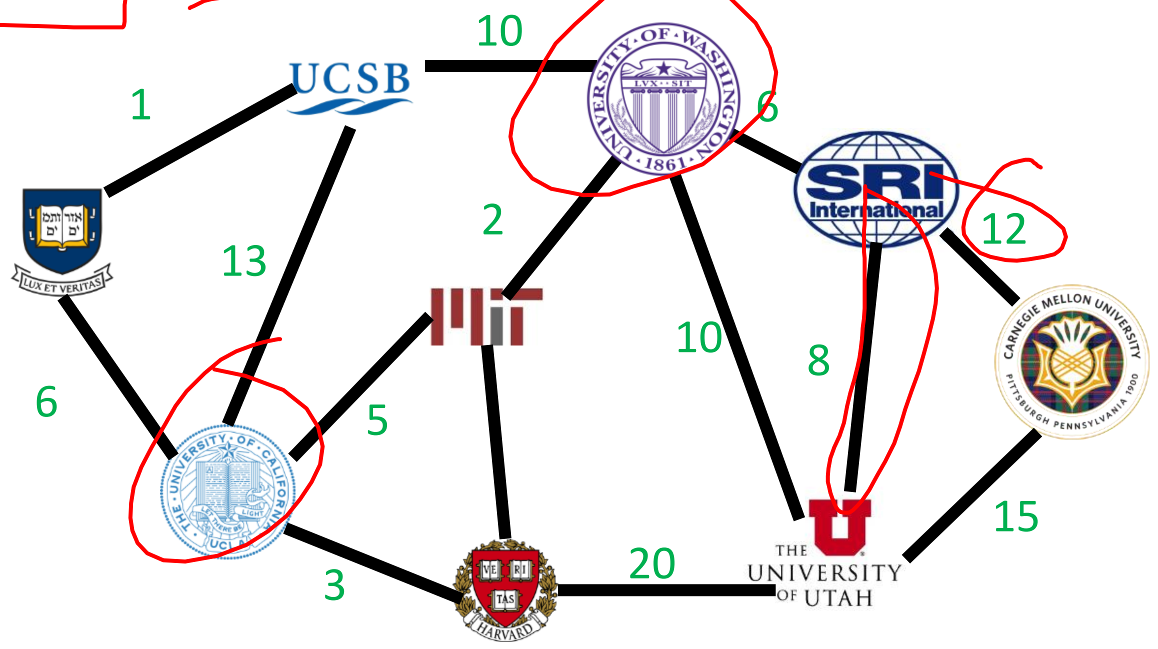
CSE 332 Winter 2024
Lecture 18: Dijkstra's, ForkJoin

Nathan Brunelle

http://www.cs.uw.edu/332

Single-Source Shortest Path



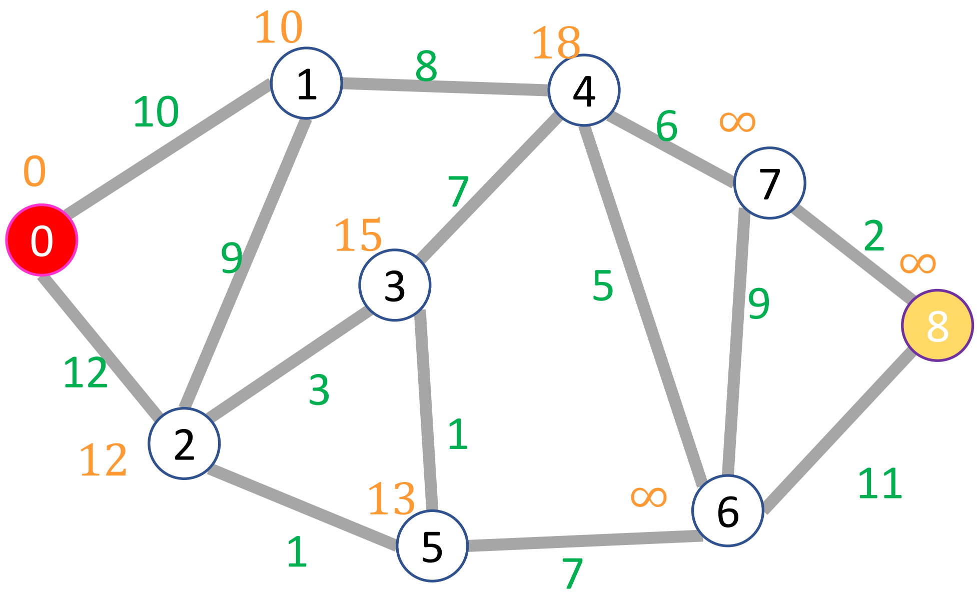
Find the quickest way to get from UVA to each of these other places

Given a graph $G = (V, E)$ and a start node $s \in V$, for each $v \in V$ find the least-weight path from $s \to v$ (call this weight $\delta(s, v)$)

(assumption: all edge weights are positive)

Dijkstra’s Algorithm

- Input: graph with **no negative edge weights**, start node s, end node t
- Behavior: Start with node s, repeatedly go to the incomplete node “nearest” to s, stop when
- Output:
  - Distance from start to end
  - Distance from start to every node



Dijkstra’s Algorithm

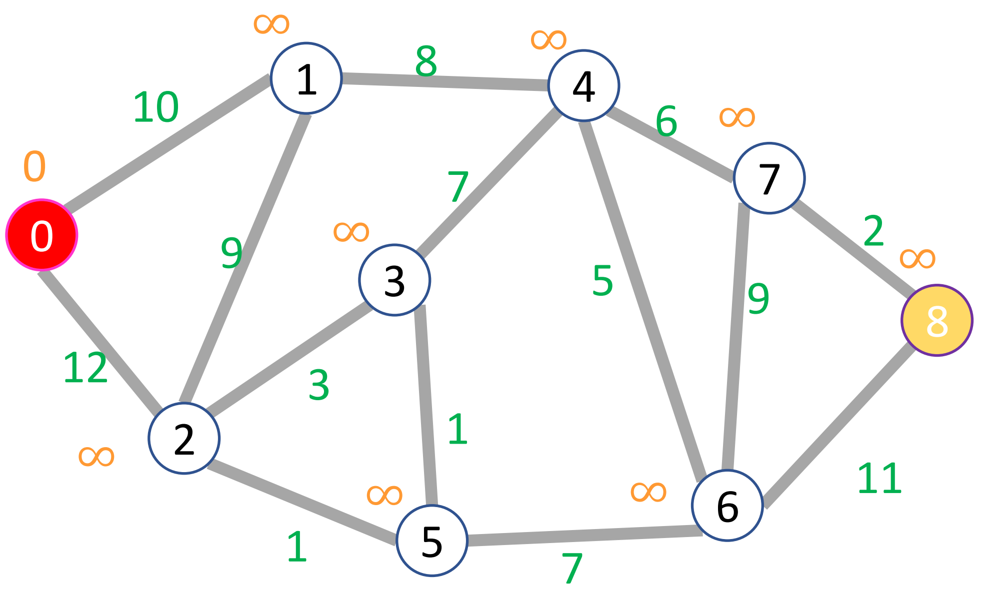
Start: 0

End: 8

Node	Done?
0	F
1	F
2	F
3	F
4	F
5	F
6	F
7	F
8	F

Node	Distance
0	0
1	∞
2	∞
3	∞
4	∞
5	∞
6	∞
7	∞
8	∞

Idea: When a node is the closest undiscovered thing to the start, we have found its shortest path



Dijkstra's Algorithm

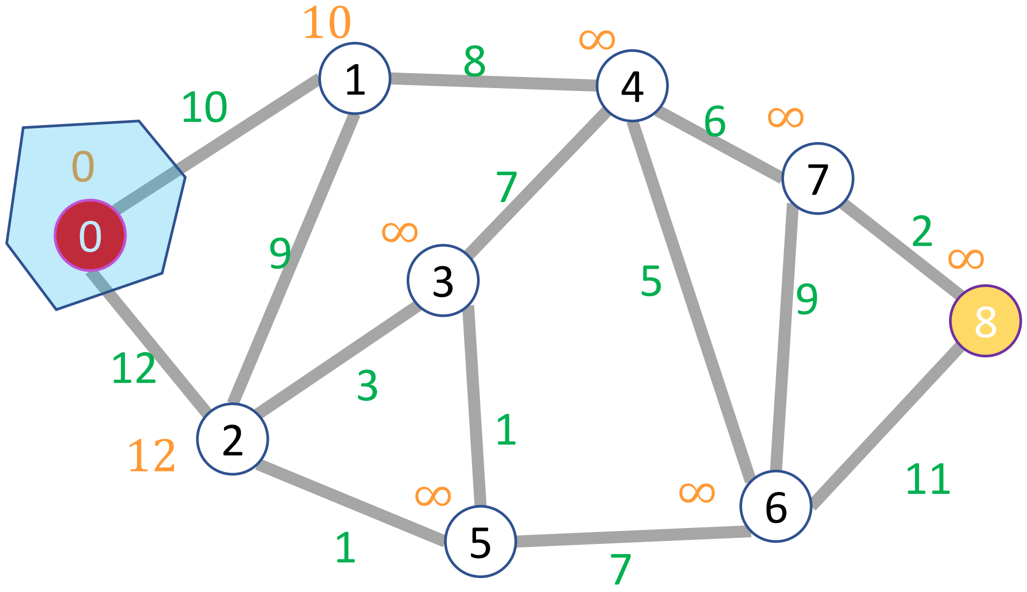
Start: 0

End: 8

Node	Done?
0	T
1	F
2	F
3	F
4	F
5	F
6	F
7	F
8	F

Node	Distance
0	0
1	10
2	12
3	∞
4	∞
5	∞
6	∞
7	∞
8	∞

Idea: When a node is the closest undiscovered thing to the start, we have found its shortest path



Dijkstra's Algorithm

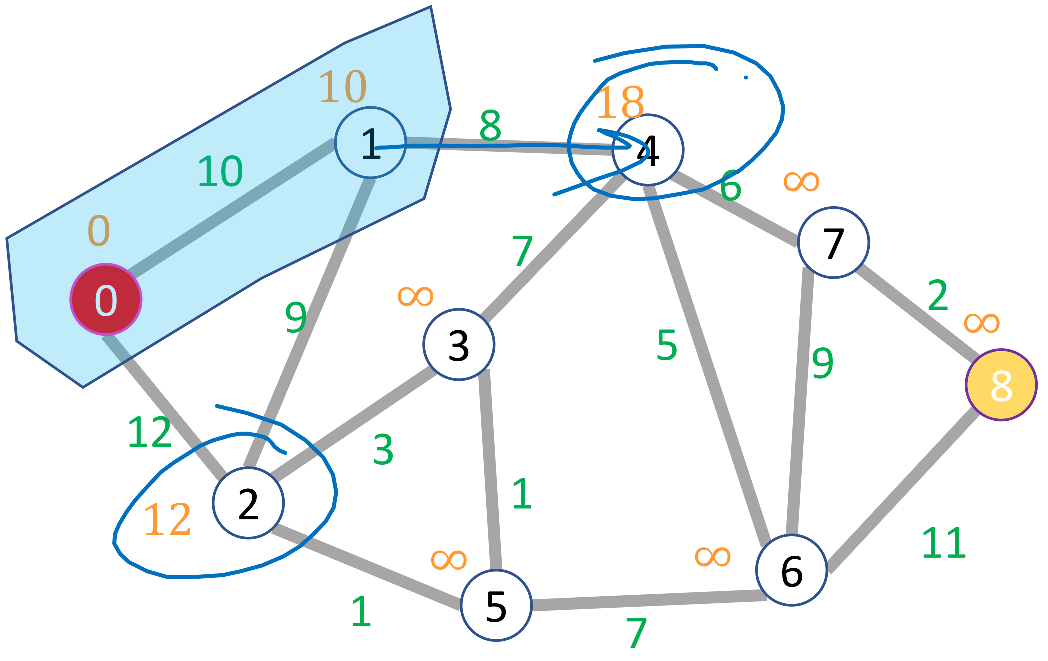
Start: 0

End: 8

Idea: When a node is the closest undiscovered thing to the start, we have found its shortest path

Node	Done?
0	T
1	T
2	F
3	F
4	F
5	F
6	F
7	F
8	F

Node	Distance
0	0
1	10
2	12
3	∞
4	18
5	∞
6	∞
7	∞
8	∞



Dijkstra’s Algorithm

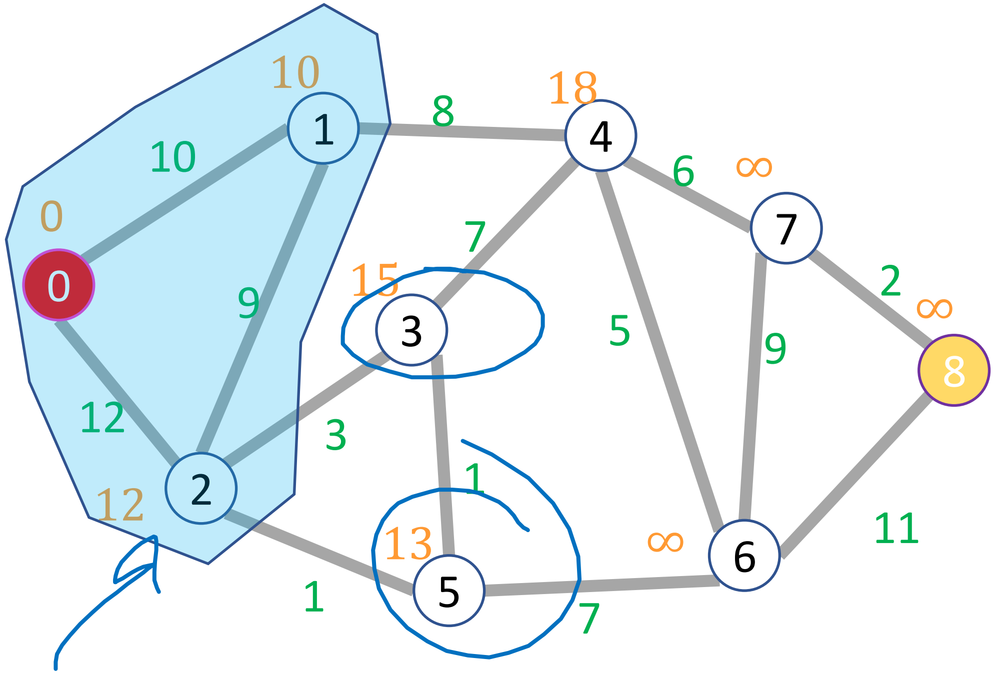
Start: 0

End: 8

Node	Done?
0	T
1	T
2	T
3	F
4	F
5	F
6	F
7	F
8	F

Node	Distance
0	0
1	10
2	12
3	15
4	18
5	13
6	∞
7	∞
8	∞

Idea: When a node is the closest undiscovered thing to the start, we have found its shortest path



Dijkstra’s Algorithm

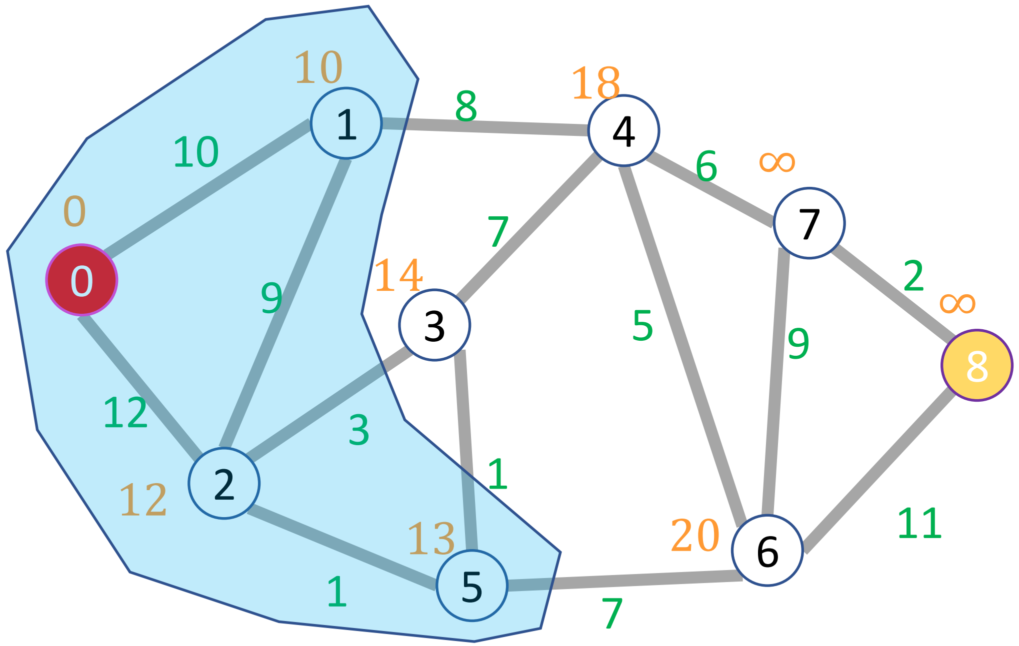
Start: 0

End: 8

Idea: When a node is the closest undiscovered thing to the start, we have found its shortest path

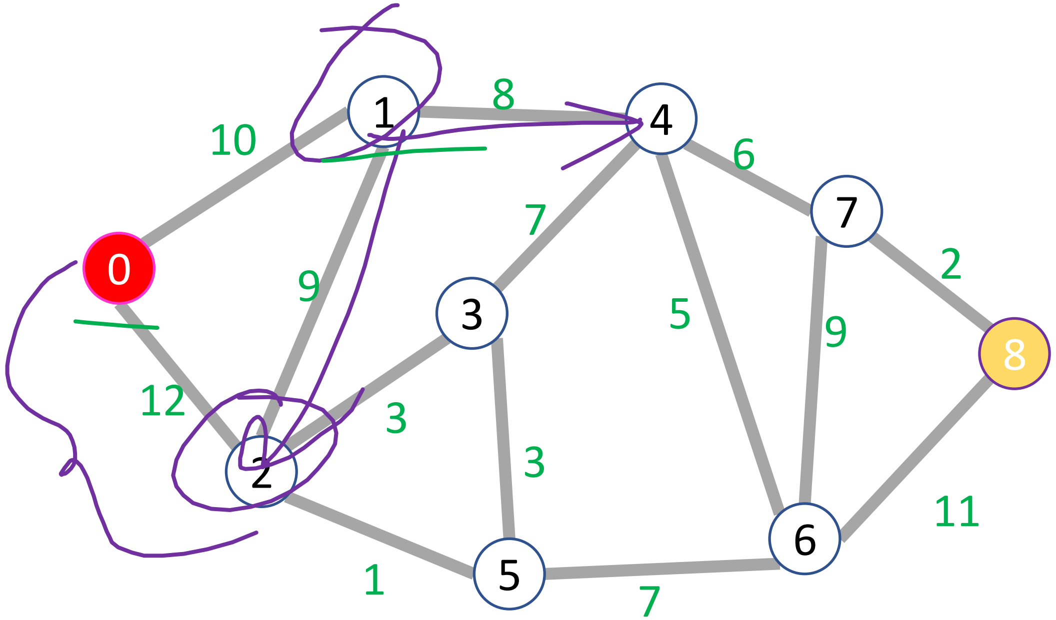
Node	Done?
0	T
1	T
2	T
3	F
4	F
5	T
6	F
7	F
8	F

Node	Distance
0	0
1	10
2	12
3	14
4	18
5	13
6	∞
7	20
8	∞



Dijkstra's Algorithm

```
int dijkstras(graph, start, end){
    distances = [∞, ∞, ∞,…];  // one index per node
    done = [False,False,False,…];  // one index per node
    PQ = new minheap();
    PQ.insert(0, start);  // priority=0, value=start
    distances[start] = 0;
    while (!PQ.isEmpty){
        current = PQ.deleteMin();
        done[current] = true;
        for (neighbor : current.neighbors){
            if (!done[neighbor]){
                new_dist = distances[current]+weight(current,neighbor);
                if new_dist < distances[neighbor]{
                    distances[neighbor] = new_dist;
                    PQ.decreaseKey(new_dist,neighbor); }
            }
        }
    }
    return distances[end]
}
```



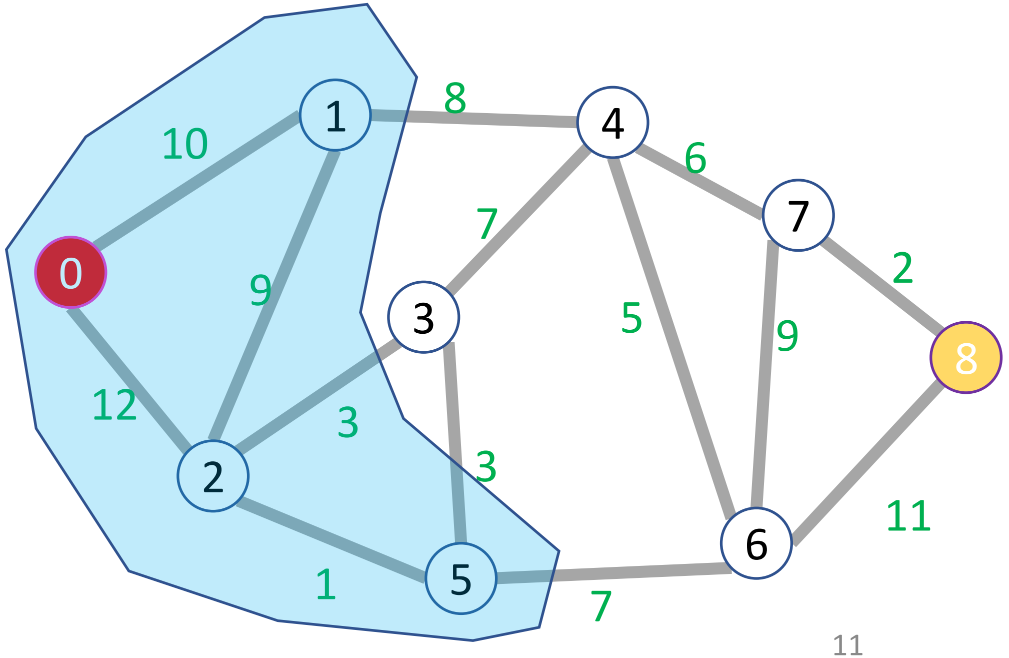
2E

Dijkstra's Algorithm: Running Time

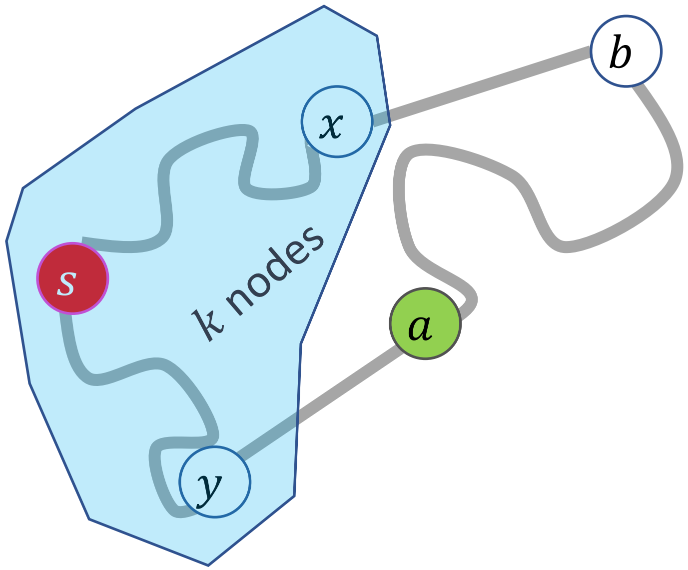
- How many total priority queue operations are necessary?
  - How many times is each node added to the priority queue?
  - How many times might a node's priority be changed?
- What's the running time of each priority queue operation?
  - $\log|V|$ $\cdot 2E$
- Overall running time:
  - $\Theta(|E|\log|V|)$

Dijkstra's Algorithm: Correctness

- Claim: when a node is removed from the priority queue, we have found its shortest path
- Induction over number of completed nodes
- Base Case:
  - 1 node done: we removed Start
    - Start node is 0 from itself always
- Inductive Step:
  - Assume that i nodes are done and we
    - Found their shortest paths
  - Want to show that when node $i+1$
    - was removed, we found its shortest



Dijkstra's Algorithm: Correctness



- Claim: when a node is removed from the priority queue, its distance is that of the shortest path
- Induction over number of completed nodes
- Base Case: Only the start node removed
  - It is indeed 0 away from itself
- Inductive Step:
  - If we have correctly found shortest paths for the first k nodes, then when we remove node $k + 1$ we have found its shortest path

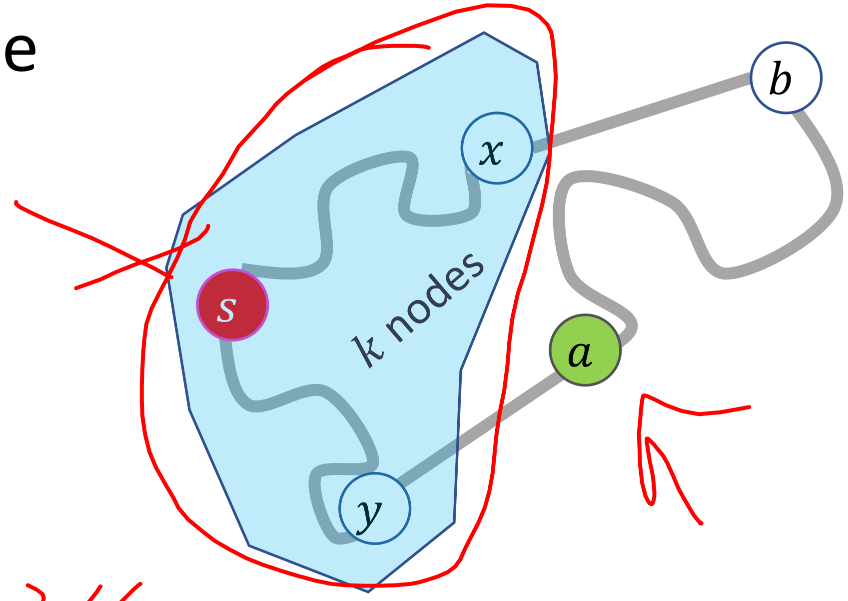
Dijkstra's Algorithm: Correctness

- Suppose a is the next node removed from the queue. What do we know bout a?

$- s \to y \to a <$

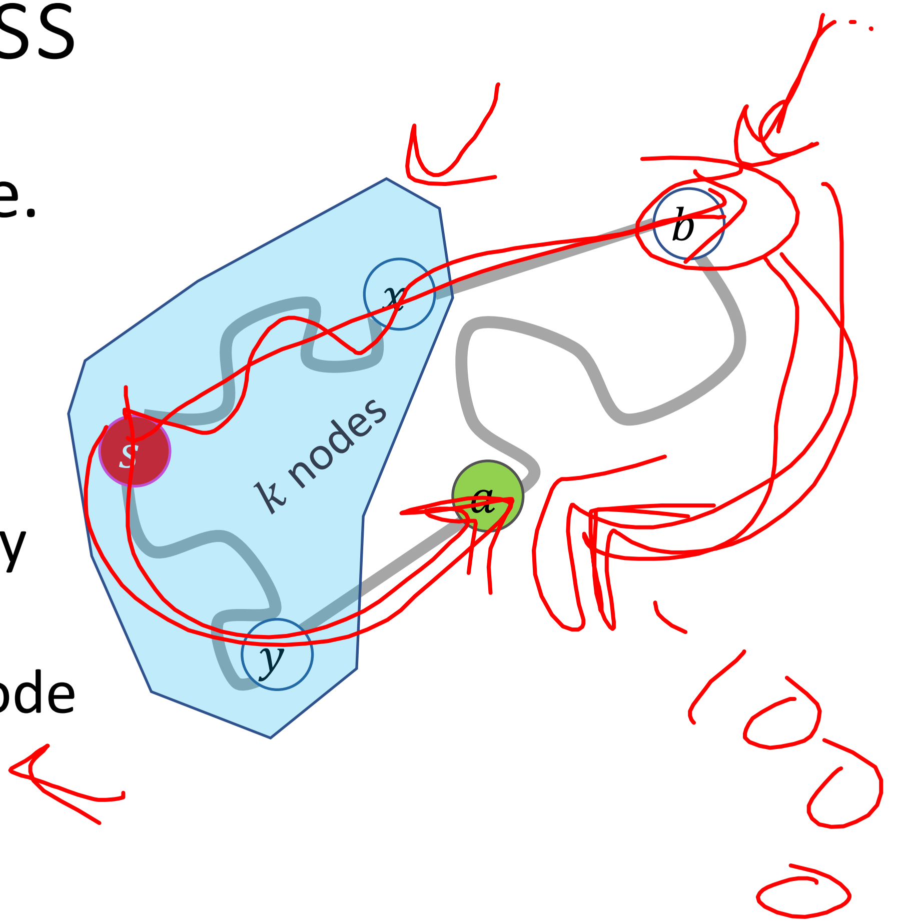
$s \to x \to b$

$- s \to a <$ any other path



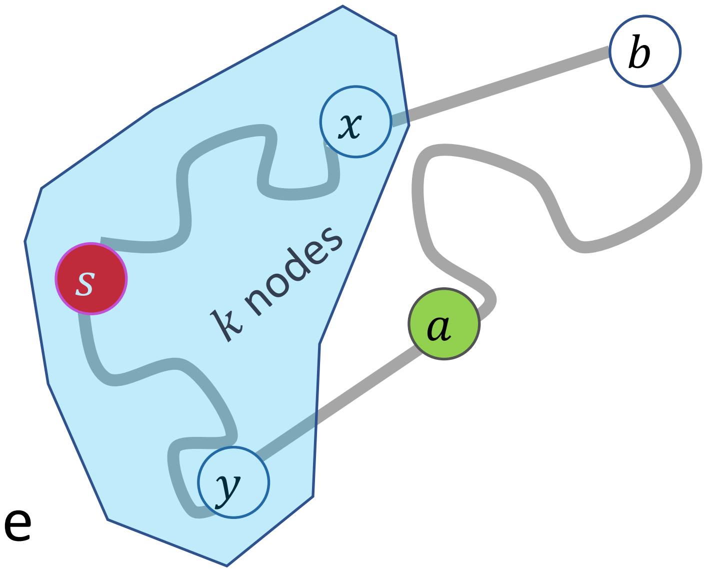
Dijkstra's Algorithm: Correctness

- Suppose a is the next node removed from the queue.
  - No other node incomplete node has a shorter path discovered so far
- Claim: no undiscovered path to a could be shorter
  - Consider any other incomplete node b that is 1 edge away from a complete node
  - a is the closest node that is one away from a complete node
  - Thus no path that includes b can be a shorter path to a
  - Therefore the shortest path to a must use only complete nodes, and therefore we have found it already!



Dijkstra’s Algorithm: Correctness

- Suppose a is the next node removed from the queue.
  - No other node incomplete node has a shorter path discovered so far
- Claim: no undiscovered path to a could be shorter
  - Consider any other incomplete node b that is 1 edge away from a complete node
  - a is the closest node that is one away from a complete node
  - No path from b to a can have negative weight
  - Thus no path that includes b can be a shorter path to a
  - Therefore the shortest path to a must use only complete nodes, and therefore we have found it already!



A Programming Assumption Reconsidered

- So far:
  - Programs run by executing one line of code at a time in the order written
  - Called **Sequential Programming**
- Removing this assumptions creates challenges and opportunities
  - Programming: Divide computation across several **parallel threads**, then coordinate (synchronize) across them.
  - Algorithms: This parallel processing can speed up computation by increasing **throughput** (operations done per unit time)
  - Data Structures: May need to support **concurrent** access (multiple parallel processes attempting to use it at once)

Why Parallelism?

moore

- Pre 2005:
  - Processors “naturally” got faster at an exponential rate (~2x faster every ~2 years)
- Since 2005:
  - Some components cannot be improved in the same way due to limitations of physics
  - Solution: increase computing speed by just adding more processors

What to do with the extra processors?

- Time Slicing:
  - Your computer is always keeping track of multiple things at once
    - running the OS, rendering the display, running Powerpoint, autosaving a document, etc.
  - Multiple processors allow for the multiple tasks to be spread across them, so each processor dedicates more time to each one
- Parallelism (our focus):
  - Multiple processors collaborate on the same task.

Parallelism Vs. Concurrency (with Potatoes)

- Sequential:
  - The task is completed by just one processor doing one thing at a time
  - There is one cook who peels all the potatoes
- Parallelism:
  - One task being completed by may threads
  - Recruit several cooks to peel a lot of potatoes faster
- Concurrency:
  - Parallel tasks using a shared resource
  - Several cooks are making their own recipes, but there is only 1 oven

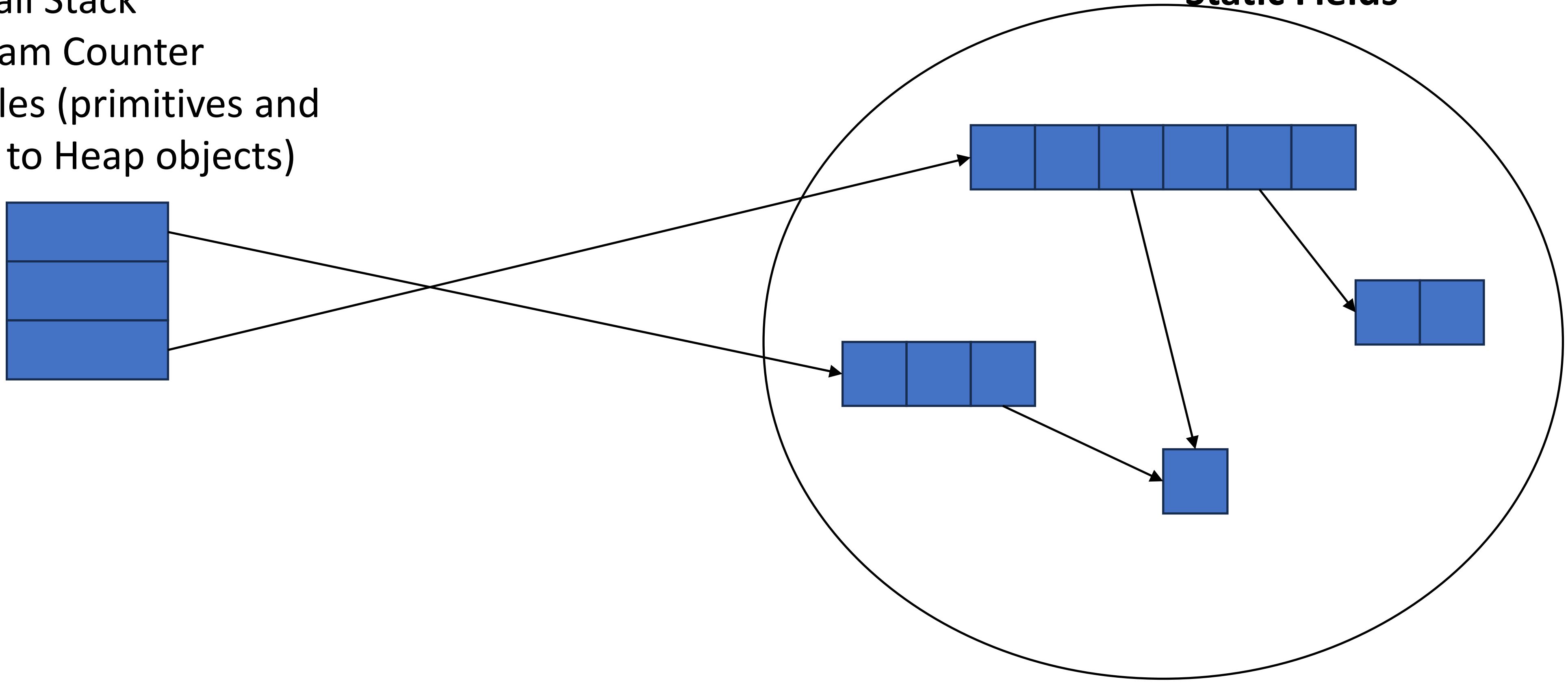
New Story of Code Execution

- Old Story:
  - One program counter (current statement executing)
  - One call stack (with each stack frame holding local variables)
  - Objects in the heap created by memory allocation (i.e., new)
    - (nothing to do with data structure called a heap)
- New Story:
  - Collection of threads each with its own:
    - Program Counter
    - Call Stack
    - Local Variables
    - References to objects in a shared heap

Old Story

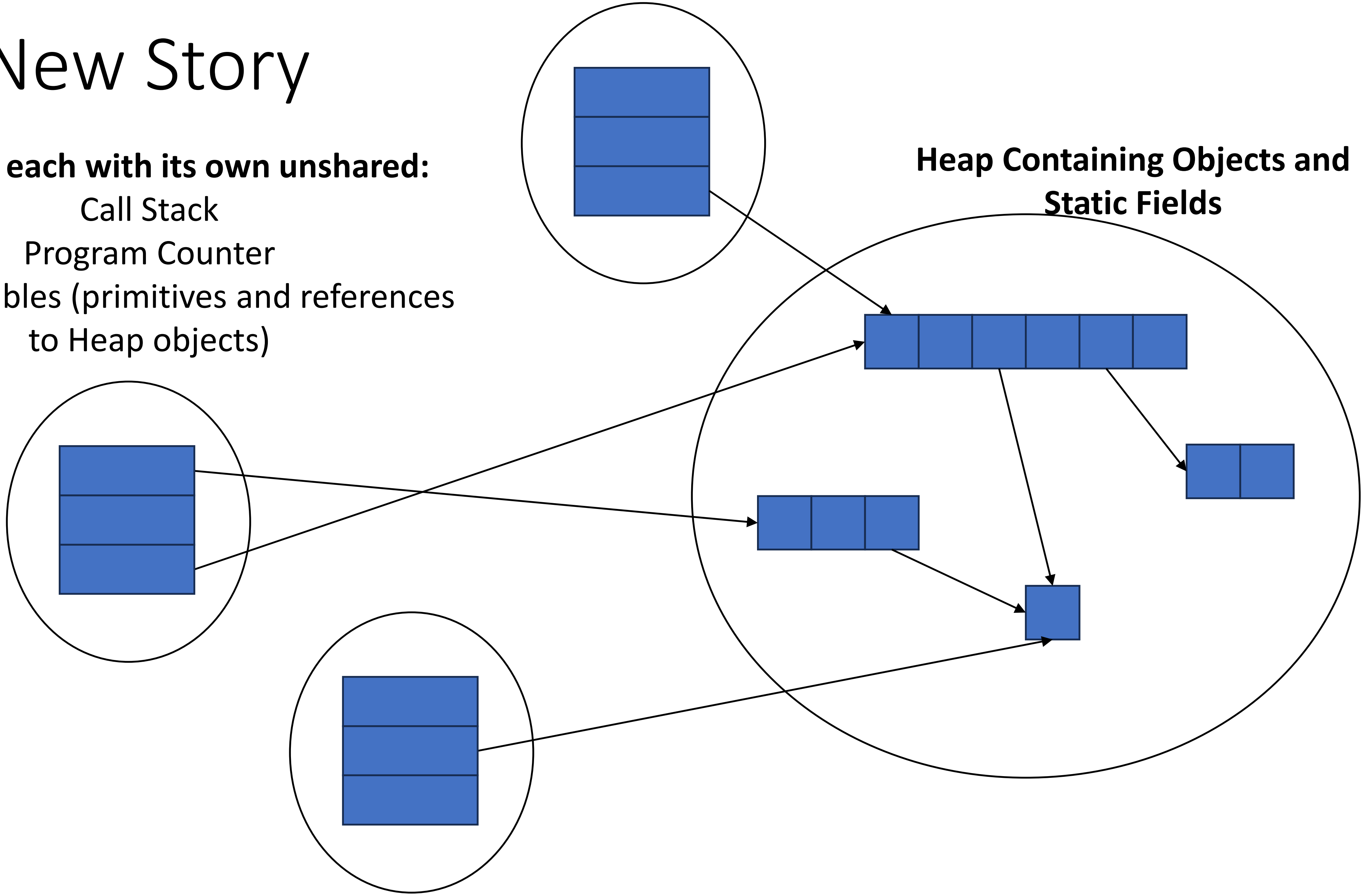
Call Stack
Program Counter
Local Variables (primitives and references to Heap objects)

Heap Containing Objects and Static Fields



New Story

Threads, each with its own unshared:
Call Stack
Program Counter
Local Variables (primitives and references to Heap objects)



Needs from Our Programming Language

- A way to create multiple things running at once
  - Threads
- Ways to share memory
  - References to common objects
- Ways for threads to synchronize
  - For now, just wait for other threads to finish their work

Parallelism Example (not real code)

- Goal: Find the sum of an array
- Idea: 4 processors will each find the sum of one quarter of the array, then we can add up those 4 results

```
int sum(int[] arr){
   res = new int[4];
   len = arr.length;
   FORALL(i=0; i < 4; i++) {   //parallel iterations
      res[i] = sumRange(arr,i*len/4,(i+1)*len/4); }
   return res[0]+res[1]+res[2]+res[3];
}
int sumRange(int[] arr, int lo, int hi) {
   result = 0;
   for(j=lo; j < hi; j++)
      result += arr[j]; return result;
}
```

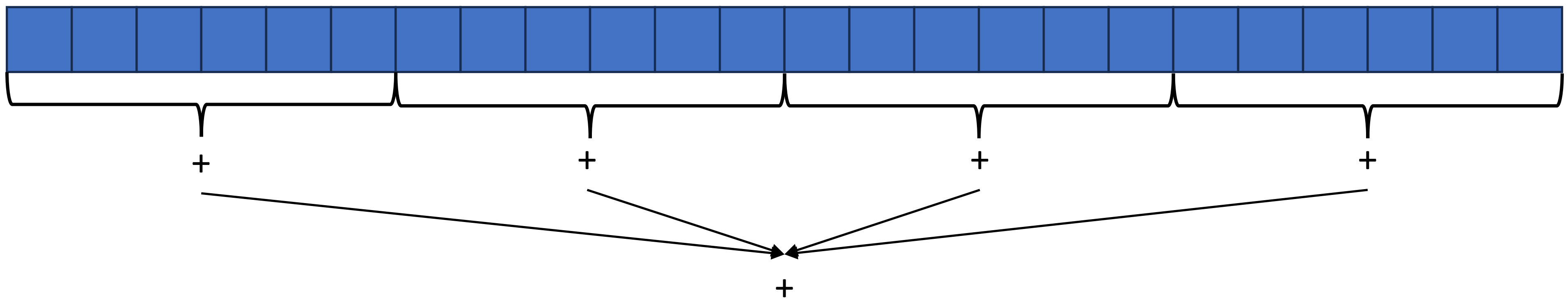
Note: This FORALL construct does not exist, but it's similar to how we'll actually do it.

Java.lang.Thread

- To run a new thread:
  1. Define a subclass **C** of java.lang.Thread, overriding **run**
  2. Create an object of class **C**
  3. Call that object's **start** method
     - **start** sets off a new thread, using **run** as its "main"
- Calling "**run**" directly causes the program to execute "**run**" sequentially

Back to Summing an Array

- Goal: Find the sum of an array
- Idea: 4 threads each find the sum of one quarter of the array
- Process:
    - Create 4 thread objects, each given a portion of the work
    - Call start() on each thread object to run it in parallel
    - Wait for threads to finish using join()
    - Add together their 4 answers for the final result



First Attempt (part 1, defining Thread Object)

```
class SumThread extends java.lang.Thread {
	int lo;	// fields, assigned in the constructor
	int hi;	// so threads know what to do.
	int[] arr;
	int ans = 0; // result

	SumThread(int[] a, int l, int h) {
		lo=l; hi=h; arr=a;
	}

	public void run() { //override must have this type
		for(int i=lo; i < hi; i++)
			ans += arr[i];
	}
```

First Attempt (part 2, Creating Thread Objects)

```
class SumThread extends java.lang.Thread {
        int lo, int hi, int[] arr; // fields to know what to do
        int ans = 0; // result
        SumThread(int[] a, int l, int h) { … }
        public void run(){ … } // override }

int sum(int[] arr){ // can be a static method
        int len = arr.length;
        int ans = 0;
        SumThread[] ts = new SumThread[4];
        for(int i=0; i < 4; i++) // do parallel computations
                ts[i] = new SumThread(arr,i*len/4,(i+1)*len/4);
        for(int i=0; i < 4; i++) // combine results
                ans += ts[i].ans;
        return ans;
}
```

First Attempt (part 3, Running Thread Objects)

```
class SumThread extends java.lang.Thread {
        int lo, int hi, int[] arr; // fields to know what to do
        int ans = 0; // result
        SumThread(int[] a, int l, int h) { … }
        public void run(){ … } // override }

int sum(int[] arr){ // can be a static method
        int len = arr.length;
        int ans = 0;
        SumThread[] ts = new SumThread[4];
        for(int i=0; i < 4; i++){ // do parallel computations
                ts[i] = new SumThread(arr,i*len/4,(i+1)*len/4);
                ts[i].start(); // start not run}
        for(int i=0; i < 4; i++) // combine results
                ans += ts[i].ans;
        return ans; }
```

First Attempt (part 4, Synchronizing)

```
class SumThread extends java.lang.Thread {
        int lo, int hi, int[] arr; // fields to know what to do
        int ans = 0; // result
        SumThread(int[] a, int l, int h) { … }
        public void run(){ … } // override }
int sum(int[] arr){ // can be a static method
        int len = arr.length;
        int ans = 0;
        SumThread[] ts = new SumThread[4];
        for(int i=0; i < 4; i++){ // do parallel computations
                ts[i] = new SumThread(arr,i*len/4,(i+1)*len/4);
                ts[i].start(); // start not run}
        for(int i=0; i < 4; i++) // combine results
                ts[i].join(); // wait for thread to finish!
                ans += ts[i].ans;
        return ans; }
```

Join

- Causes program to pause until the other thread completes its **run** method
- Avoids a **race condition**
  - Without join the other thread's **ans** field may not have its final answer yet

Flaws With this Attempt

Different machines have different numbers of processors!

Making the thread count a parameter helps make your program more efficient and reusable across computers

```
int sum(int[] arr, int numTs){ // can be a static method
        int len = arr.length;
        int ans = 0;
        SumThread[] ts = new SumThread[numTs];
        for(int i=0; i < numTs; i++){ // do parallel computations
                ts[i] = new SumThread(arr,i*len/numTs,(i+1)*len/numTs);
                ts[i].start(); // start not run}
        for(int i=0; i < numTs; i++) // combine results
                ts[i].join(); // wait for thread to finish!
                ans += ts[i].ans;
        return ans; }
```

Flaws With this Attempt

- Even If we make the number of threads equal the number of processors, the OS is doing time slicing, so we might not have all processors available right now
- For some problems, not all subproblems will take the same amount of time:
  - E.g. determining whether all integers in an array are prime.

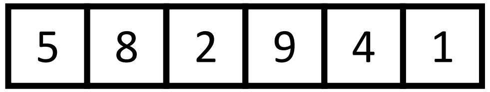
One Potential Solution: More Threads!

- Identify an “optimal” workload per thread
  - E.g. maybe it’s not worth splitting the work if the array is shorter than 1000
- Split the array into chunks using this “sequential Cutoff”
  - numTs = len/SEQ_CUTOFF;

- Problem: One process is still responsible for summing all len/1000 results
  - Process is still linear time

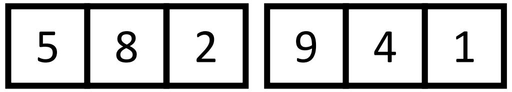
A Better Solution: Divide and Conquer!

- Idea: Each thread checks its problem size. If its smaller than the sequential cutoff, it will sum everything sequentially. Otherwise it will split the problem in half across two separate threads.

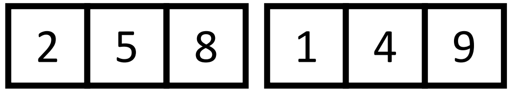


Merge Sort

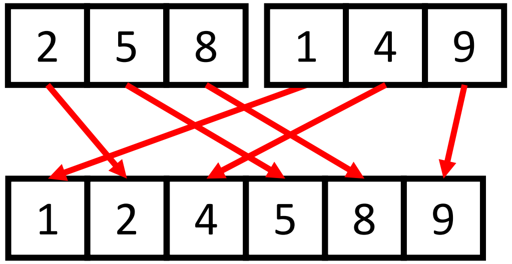
- **Base Case**:
  - If the list is of length 1 or 0, it’s already sorted, so just return it



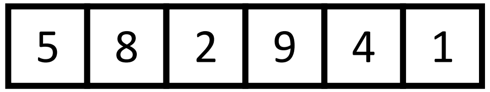
- **Divide**:
  - Split the list into two “sublists” of (roughly) equal length



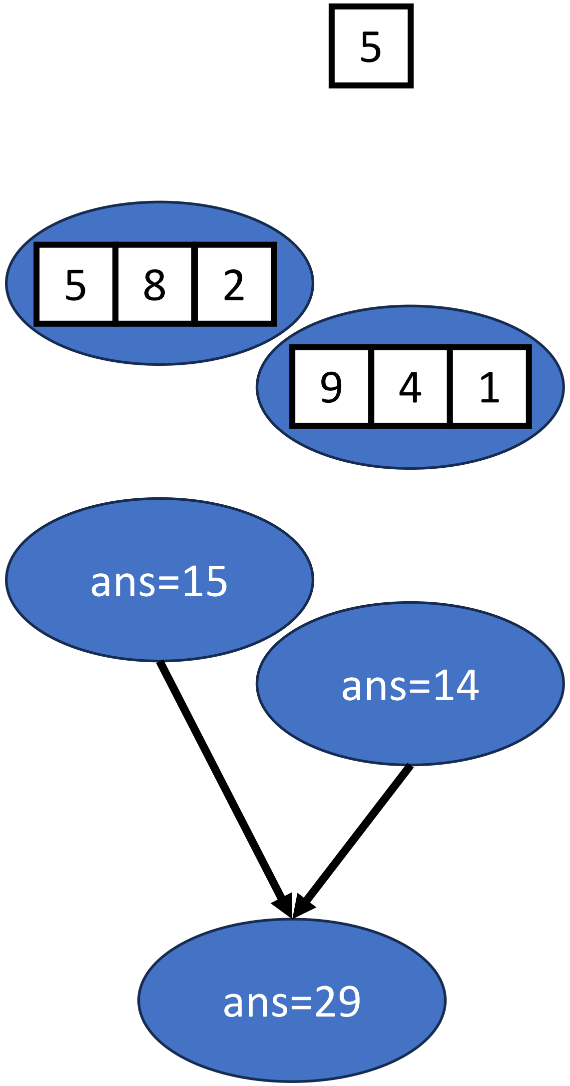
- **Conquer**:
  - Sort both lists recursively



- **Combine**:
  - **Merge** sorted sublists into one sorted list



Parallel Sum



- **Base Case**:
  - If the list's length is smaller than the Sequential Cutoff, find the sum sequentially
- **Divide**:
  - Split the list into two "sublists" of (roughly) equal length, create a thread to sum each sublist.
- **Conquer**:
  - Call **start()** for each thread
- **Combine**:
  - Sum together the answers from each thread

Divide and Conquer with Threads

```
class SumThread extends java.lang.Thread {
	public void run(){ // override
		if(hi – lo < SEQUENTIAL_CUTOFF) // “base case”
			for(int i=lo; i < hi; i++) ans += arr[i];
		else {
			SumThread left = new SumThread(arr,lo,(hi+lo)/2); // divide
			SumThread right= new SumThread(arr,(hi+lo)/2,hi); // divide
			left.start(); // conquer
			right.start(); // conquer
			left.join(); // don’t move this up a line – why?
			right.join();
			ans = left.ans + right.ans; // combine
		}
	}
}
int sum(int[] arr){ // just make one thread!
	SumThread t = new SumThread(arr,0,arr.length);
	t.run();
	return t.ans; }
```

Small optimization

- Instead of calling two separate threads for the two subproblems, create one parallel thread (using **start**) and one sequential thread (using **run**)

Divide and Conquer with Threads (optimized)

```
class SumThread extends java.lang.Thread {
	public void run(){ // override
		if(hi – lo < SEQUENTIAL_CUTOFF) // “base case”
			for(int i=lo; i < hi; i++) ans += arr[i];
		else {
			SumThread left = new SumThread(arr,lo,(hi+lo)/2); // divide
			SumThread right= new SumThread(arr,(hi+lo)/2,hi); // divide
			left.start(); // conquer
			right.run(); // conquer
			left.join(); // don’t move this up a line – why?
			//right.join();
			ans = left.ans + right.ans; // combine
		}
	}
}
int sum(int[] arr){ // just make one thread!
	SumThread t = new SumThread(arr,0,arr.length);
	t.run();
	return t.ans; }
```

ForkJoin Framework

- This strategy is common enough that Java (and C++, and C#, and…) provides a library to do it for you!

What you would do in Threads	What to instead in ForkJoin
Subclass **Thread**	Subclass **RecursiveTask<V>**
Override **run**	Override **compute**
Store the answer in a field	Return a V from compute
Call **start**	Call **fork**
join synchronizes only	**join** synchronizes and returns the answer
Call **run** to execute sequentially	Call **compute** to execute sequentially
Have a topmost thread and call **run**	Create a pool and call **invoke**

Divide and Conquer with ForkJoin

```
class SumTask extends RecursiveTask {
        int lo; int hi; int[] arr; // fields to know what to do
        SumTask(int[] a, int l, int h) { … }
        protected Integer compute(){// return answer
                if(hi – lo < SEQUENTIAL_CUTOFF) {  // base case
                        int ans = 0; // local var, not a field
                        for(int i=lo; i < hi; i++) {
                                ans += arr[i]; return ans; }
                else {
                        SumTask left = new SumTask(arr,lo,(hi+lo)/2); // divide
                        SumTask right= new SumTask(arr,(hi+lo)/2,hi); // divide
                        left.fork(); // fork a thread and calls compute (conquer)
                        int rightAns = right.compute(); //call compute directly (conquer)
                        int leftAns = left.join(); // get result from left
                        return leftAns + rightAns; // combine
                }
        }
}
```

Divide and Conquer with ForkJoin (continued)

```
static final ForkJoinPool POOL = new ForkJoinPool();
int sum(int[] arr){
	SumTask task = new SumTask(arr,0,arr.length)
	return POOL.invoke(task); // invoke returns the value compute returns
}
```

Section

- Working with examples of ForkJoin
- Make sure to bring your laptops!
  - And charge it!