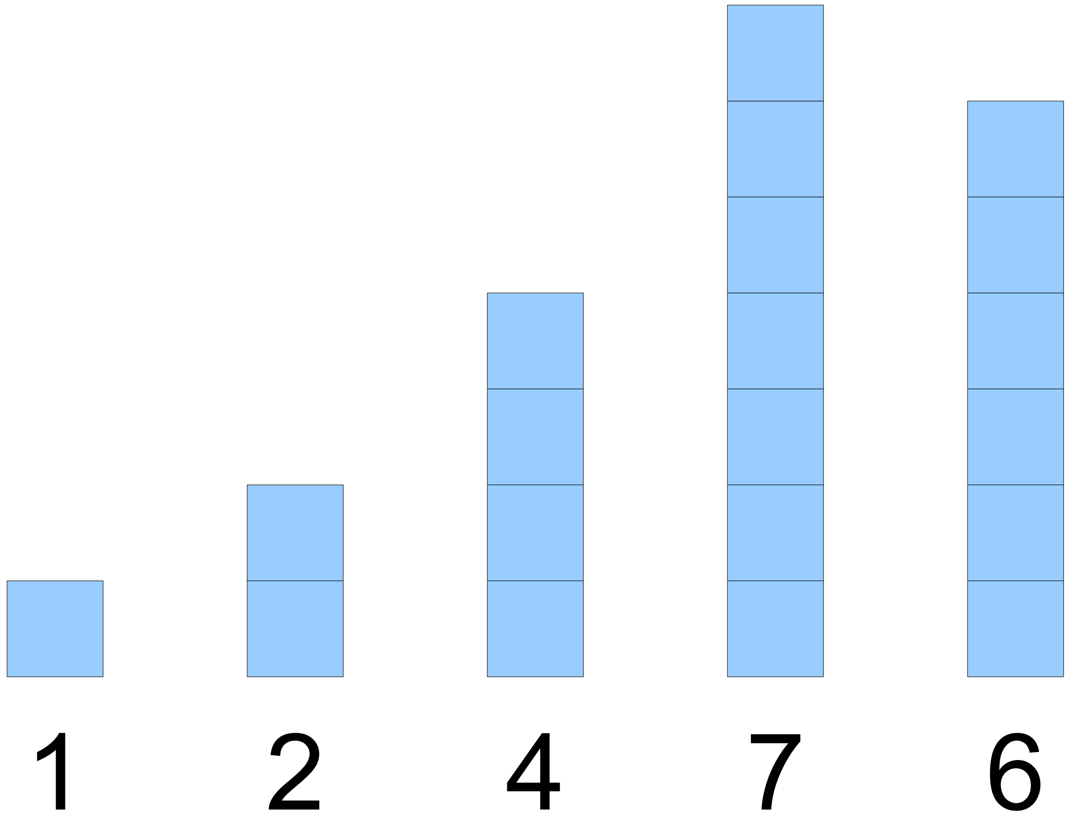
An Initial Idea: **Selection Sort**



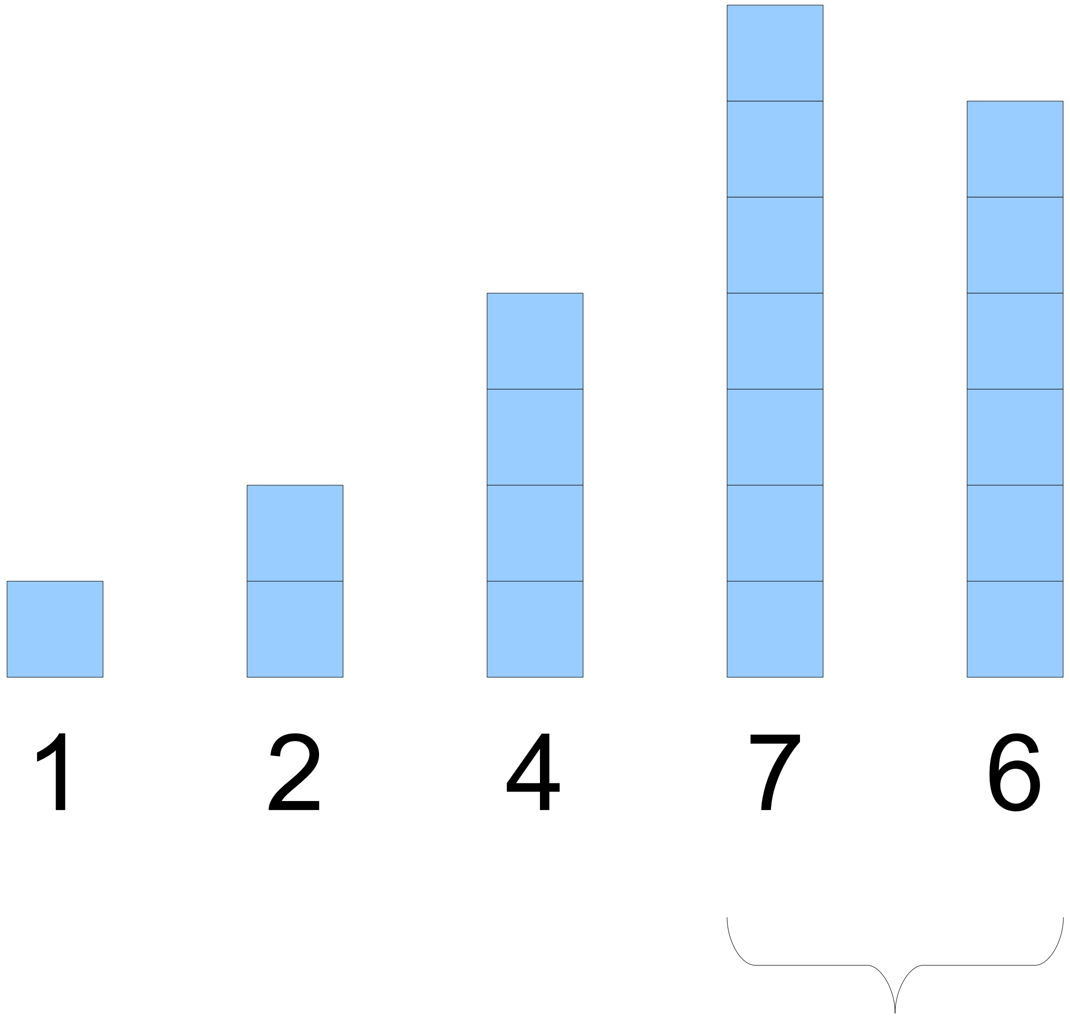
An Initial Idea: **Selection Sort**



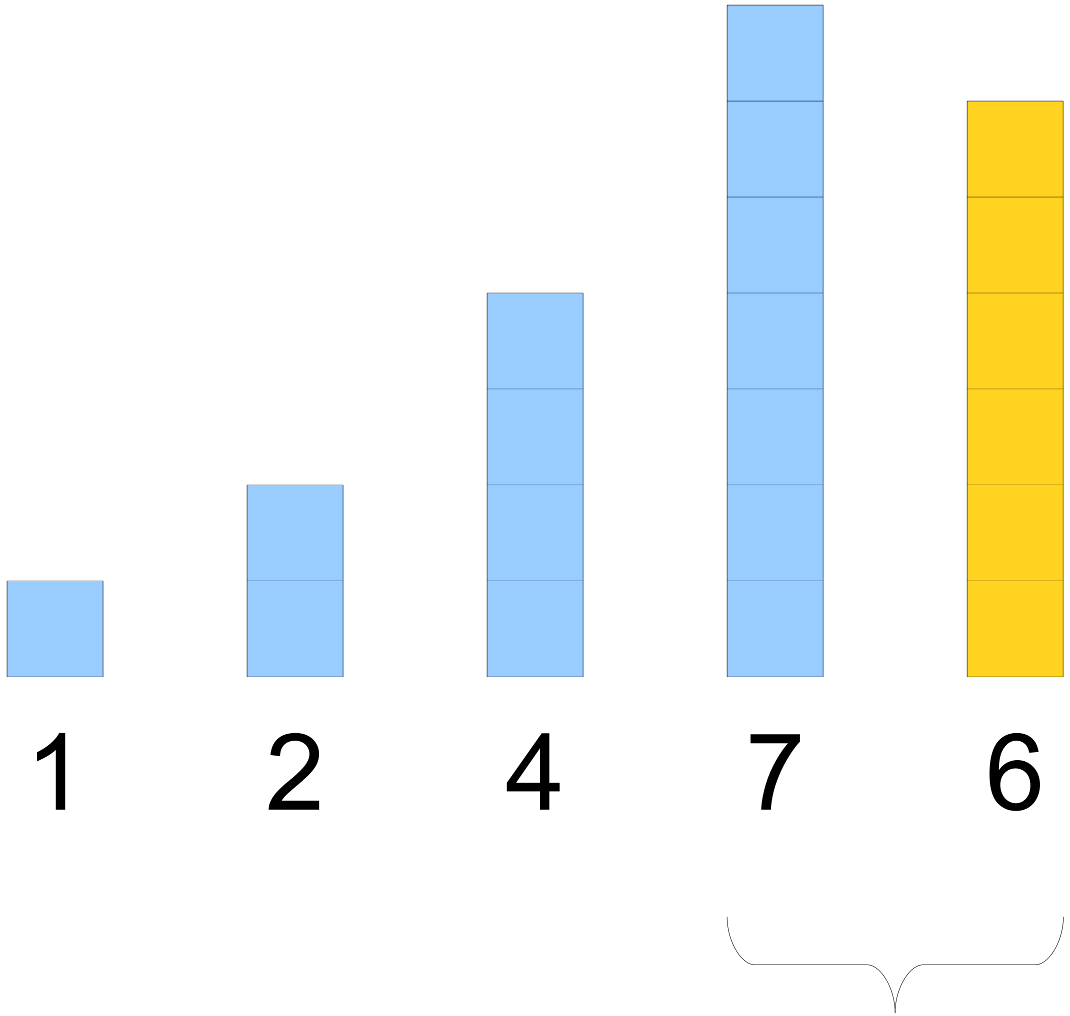
An Initial Idea: **Selection Sort**



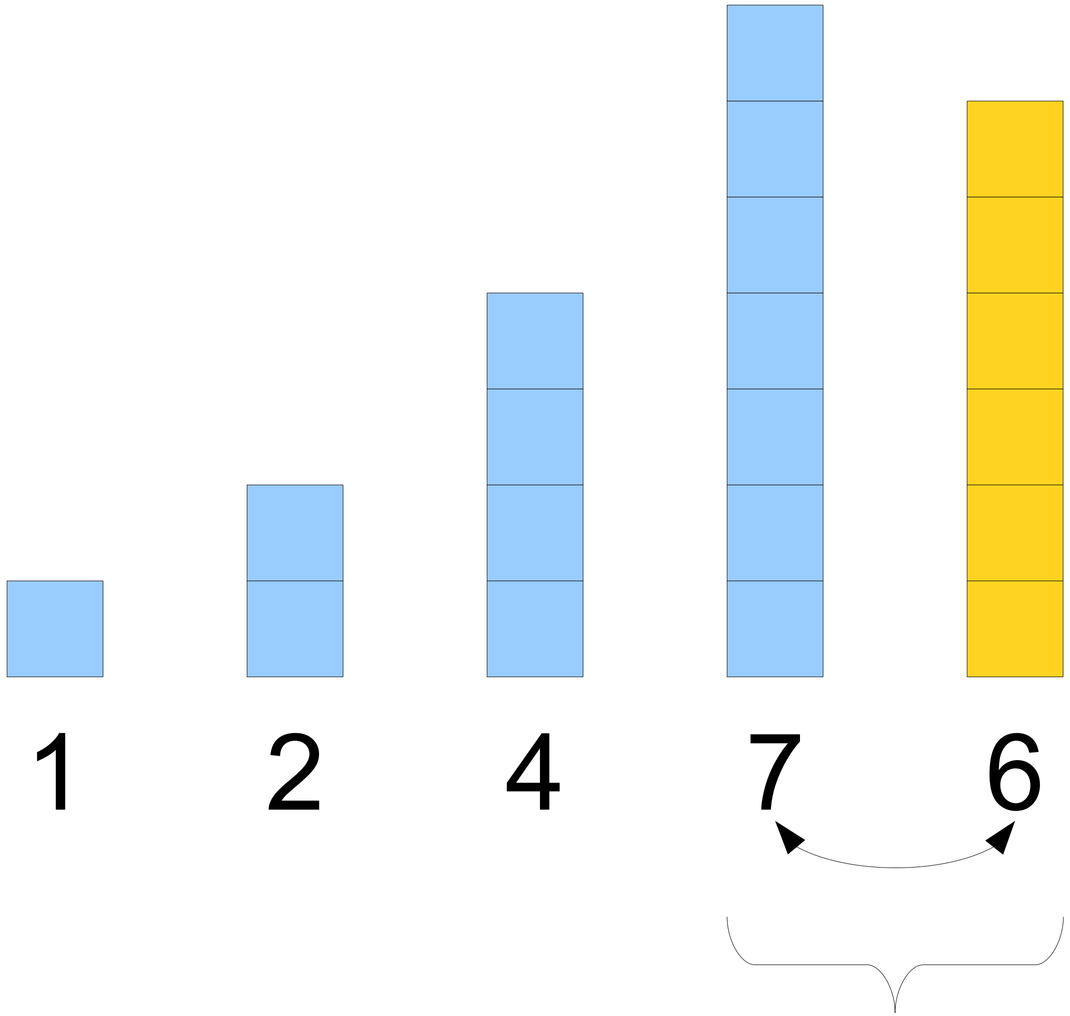
An Initial Idea: **Selection Sort**



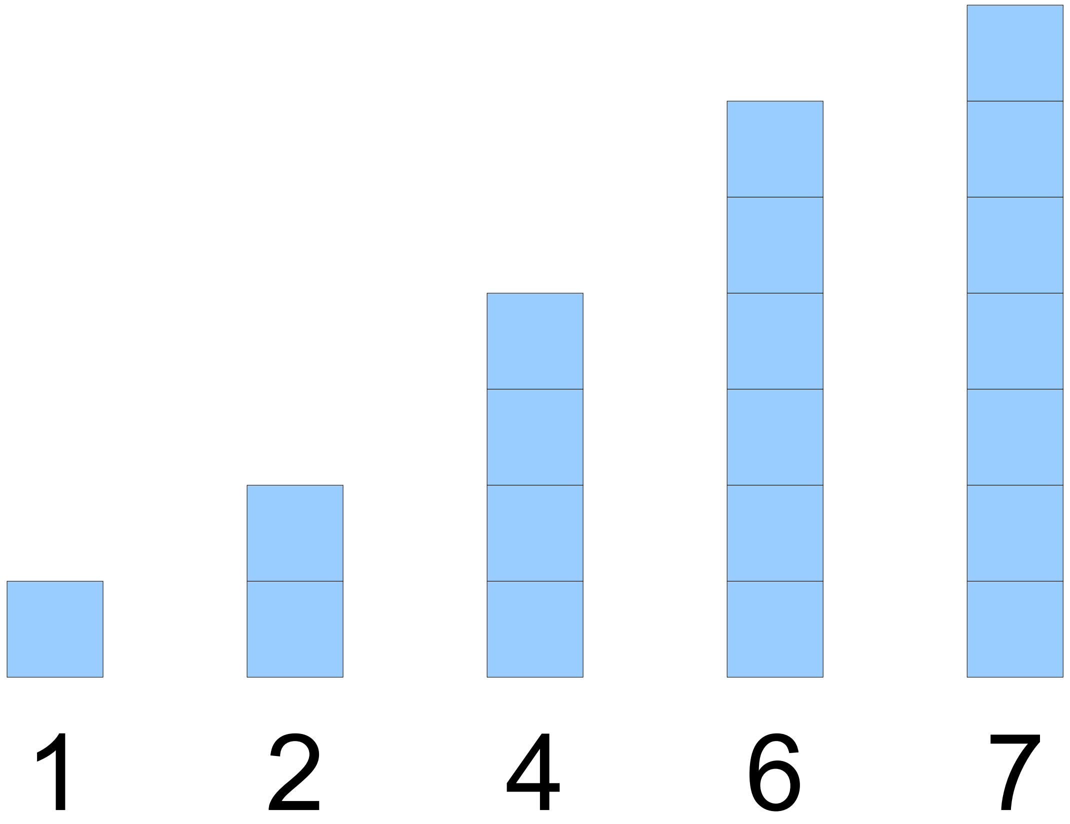
An Initial Idea: **Selection Sort**



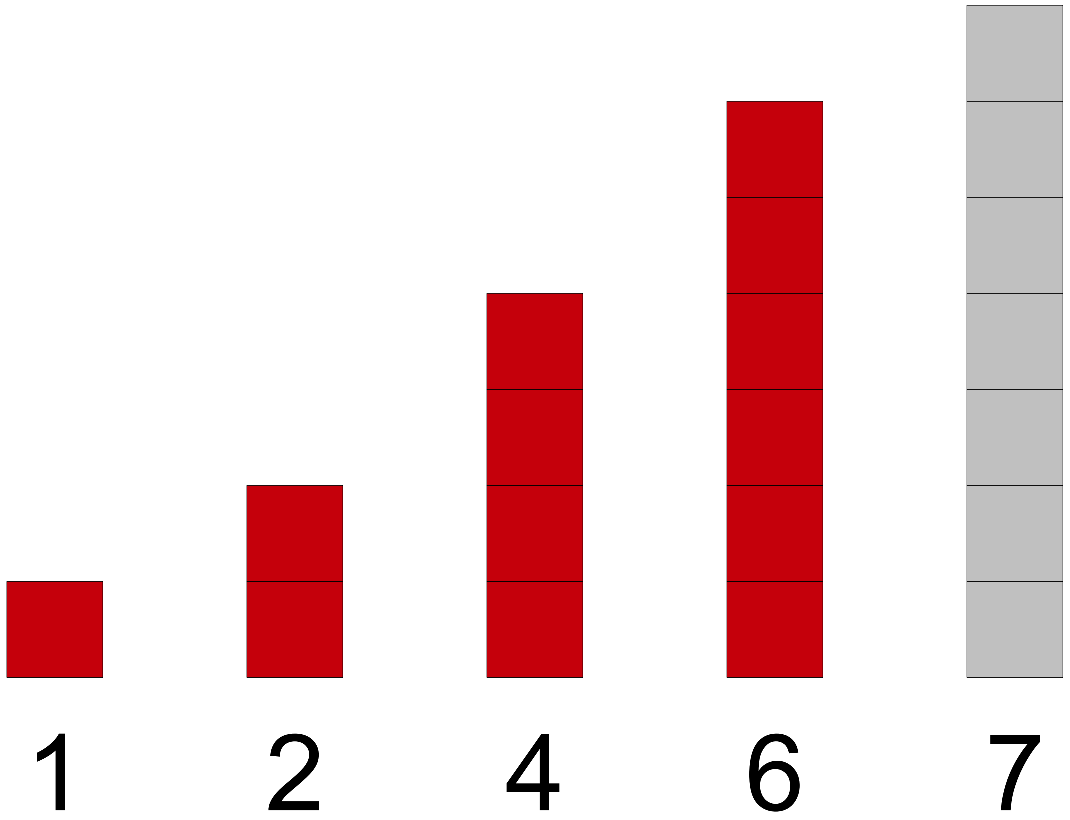
An Initial Idea: **Selection Sort**



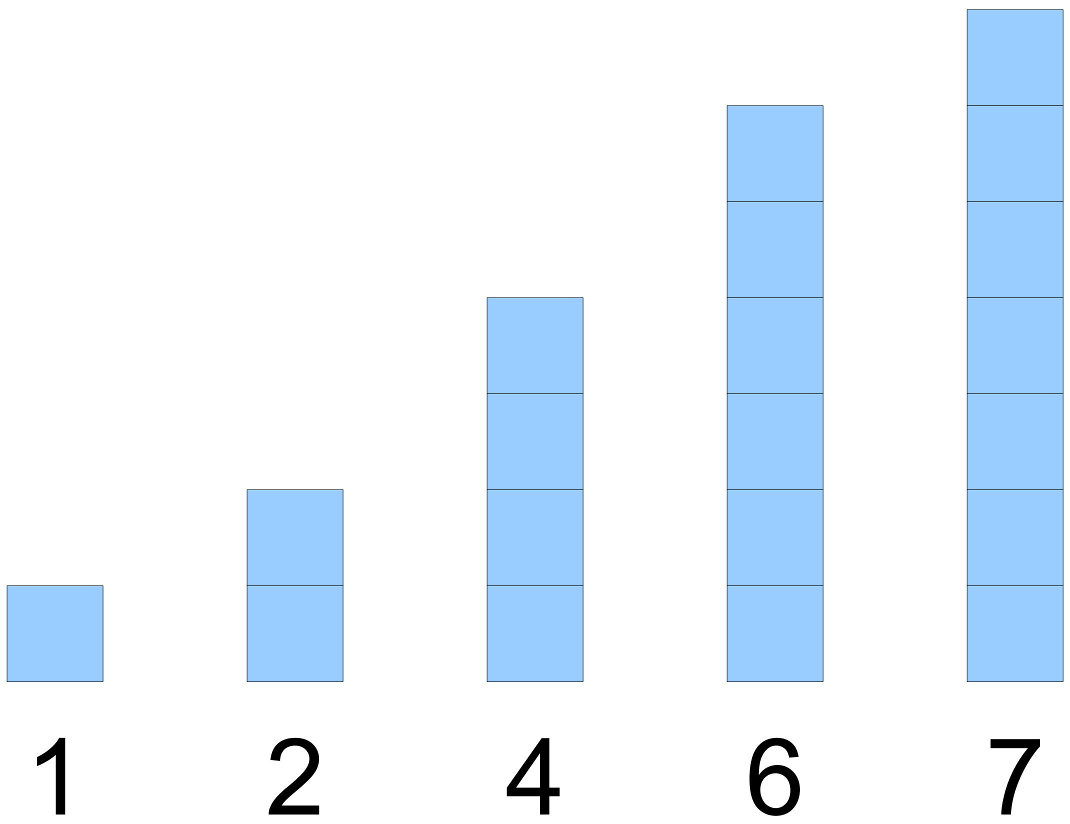
An Initial Idea: **Selection Sort**



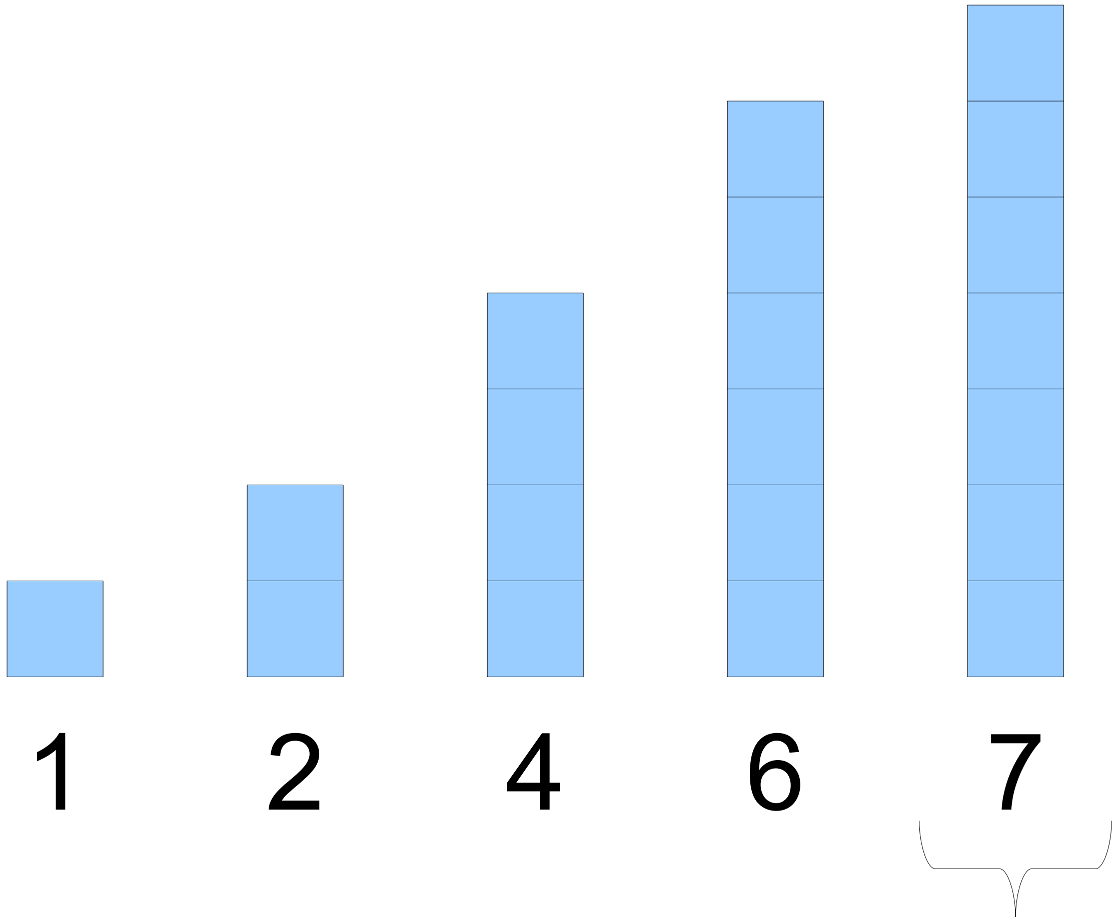
An Initial Idea: **Selection Sort**



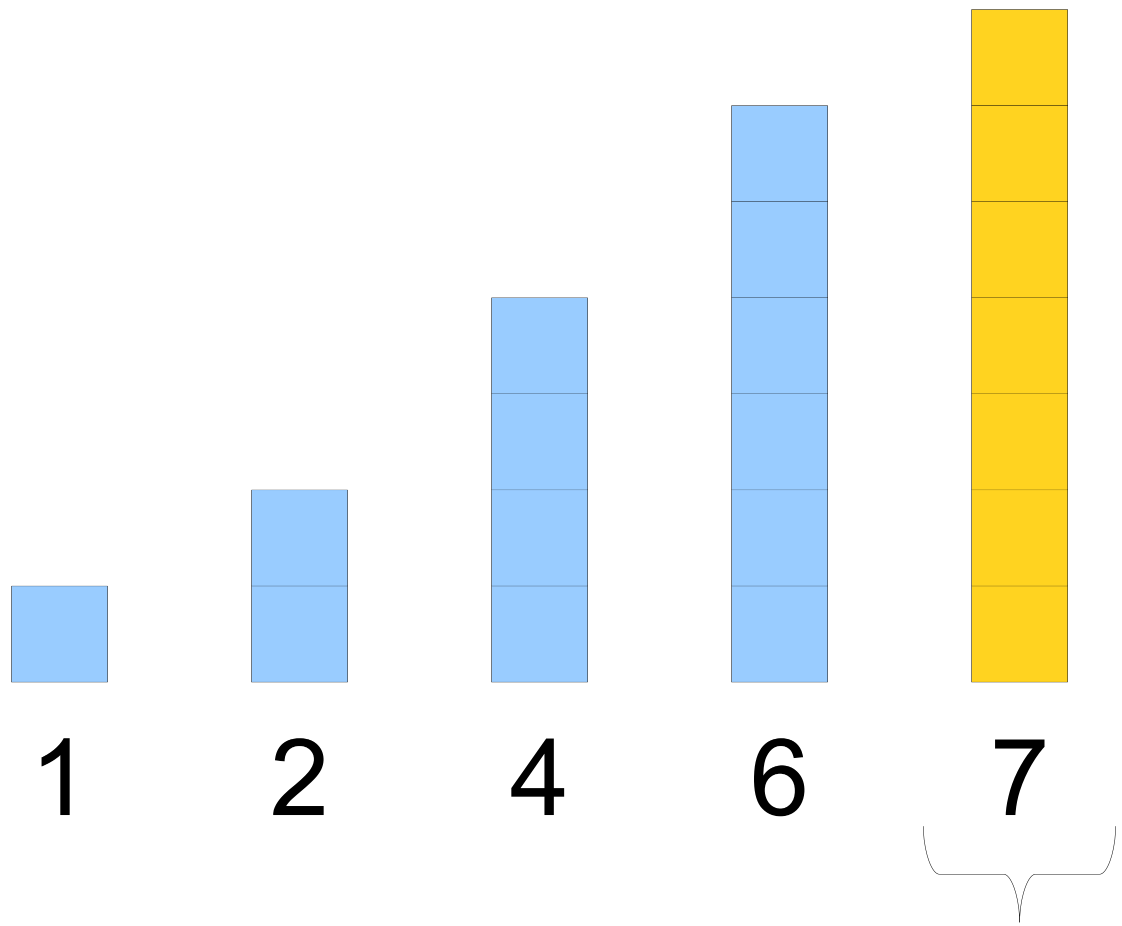
An Initial Idea: **Selection Sort**



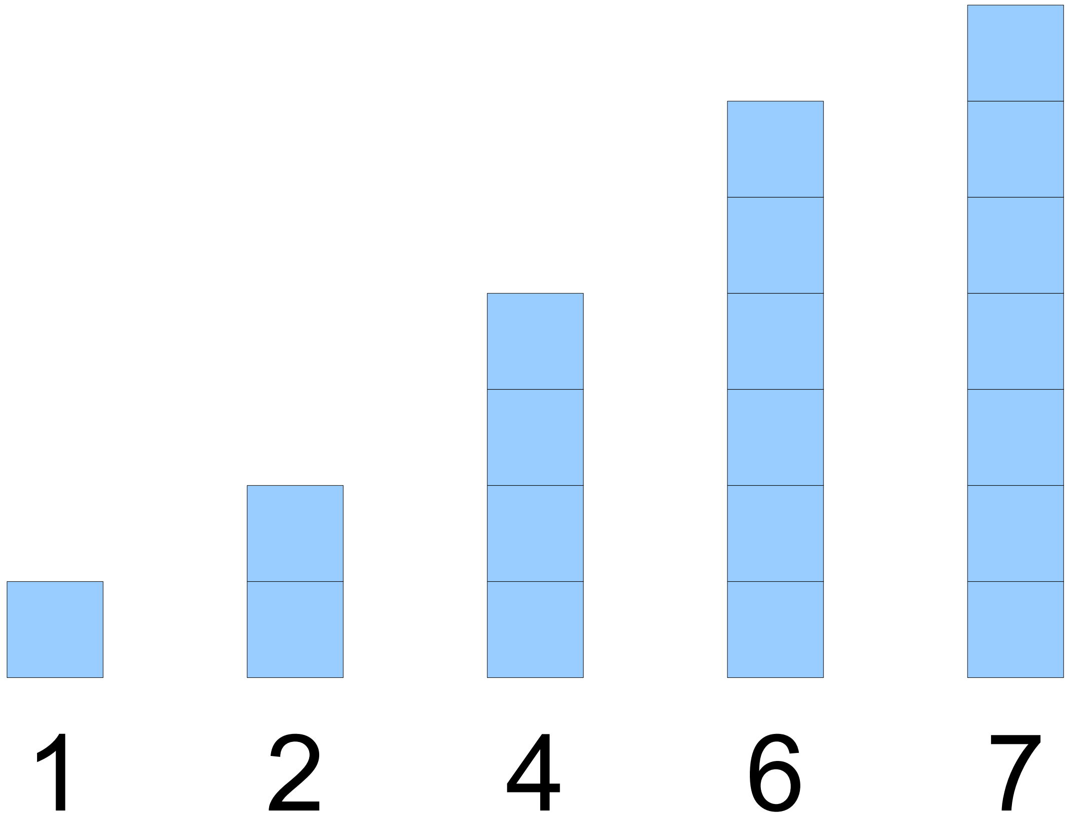
An Initial Idea: **Selection Sort**



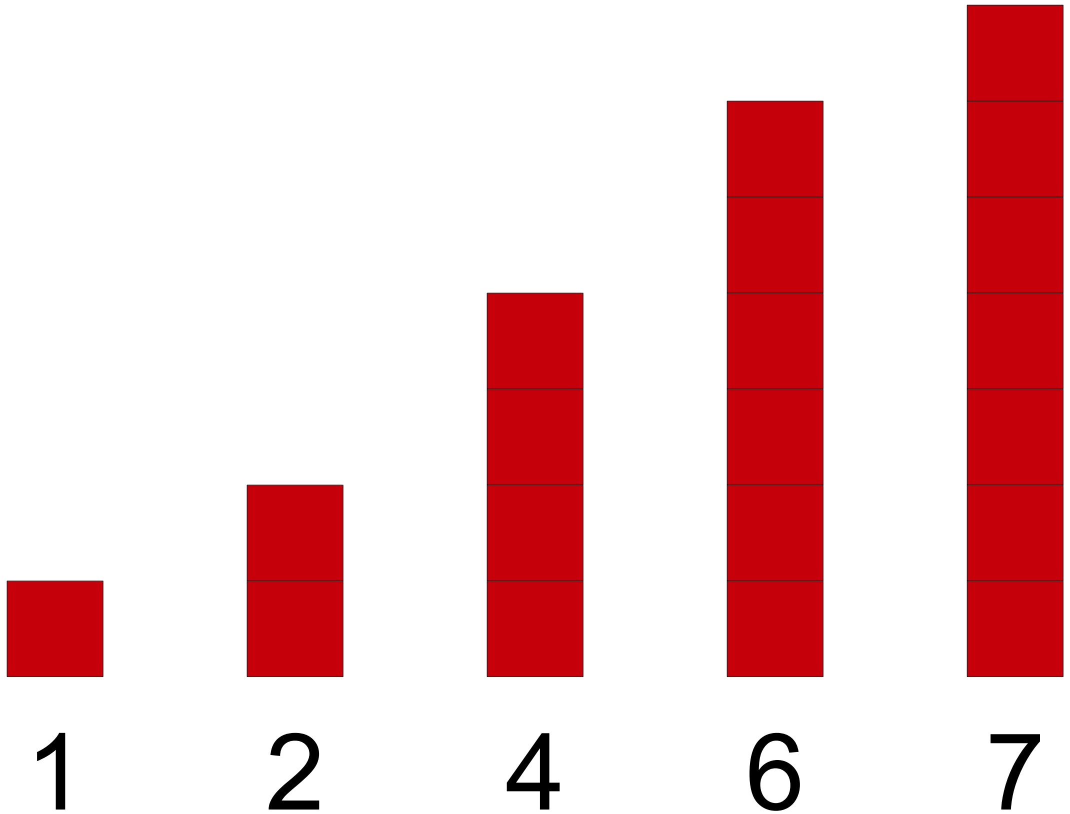
An Initial Idea: **Selection Sort**



An Initial Idea: **Selection Sort**



An Initial Idea: **Selection Sort**



Selection Sort

- Find the smallest element and move it to the first position.
- Find the second-smallest element and move it to the second position.
- (etc.)

Code for Selection Sort

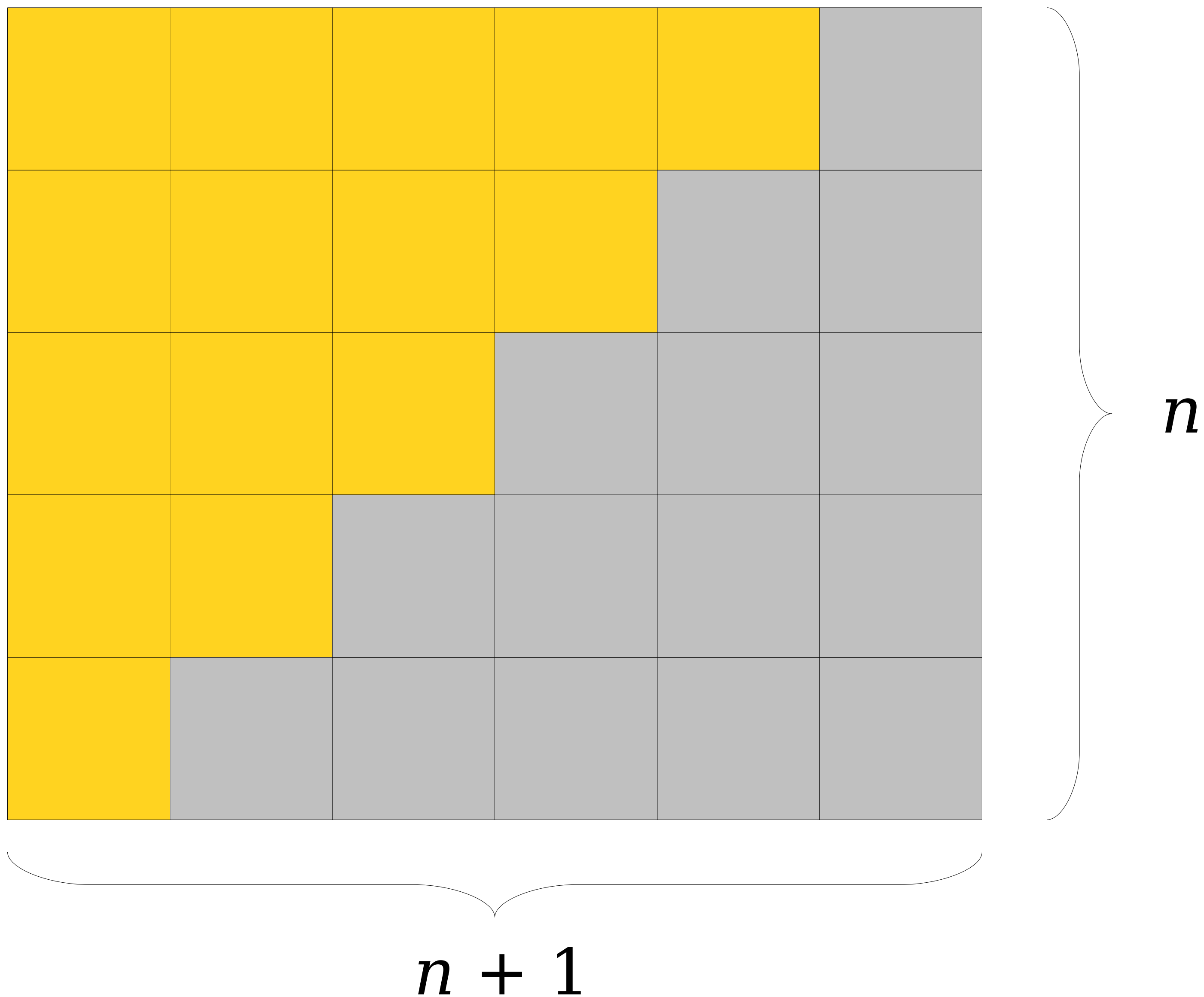
```
void selectionSort(Vector<int>& elems) {
  for (int index = 0; index < elems.size(); index++) {
    int smallestIndex = indexOfSmallest(elems, index);
    swap(elems[index], elems[smallestIndex]);
  }
}

int indexOfSmallest(Vector<int>& elems, int startPoint) {
  int smallestIndex = startPoint;
  for (int i = startPoint + 1; i < elems.size(); i++) {
    if (elems[i] < elems[smallestIndex])
      smallestIndex = i;
  }
  return smallestIndex;
}
```

Analyzing Selection Sort

- How much work do we do for selection sort?
- To find the smallest value, we need to look at all n array elements.
- To find the second-smallest value, we need to look at $n - 1$ array elements.
- To find the third-smallest value, we need to look at $n - 2$ array elements.
- Work is $n + (n - 1) + (n - 2) + \ldots + 1$.

$$n + (n\text{-}1) + \ldots + 2 + 1 = n(n+1) / 2$$



The Complexity of Selection Sort

$$O(n\ (n + 1) / 2)$$

$$= O(n\ (n + 1))$$

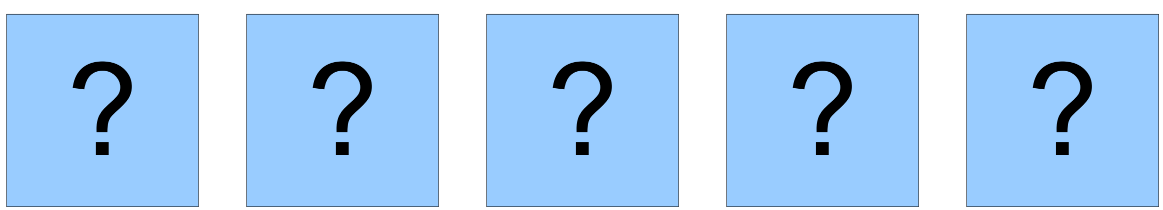
$$= O(n^2 + n)$$

$$= O(n^2)$$

So selection sort runs in time $\mathbf{O(n^2)}$.

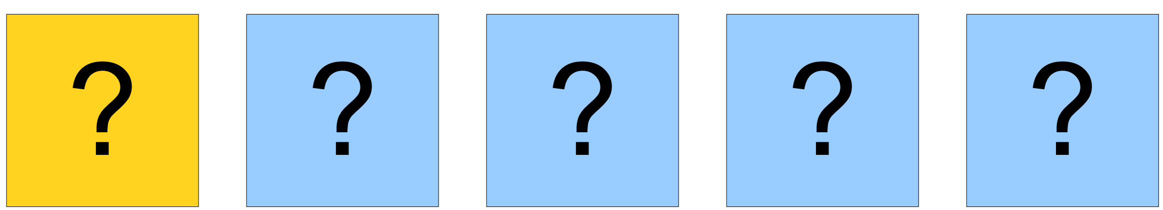
Notes on Selection Sort

- Selection sort has runtime $O(n^2)$ in the worst case.
- How about the best case?



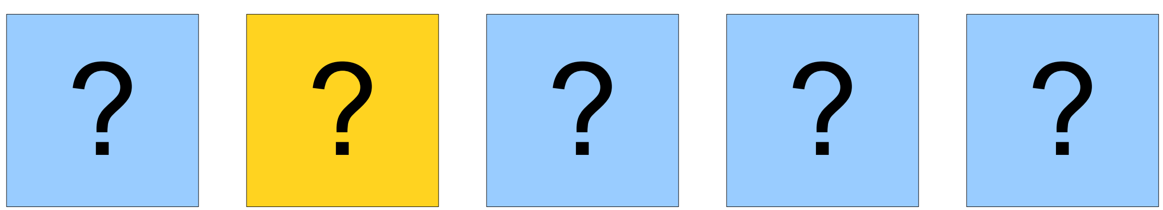
Notes on Selection Sort

- Selection sort has runtime $O(n^2)$ in the worst case.
- How about the best case?



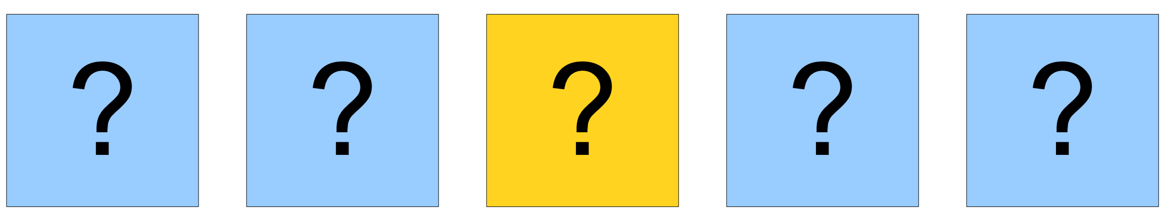
Notes on Selection Sort

- Selection sort has runtime O(n^2) in the worst case.
- How about the best case?



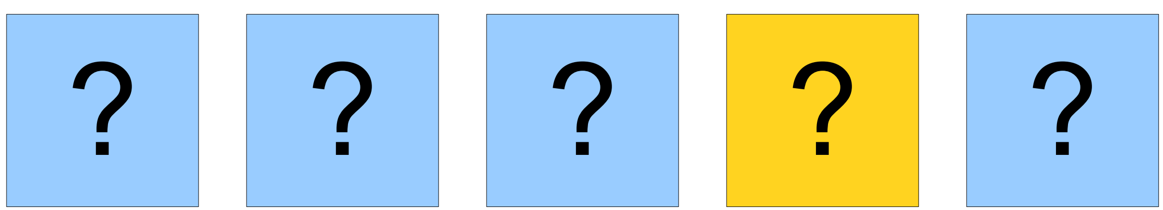
Notes on Selection Sort

- Selection sort has runtime O(n^2) in the worst case.
- How about the best case?



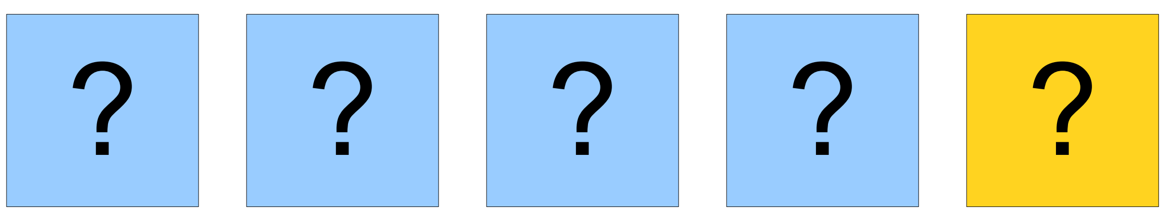
Notes on Selection Sort

- Selection sort has runtime $O(n^2)$ in the worst case.
- How about the best case?



Notes on Selection Sort

- Selection sort has runtime O(n^2) in the worst case.
- How about the best case?

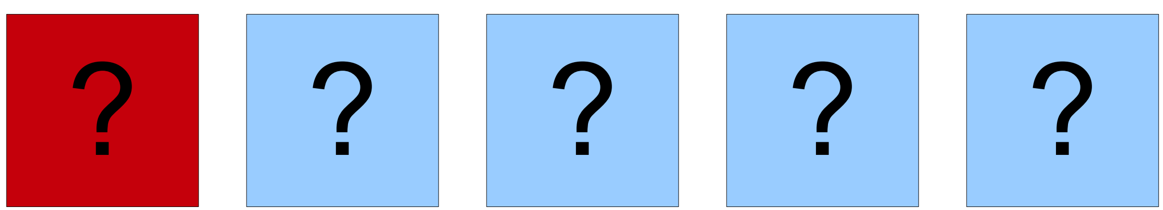


Notes on Selection Sort

- Selection sort has runtime $O(n^2)$ in the worst case.
- How about the best case?

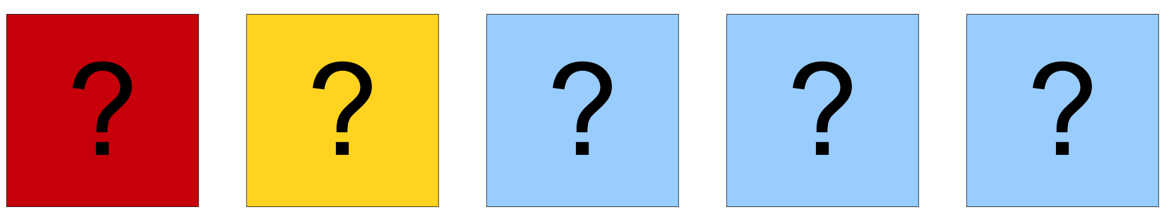
Notes on Selection Sort

- Selection sort has runtime O(n^2) in the worst case.
- How about the best case?



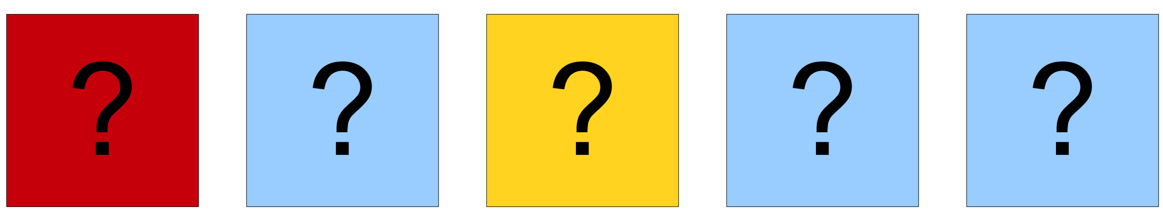
Notes on Selection Sort

- Selection sort has runtime O(n^2) in the worst case.
- How about the best case?



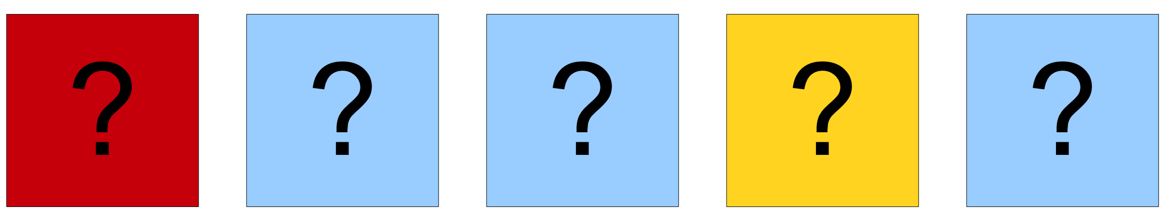
Notes on Selection Sort

- Selection sort has runtime $O(n^2)$ in the worst case.
- How about the best case?



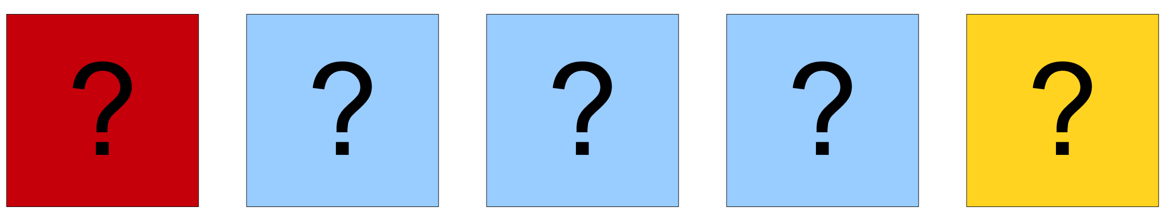
Notes on Selection Sort

- Selection sort has runtime $O(n^2)$ in the worst case.
- How about the best case?



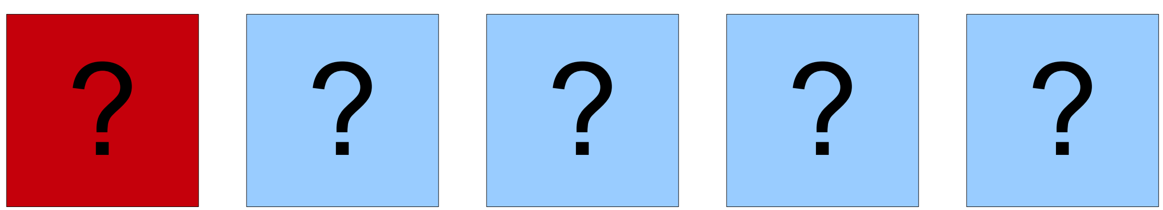
Notes on Selection Sort

- Selection sort has runtime O(n^2) in the worst case.
- How about the best case?



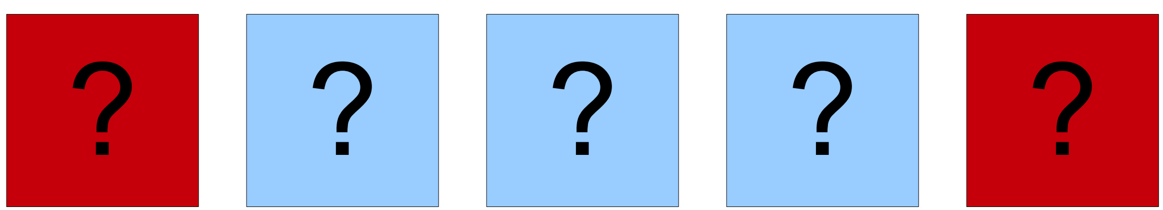
Notes on Selection Sort

- Selection sort has runtime O(n^2) in the worst case.
- How about the best case?



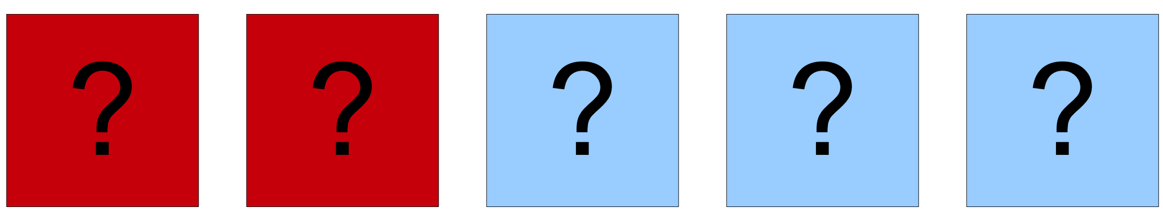
Notes on Selection Sort

- Selection sort has runtime O(n^2) in the worst case.
- How about the best case?



Notes on Selection Sort

- Selection sort has runtime $O(n^2)$ in the worst case.
- How about the best case?



Notes on Selection Sort

- Selection sort has runtime O(n^2) in the worst case.
- How about the best case?
- Also O(n^2)
- Selection sort *always* takes O(n^2) time.
- Notation: Selection sort is Θ(n^2).

Thinking About $O(n^2)$

Thinking About O(n^2)

14	**6**	**3**	**9**	**7**	**16**	**2**	**15**

Thinking About O(n^2)

$$\underbrace{\begin{array}{|c|c|c|c|c|c|c|c|}\hline \mathbf{14} & \mathbf{6} & \mathbf{3} & \mathbf{9} & \mathbf{7} & \mathbf{16} & \mathbf{2} & \mathbf{15} \\ \hline \end{array}}_{T(n)}$$

Thinking About O(n^2)

14	6	3	9	7	16	2	15

T(n)

14	6	3	9	7	16	2	15	5	10	8	11	1	13	12	4

Thinking About O(n^2)

14	6	3	9	7	16	2	15

$\mathrm{T}(n)$

14	6	3	9	7	16	2	15	5	10	8	11	1	13	12	4

$\mathrm{T}(2n) \approx 4\mathrm{T}(n)$

Selection Sort Times

Size	Selection Sort
10000	0.304
20000	1.218
30000	2.790
40000	4.646
50000	7.395
60000	10.584
70000	14.149
80000	18.674
90000	23.165

Next Time

- **Faster Sorting Algorithms**
  - Can you beat O(n^2) time?
- **Hybrid Sorting Algorithms**
  - When might selection sort be useful?