Limits of What Computers Can Do

Announcements

- Assignment 6 due Friday at 11AM
  - Cannot be turned in late!
- Regular Office Hours today
- Extended Office Hours this Week
  - Wednesday, Thursday: Noon-5PM
- Graded midterms will be returned tomorrow (Wednesday)
- Please fill out course evaluations!

Limits of Programs

- We've spent a lot of time going over cool stuff computers can do
  - Quickly Sorting, Searching
    - Binary Search, Quicksort
  - Quickly storing and retrieving data
    - Hashing, Binary Search Trees
- An interesting question to consider is what *can't* computers do

Limits of Programs

- There are three I want to consider:
  - What can't a computer *do any faster*?
  - What can't a computer *do fast*?
  - What can't a computer *do at all*?

Limits of Programs

- There are three I want to consider:
  - **What can't a computer *do any faster*?**
  - What can't a computer *do fast*?
  - What can't a computer *do at all*?

Lower Bounds on Sorting

- Run times of various sorting algorithms:
  - QuickSort: O(n log n)
  - MergeSort: O(n log n)
  - HeapSort: O(n log n)
  - SmoothSort: O(n log n)

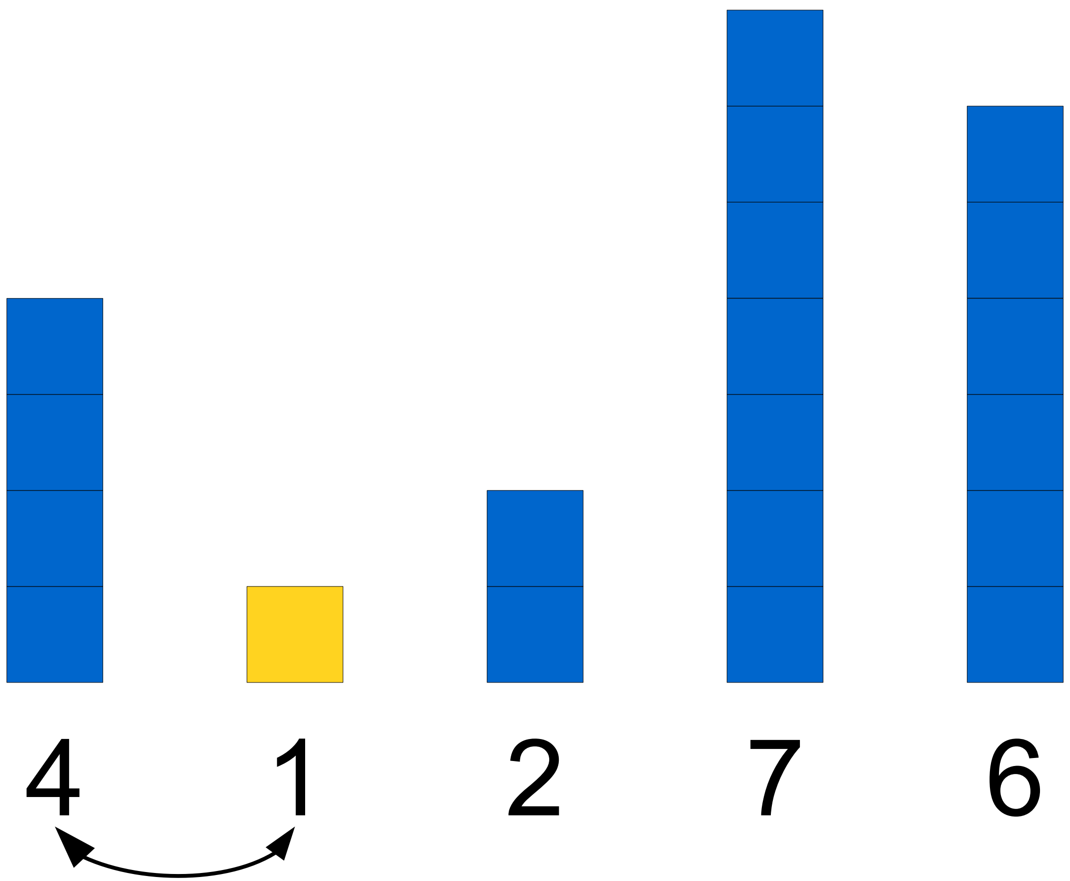
  ...
- Notice a pattern?

All of our fast sorting algorithms run in O(n log n) – what's up with that?

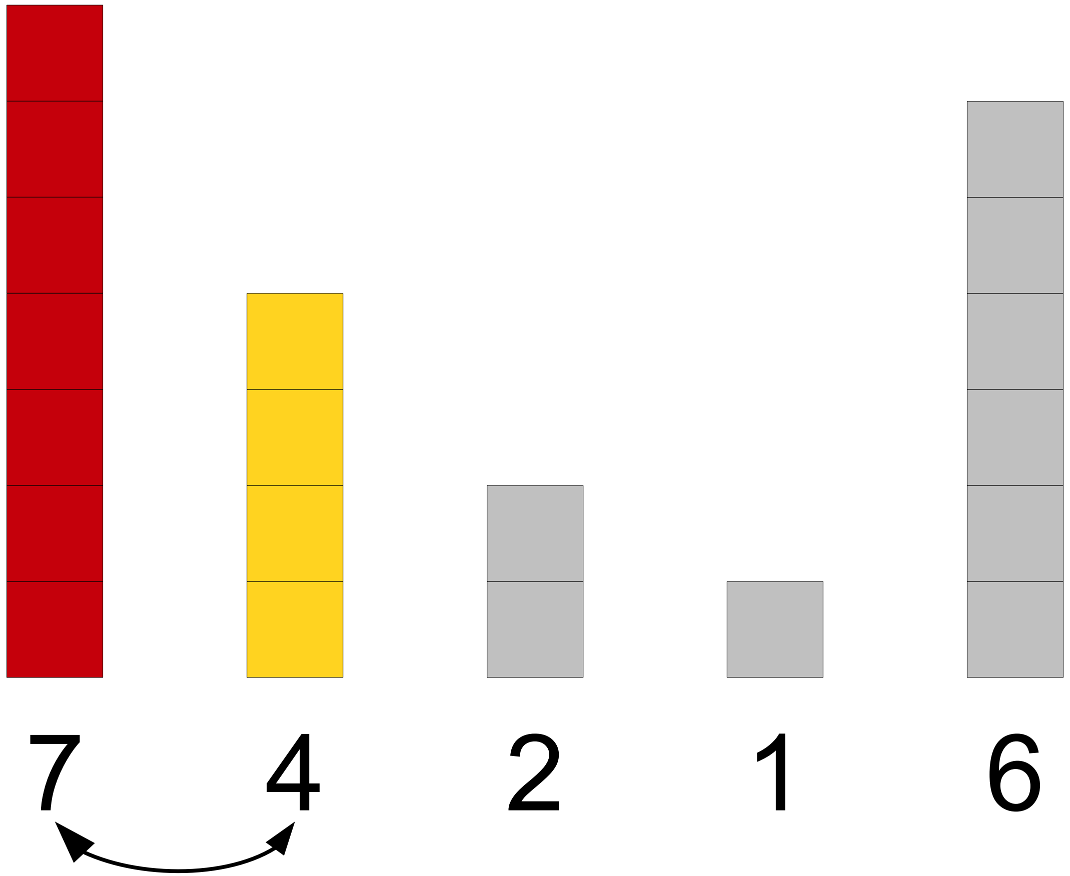
Lower Bounds on Sorting

- I haven't been holding back – we don't have any general-purpose sorting algorithms that are asymptotically faster than O(n log n).
- In fact, we can *prove* that we can't do any better (for general purpose algorithms).
- In order to do this we need to find what all our sorting algorithms have in common...

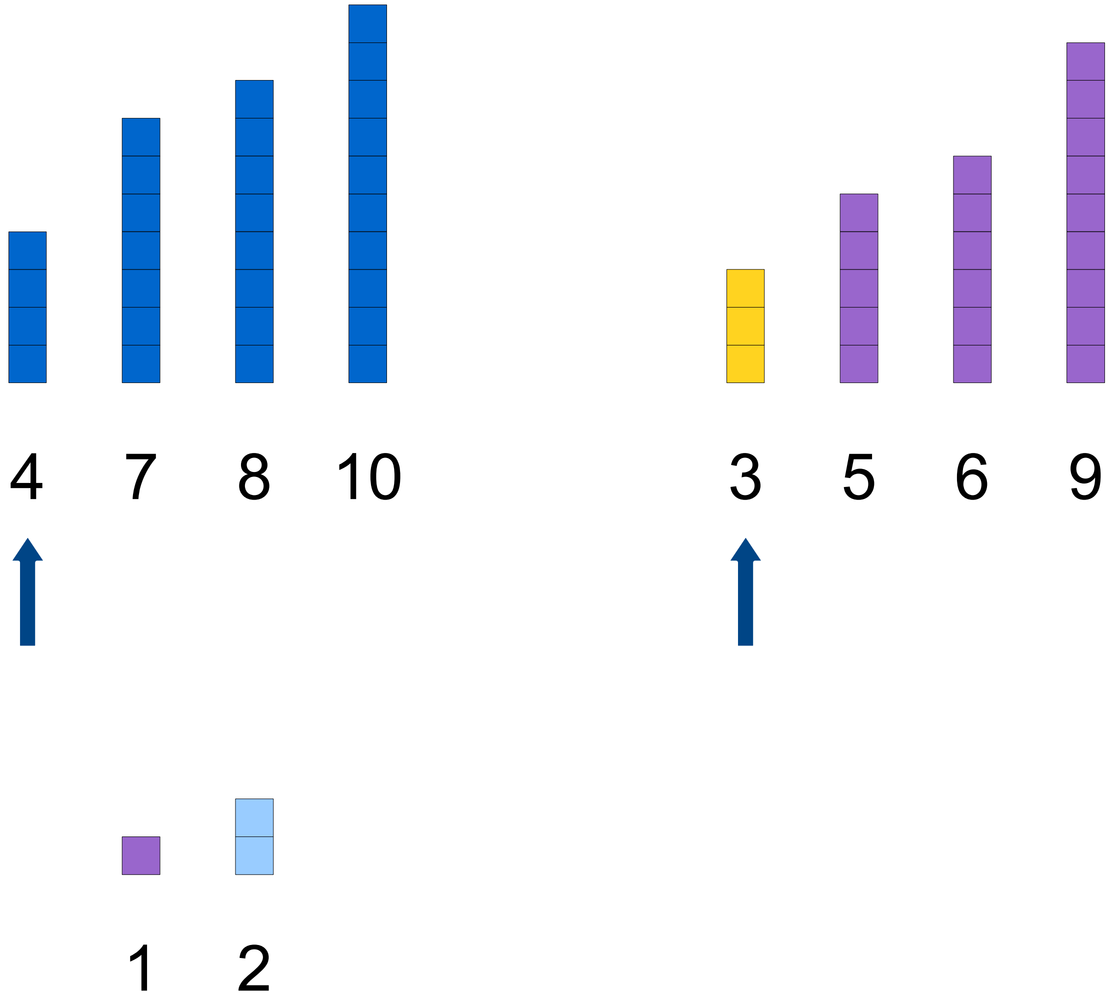
An Initial Idea: **Selection Sort**



Another Idea: **Insertion Sort**



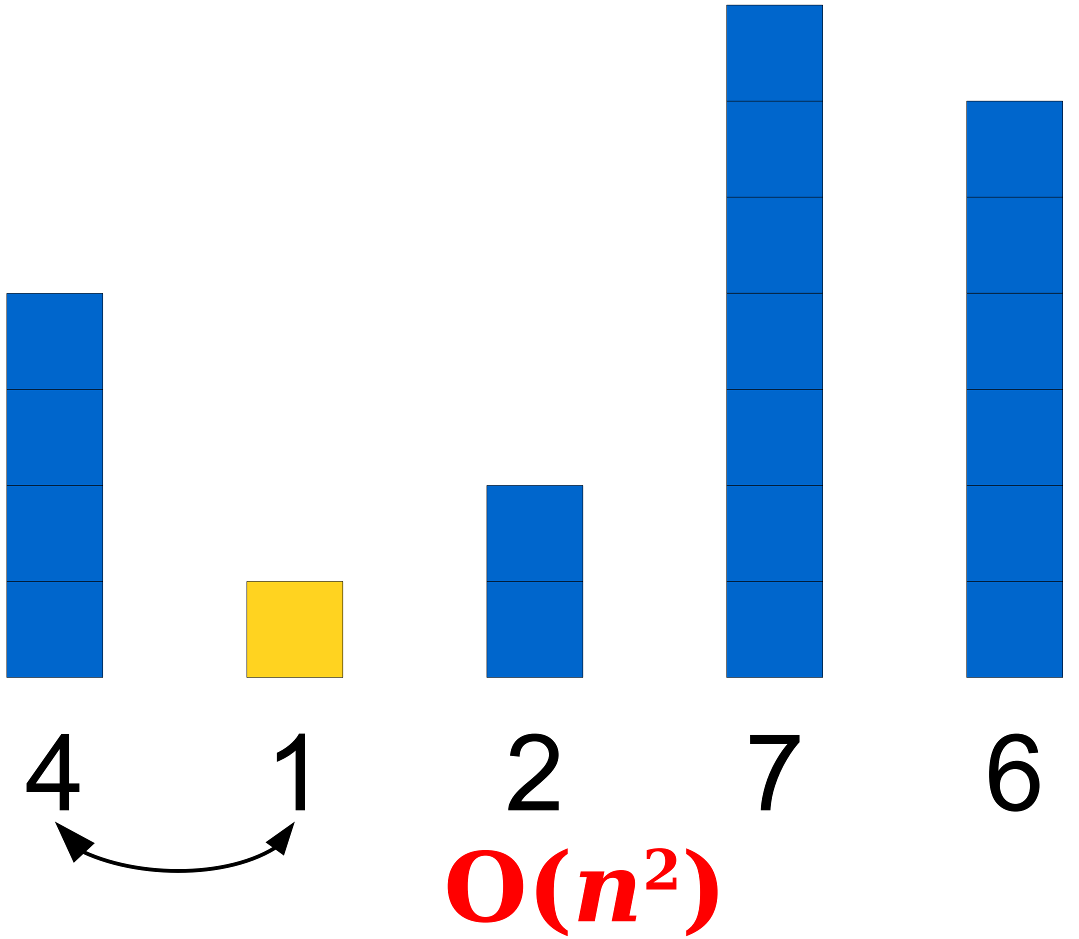
The Key Insight: **Merge**



Lower Bounds on Sorting

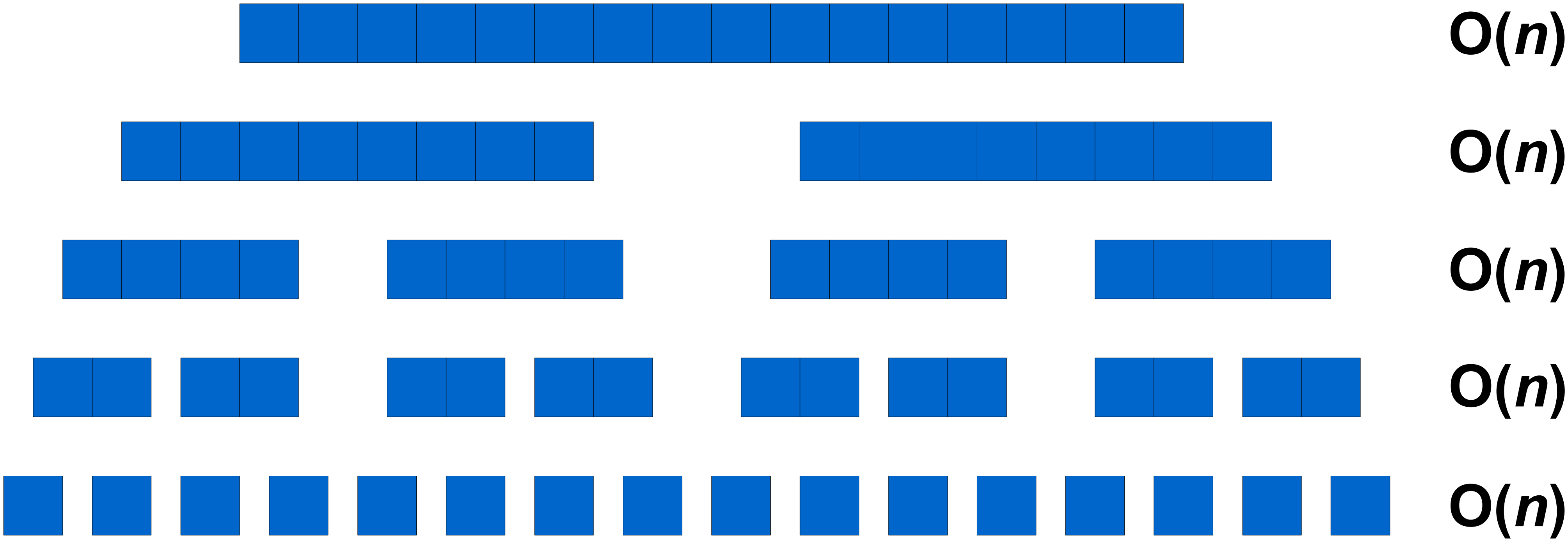
- Observation: All our sorting algorithms involve repeatedly comparing pairs of elements in the array
- One way of measuring the amount of work our sorting algorithms do is by counting how many comparisons are performed

An Initial Idea: **Selection Sort**



O(n) comparisons per element → O(n^2) runtime!

Merge Sort



O(n log n)

O(n log n) runtime → O(log n) comparisons per node!

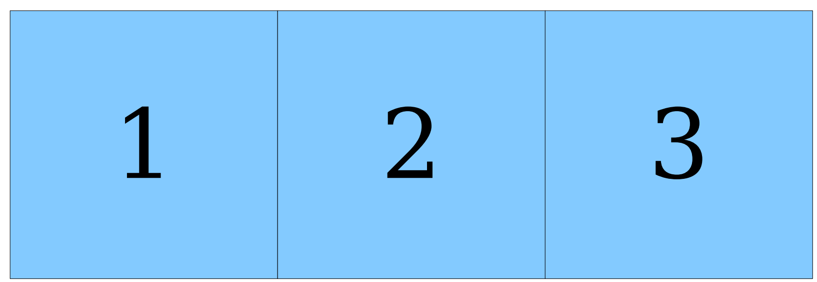
Lower Bounds on Sorting

- All our algorithms compare pairs of elements and their runtime is determined by how many comparisons are made.
  - These are all **comparison based** sorting algorithms
- Can we prove that all comparison based sorting algorithms require some minimum number of comparisons?
  - If we can do this, then we can prove a **lower bound** on the runtime of all comparison based sorting algorithms.

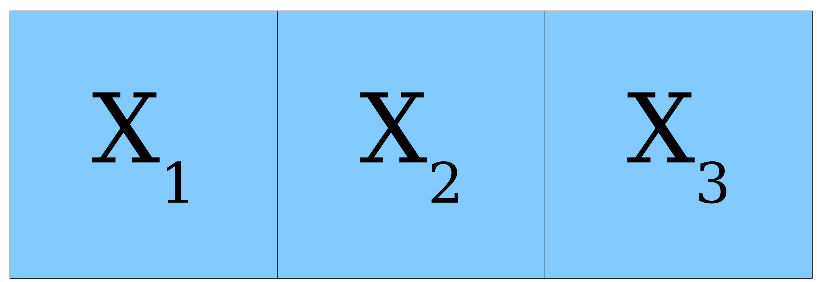
Lower Bounds on Sorting

- All our algorithms compare pairs of elements and their runtime is determined by how many comparisons are made.
  - These are all **comparison based** sorting algorithms
- Can we prove that all comparison based sorting algorithms require some minimum number of comparisons?
  - If we can do this, then we can prove a **lower bound** on the runtime of all comparison based sorting algorithms.

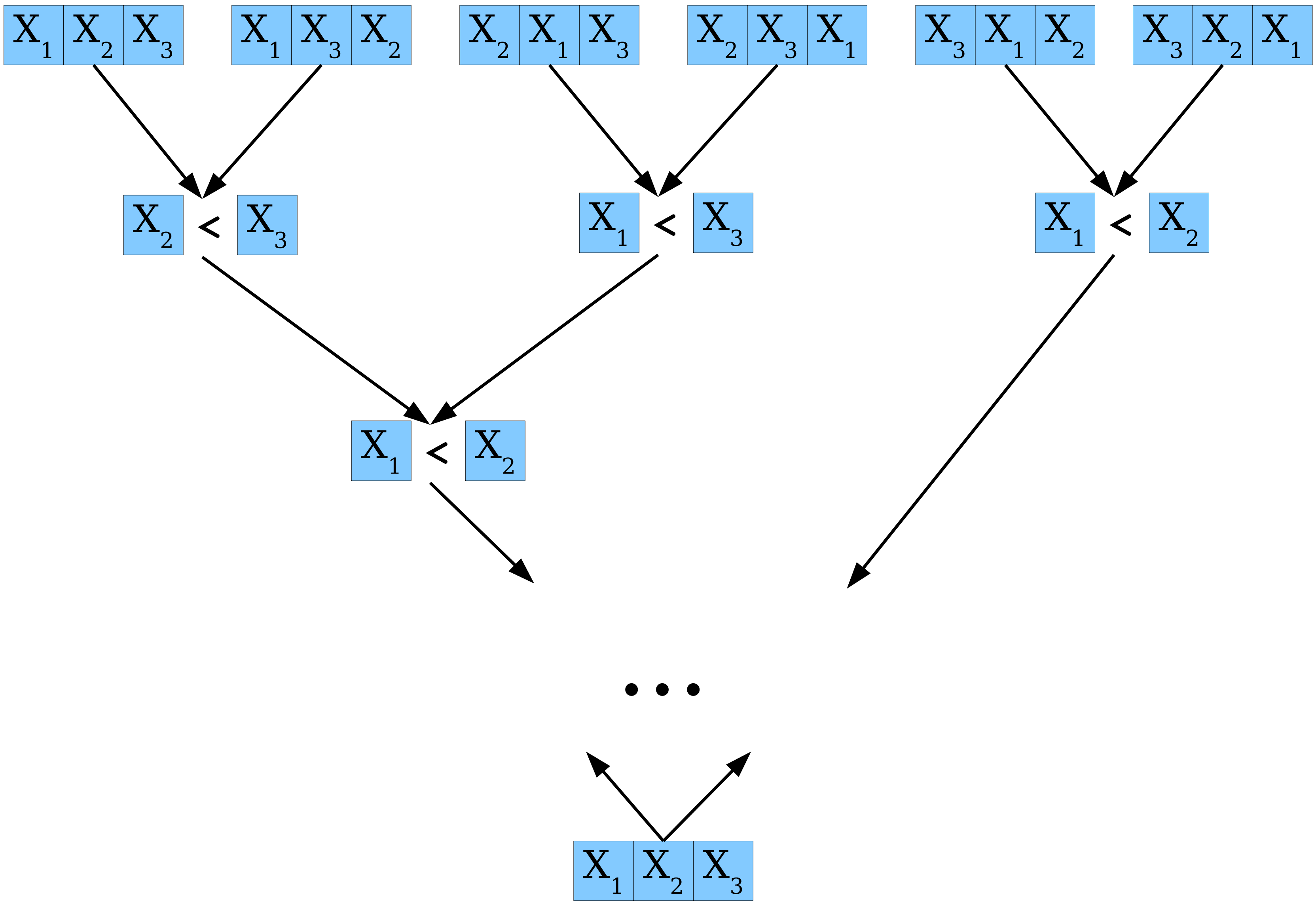
Intuition Behind Proof



Intuition Behind Proof



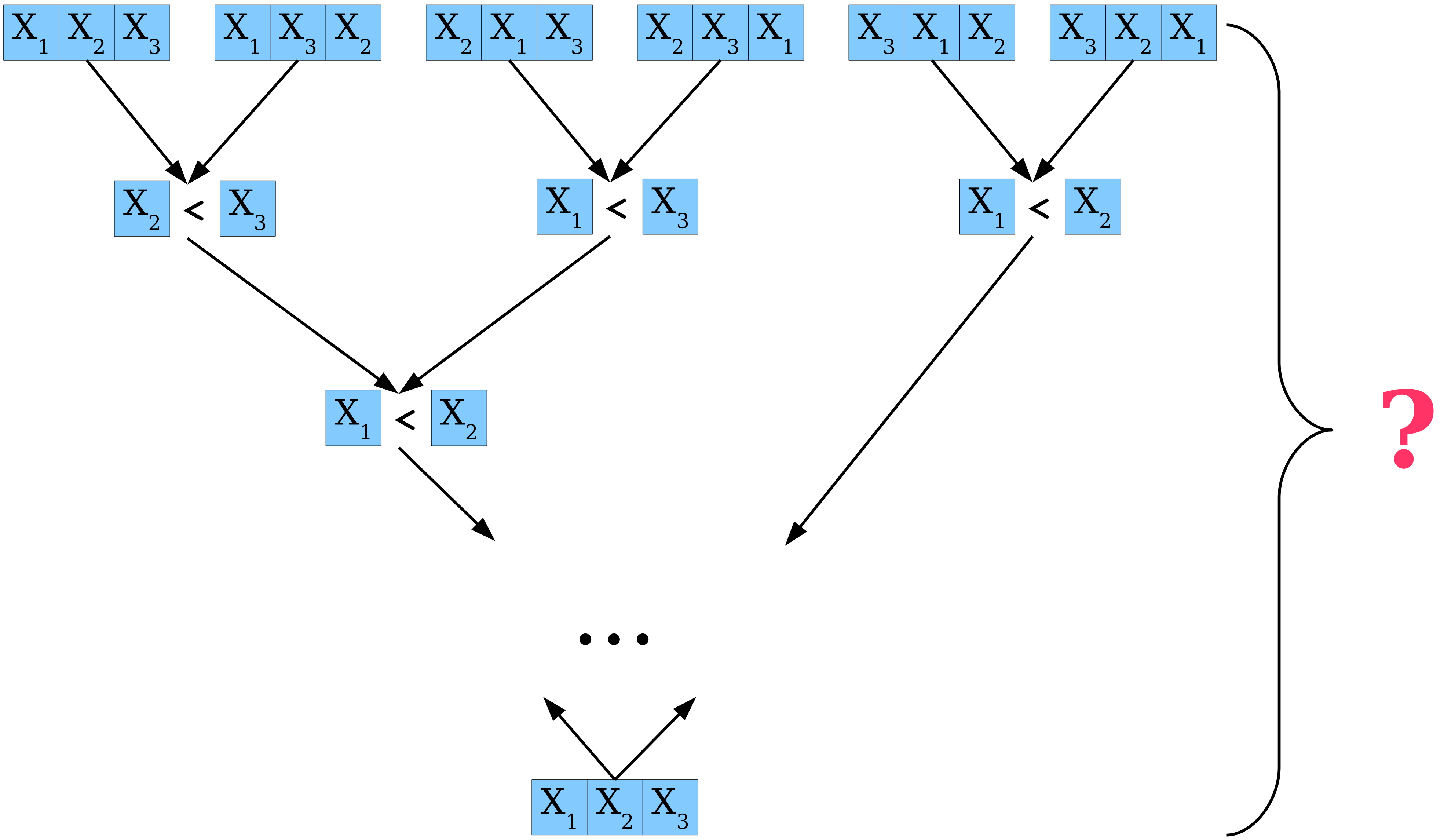
Intuition Behind Proof



Intuition Behind Proof

- Every sorting algorithm needs to be able to sort every possible permutation of ***n*** elements.
- The number of comparisons needed is proportional to the height of the tree.

Intuition Behind Proof



Intuition Behind Proof

- Because any list of elements has ***n*!** permutations, we know the tree has ***n*!** leaves.
- The height of a balanced binary tree with *L* leaves is **O(log *L*)**
- Therefore, the height of our tree is **O(log *n*!)**
- Sterling's Approximation
  - **O(log *n*!)** = **O(*n* log *n*)**
- The height of our tree is **O(*n* log *n*)**

Therefore, **all** comparison based sorting algorithms require **$O(n \log n)$** comparisons in the worst case.

This implies the best we can do is **$O(n \log n)$** worst case runtime.

(QED)

Other Sorting Algorithms

- Summary: No "comparison-based" sorting algorithms can do better than worse case **O(*n* log *n*)**.
- Should we give up? No!
- Two ways we can get around this:
  - Make additional assumptions about the data
  - Use a non-comparison based sorting algorithm

Additional Assumptions

- If we have an unsorted array in which we knew every element was within *k* indices of where it should be and ran HeapSort

***k* = 3**

4	3	1	5	2	6	9	7	8	12	11	10

Heap

Additional Assumptions

- If we have an unsorted array in which we knew every element was within *k* indices of where it should be and ran HeapSort

***k* = 3**

3	1	5	2	6	9	7	8	12	11	10

Heap	4

Additional Assumptions

- If we have an unsorted array in which we knew every element was within *k* indices of where it should be and ran HeapSort

***k* = 3**

1	5	2	6	9	7	8	12	11	10

Heap	4	3

Additional Assumptions

- If we have an unsorted array in which we knew every element was within *k* indices of where it should be and ran HeapSort

***k* = 3**

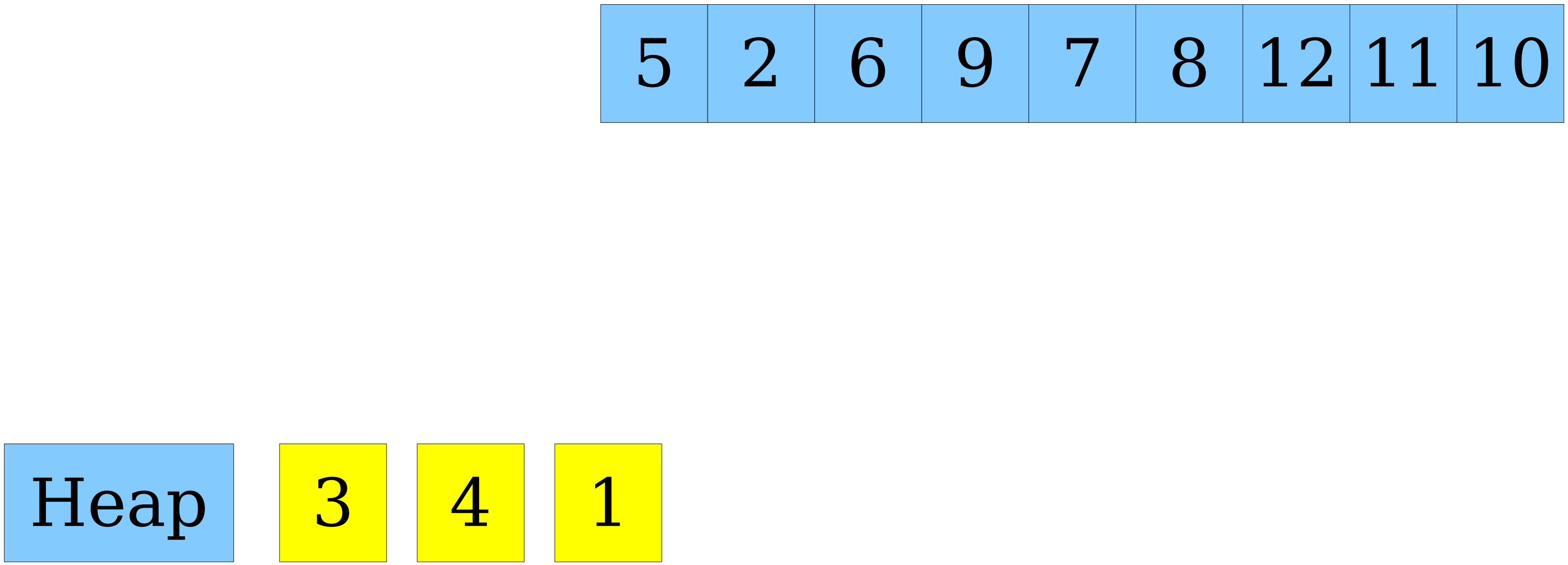
1	5	2	6	9	7	8	12	11	10

Heap	3	4

Additional Assumptions

- If we have an unsorted array in which we knew every element was within *k* indices of where it should be and ran HeapSort

***k* = 3**



Additional Assumptions

- If we have an unsorted array in which we knew every element was within *k* indices of where it should be and ran HeapSort

***k* = 3**

5	2	6	9	7	8	12	11	10

Heap	1	4	3

Additional Assumptions

- If we have an unsorted array in which we knew every element was within *k* indices of where it should be and ran HeapSort

***k* = 3**

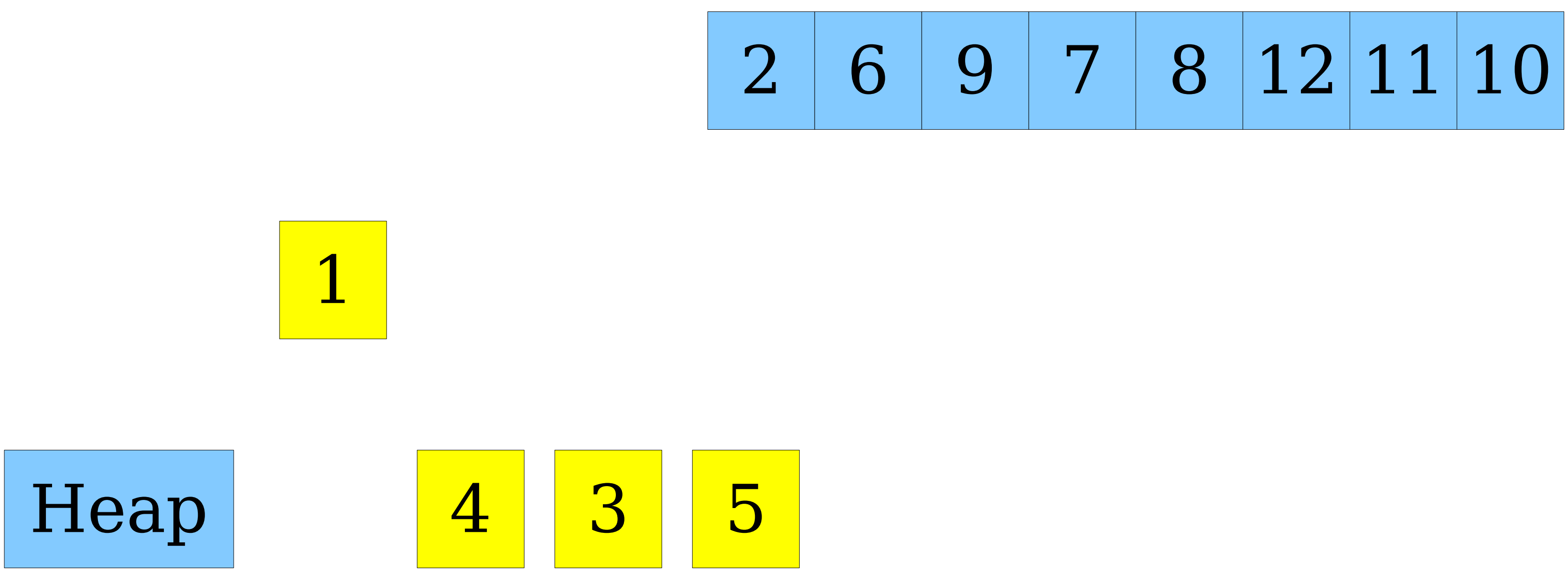
2	6	9	7	8	12	11	10

Heap	1	4	3	5

Additional Assumptions

- If we have an unsorted array in which we knew every element was within k indices of where it should be and ran HeapSort

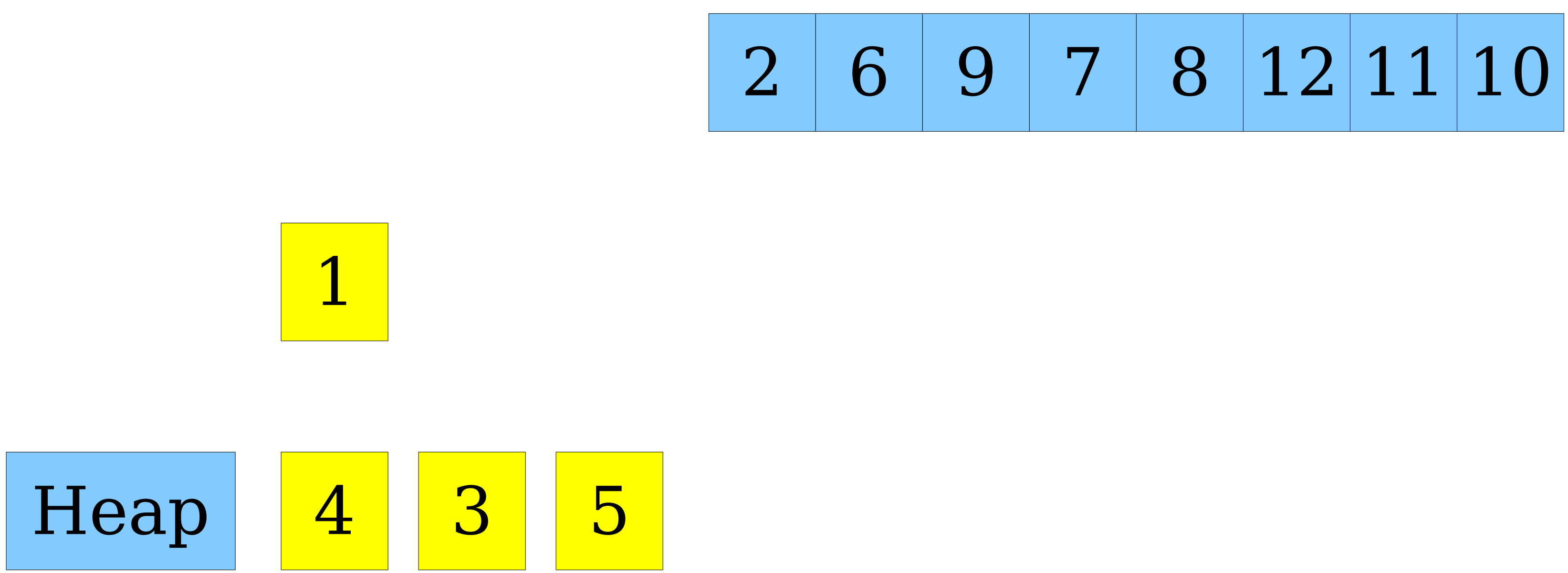
***k* = 3**



Additional Assumptions

- If we have an unsorted array in which we knew every element was within *k* indices of where it should be and ran HeapSort

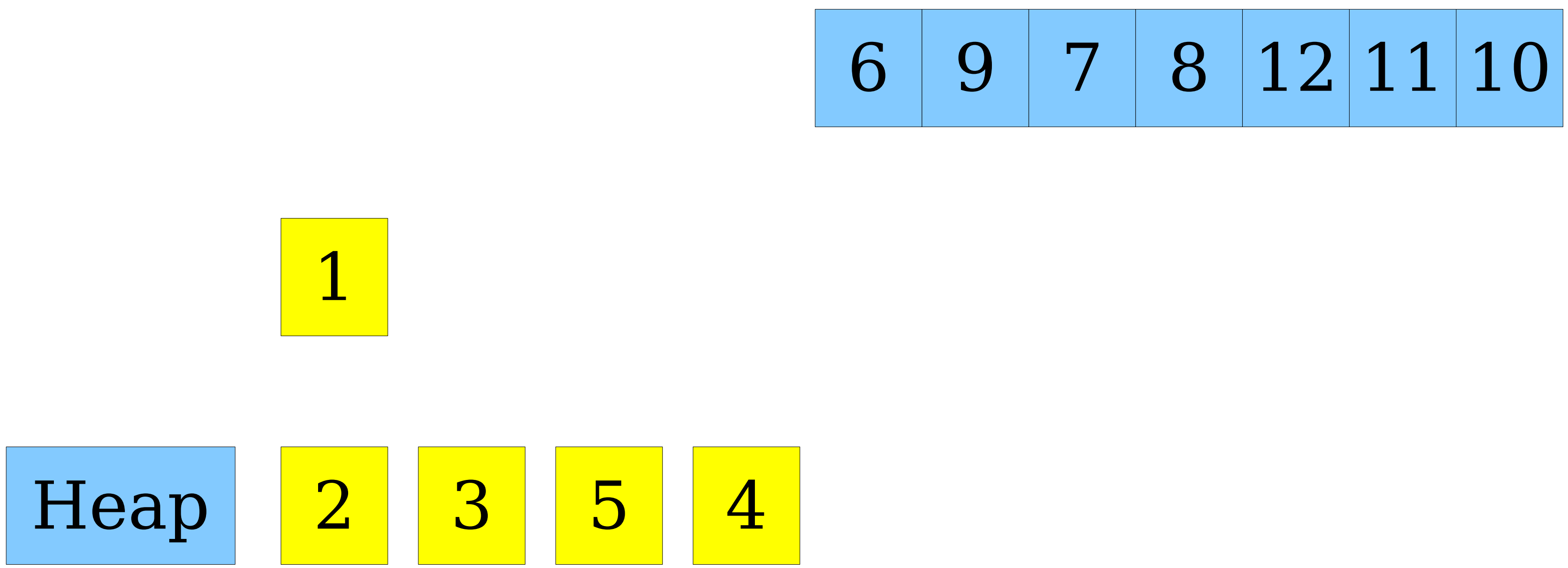
***k* = 3**



Additional Assumptions

- If we have an unsorted array in which we knew every element was within *k* indices of where it should be and ran HeapSort

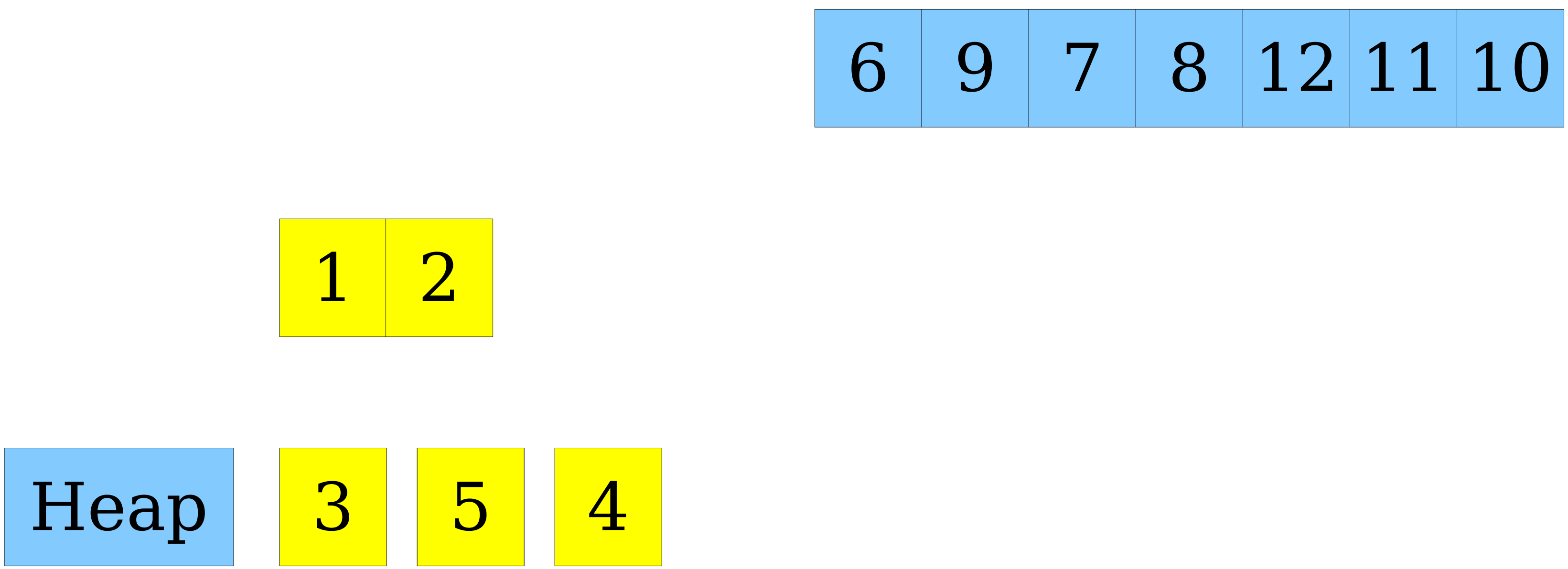
***k* = 3**



Additional Assumptions

- If we have an unsorted array in which we knew every element was within *k* indices of where it should be and ran HeapSort

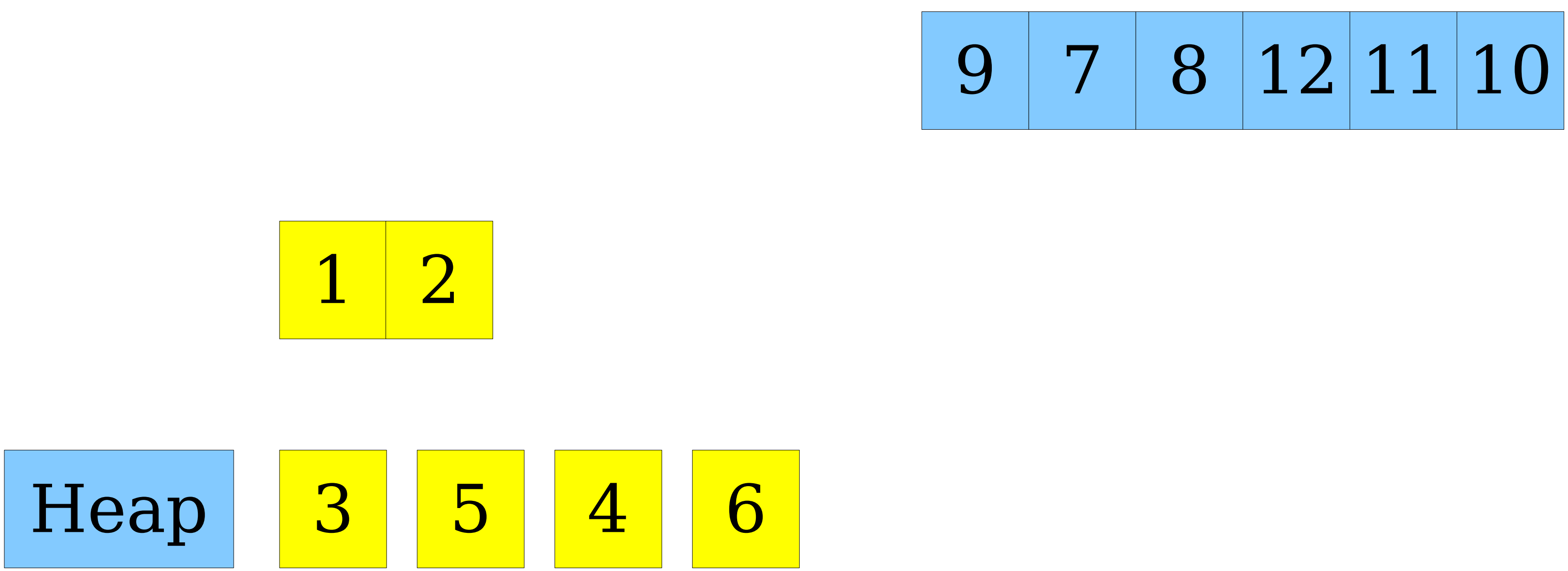
***k* = 3**



Additional Assumptions

- If we have an unsorted array in which we knew every element was within *k* indices of where it should be and ran HeapSort

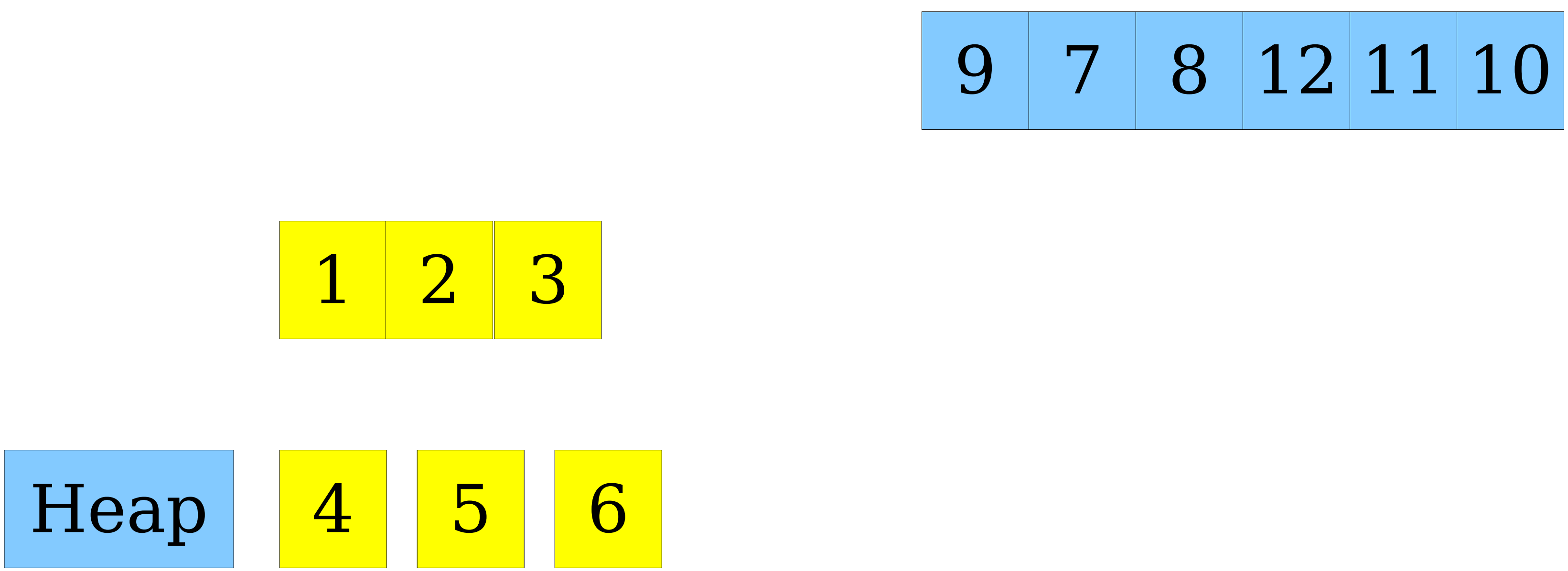
***k* = 3**



Additional Assumptions

- If we have an unsorted array in which we knew every element was within *k* indices of where it should be and ran HeapSort

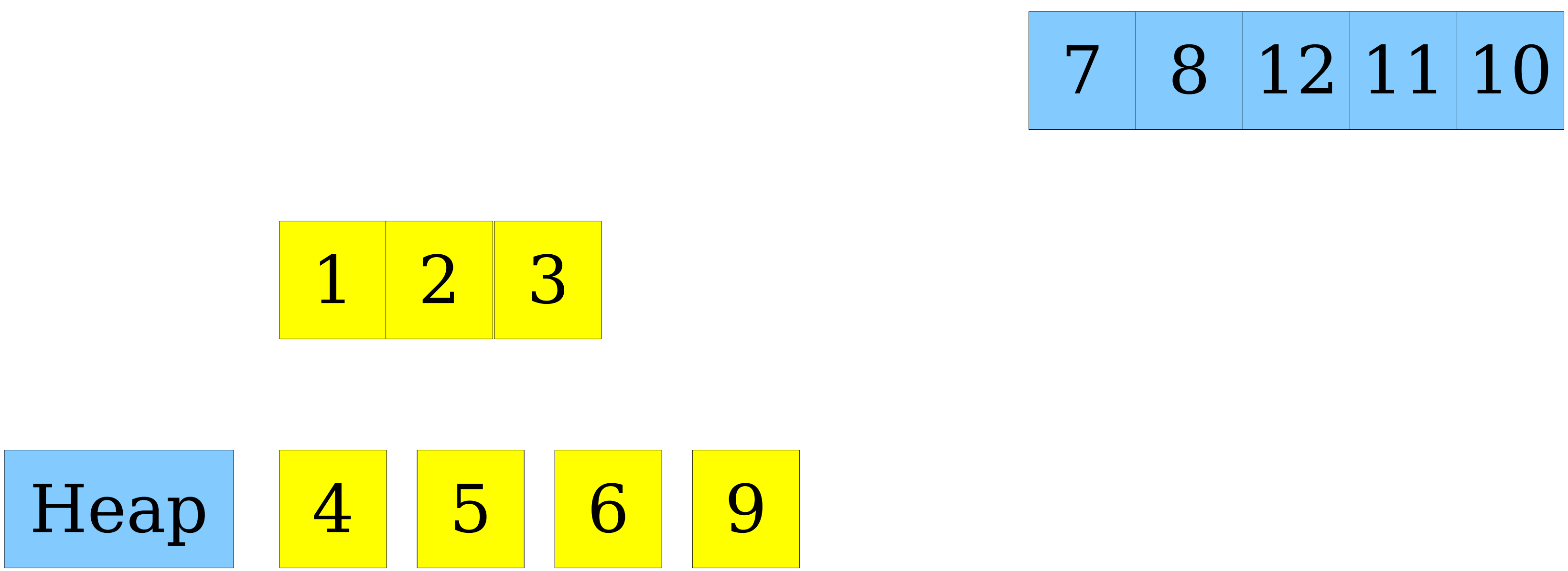
***k* = 3**



Additional Assumptions

- If we have an unsorted array in which we knew every element was within k indices of where it should be and ran HeapSort

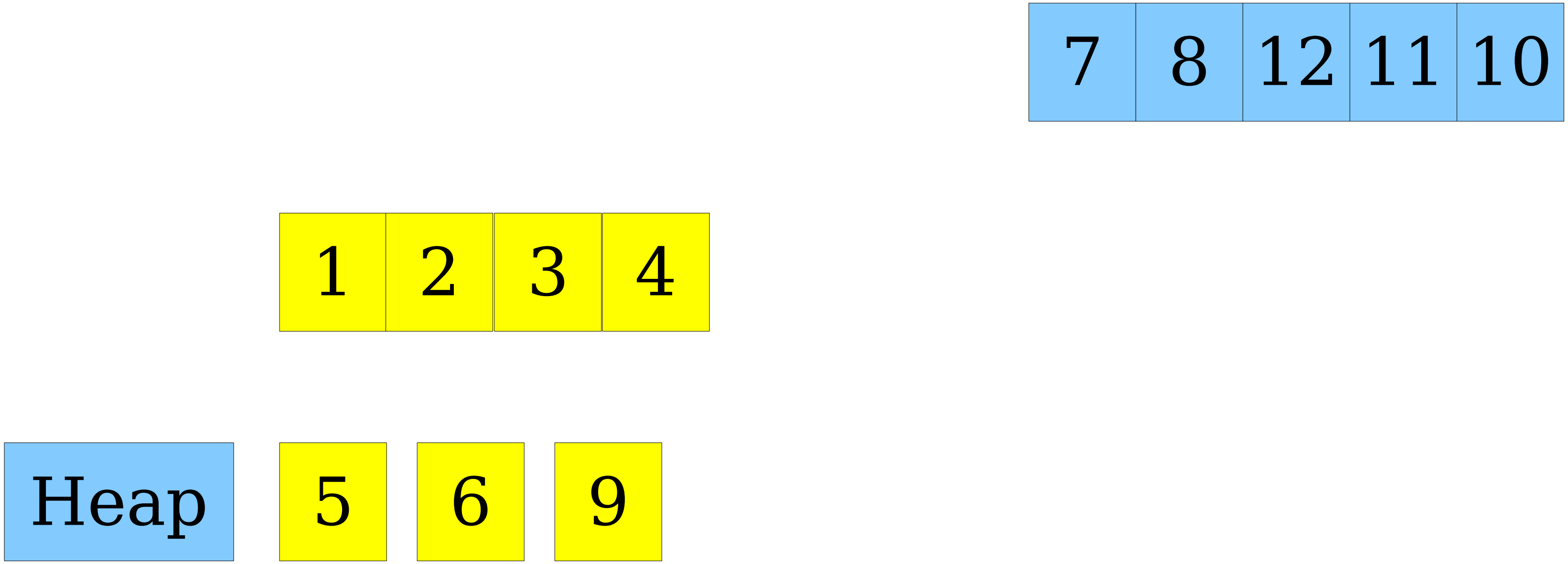
***k* = 3**



Additional Assumptions

- If we have an unsorted array in which we knew every element was within *k* indices of where it should be and ran HeapSort

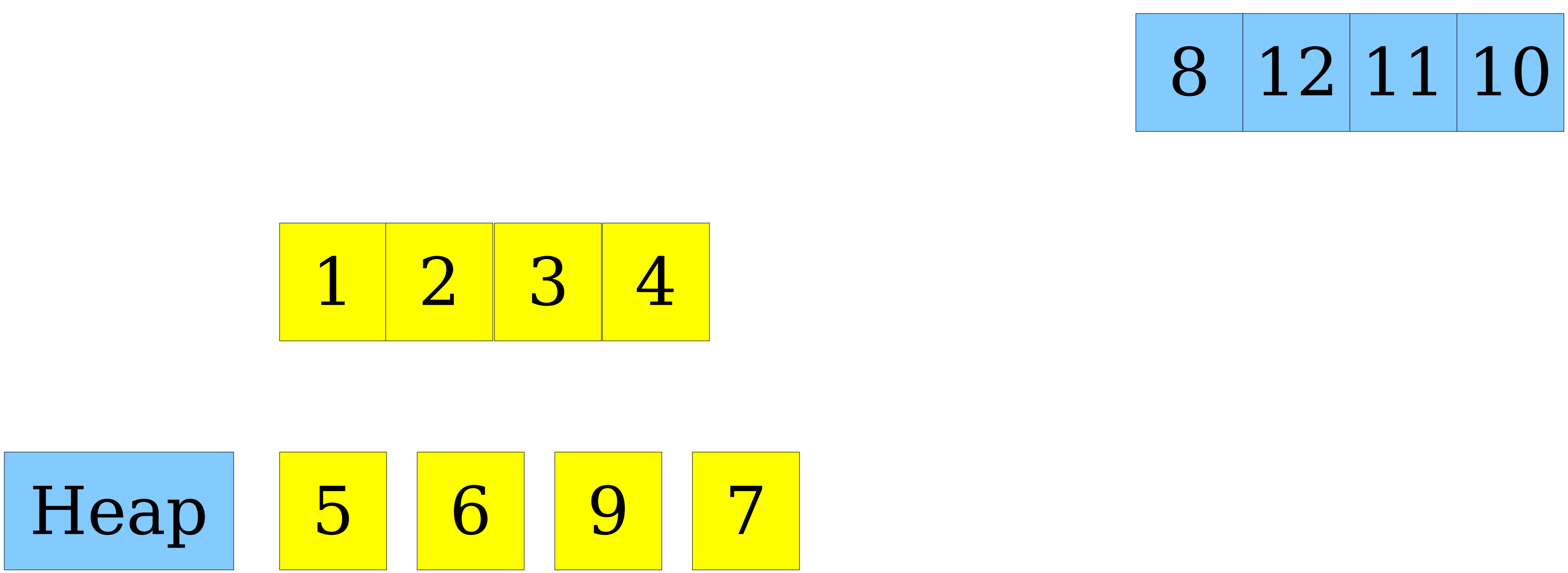
***k* = 3**



Additional Assumptions

- If we have an unsorted array in which we knew every element was within *k* indices of where it should be and ran HeapSort

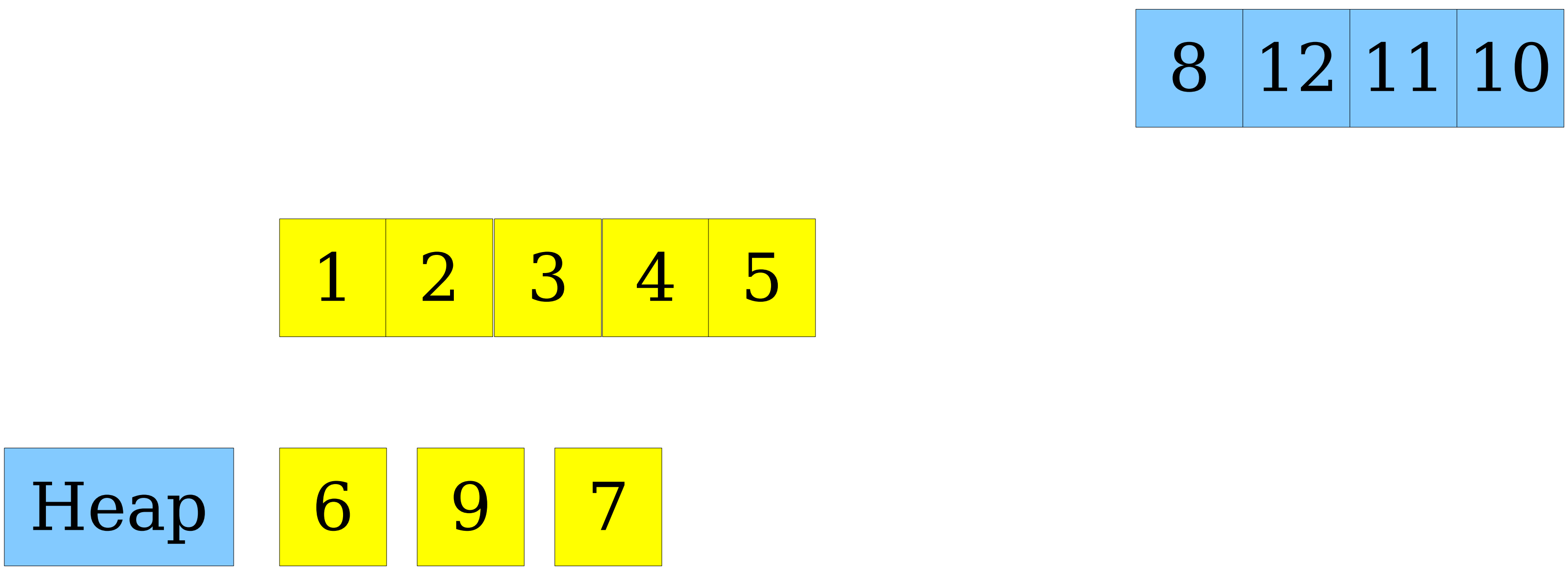
***k* = 3**



Additional Assumptions

- If we have an unsorted array in which we knew every element was within *k* indices of where it should be and ran HeapSort

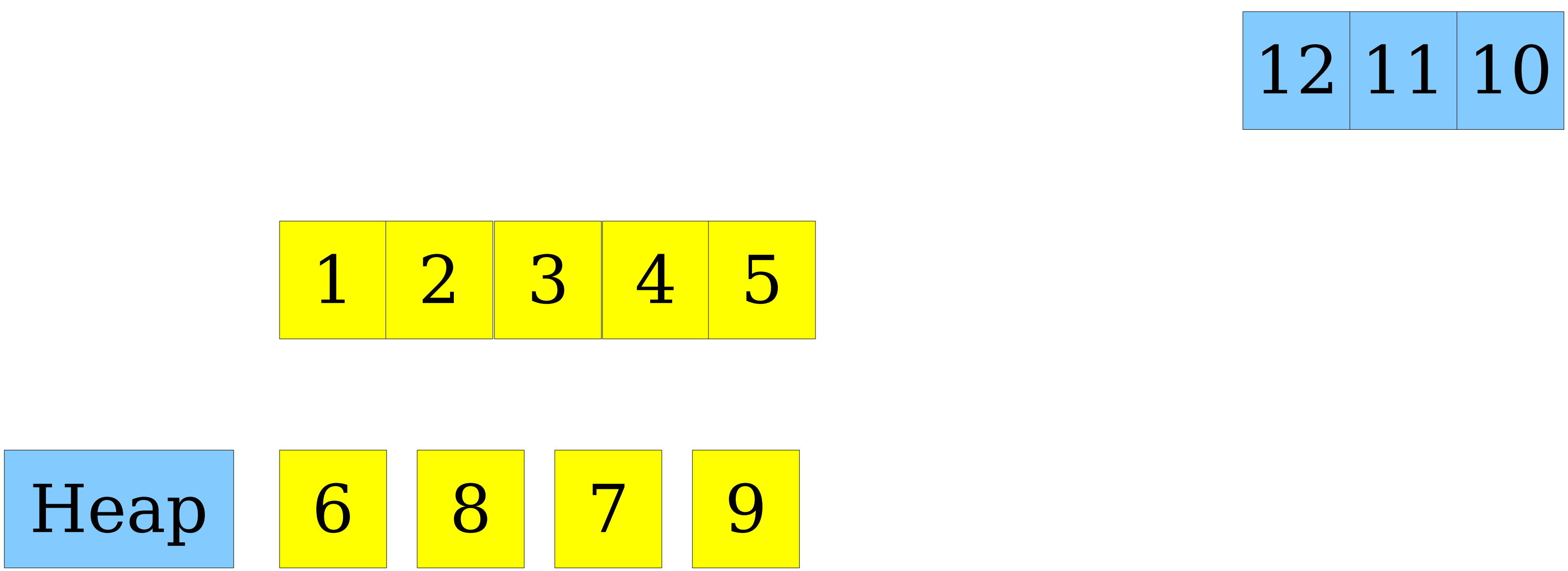
***k* = 3**



Additional Assumptions

- If we have an unsorted array in which we knew every element was within *k* indices of where it should be and ran HeapSort

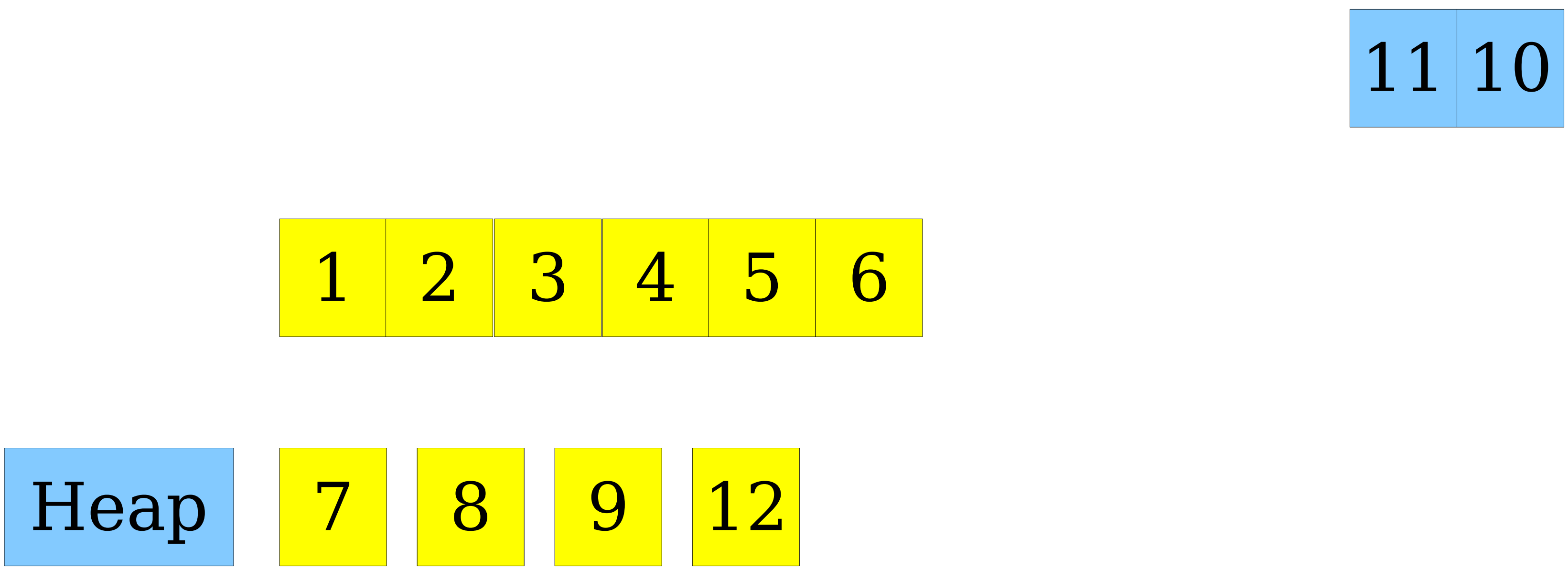
***k* = 3**



Additional Assumptions

- If we have an unsorted array in which we knew every element was within k indices of where it should be and ran HeapSort

k = 3



Additional Assumptions

- If we have an unsorted array in which we knew every element was within *k* indices of where it should be and ran HeapSort

***k* = 3**

1	2	3	4	5	6	7	8	9	10	11	12

Heap

Heap Sort

- If we know every element is within k indices of its correct location, then we can dequeue whenever the heap has $k + 1$ elements
- What is the runtime of this algorithm?
  - Each element is added and removed
  - Both operations are logarithmic in the size of the Heap = $k + 1$
  - Therefore and remove are **O(log *k*)**
  - We have **O(*n*)** elements
  - **O(*n* log *k*)!!!!**

Heap Sort

- The smaller we can make k, the faster HeapSort will run.
- When $k = n$ it devolves into regular HeapSort with **O(n log n)** runtime

Non-Comparison Based Algorithms

- Another way to beat the **O(*n* log *n*)** bound is to use non-comparison based sorting algorithms:
  - Bucket Sort: Construct a histogram of the elements in the array

Bucket Sort

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z

b	**a**	**n**	**a**	**n**	**a**

Bucket Sort

0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z

b	**a**	**n**	**a**	**n**	**a**

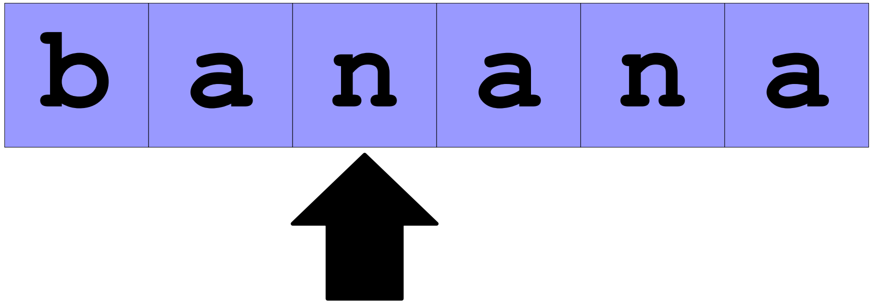
Bucket Sort

1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z

b	**a**	**n**	**a**	**n**	**a**

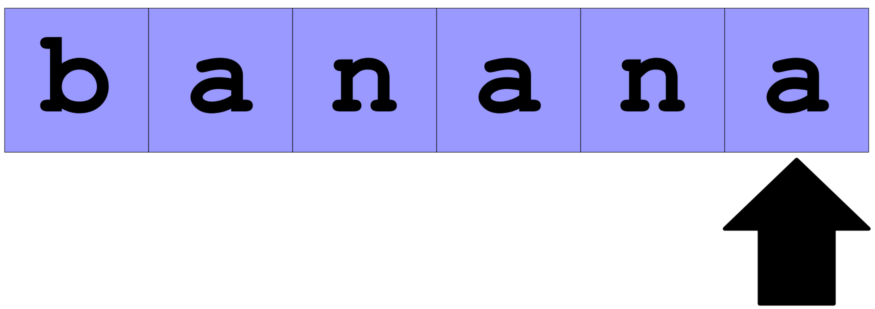
Bucket Sort

1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z



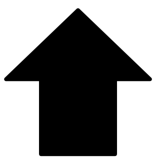
Bucket Sort

3	1	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0
a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z



Bucket Sort

3	1	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0
a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z



Bucket Sort

3	1	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0
a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z

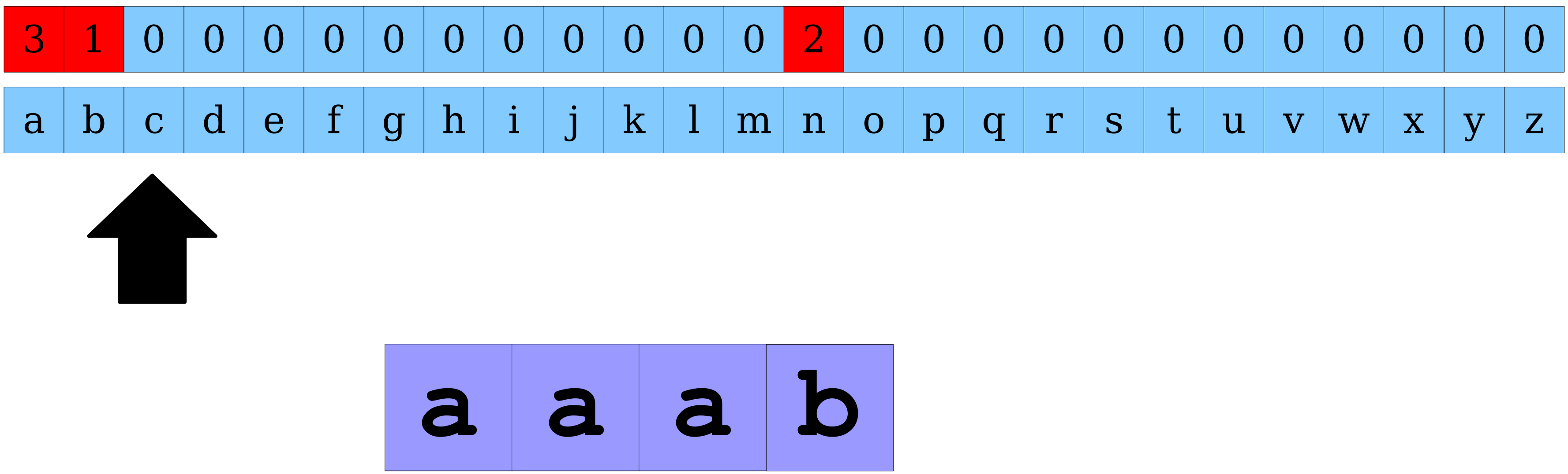
a	**a**	**a**

Bucket Sort

3	1	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0
a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z

a	**a**	**a**	**b**

Bucket Sort

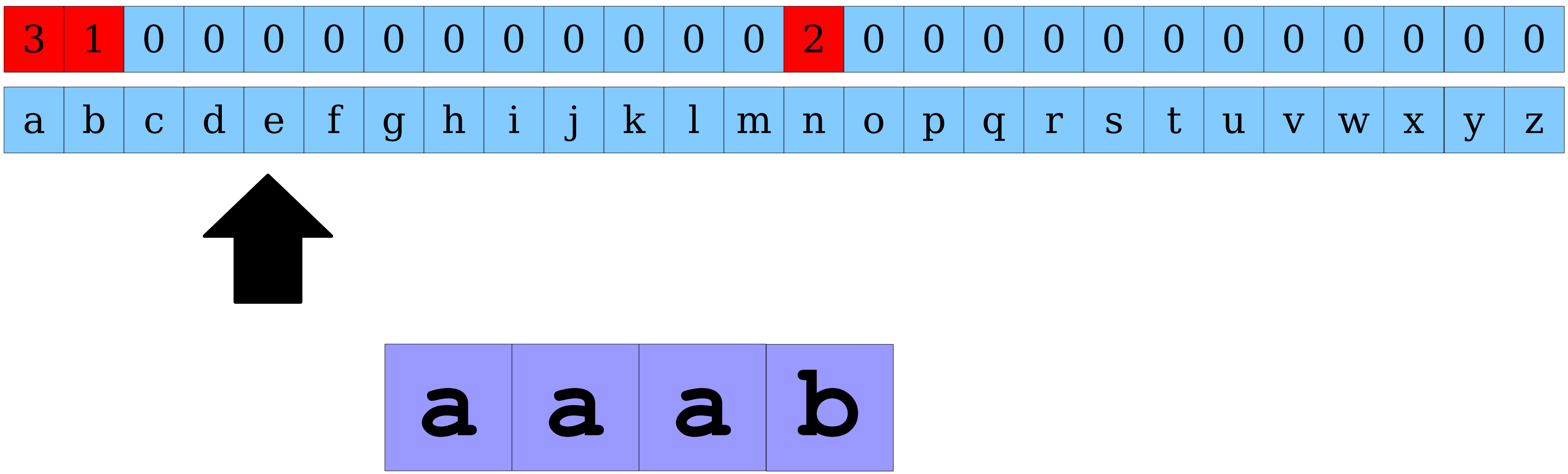

3 1 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0
a b c d e f g h i j k l m n o p q r s t u v w x y z
a a a b

Bucket Sort

3	1	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0
a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z

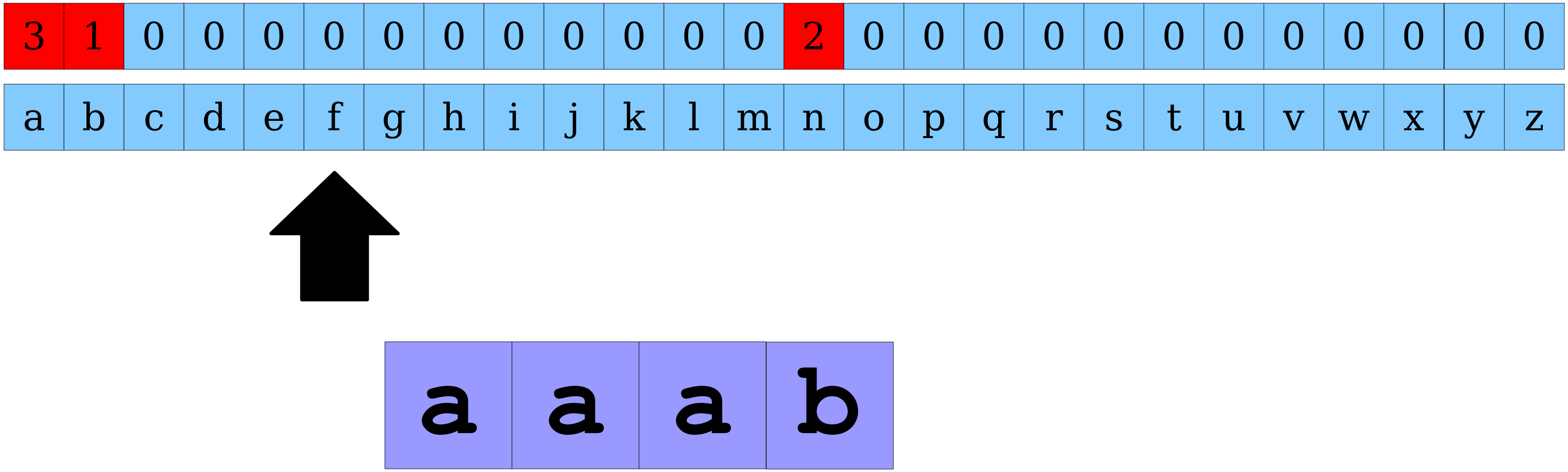
a	**a**	**a**	**b**

Bucket Sort

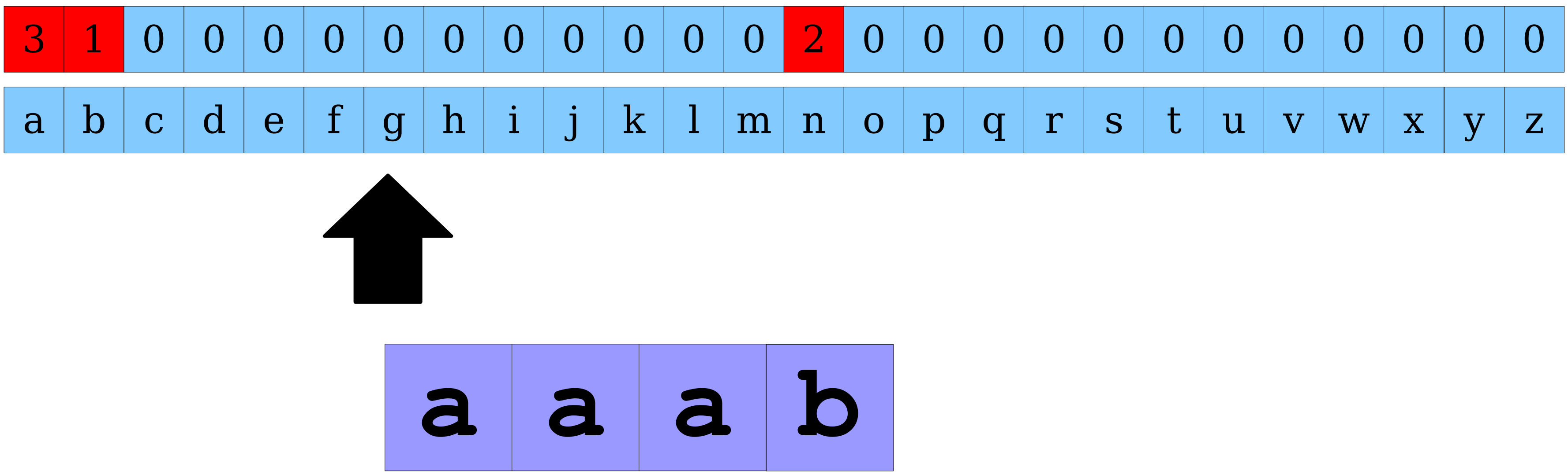


3 1 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0
a b c d e f g h i j k l m n o p q r s t u v w x y z
a a a b

Bucket Sort


3 1 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0
a b c d e f g h i j k l m n o p q r s t u v w x y z
a a a b

Bucket Sort


3 1 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0
a b c d e f g h i j k l m n o p q r s t u v w x y z
a a a b

Bucket Sort

3	1	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0
a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z

a	**a**	**a**	**b**	**n**	**n**

Bucket Sort

- Pseudocode:
  - Create an array **`histogram`** of length *d* where *d* is the number of possible values elements can take in the original array.
  - For each element in the array we're sorting, update the **`histogram`**
  - For each index in the **`histogram`**, output the corresponding element **`histogram[i]`** times
- Runtime?
  - **O(*d* + *n*)**
- Generally used if *d* is small (e.g. **`char`**)

Bucket Sort for `ints`

0	0	0	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9	10

…

0	0	0	0	0	0
2^{32}-6	2^{32}-5	2^{32}-4	2^{32}-3	2^{32}-2	2^{32}-1

8	**112**	**240**	**62**	**987**	**500**

Limits of Programs

- There are three I want to consider:
  - What can't a computer *do any faster*?
  - **What can't a computer *do fast*?**
  - What can't a computer do at all?

Traveling Salesperson

https://www.google.com/maps/vt/data=VLHX1wd2Cgu8wR6jwyh-km8JBWAkEzU4,2bUCUBVs3YYr-KB4ccFl-1Q1nWYcyKzmW0Ggf8ar4OOyEuuN9txRnTiKzIvmH6qy6B4vSoZvopndG7VjMIsOIDayhdkqKblOykP1wZYm9RcF8-Y6pkecPwDi3xc98B3gNGLchfR7xnPKzCGEmRocrv9OczmELzORvRseZHLyjWOvL0GzUeg0WFJGA4Y

Traveling Salesperson

https://www.google.com/maps/vt/data=VLHX1wd2Cgu8wR6jwyh-km8JBWAkEzU4,2bUCUBVs3YYr-KB4ccFl-1Q1nWYcyKzmW0Ggf8ar4OOyEuuN9txRnTiKzIvmH6qy6B4vSoZvopndG7VjMIsOIDayhdkqKbIOykP1wZYm9RcF8-Y6pkecPwDi3xc98B3gNGLchfR7xnPKzCGEmRocrv9OczmELzORvRseZHLyjWOvL0GzUeg0WFJGA4Y

Traveling Salesperson

- Find a minimal cost tour (visits every city and returns to starting city)
- How can we solve this?
- Algorithm 1: Consider all possible permutations of cities and return the cheapest permutation.
  - Worst case **$O(n!)$**
- Algorithm 2: Dynamic Programming.
  - Technique similar in spirit to memoization except you build up longer and longer paths
  - Worst case **$O(2^n)$**

Traveling Salesperson

- **$O(n!)$** and **$O(2^n)$** are both **exponential runtimes**
  - i.e. The runtime of the algorithm grows exponential in the size of the input
- How long it takes to compute depends on constant factors, but if each operation takes 1 millisecond…

Comparison of Runtimes

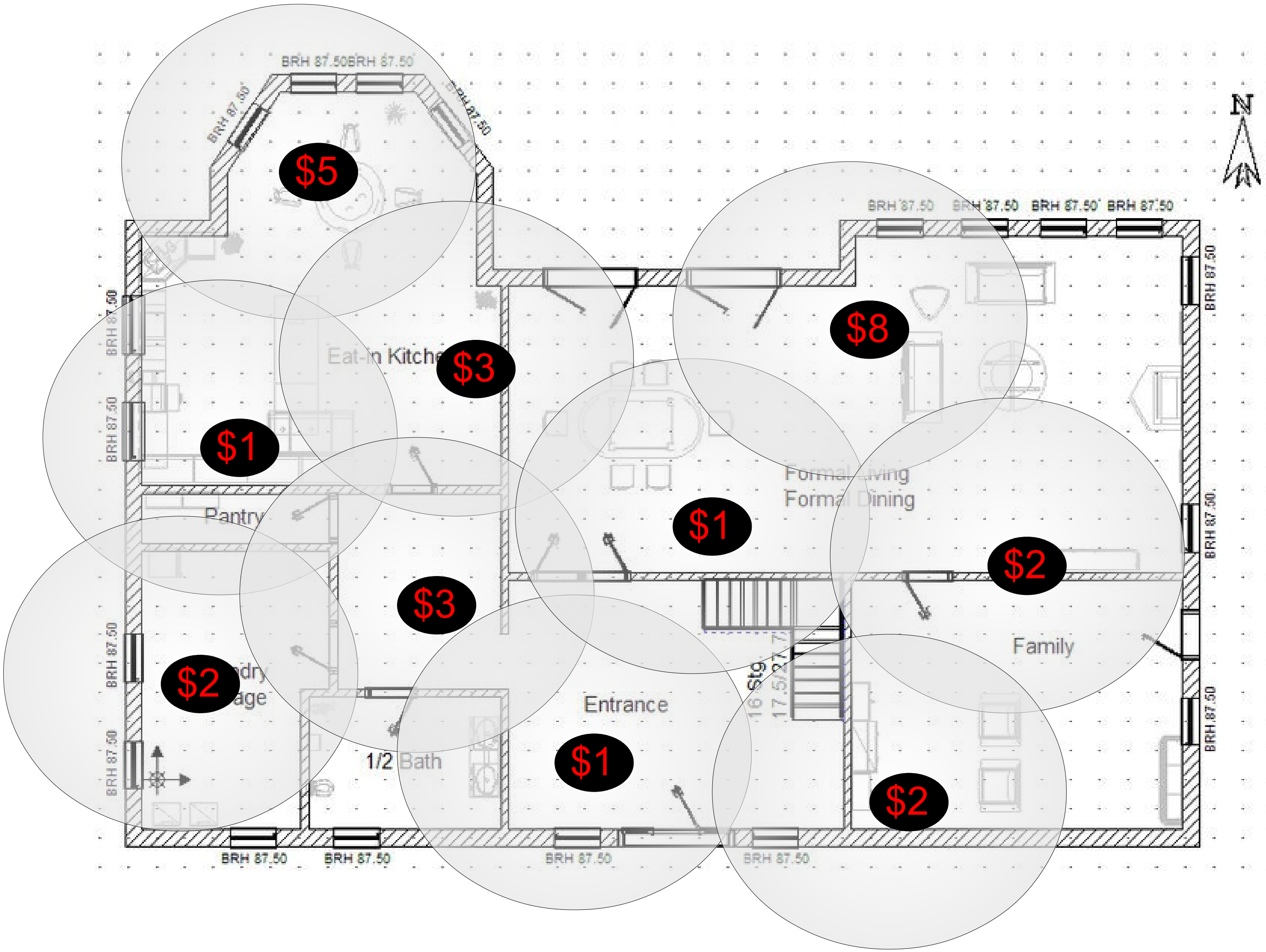
(1 operation = 1 microsecond)

Size	n	n log n	n^2	n^3	2^n	n!
10	10μs	33μs	100μs	1ms	1ms	1 hour
20	20μs	86μs	400μs	8ms	17min	8 years
30	30μs	147μs	900μs	27ms	12 days	2 sixtillion years
40	40μs	212μs	1.6ms	64ms	34 years	...
50	50μs	282μs	2.5ms	125ms	$3.56e^2$ years	
60	60μs	354μs	3.6ms	216ms	$3.65e^7$ years	
70	70μs	429μs	4.9ms	343ms	$3.74e^{10}$ years	
80	80μs	506μs	6.4ms	512ms	$3.83e^{13}$ years	
90	90μs	584μs	8.1ms	729ms	$3.92e^{16}$ years	
100	100μs	664μs	10ms	1s	40 quintillion years	

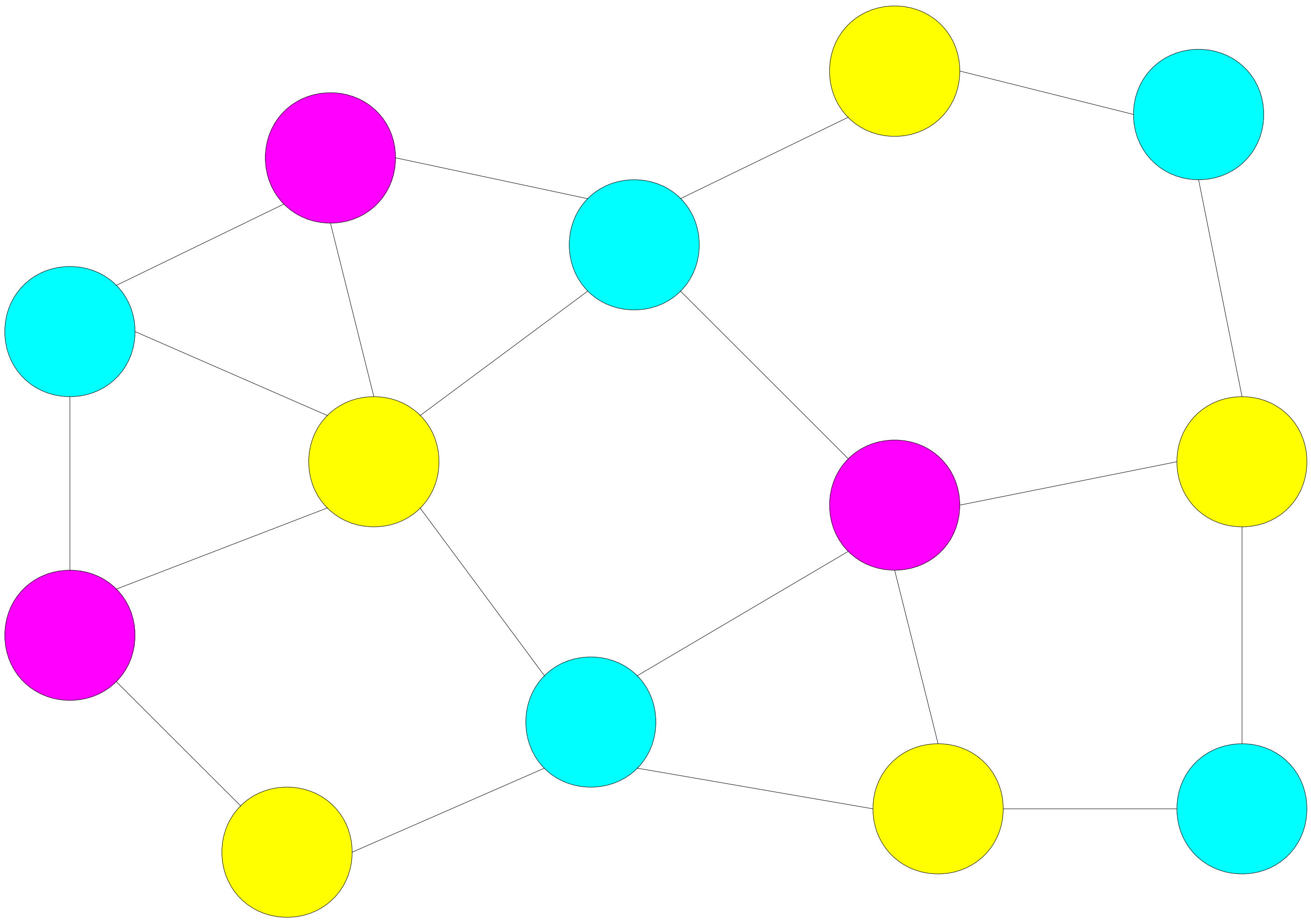
Traveling Salesperson

- There are many problems in which the best known algorithms run in worst case exponential time…

Sensor Placement



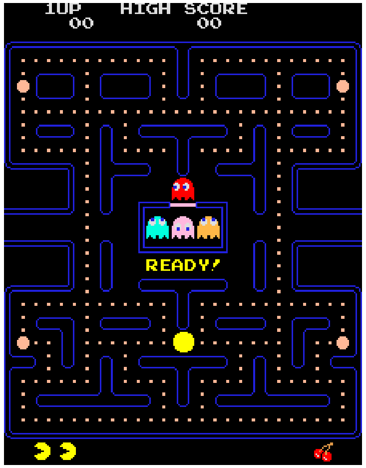
Graph Coloring



Games…

http://kickdes.files.wordpress.com/2011/04/classicbattleship.jpg

http://alum.mit.edu/pages/sliceofmit/files/2012/03/SuperMarioBros.jpg



http://www.technologyreview.com/blog/arxiv/files/80466/Pac-Man.png

Complexity Classes

- In Complexity Theory computing problems are put into different **complexity classes**
- **P**: The set of problems that can be solved in polynomial time
  - e.g. sorting, searching an array for a value
- **NP**: The set of problems that can be solved in exponential time
  - e.g. Traveling Salesperson, Graph Coloring
- It has not been proved, but it's assumed that **P** != **NP**

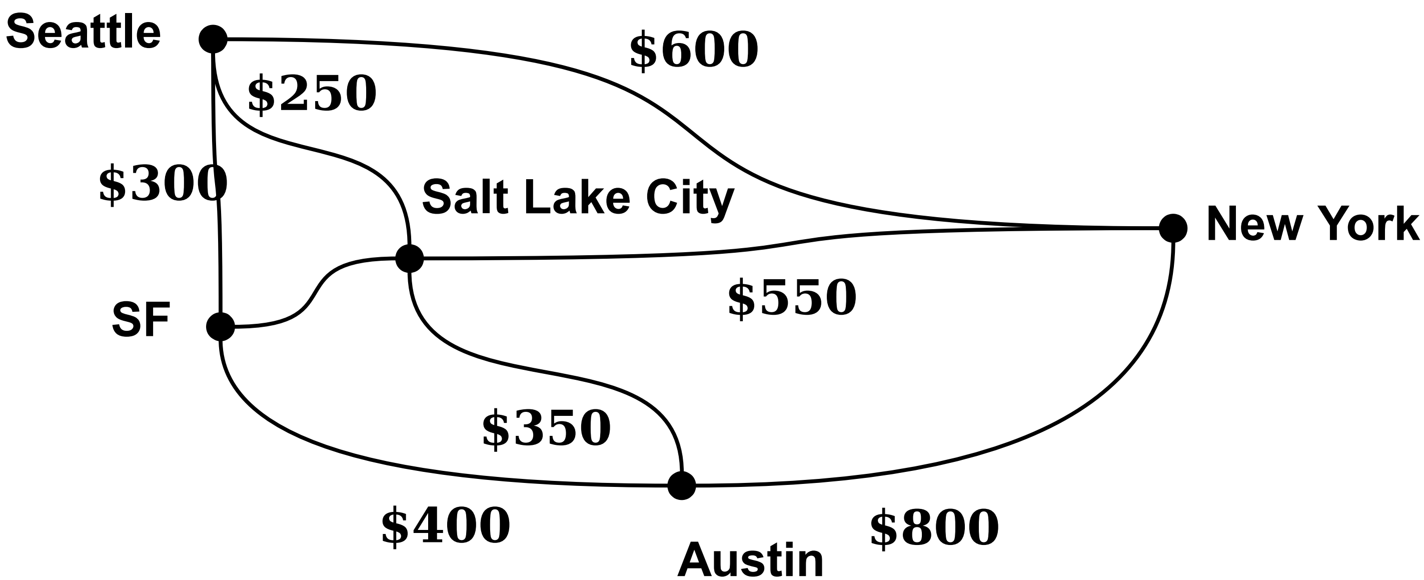
Beating Exponential Time

- We have two options to beat exponential time algorithms:
  - Approximation Algorithms
  - Heuristics

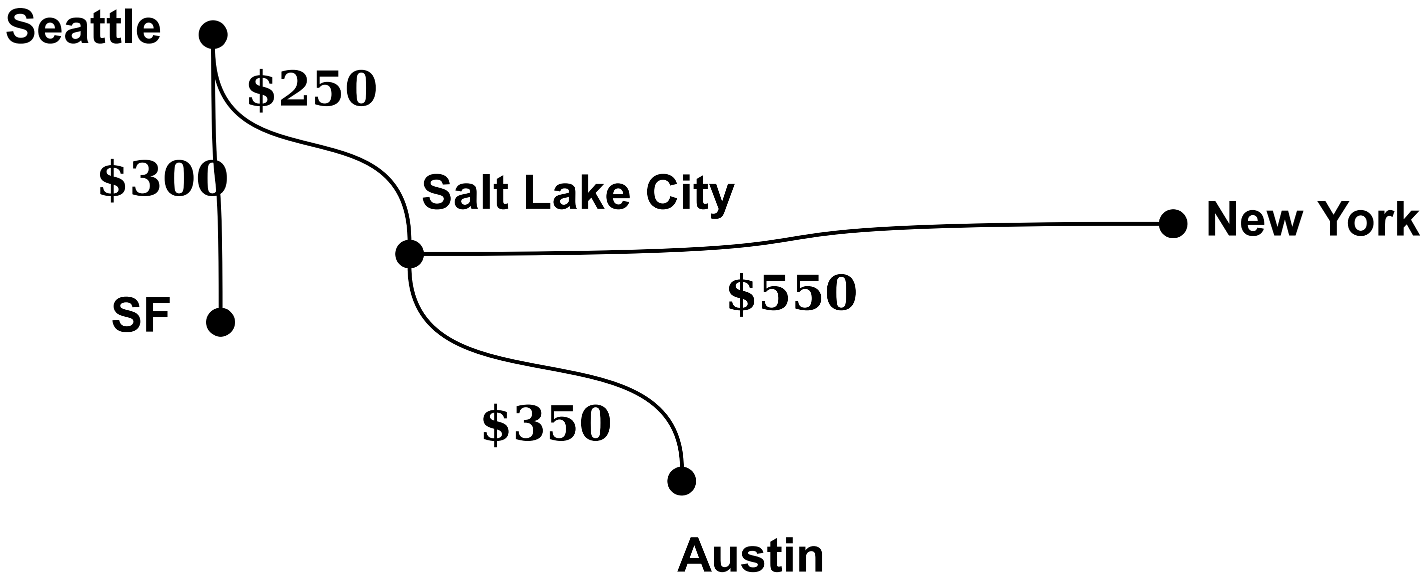
Approximation Algorithms

- A **k-Approximation Algorithm** is an algorithm that you can prove gets within a factor k of an optimal solution in the worst case
- A simple 2-Approxmiation Algorithm for traveling salesperson…
  - Compute a Minimum Spanning Tree of the graph and return a “depth first” path of the tree

2-Approximation TSP



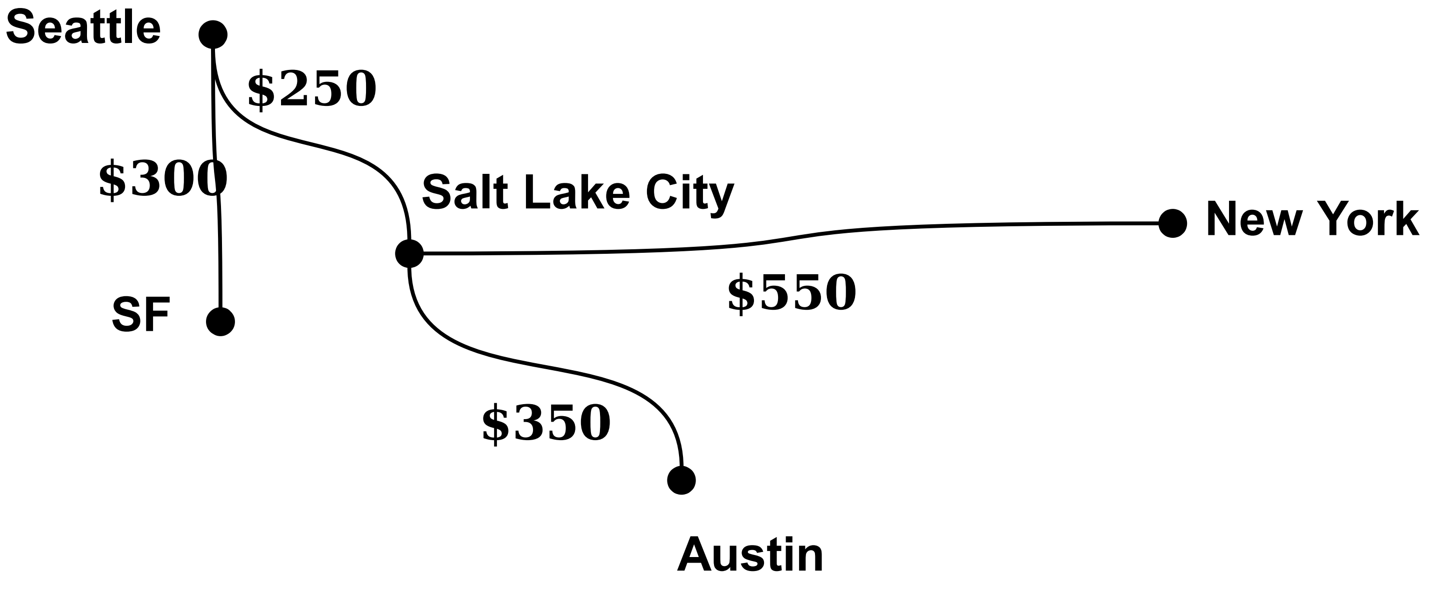
2-Approximation TSP



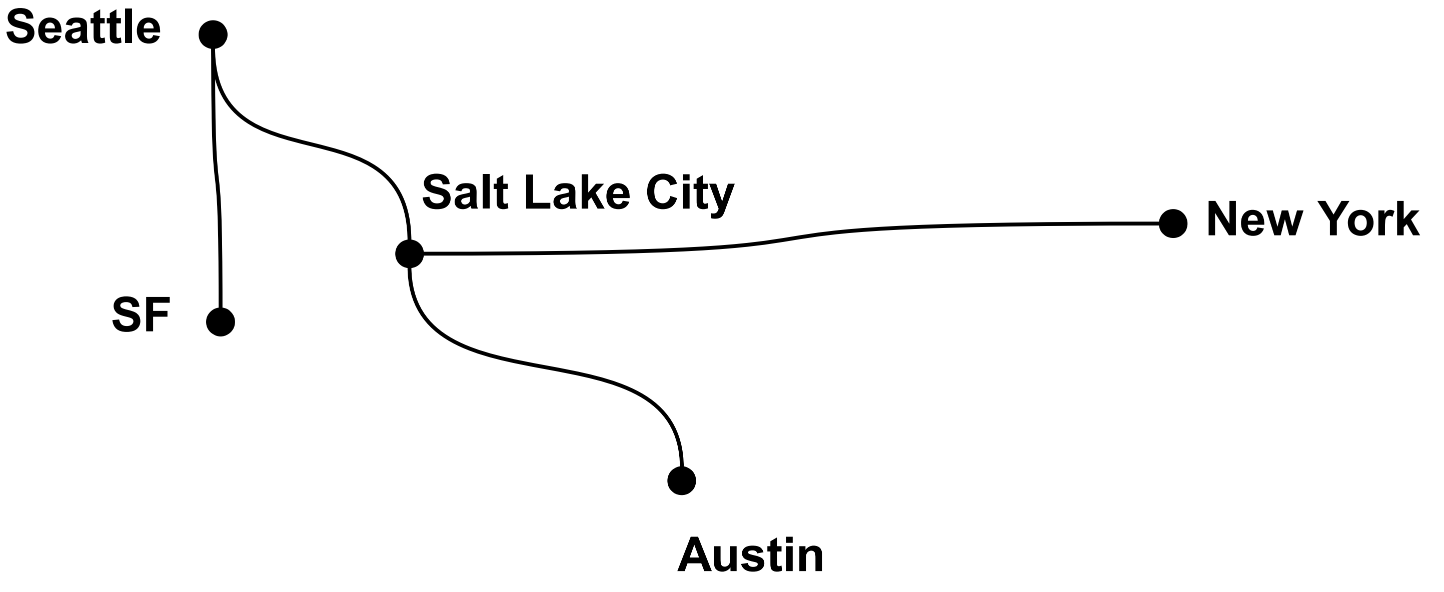
WHY????

- Remember we are computing an optimal tour – visit every node at least once and end at the starting node.
- The cost of **every** optimal tour is going to be less than the cost of a Minimum Spanning Tree
- The cost of our MST is a **lower bound** of the cost of an optimal tour

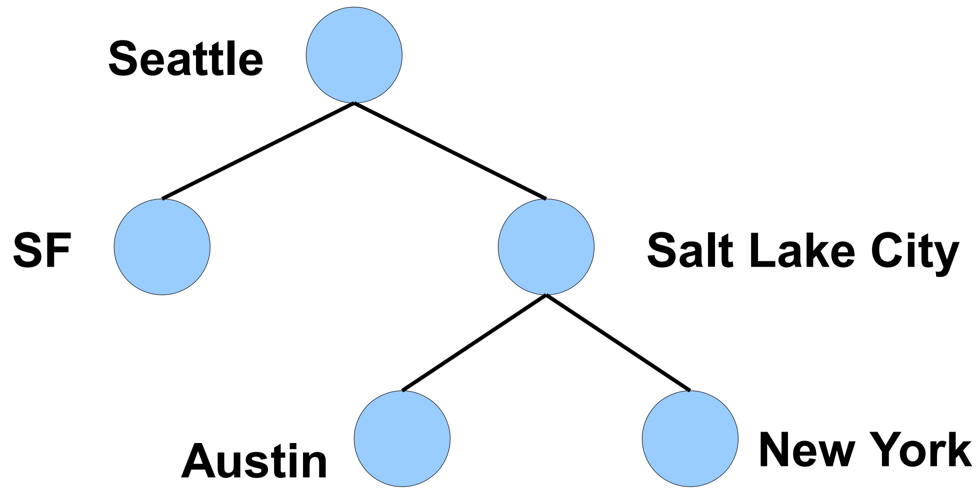
2-Approximation TSP



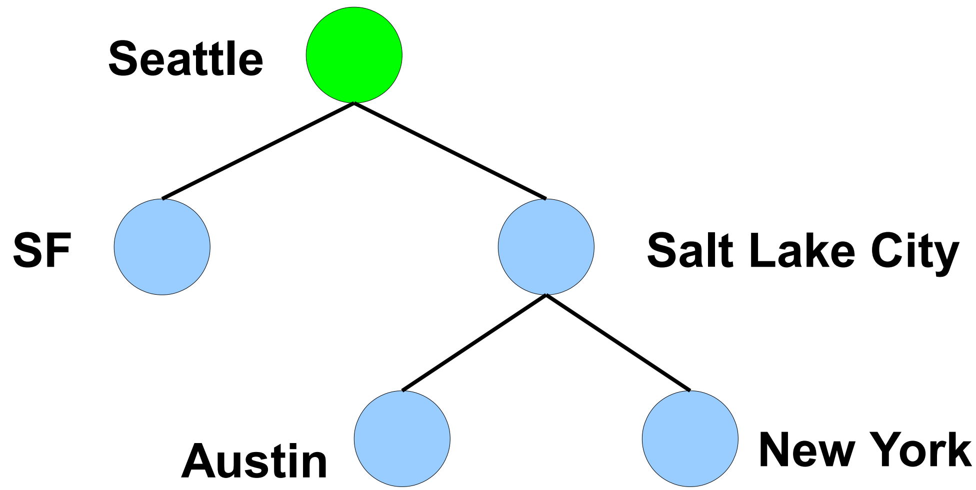
2-Approximation TSP



2-Approximation TSP

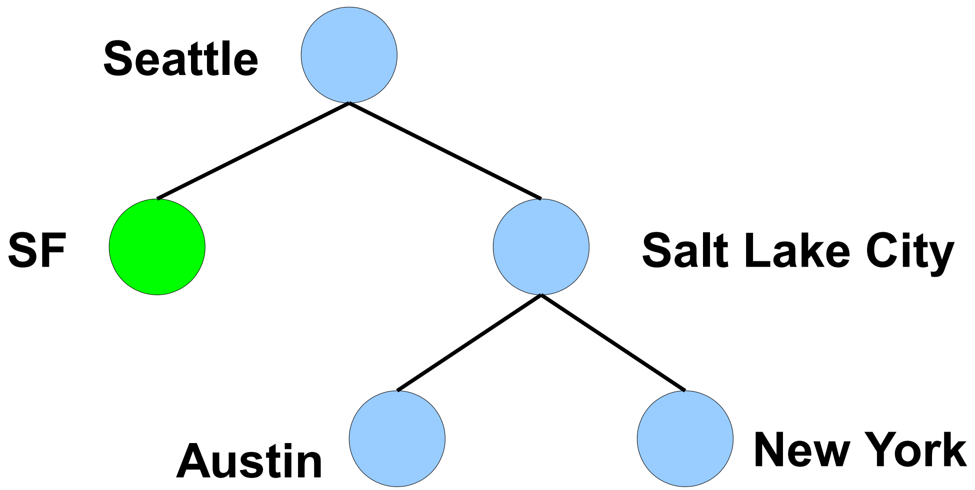


2-Approximation TSP



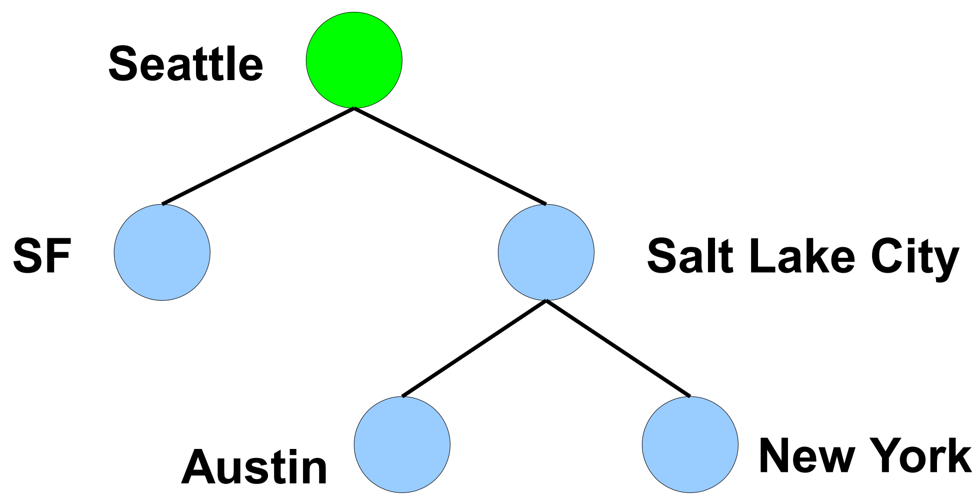
Seattle

2-Approximation TSP



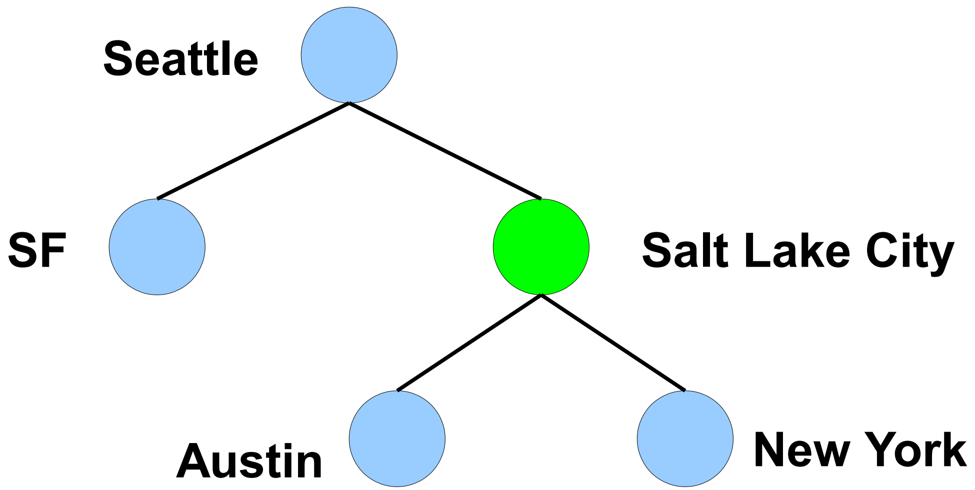
Seattle → SF

2-Approximation TSP



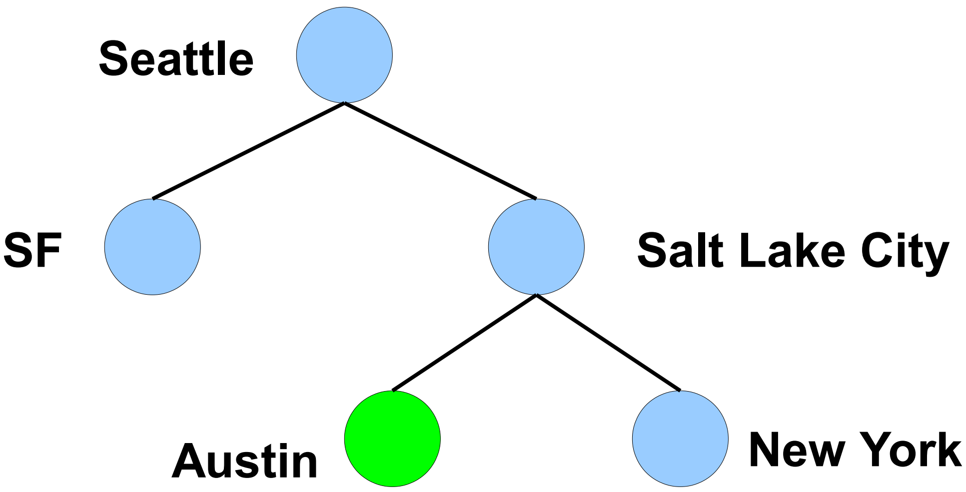
Seattle → SF → Seattle

2-Approximation TSP



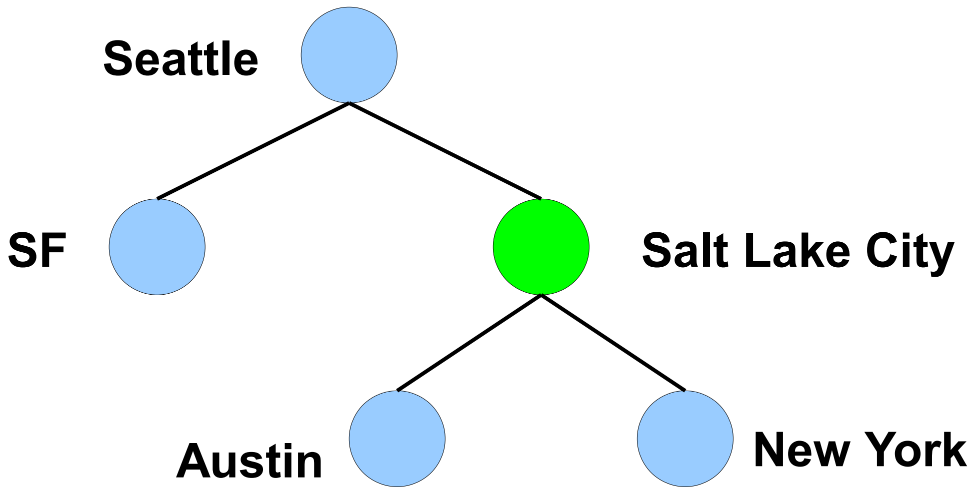
Seattle → SF → Seattle → SLC

2-Approximation TSP



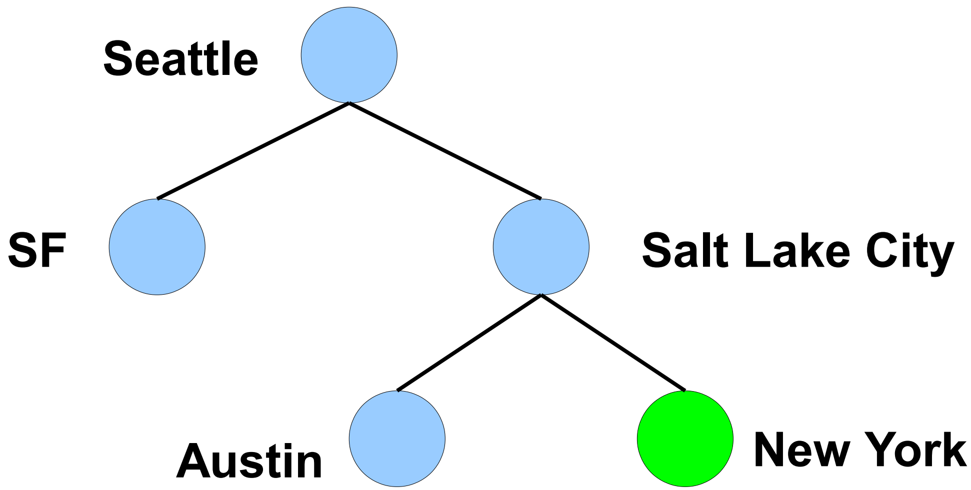
Seattle → SF → Seattle → SLC → Austin

2-Approximation TSP



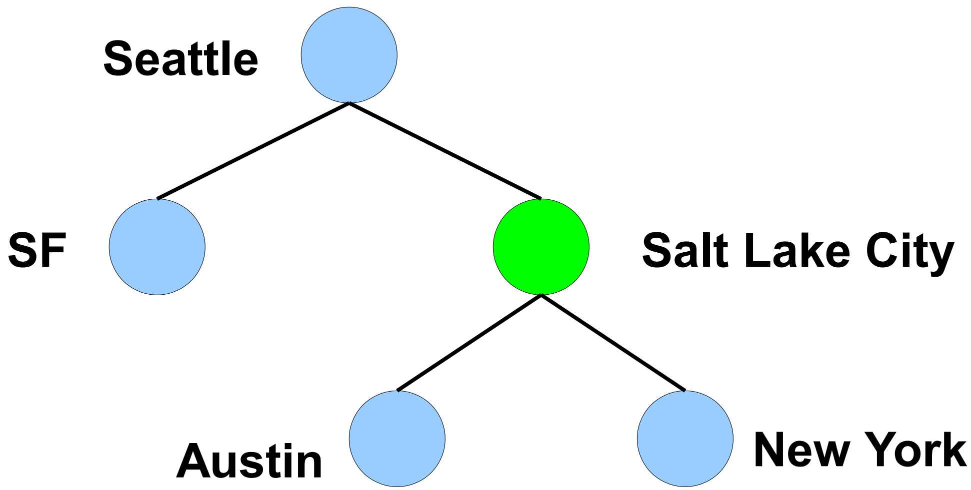
Seattle → SF → Seattle → SLC → Austin → SLC

2-Approximation TSP



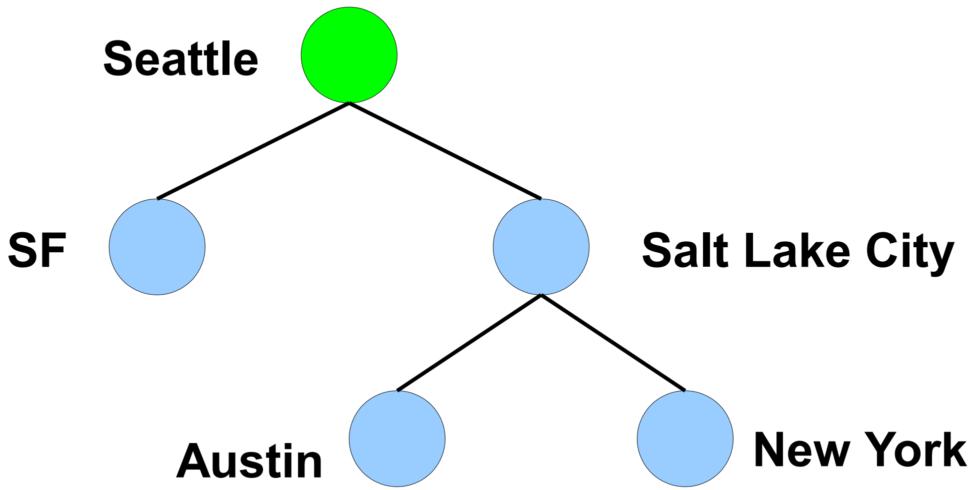
Seattle → SF → Seattle → SLC → Austin → SLC → NY

2-Approximation TSP



Seattle → SF → Seattle → SLC → Austin → SLC → NY → SLC

2-Approximation TSP



Seattle → SF → Seattle → SLC → Austin → SLC → NY → SLC → Seattle

2-Approximation TSP

- Because we use every edge twice, the cost of this tour is going to be twice the cost of the MST.
- The cost of the MST is less than or equal to the cost of an optimal tour.
- Therefore, the cost of our tour is less than or equal to twice the cost of an optimal tour.
  - Hence, this is a 2-approximation

Approximation Algorithms

- Better approximation algorithms exist for TSP (but they are more difficult to prove)
- Many approximation algorithms exist for different problems in **NP**

Heuristics

- A different Idea: Construct a heuristic that will give a “good” solution.
  - Even if it performs terribly in the “worst case”, it may perform well in “most” cases.
- **Nearest Neighbor Heuristic**
  - Iteratively extend path by picking cheapest edge that will get us to an unvisited node
  - Works reasonably well with high probability
  - Has terrible worst case behavior.
    - Okay because worst case is unlikely

Limits of Programs

- There are three I want to consider:
  - What can't a computer *do any faster*?
  - What can't a computer *do fast*?
  - **What can't a computer do at all?**

A Useful Tool

- It would be incredibly useful if Visual Studio and Xcode would detect the following issues before running a program:
  - Infinite Loops/Recursion
  - Memory Leaks
  - Issues dereferencing NULL and uninitialized pointers
  - Automatic grading of assignments

Problem: It is *impossible* to write a program which can, for all input programs, successfully do these tasks!

A Useful Tool

- Example: It is impossible to write a program that, given any program and input, detects if the program will terminate on that input.
  - Called the **Halting Problem**
  - We say that the Halting Problem is **undecideable**
- Wait…really?

What about this?

```
int main() {
    while (true) {
        cout << “Counter Example?” << endl;
    }
}
```

Or this?

```
int main() {
    for (int i = 0; i < 10; i++)
        cout << "This isn't hard!" << endl;
}
```

Or even this?

```
int main() {
    return 0;
}
```

Halting Problem

- For many program-input pairs we can easily tell if they terminate.
- We cannot do this for **all** programs.
- So how can we construct one?
  - It's tricky. Take CS161 to learn more about this.
- We're just going to go over the intuition...

Proof Sketch

- The way we prove the Halting Problem is undecideable is through **proof by contradiction**: we start by assuming that it is decideable then derive a contradiction.
  - Common proof technique for proving something *cannot* exist
- Proof Sketch:
  - Assume a program **P** exists that solves the halting problem *for all inputs*
  - Construct a new program **Q** from **P**
  - Show **P** cannot decide if **Q** terminates

Proof Intuition

- Constructing **Q** from **P** is the heart of the proof.
- It's somewhat confusing, but is similar in spirit to the following contradiction:
  - "The barber of Seville shaves everyone in Seville who doesn't shave himself. Does the barber shave himself?"
- Idea is to run **P** with input **P**

Halting Problem

- As a corollary, many other useful questions regarding arbitrary programs are also undecideable:
  - Memory Leaks?
  - Dereferencing NULL pointers?
  - Many many more...

How Bad is This?

- As a result of this we run into some issues...
  - Can't prove arbitrary programs are correct – need to test them
  - Tools to detect memory leaks don't catch everything
- Modern tools that detect these types of issues can't detect everything, but can still be useful.

Tomorrow

- Introduction to **Machine Learning**