Ian Piper
CSCI203

# Algorithms and Data Structures

Week 3 – Lecture A

# Applied Algorithms

- Simulation
  - Two basic kinds:
    - Continuous (time steps by a constant amount)
    - Discrete (time steps from event to event)
- Each type is suited to different kinds of simulations
  - Continuous – processes where there are no time-based events
    - Physical processes (e.g. Explosions)
  - Discrete – processes where time-based events control what is happening.
    - (e.g. Queues)

# Note

- This terminology is not especially obvious or clear.
  - Continuous:
    - Time is broken into discrete chunks (ticks)
    - Usually used to model a continuous process.
  - Discrete:
    - Time can take any value
    - Usually used to model discrete events

# Continuous Simulation

Clock Driven

# Continuous Simulation.

- Often dependent on complex mathematics.
- Often based on gridded algorithms.
  - Implicit solution
  - Explicit solution
- Often requires extreme computing resources.
  - Supercomputers
- We will not be looking at continuous simulation in this subject.

# Examples of Continuous Simulation: 1

- Mine explosions
  - Solve 10 differential equations for each of several thousand cells for every millisecond of the simulated event.
  - Parallel supercomputer.
  - Still over 24 hours per run.
- Some videos of the results...

# Examples of Continuous Simulation: 2

- Social simulation
  - Track the state of individuals in a simulated environment.
- Two entities
  - Peeps – simulated individuals (not always people)
  - Cells – simulated locations
- Used to examine response to threat
  - Spread of disease
  - Natural disaster
  - Man-made disaster (terrorist attack)
- Again, lots of time/computer power required.

# Discrete Simulation

Event driven

# Discrete Simulation

- Normally a lot less mathematically complex.
- Usually requires a lot less computer resources.
- E.g. Queue simulation
  - Widely used to evaluate queue-based processes
    - Shops
    - Production lines
    - Industrial processes
  - We will look at some simple examples and see how we might implement them.

# Scenario 1:

- A single server queue.
  - Customers arrive at random intervals to be served
  - If the server is not busy the customer will be served immediately
  - If the server is busy the customer will join the end of the (possibly empty) queue.
  - When the server has finished with the customer the next customer (if any) begins service – first customer in queue.

# Scenario 1:

- Events.
  - Customer arrives
  - Customer starts service
  - Customer ends service and leaves
- What we know:
  - When each customer arrives
  - How long they take to serve
- What we want to know:
  - How big is the queue on average?
  - How busy is the server?
  - What proportion of customers have to wait in a queue?

# Scenario 1:

- Data
  - Input data is a file consisting of a set of records containing
    - Arrival time
    - Service time
  - For each customer
  - Sorted by arrival time

```
0.24 0.55
0.59 0.16
0.90 0.07
1.87 0.69
…
```

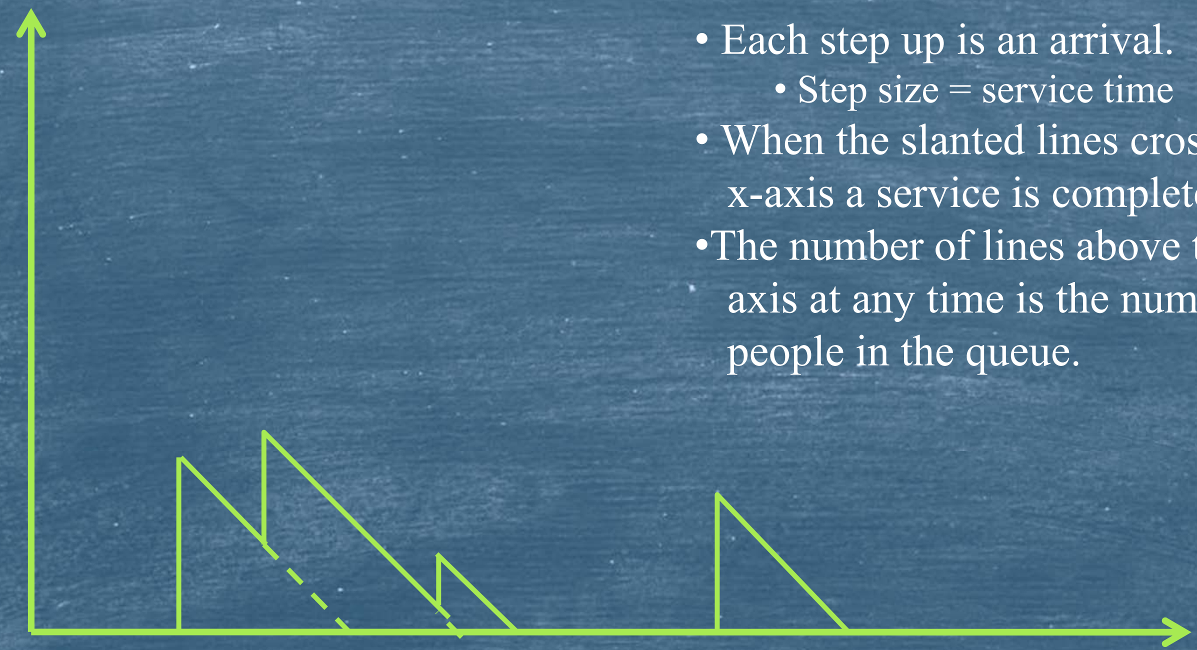
# Scenario 1:

- Manual Simulation
  - From the data file we can get a feel for what is happening
  - At time 0.00 the simulation starts
    - The server is idle
    - The queue is empty
  - At time 0.24 the first customer arrives
    - The server is busy for the next 0.55 (until 0.79)
    - The queue is empty
  - At time 0.59 the second customer arrives
    - The server is still busy
    - The queue now contains 1 customer (customer 2)
  - At time 0.79 the server finishes with customer 1
    - The server stays busy for the next 0.16 (until 0.95)
    - The queue is empty

```
0.24 0.55
0.59 0.16
0.90 0.07
1.87 0.69
…
```

# Graphs of Queues

- We can represent what is happening with a graph:



- Each step up is an arrival.
  - Step size = service time
- When the slanted lines cross the x-axis a service is complete.
- The number of lines above the axis at any time is the number of people in the queue.

# The Algorithm

- Starting to design the algorithm.
  - Data structures
    - We need to hold the queue
      - What should we put in it?
    - We need to keep track of the time
    - We need to know if the server is busy
      - If so we need to know when they will finish
    - We need to know when the next customer will arrive
    - We need to track statistics

# The Algorithm

- Starting to design the algorithm.
  - Procedures
    - Initialise the simulation
    - Process an arrival
    - Process a service completion
    - Finish the simulation
- Once the simulation is running how do we decide what to do next?
  - Compare the next arrival time with the end of service time

# The Algorithm

- Initialise the simulation

```
time = 0
busy = false
queue = empty
read  next_arrival, next_service
```

- Set up for statistics collection

# The Algorithm

- The main program loop

```
initialise
repeat
        if  busy = true  then
                if  service_end < next arrival  then
                        process_service_end
                else
                        process_arrival
        else
                process_arrival
        fi
until arrival file is empty and busy = false
finish
```

# The Algorithm

- Process an arrival

```
time = next_arrival
if  busy  then
        enqueue (next_service)
else
        busy = true
        service_end = time + next_service
fi
read (next_arrival next_service)
```

# The Algorithm

- Process a service completion

```
time = service_end
if  queue_empty  then
        busy = false
else
        service_end = time + dequeue()
fi
```

# Getting Complicated

- What if there is more than one server?
  - Two possible situations
    - One queue for all servers (like a bank)
    - One queue per server (like a supermarket)

# “One Queue to Serve Them All…”

- One queue for all servers
  - If all servers are busy add arrival to queue
    - Otherwise make one of the idle servers busy
  - When a server finishes have them serve the head of the queue
    - Otherwise make them idle

# Single Queue

- The events we are interested in are now:
  - Customer arrives
  - Server 1 finishes
  - Server 2 finishes
  - ...
  - Server *n* finishes
- How do we keep track of which event will happen next?
- Do we really need to know which servers are busy or can we just keep track of how many are busy?

# Single Queue

- Keeping track of servers:
  - Two possible solutions
    - An array of servers with busy[i] and end_time[i]
      - This lets us track who is doing what
      - Finding what happens next is in O(*n*)
    - A heap of end times (smallest on top)
      - This does not let us track who is doing what
      - Finding what happens next is O(log *n*)
    - Can we get the best of both worlds?

# Single Queue

- Using a heap
  - If we are clever, we can store all event times on the heap
  - All we need is a second array telling us what is what
  - If we are really clever, we can partition the heap into two parts:
    - The heap itself
    - The idle servers
  - How do we manipulate the heap as events occur?

# Single Queue

- Customer arrives:

```
time = heap[0]
read next arrival into heap[0], next_service_time
sift_down(heap)
if  n_busy < n_servers then
        n_busy = n_busy + 1
        heap[n_busy] = time + service_time
        sift_up(heap)
else
        enqueue(service_time)
fi
service_time = next_service_time
```

# Single Queue

- Server finishes:

```
time = heap[0]
if queue_empty then
    heap[0] = heap[n_busy]
    n_busy = n_busy - 1
else
    heap[0] = time + dequeue()
fi
sift_down(heap)
```

# Single Queue

- In summary:
  - Heap Grows if a customer arrives and a server is idle
  - Heap shrinks if a customer is served and the queue is empty
  - Heap stays the same size otherwise
- If we keep a second array (id) initially filled with integers 0..*n* we can use it to track who is doing what
  - 0 is the next arrival
  - 1 is server 1's completion time
  - 2 is server 2's completion time
  - …
  - *n* is server *n*'s completion time

## Single Queue

- Customer arrives:

```
time = heap[0]
read next arrival into heap[0], next_service_time
sift_down(heap, id)
if  n_busy < n_servers then
        n_busy = n_busy + 1
        heap[n_busy] = time + service_time
        sift_up(heap, id)
        service_time = next_service_time
else
        enqueue(next_service_time)
fi
```

# Single Queue

- Server finishes:

```
time = heap[0]
if queue_empty then
	swap (id[0], id[n_busy])
	heap[0] = heap[n_busy]
	n_busy = n_busy - 1
else
	heap[0] = time + dequeue()
fi
sift_down(heap, id)
```

# Single Queue

- Every time we move an entry in the heap we move the corresponding entry in the id array.
- If the top of the id array is a zero the next event is an arrival
- If the top of the id array is non_zero the next event is a service completion for server id[0]
- The simulation starts with the first arrival time in heap[0] and n_busy = 0

# Multiple Queues

- In this case we have an array of *n* queues, 1 per server
- When a customer arrives and all servers are busy we place the customer on one of the queues (which one?)
- When a server finishes we only make them busy if their queue is not empty
- NOTE: This means that we can have a queue even if a server is idle.

# Priority queues

- How would we handle the situation where customers are given different service priorities?
  - One queue for each priority empty the highest priority queue first
  - This is only efficient if there are a small number of priorities
- What do we do if each priority may be different?
  - E.g. priority is a float between 0 and 1
  - 0 is the lowest customer priority
  - 1 is the highest priority customer
  - We have an infinite number of different priorities so we can’t have a queue for each one.

# Priority queues

- The solution is to replace the queue with a heap ordered on priority
- Each time we remove a customer from the heap we
  - Move the last entry to the top of the heap
  - Reduce the heap size by one
  - Sift down the top entry
- Each time we add a customer to the heap we:
  - Increase the heap size by one
  - Add the customer to the end of the heap
  - Sift up the last entry