COMP9334
Capacity Planning for Computer Systems and Networks

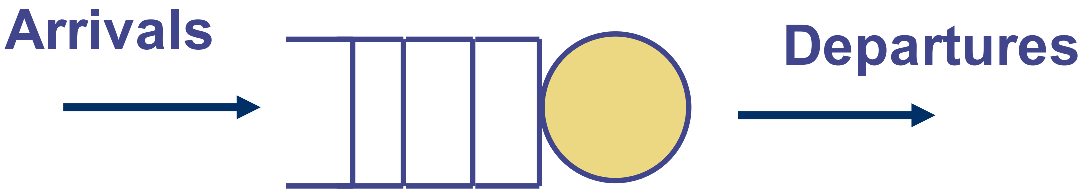
Week 7: Discrete event simulation (1)

Week 3: Queues with Poisson arrivals

- Single-server M/M/1

Exponential inter-arrivals (λ)

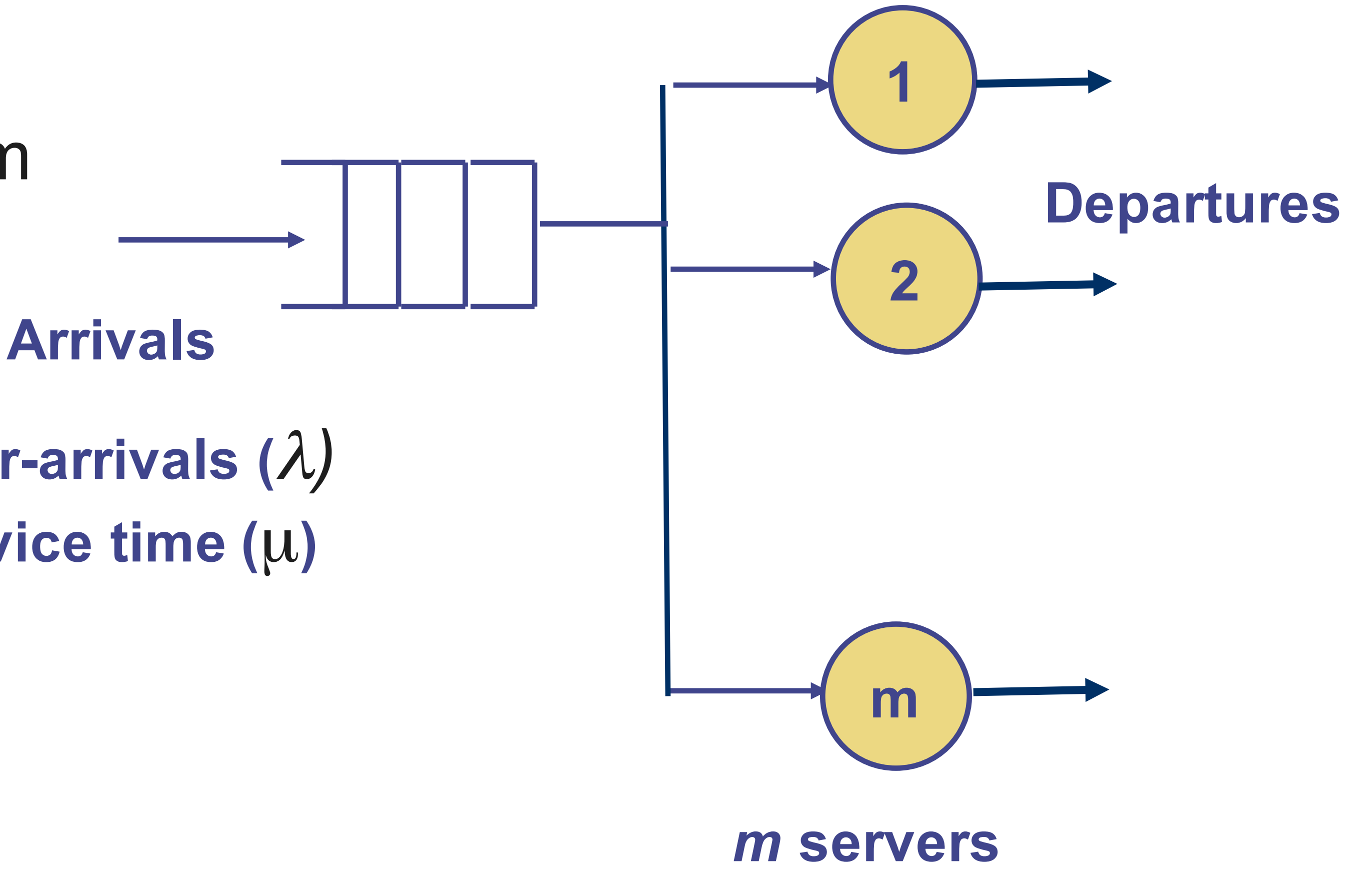
Exponential service time (μ)



Arrivals

Departures

- Multi-server M/M/m



Arrivals

1

2

m

Departures

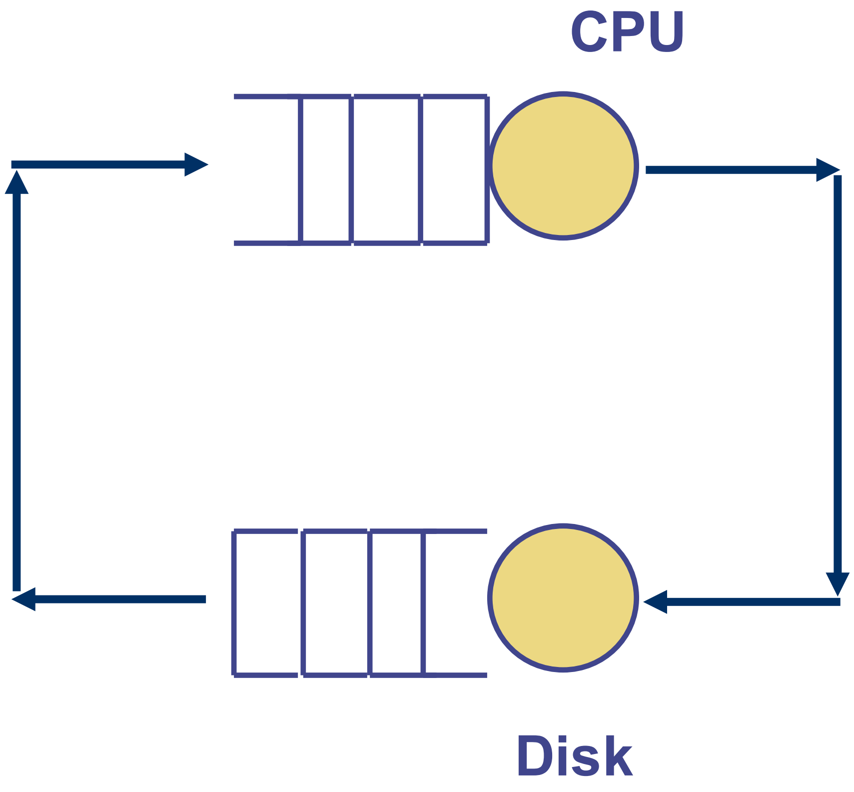
***m* servers**

Exponential inter-arrivals (λ)

Exponential service time (μ)

Week 4: Closed-queueing networks

- Analyse closed-queueing network with Markov chain
  - The transition between states is caused by an arrival or a departure according to exponential distribution

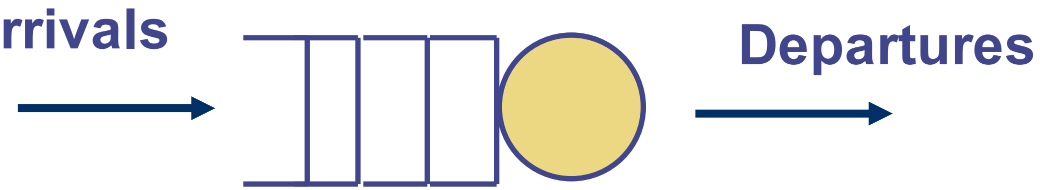


- General procedure
  - Identify the states
  - Find the state transition rates
  - Set up the balance equations
  - Solve for the steady state probabilities
  - Find the response time etc.

Week 5: Queues with general arrival & service time

- Queues with general inter-arrival and service time distributions

General Inter-arrivals time distribution
General service time distribution



Arrivals

Departures

- M/G/1 queue
  - Can calculate delay with the P-K formula

$$W = \frac{\lambda E[S^2]}{2(1-\rho)}$$

- G/G/1 queue
  - No explicit formula, get a bound or approximation

$$W \leq \frac{\lambda(\sigma_a^2 + \sigma_s^2)}{2(1-\rho)}$$

Analytical methods for queues

- You had learnt how to solve a number of queues analytically (= mathematically) given their
  - Inter-arrival time probability distribution
  - Service time probability distribution

- Queues that you can solve now include M/M/1, M/M/m, M/G/1, M/G/1 with priorities etc.
  - If you know the analytical solution, this is often the most straightforwad way to solve a queueing problem
- Unfortunately, *many queueing problems are still analytically intractable!*

- What can you do if we have an analytically intractable queueing problem?

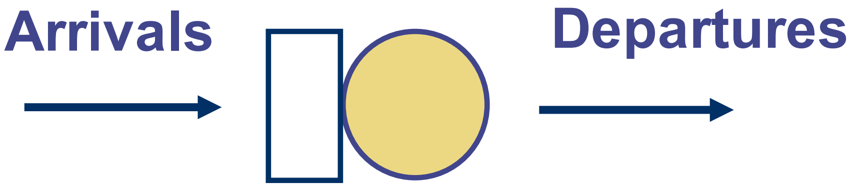
This lecture and next lecture

- The lectures for these two weeks will focus on using *discrete event simulation for queueing problems*
  - *Simulation is an imitation of the operation of real-life system over time.*

- The topics for this week are
  - What are discrete event simulation?
  - How to structure a discrete event simulation?
  - How to generate pseudo-random numbers for simulation?

- Next week
  - How to choose simulation parameters?
  - How to analyse data?
  - What are the pitfalls that you need to avoid?

Motivating example



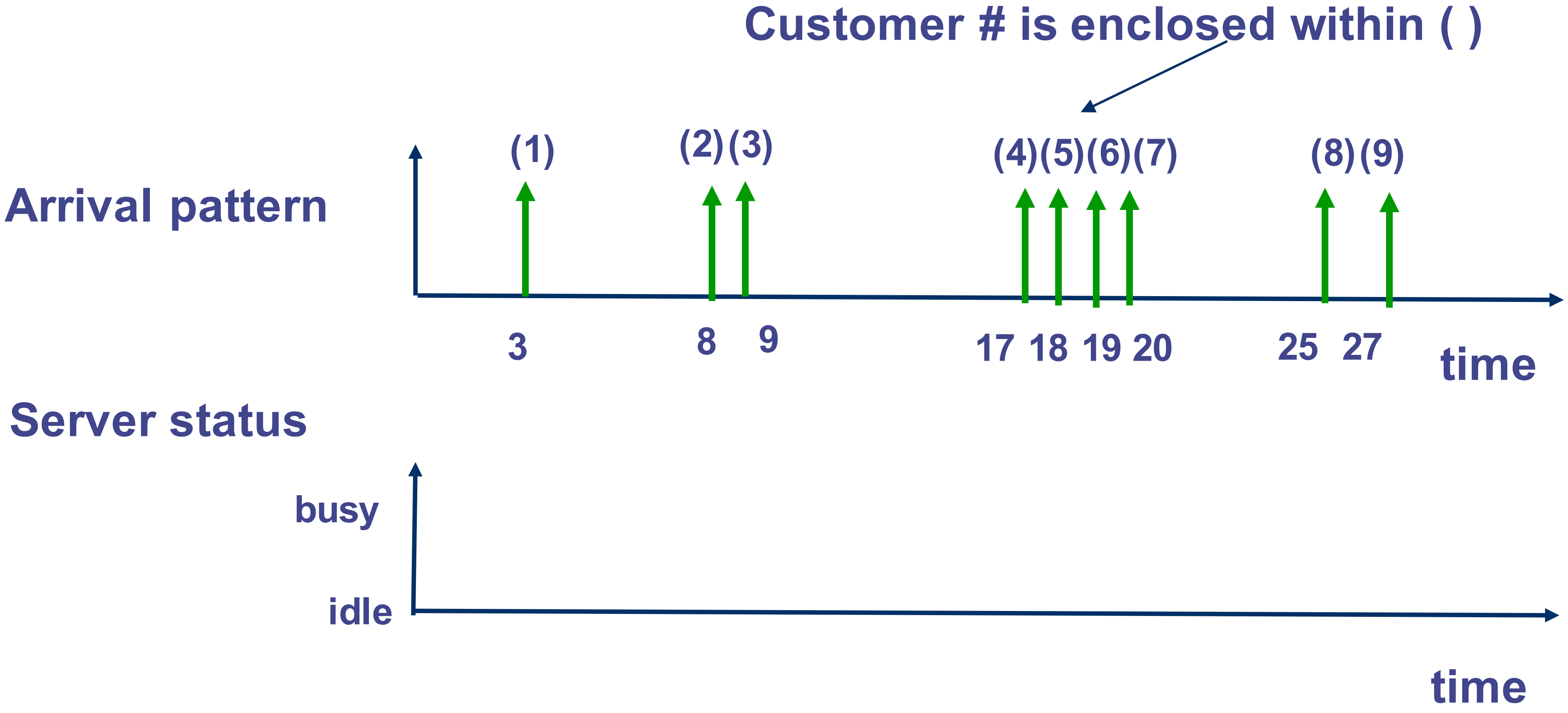
- Consider a single-server queue with only one buffer space (= waiting room)
- If a customer arrives when the buffer is occupied, the customer is rejected.
- Given the arrival times and service times in the table on the right, find
  - The mean response time
  - % of rejected customers

  Assuming an idle server at time = 0.

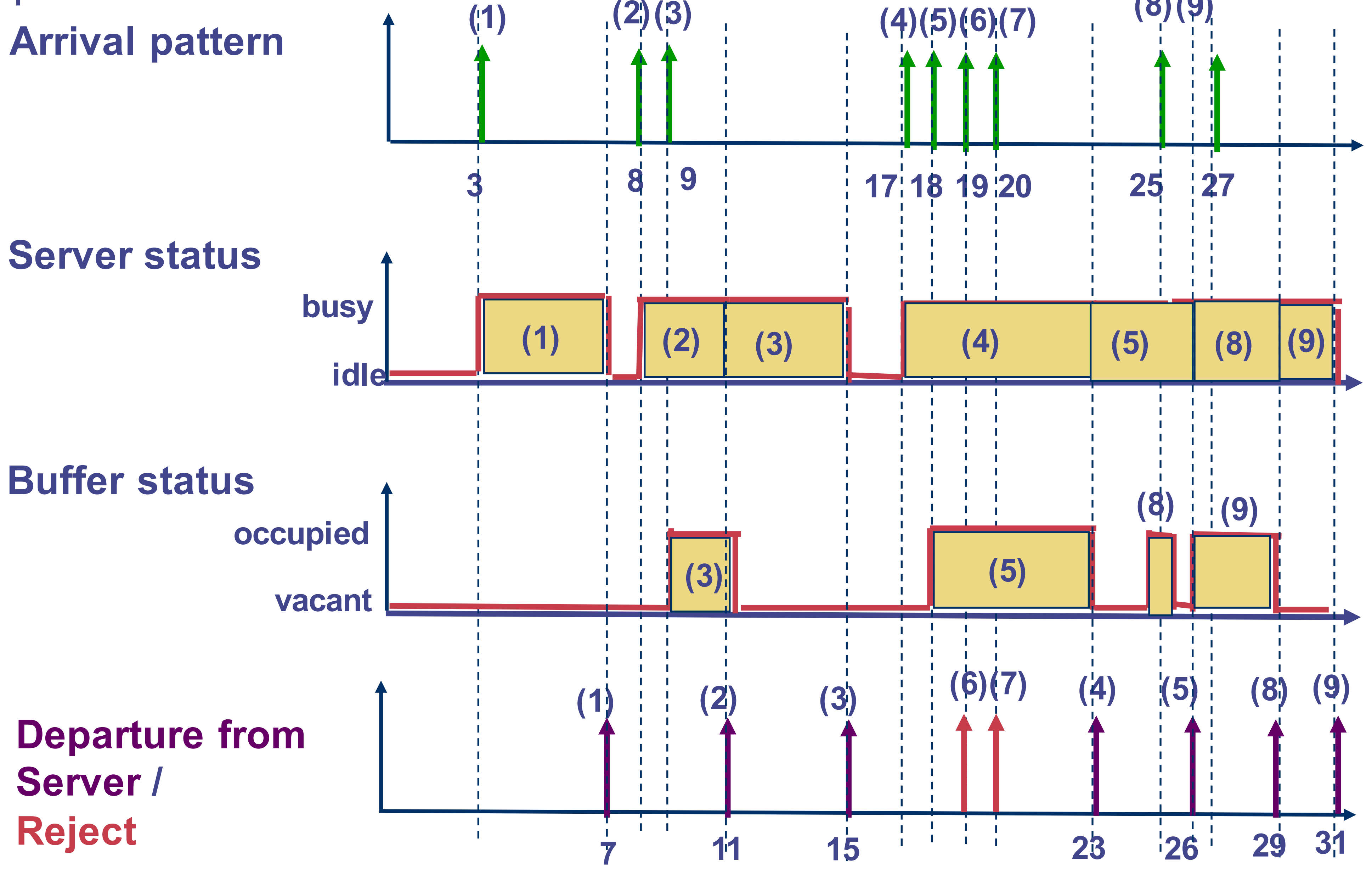
Customer number	Arrival time	Service time
1	3	4
2	8	3
3	9	4
4	17	6
5	18	3
6	19	2
7	20	2
8	25	3
9	27	2

Let us try a graphical solution

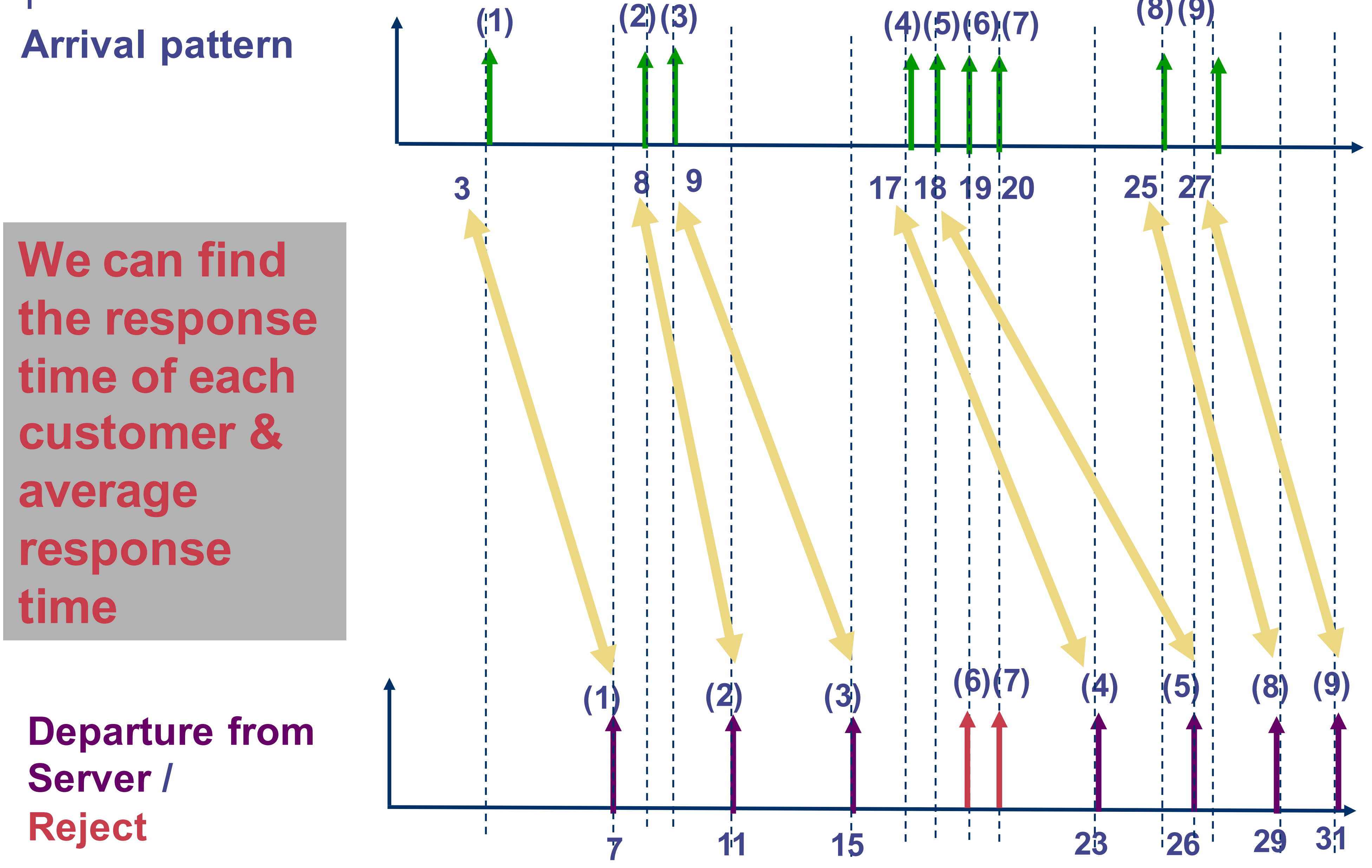
- In the graphical solution, we will keep track of
  - The status of the server: busy or idle
  - The status of the buffer: occupied or vacant



A graphical solution

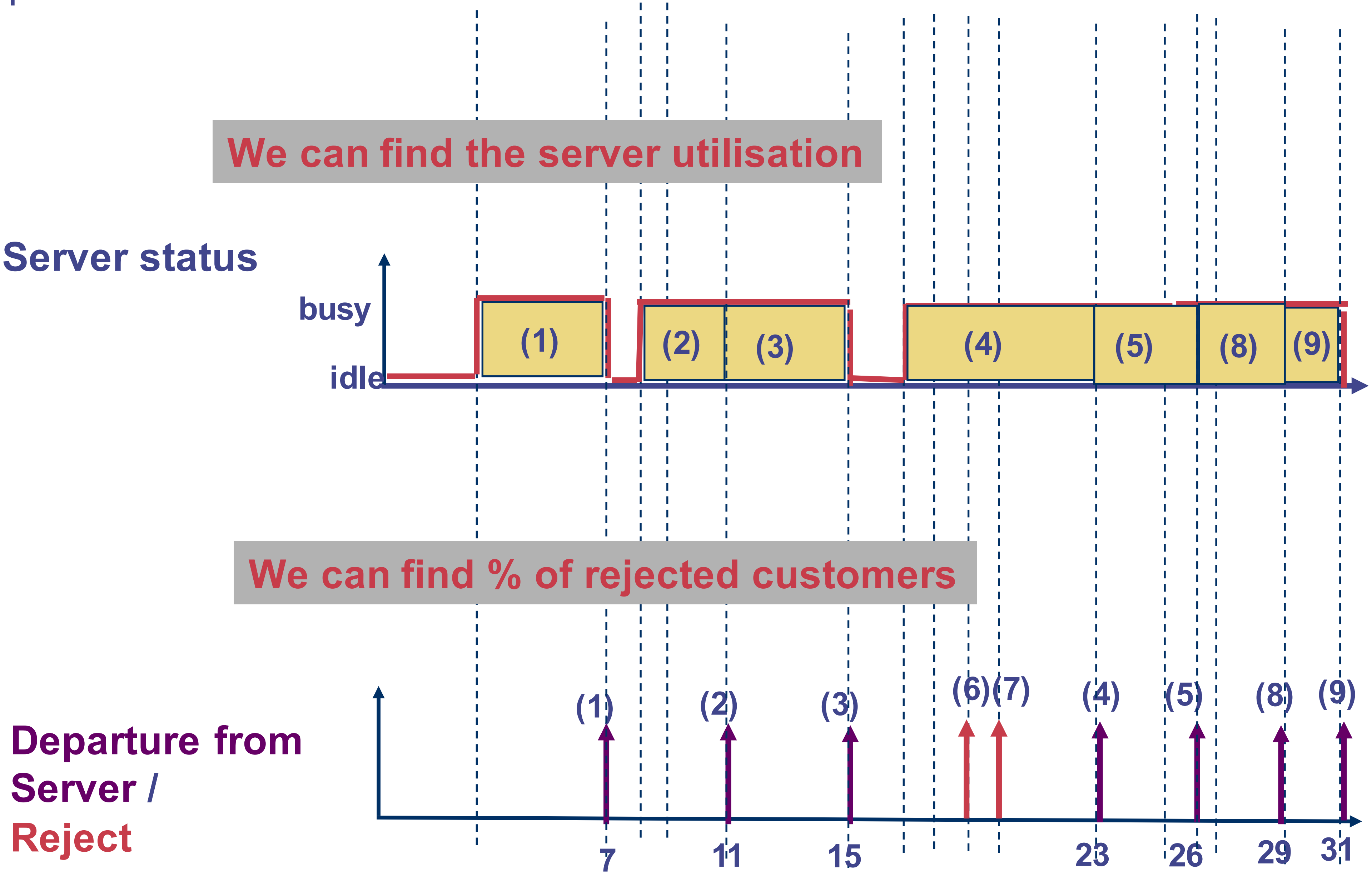


Using the graphical solution (1)



We can find the response time of each customer & average response time

Using the graphical solution (2)



We can find the server utilisation

We can find % of rejected customers

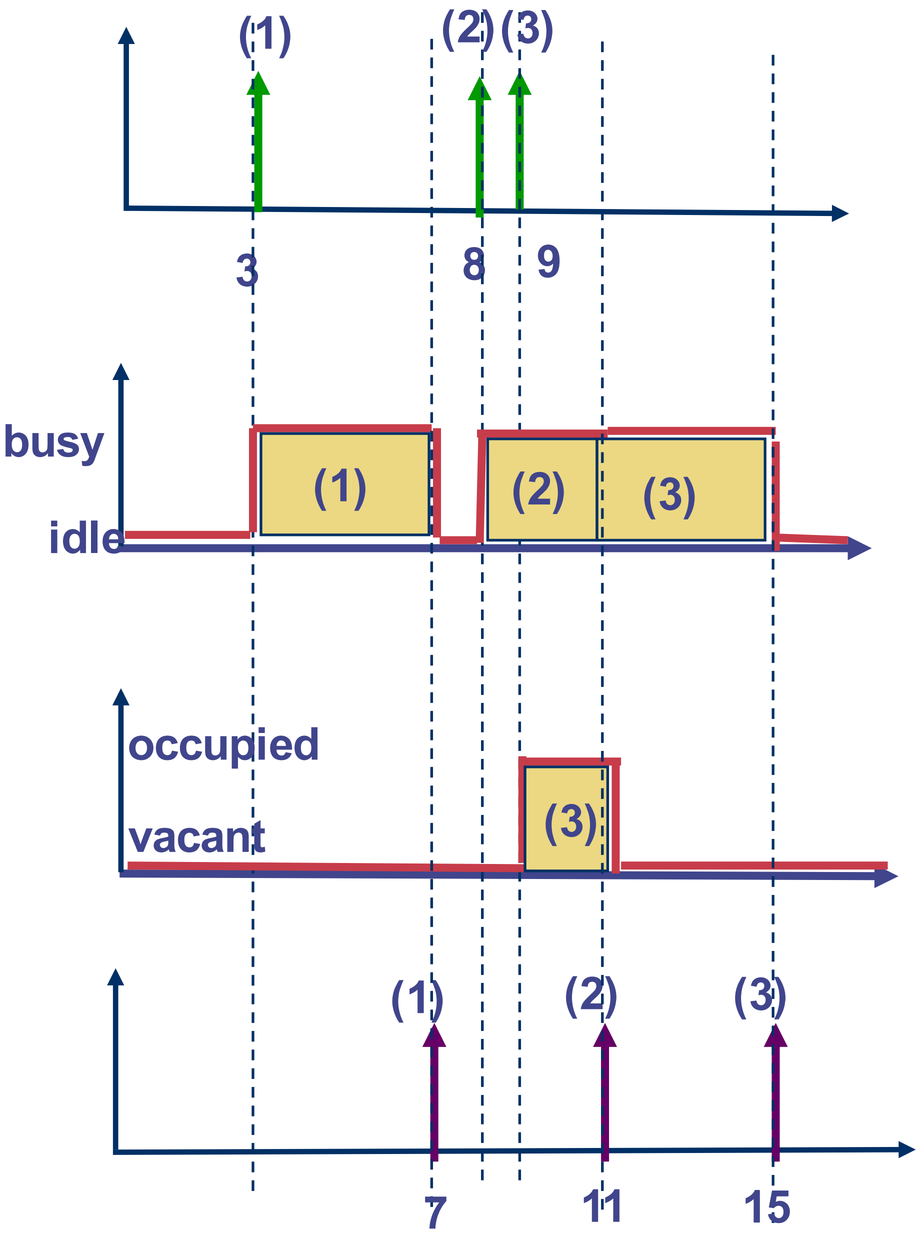
From graphical solution to computer solution (1)

- How can we turn this graphical solution into a computer solution, i.e. a computer program that can solve the problem for us

- We need to keep track of the status of the server and the status of the buffer,
  - This allows us to make decisions
  - E.g. If server is BUSY and buffer is OCCUIPIED, an arriving customer is rejected.
  - E.g. If server is BUSY and buffer is VACANT, an arriving customer goes to the buffer.
  - E.g. If server is IDLE, an arriving customer goes to the sever
- *What this means*: We need to keep track of the status of some variables in our computer solution.

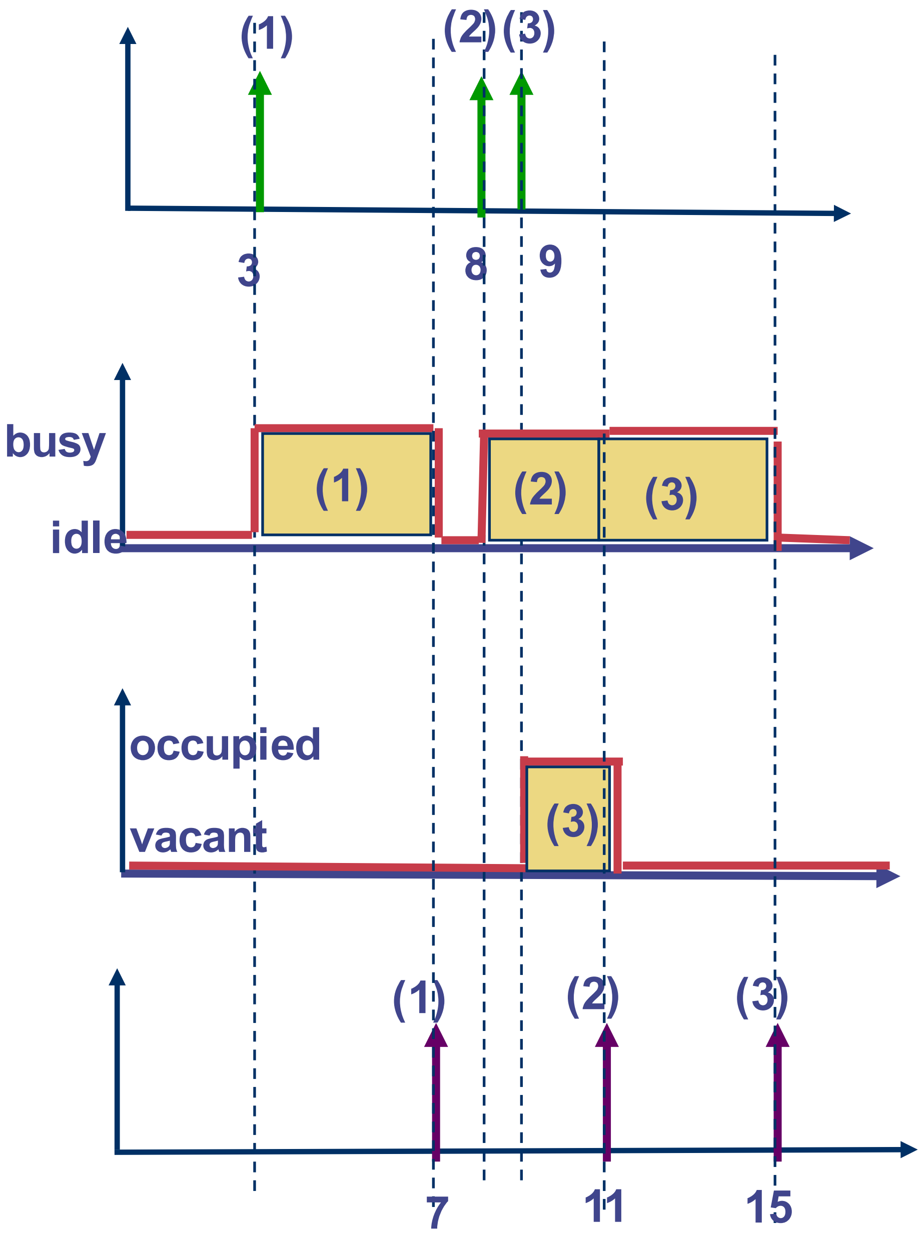
From graphical solution to computer solution (2)

- Observation #1:
    - An arriving or departing customer causes the server or buffer status to change
    - Examples:
        - At time = 3, the arrival of customer #1 causes the server to switch from IDLE to BUSY
        - At time = 7, the departure of customer #1 causes the server to switch from BUSY to IDLE
        - At time = 9, the arrival of customer #3 causes the buffer to switch from VACANT to OCCUPIED
        - Etc.



From graphical solution to computer solution (3)

- Let us call the arrival of a customer or the departure of a customer an ***event***
- Observation #2:
  - The status of the server and the status of the buffer remain the same between two consecutive events
- What this means:
  - We need to keep track of the timing of the events
    - Events can cause status transitions
    - In between events, status remain the same



From graphical solution to computer solution (4)

- In our computer solution, we will use a ***master clock*** to keep track of the current time
- We will advance the master clock from event to event
- In order to see how the computer solution works, let us try it out on paper first

On paper simulation

- In our simulation, we keep track of a number of variables
  - MC = Master clock
  - Status of
    - Server: 1 = BUSY, 0 = IDLE
    - Buffer: 1 = OCCUPIED, 0 = VACANT
  - Event time:
    - Next arrival event and service time of this arrival
    - Next departure event and arrival time of this departure
  - The (arrival time, service time) of the customer in buffer
  - In order to compute the response time, we keep track of
    - The cumulative response time (T)
    - Cumulative number of customers rejected (R)

MC	Next arrival		Next departure		Server status	Buffer status + customer in buffer	T	R
	Arrival time	Service time	Departure time	Arrival time of this departure				
0	3	4	-	-	0	0	0	0
3	8	3	7	3	1	0	0	0
7	8	3	-	-	0	0	4	0

On paper simulation

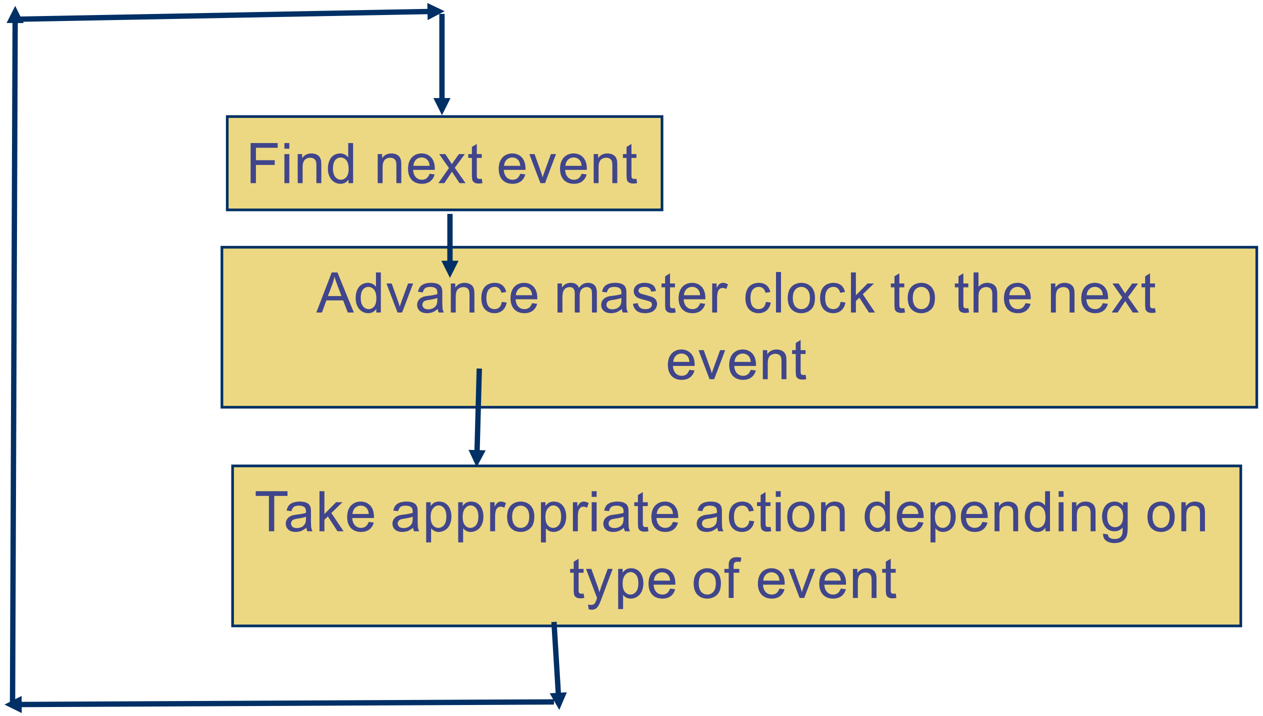
MC	Next arrival		Next departure		Server status	Buffer status + Customer in buffer	T	R
	Arrival time	Service time	Departure time	Arrival time of this departure				
0	3	4	-	-	0	0	0	0
3	8	3	7	3	1	0	0	0
7	8	3	-	-	0	0	4	0
8	9	4	11	8	1	0	4	0
9	17	6	11	8	1	1 (9,4)	4	0
11	17	6	15	9	1	0	7	0
15	17	6	-	-	0	0	13	0

Can you continue?

(Arrival time, service time) of the customer in the buffer.

Logic of the program (1)

- At each step, we advance to the next event that will take place



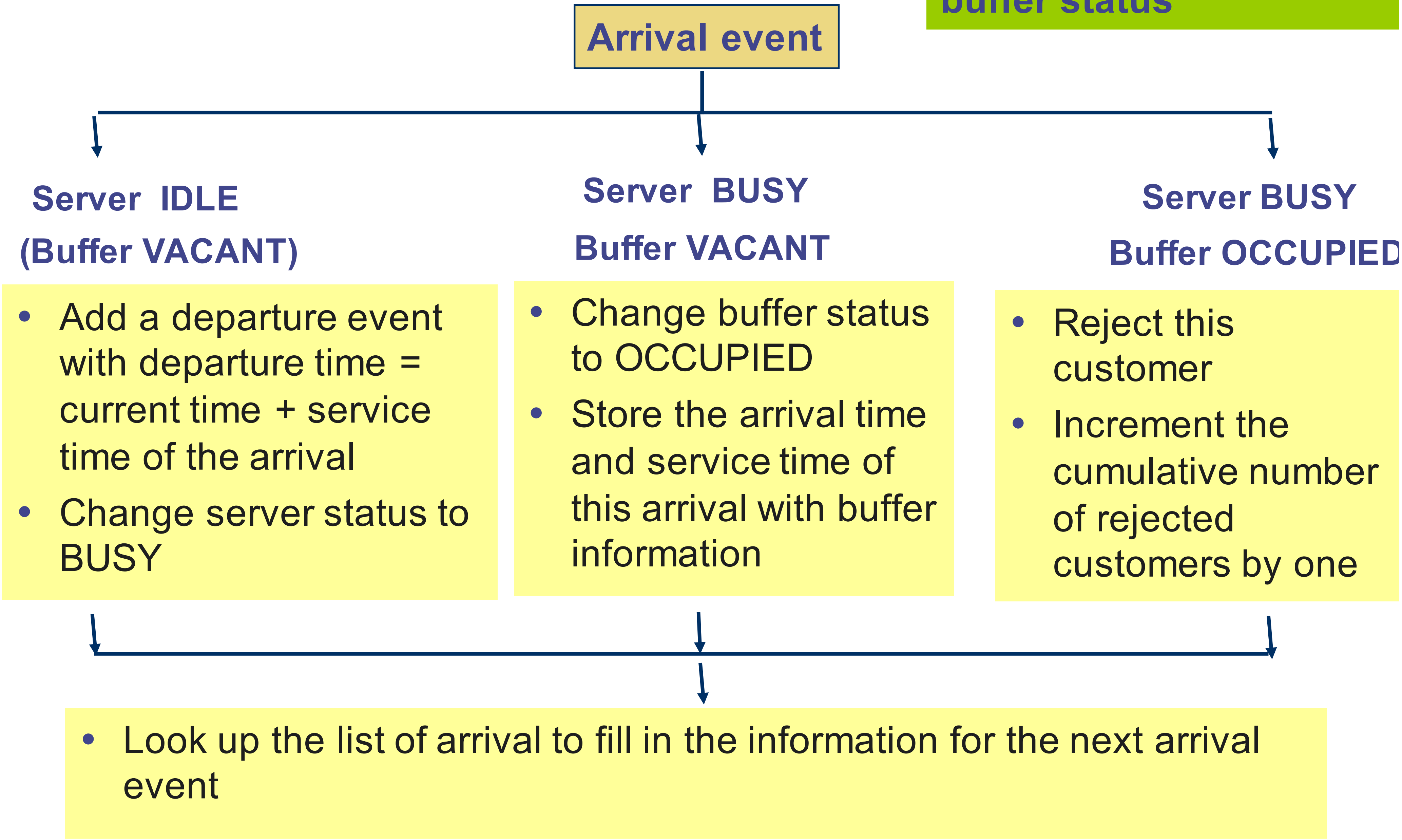
Find next event

Advance master clock to the next event

Take appropriate action depending on type of event

Handling an arrival event

Three cases according to the server and/or buffer status



Arrival event

Server IDLE

(Buffer VACANT)

- Add a departure event with departure time = current time + service time of the arrival
- Change server status to BUSY

Server BUSY

Buffer VACANT

- Change buffer status to OCCUPIED
- Store the arrival time and service time of this arrival with buffer information

Server BUSY

Buffer OCCUPIED

- Reject this customer
- Increment the cumulative number of rejected customers by one

- Look up the list of arrival to fill in the information for the next arrival event

Handling an departure event

Two cases according to the buffer status

Departure event

- Update the cumulative response time
  - T 🡐 T + current time - arrival time of the departing customer

Buffer VACANT

- Change server status to IDLE
- Next departure event becomes empty

Buffer OCCUPIED

- Update the departure event with information of the customer in the buffer
- Next departure time =

  current time + service time of the customer in the buffer
- Change buffer status to VACANT

Discrete event simulation

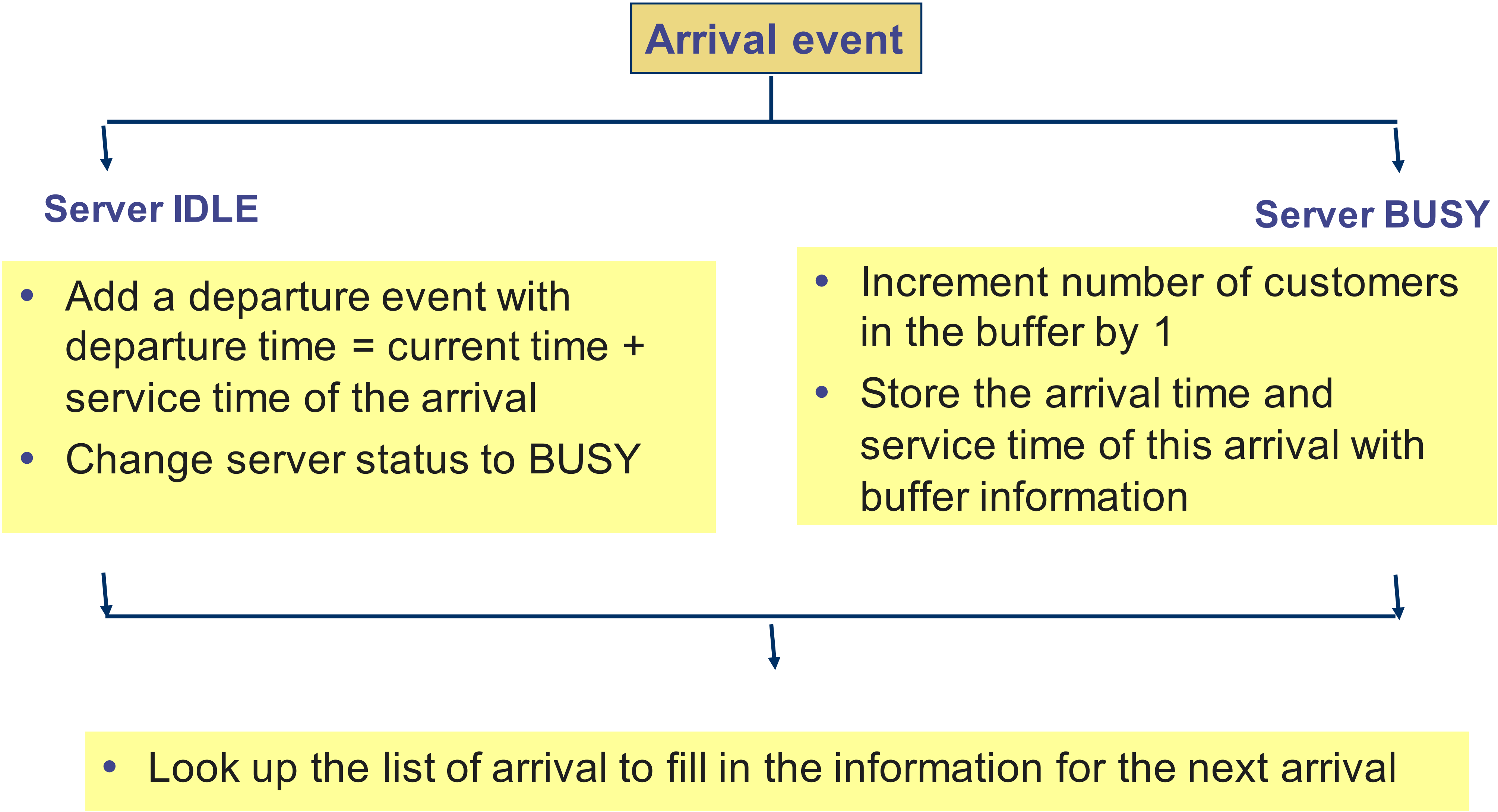
- The above computer program is an example of a discrete event simulation
- It allows you to solve a queueing problem with one server and one buffer space
- You can generalise the above procedure to
    - Multi-server
    - Finite or infinite buffer space
    - Different queueing disciplines
- Let us generalise it to the case of single-server with infinite buffer

Single server with infinite buffer simulation

- In this case, we will use buffer status to denote the number of customers in buffer
    - Buffer status = 0, 1, 2, 3, …
- We also need to store all the (arrival time, service time) of all the customers in the buffer
- Compare with the single-server single-buffer case, we only need to change the handling of
    - An arrival event
    - A departing event

Handling an arrival event

Two cases according to the server status



Arrival event

Server IDLE

- Add a departure event with departure time = current time + service time of the arrival
- Change server status to BUSY

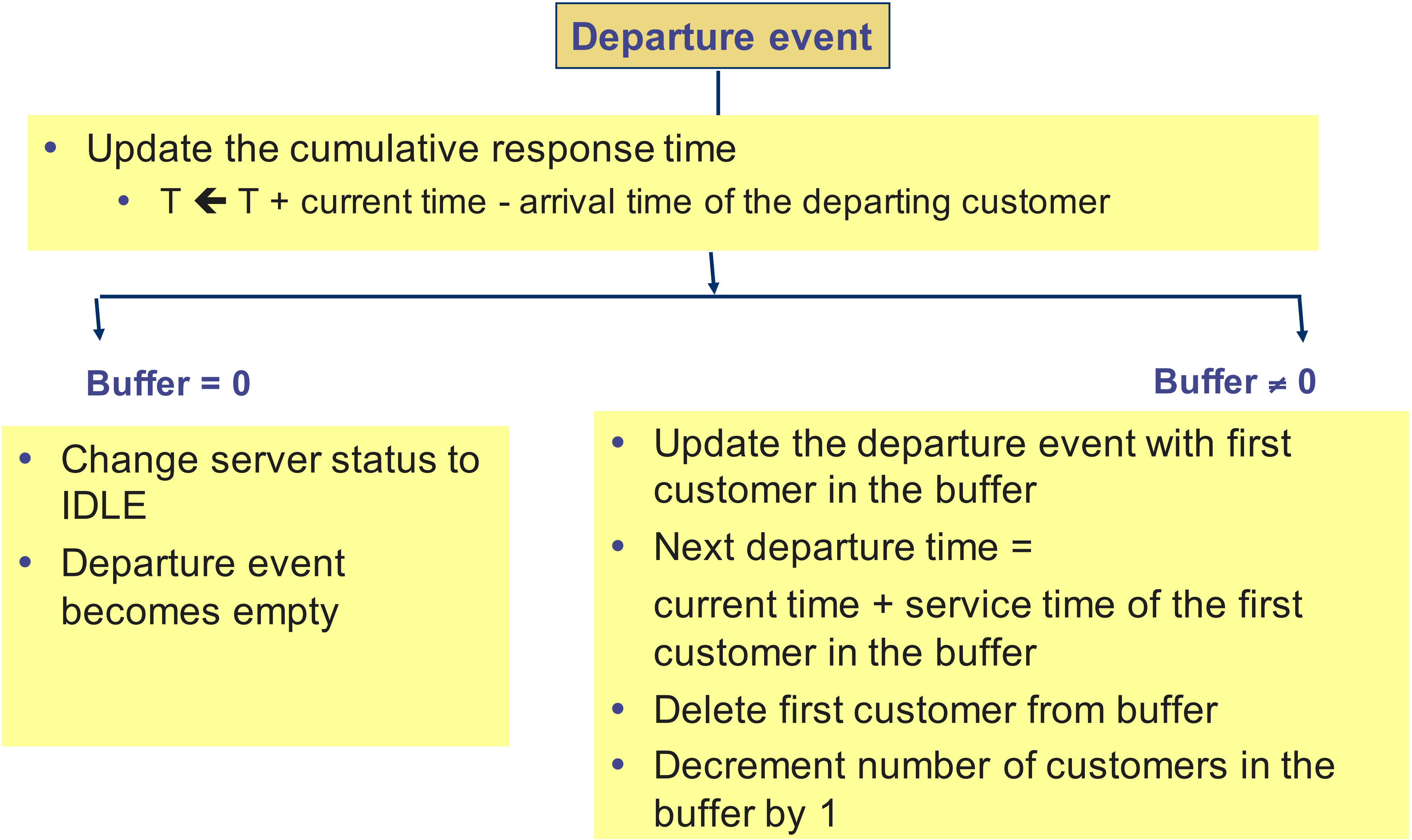
Server BUSY

- Increment number of customers in the buffer by 1
- Store the arrival time and service time of this arrival with buffer information

- Look up the list of arrival to fill in the information for the next arrival

Handling an departure event

Two cases according to the buffer status



Departure event

- Update the cumulative response time
    - T 🡐 T + current time - arrival time of the departing customer

Buffer = 0

- Change server status to IDLE
- Departure event becomes empty

Buffer ≠ 0

- Update the departure event with first customer in the buffer
- Next departure time =

  current time + service time of the first customer in the buffer
- Delete first customer from buffer
- Decrement number of customers in the buffer by 1

Generating random numbers

- We have so far assume that you can look up a list of arrival times and service times for the next customer
- However, sometimes you want to solve a queue with some specific inter-arrival time and service time probability distribution
  - For example, if
    - inter-arrival time x is drawn from $1/x^2$ with $x \geq 1$
    - Service time y is drawn from $2/y^3$ with $y \geq 1$
  - In this case, you will need to generate random numbers with the given probability distribution
- We will now study how we can generate random numbers

Random number generator in C

- In C, the function *rand()* generates random integers between 0 and RAND_MAX
- E.g. The following program generates 10 random integers:

```
#include <stdio.h>
#include <stdlib.h>

int main ()
{
  int i;

   for (i = 0; i < 10; i++)
    printf("%d\n",rand());

  return;

}
```

Let us generate 10,000 random integers using rand() and see how they are distributed

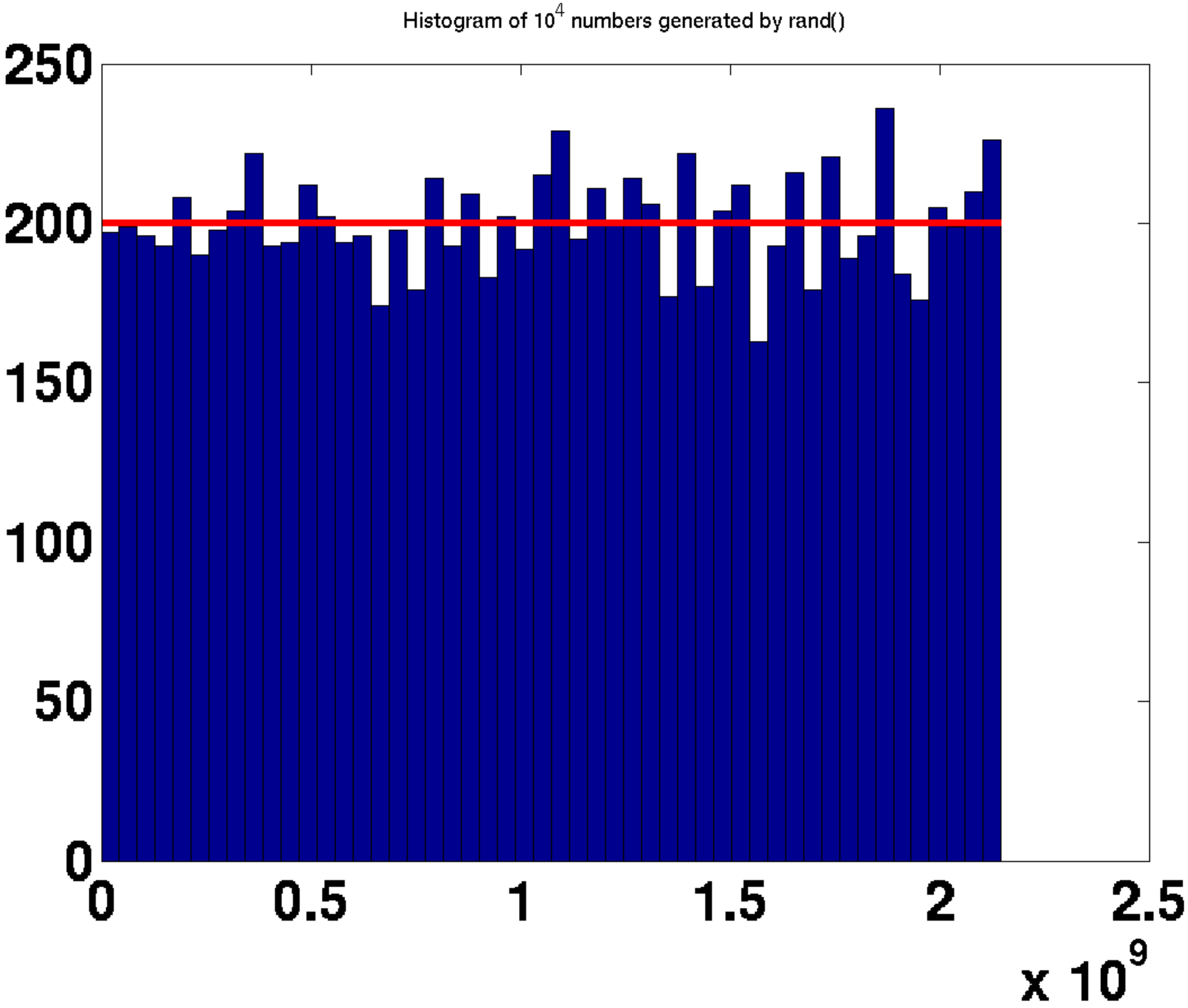
This C file "genrand1.c" is available from the course web site.

Distribution of 10000 entries from rand()

Sort into 50 bins

If the numbers are really uniformly distributed, we expect 200 numbers in each bin.

The numbers are almost uniformly distributed



LCG

- The random number generator in C is a Linear Congruential Generator (LCG)
- LCG generates a sequence of integers $\{Z_1, Z_2, Z_3, \ldots\}$ according to the recursion

  $$Z_k = a\, Z_{k-1} + c \pmod{m}$$

  where *a, c* and *m* are integers
- By choosing a, c, m, Z_1 appropriately, we can obtain a sequence of seemingly random integers
- If a = 3, c = 0, m = 5, Z_1 = 1, LCG generates the sequence 1, 3, 4, 2, 1, 3, 4, 2, …
- *Fact:* The sequence generated by LCG has a cycle of *m-1*
- We must choose *m* to be a large integer
  - For C, m = 2^{31}
- The proper name for the numbers generated is *pseudo-random numbers*

Seed

- LCG generates a sequence of integers $\{Z_1, Z_2, Z_3, \ldots\}$ according to the recursion

  $$Z_k = a Z_{k-1} + c \pmod{m}$$

  where *a, c* and *m* are integers
- The term Z_1 is call a seed
- By default, C also uses 1 as the seed and it will generate the same random sequence
- However, sometimes you need to generate different random sequences and you can change the seed by calling the function *srand()* before using *rand()*
  - Demo genrand1.c, genrand2.c and genrand3.m
  - *genrand1.c – uses the default seed*
  - *genrand2.c – sets the seed using command line argument*
  - *genrand3.c – sets the seed using current time*

Uniformly distributed random numbers between (0,1)

- With rand() in C, you can generate uniformly distributed random numbers in between 1 and 2^{31}-1(= RAND_MAX)
  - By dividing the numbers by RAND_MAX, you get randomly distributed numbers in (0,1)

- In Matlab, *rand(n,1)* generates a sequence of *n* uniformly distributed random numbers in (0,1)
  - Matlab uses the Mersenne Twister random number generator with a period of 2^{19937} - 1
  - If you use 10^9 random number in a second, the sequence will only repeat after 10^{5985} years

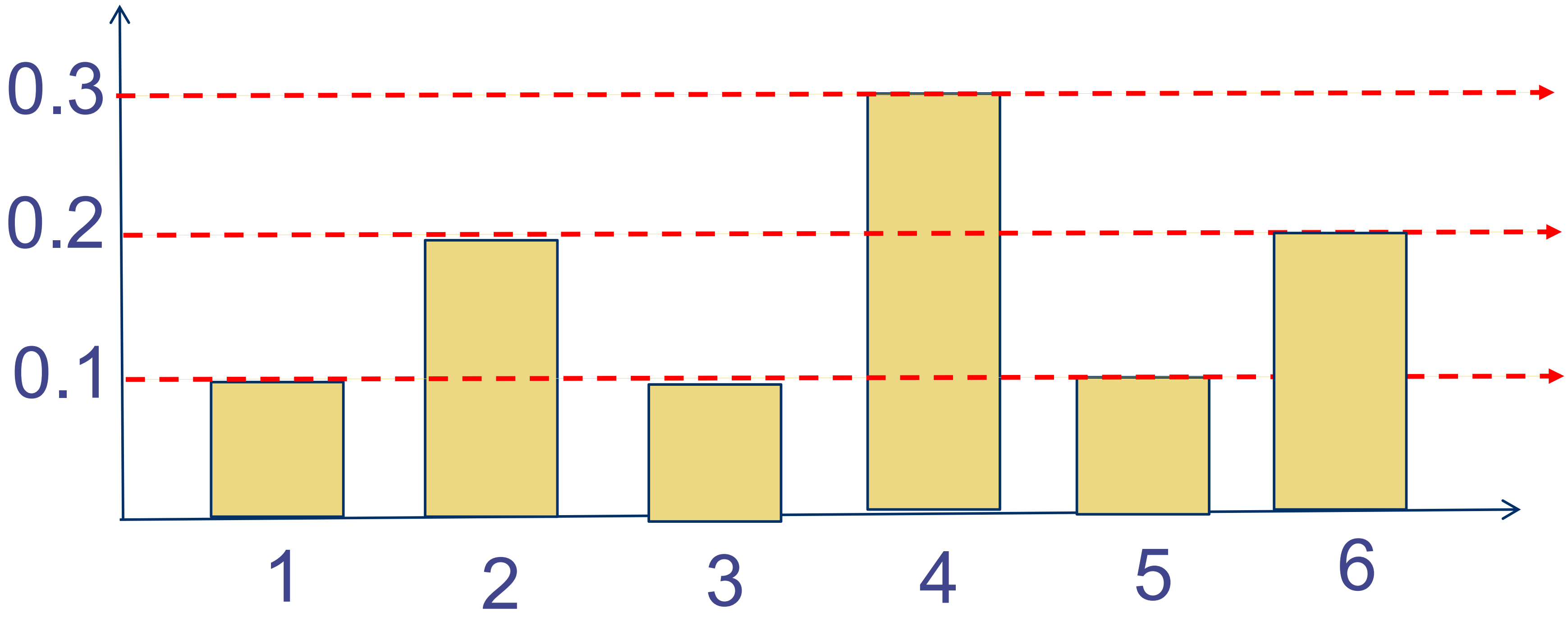
- Why are uniformly distributed random numbers important?
  - If you can generate uniformly distributed random numbers between (0,1), you can generate random numbers for any probability distribution

Fair coin distribution

- You can generate random numbers between 0 and 1
- You want to use these random numbers to imitate fair coin tossing, i.e.
    - Probability of HEAD = 0.6
    - Probability of TAIL = 0.4
- You can do this using the following algorithm
    - Generate a random number *u*
    - If u < [redacted], output HEAD
    - If u ≥ [redacted], output TAIL

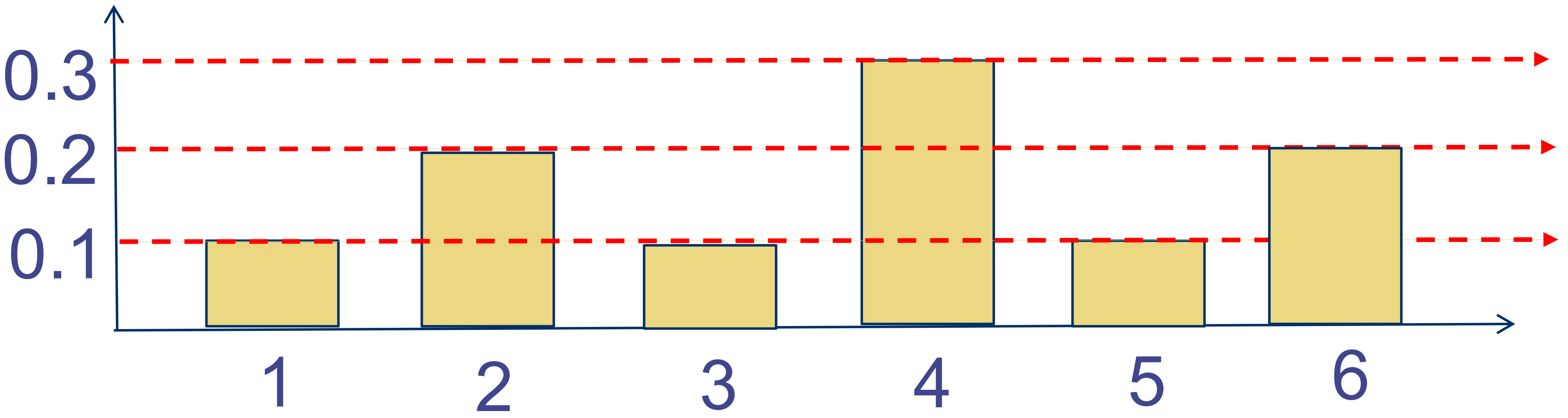
A loaded dice

- You want to create a loaded dice with probability mass function



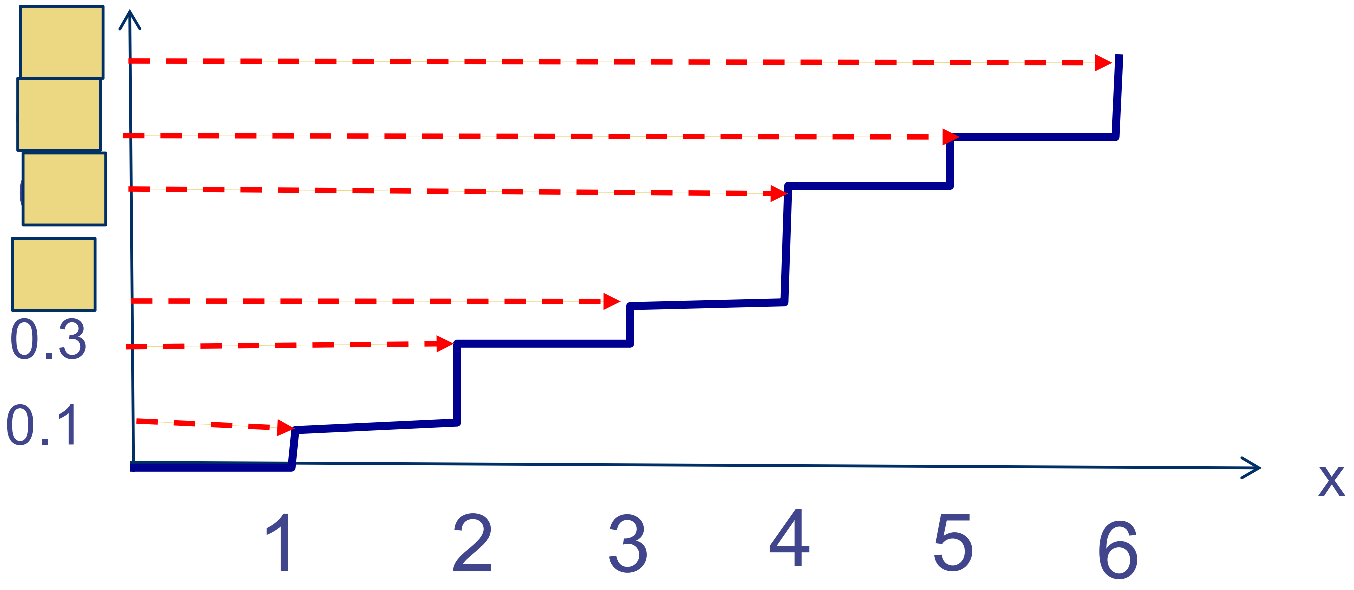
- The algorithm is:
  - Generate a random number u
  - If u < 0.1, output 1
  - If [redacted] ≤ u < [redacted], output 2
  - If [redacted] ≤ u < [redacted], output 3
  - If [redacted] ≤ u < [redacted], output 4
  - If [redacted] ≤ u < [redacted], output 5
  - If [redacted] ≤ u , output 6

Cumulative probability distribution



Probability that the dice gives a value ≤ x

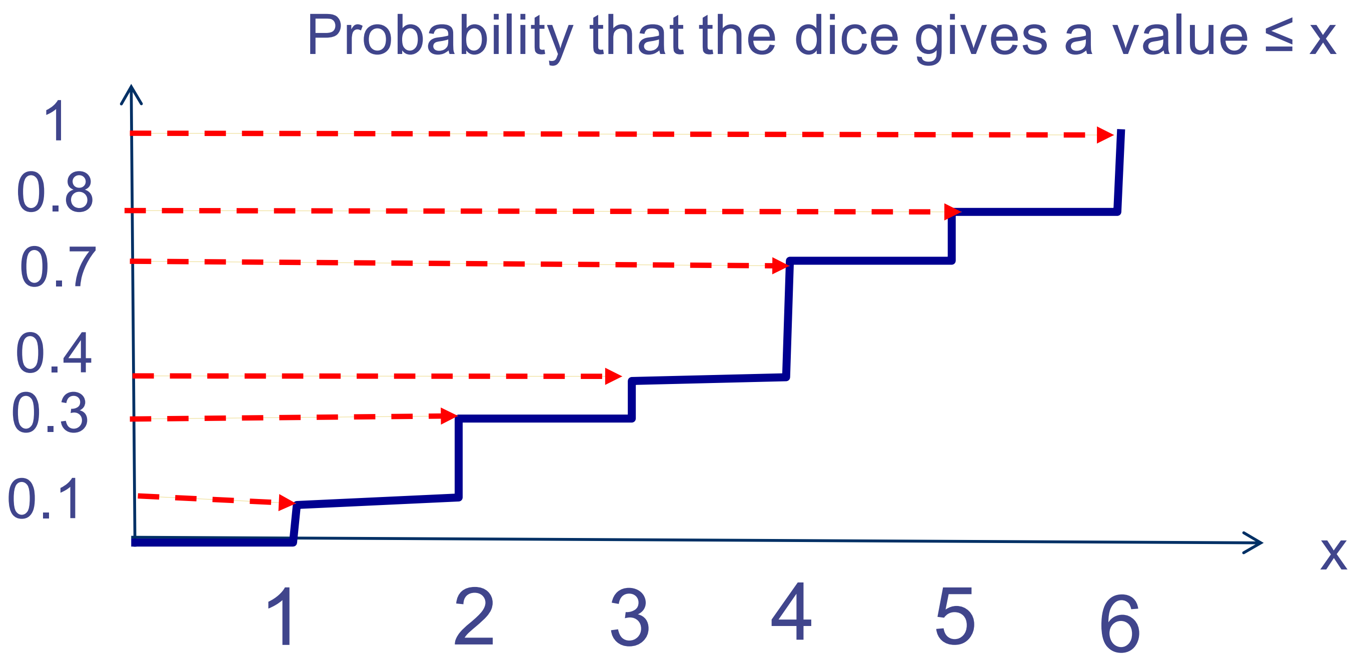
Ex: Can you work out what these levels should be



Comparing algorithm with cumulative distribution

- The algorithm is:
  - Generate a random number u
  - If $u < 0.1$, output 1
  - If $0.1 \leq u < 0.3$, output 2
  - If $0.3 \leq u < 0.4$, output 3
  - If $0.4 \leq u < 0.7$, output 4
  - If $0.7 \leq u < 0.8$, output 5
  - If $0.8 \leq u$, output 6

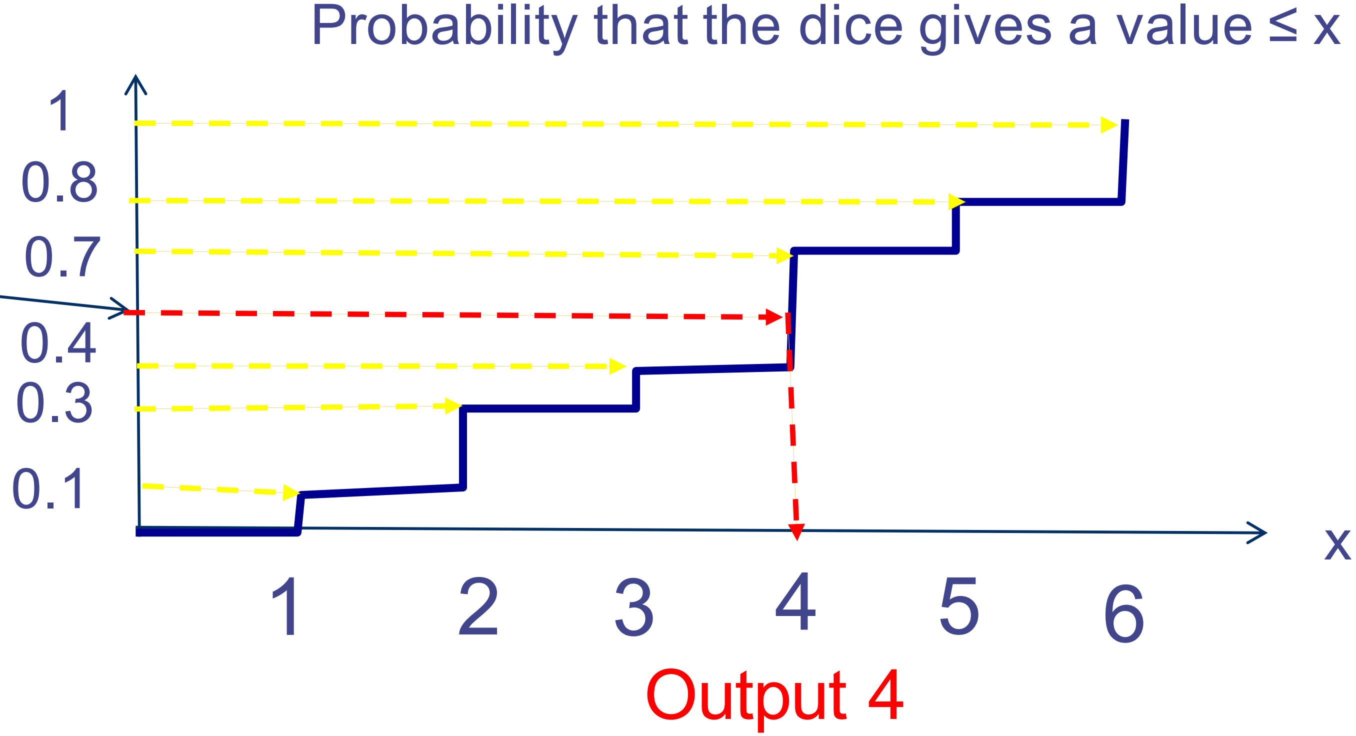
Ex: What do you notice about the intervals in the algorithm and the cumulative distribution?



Graphical interpretation of the algorithm

- The algorithm is:
  - Generate a random number u
  - If u < 0.1, output 1
  - If 0.1 ≤ u < 0.3, output 2
  - If 0.3 ≤ u < 0.4, output 3
  - If 0.4 ≤ u < 0.7, output 4
  - If 0.7 ≤ u < 0.8, output 5
  - If 0.8 ≤ u , output 6

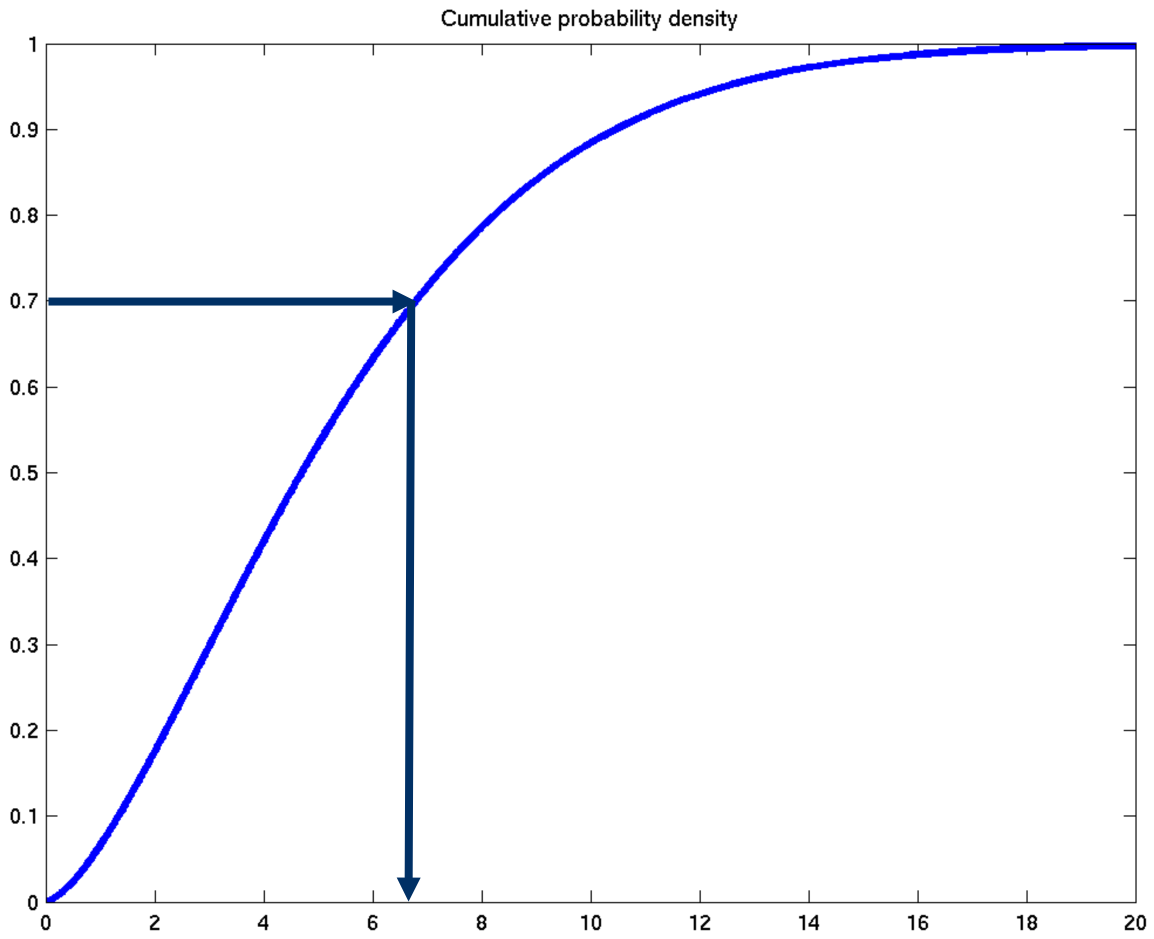
Ex: Let us assume u = 0.5126, what should the algorithm output?



Graphical representation of inverse transform method

- Consider the cumulative density function (CDF) y = *F(x)*, showed in the figure below

For this particular F(x), if u = 0.7 is generated then $F^{-1}(0.7)$ is 6.8



Inverse transform method

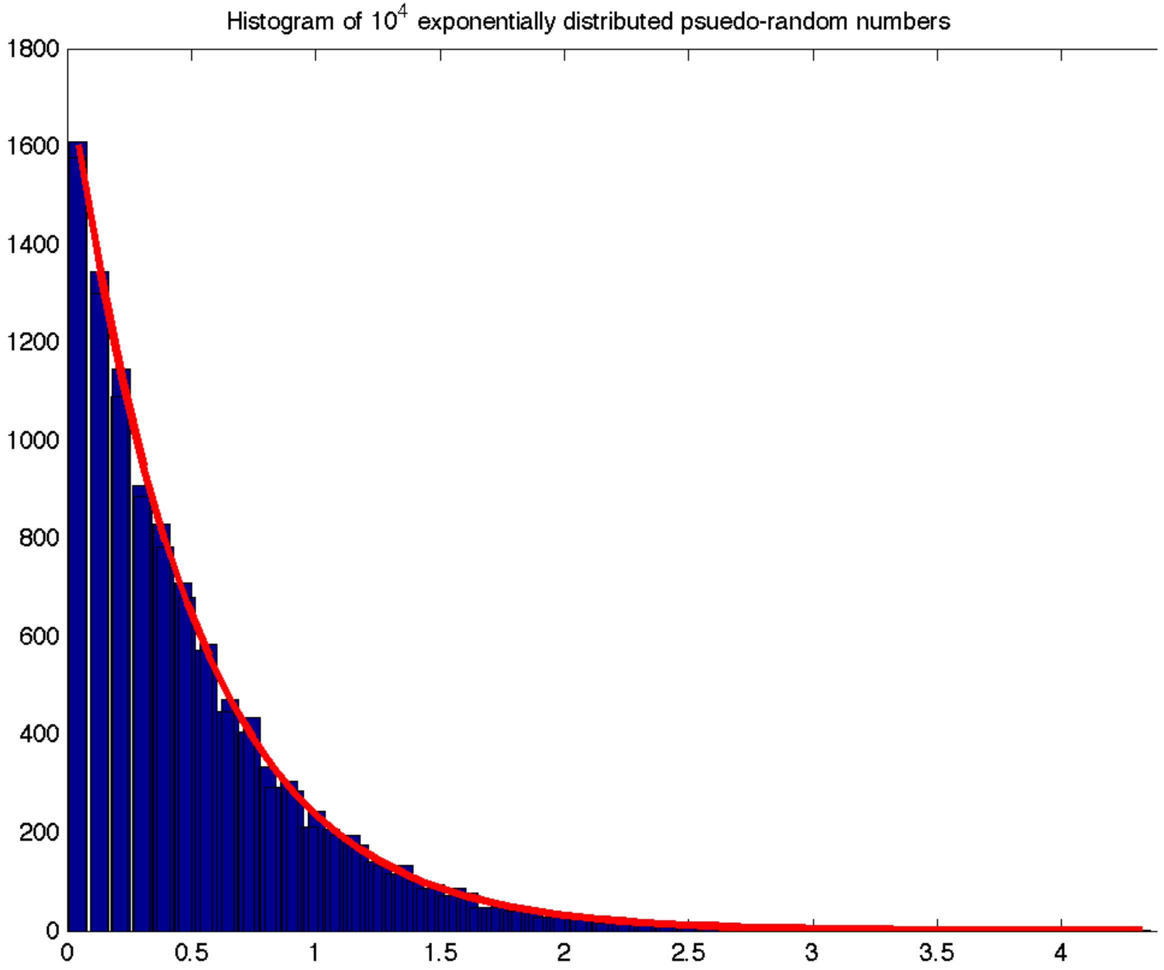
- A method to generate random number from a particular distribution is the *inverse transform method*
- In general, if you want to generate random numbers with cumulative density function (CDF) $F(x) = Prob[X \leq x]$, you can use the following procedure:
  - Generate a number u which is uniformly distributed in (0,1)
  - Compute the number $F^{-1}(u)$
- Example: Let us apply the inverse transform method to the exponential distribution
  - *CDF is* $1 - \exp(-\lambda x)$

Generating exponential distribution

- Given a sequence $\{U_1, U_2, U_3, \ldots\}$ which is uniformly distributed in (0,1)
- The sequence $-\log(1 - U_k)/\lambda$ is exponentially distributed with rate λ

- (Matlab file hist_expon.m)

1. Generate 10,000 uniformly distributed numbers in (0,1)
2. Compute $-\log(1-u_k)/2$ where u_k are the numbers generated in Step 1
3. The plot shows
   1. The histogram of the numbers generated in Step 2 in 50 bins
   2. The red line show the expected number of exponential distributed numbers in each bin



Putting everything together

- We know how to write a discrete event simulation program to simulate a single-server queue with infinite buffer
- We know how to generate random numbers
- This will allow us to simulate a G/G/1 queue provided that we can generate the probability distribution
- In order to test how well our discrete event simulation program works, we will use it to simulate an M/M/1 queue and compare it with the expected result
- An M/M/1 simulation program (based on Matlab) is given in *sim_mm1.m* (available on the course web site)

Reproducible simulation

- We run the simulation sim_mm1.m a few times, we get mean response times of 0.98623, 0.98445, 1.0034, …
- Each simulation run gives a different result because different set of random numbers is used
- In order to realise reproducibility of results, you can save the setting of the random number generator before simulation. If you reuse the setting later, you can reproduce the result

```
% obtain setting and save it in a file
rand_setting = rng;
save saved_rand_setting rand_setting
sim_mm1
```

```
% load the save setting and apply it
load saved_rand_setting
rng(rand_setting)
sim_mm1
```

References

- Discrete event simulation of single-server queue
  - Winston, “Operations Research”, Sections 23.1-23.2
  - Law and Kelton, “Simulation modelling and analysis”, Section 1.4
- Generation of random numbers
  - Raj Jain, “The Art of Computer Systems Performance Analysis”
    - Sections 26.1 and 26.2 on LCG
    - Section 28.1 on the inverse transform methods
- Note: We have only touched on the basic of discrete event simulations. For a more complete treatment, see
  - Law and Kelton, “Simulation modelling and analysis”
  - Harry Perros, “Computer Simulation Techniques: The definitive introduction”, an e-book that can be downloaded from
    - http://www4.ncsu.edu/~hp/files/simulation.pdf