
Cmpt-225

Simulation

Application: Simulation

■ Simulation

- A technique for modeling the behavior of both natural and human-made systems
 - Goal
 - Generate statistics that summarize the performance of an existing system
 - Predict the performance of a proposed system
 - Example
 - A simulation of the behavior of a bank
-

Application: Simulation

- The bank simulation is concerned with
 - Arrival events
 - Indicate the arrival at the bank of a new customer
 - External events: the input file specifies the times at which the arrival events occur
 - Departure events
 - Indicate the departure from the bank of a customer who has completed a transaction
 - Internal events: the simulation determines the times at which the departure events occur
-

Input file

Arrival	transaction length
20	5
22	4
23	2
30	3

Arrival	transaction length
20	5
22	4
23	2
30	3

Queue

C1

Time	Event
20	1 st Customer Arrives
25	1 st Customer leaves

Arrival	transaction length
20	5
22	4
23	2
30	3

Queue

C1
C2

Time	Event
20	1 st Customer Arrives
22	2 nd Customer Arrives
25	1 st Customer leaves

Arrival	transaction length
20	5
22	4
23	2
30	3

Queue

C1
C2
C3

Time	Event
20	1 st Customer Arrives
22	2nd Customer Arrives
23	3rd Customer Arrives
25	1 st Customer leaves

Arrival	transaction length
20	5
22	4
23	2
30	3

Queue

C2
C3

Time	Event
20	1 st Customer arrives
22	2 nd Customer arrives
23	3 rd Customer arrives
25	1 st Customer leaves
29	2 nd Customer leaves

Arrival	transaction length
20	5
22	4
23	2
30	3

Queue

C3

Time	Event
20	1 st Customer arrives
22	2 nd Customer arrives
23	3 rd Customer arrives
25	1 st Customer leaves
29	2 nd Customer leaves
31	3 rd Customer leaves

Arrival	transaction length
20	5
22	4
23	2
30	3

Queue

C3
C4

Time	Event
20	1 st Customer arrives
22	2 nd Customer arrives
23	3 rd Customer arrives
25	1 st Customer leaves
29	2 nd Customer leaves
30	4 th Customer arrives
31	3 rd Customer leaves

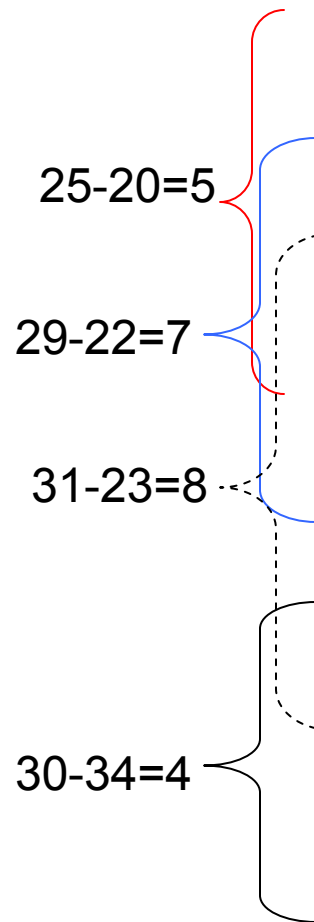
Arrival	transaction length
20	5
22	4
23	2
30	3

Queue

C4

Time	Event
20	1 st Customer arrives
22	2 nd Customer arrives
23	3 rd Customer arrives
25	1 st Customer leaves
29	2 nd Customer leaves
30	4 th Customer arrives
31	3 rd Customer leaves
34	4 th Customer leaves

Arrival	transaction length
20	5
22	4
23	2
30	3



Average wait:
 $(5 + 7 + 8 + 4) / 4 = 6$

Time	Event
20	1 st Customer arrives
22	2 nd Customer arrives
23	3 rd Customer arrives
25	1 st Customer leaves
29	2 nd Customer leaves
30	4 th Customer arrives
31	3 rd Customer leaves
34	4 th Customer leaves

```
while (events remain to be processed){  
    currentTime=time of the next event;  
    if(event is an arrival event)  
        process the arrival event  
    else  
        process the departure event  
}
```

Application: Simulation

- An event list is needed to implement an event-driven simulation
 - An event list
 - Keeps track of arrival and departure events that will occur but have not occurred yet
 - Contains at most one arrival event and one departure event

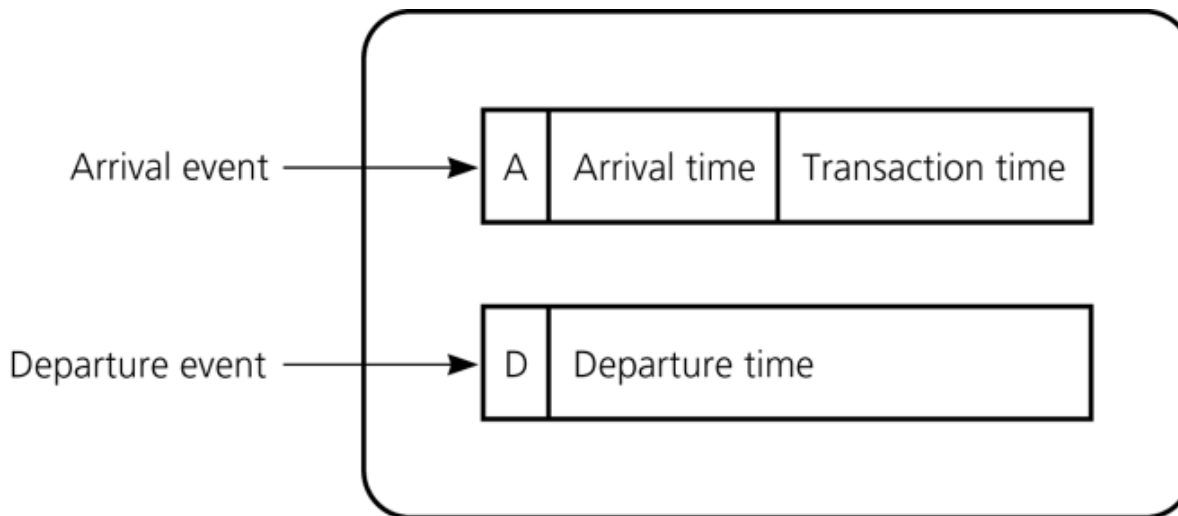


Figure 8-15
A typical instance
of the event list

■ When a customer arrives

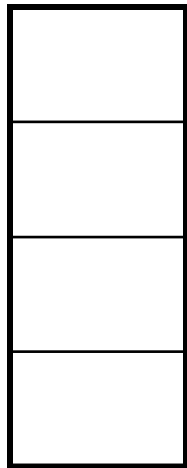
- ❑ Put him/her in the queue to be served.
- ❑ If the queue is empty then schedule his/her departures.
- ❑ Schedule the arrival of the next customer.

■ When a customer leaves

- ❑ Remove him/her from the queue.
 - ❑ Schedule the departure of the next customer in the queue if there's any.
-

Arrival	transaction length
20	5
22	4
23	2
30	3

Queue

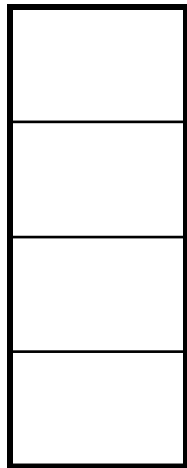


Time	Event
20	1 st Customer arrives

Current Event

Arrival	transaction length
20	5
22	4
23	2
30	3

Queue



Time	Event

Current Event
1 st Customer arrives, 20

Arrival	transaction length
20	5
22	4
23	2
30	3

Queue

C1

Time	Event
22	2 nd Customer arrives
25	1 st Customer leaves

Current Event
1 st Customer arrives

Arrival	transaction length
20	5
22	4
23	2
30	3

Queue

C1

Time	Event
25	1 st Customer leaves

Current Event
2 nd Customer arrives, 22

Arrival	transaction length
20	5
22	4
23	2
30	3

Queue

C1
C2

Time	Event
23	3 rd Customer arrives
25	1 st Customer leaves

Current Event
2 nd Customer arrives, 22

Arrival	transaction length
20	5
22	4
23	2
30	3

Queue

C1
C2

Time	Event
25	1 st Customer leaves

Current Event
3 rd Customer arrives, 23

Arrival	transaction length
20	5
22	4
23	2
30	3

Queue

C1
C2
C3

Time	Event
25	1 st Customer leaves
30	4 th Customer arrives

Current Event
3 rd Customer arrives, 23

Arrival	transaction length
20	5
22	4
23	2
30	3

Queue

C1
C2
C3

Time	Event
30	4 th Customer arrives

Current Event
1 st Customer leaves, 25

Arrival	transaction length
20	5
22	4
23	2
30	3

Queue

C2
C3

Time	Event
29	2 nd Customer leaves
30	4 th Customer arrives

Current Event
1 st Customer leaves, 25

Arrival	transaction length
20	5
22	4
23	2
30	3

Queue

C2
C3

Time	Event
29	2 nd Customer leaves
30	4 th Customer arrives

Current Event
2 nd Customer leaves, 29

Arrival	transaction length
20	5
22	4
23	2
30	3

Queue

C3

Time	Event
30	4 th Customer arrives
31	3 rd Customer leaves

Current Event
2 nd Customer leaves, 29

Arrival	transaction length
20	5
22	4
23	2
30	3

Queue

C3

Time	Event
30	4 th Customer arrives
31	3 rd Customer leaves

Current Event
4 th Customer arrives, 30

Arrival	transaction length
20	5
22	4
23	2
30	3

Queue

C3
C4

Time	Event
31	3 rd Customer leaves

Current Event
4 th Customer arrives, 30

Arrival	transaction length
20	5
22	4
23	2
30	3

Queue

C3
C4

Time	Event

Current Event
3 rd Customer leaves, 31

Arrival	transaction length
20	5
22	4
23	2
30	3

Queue

C4

Time	Event
34	4 th Customer leaves

Current Event
3 rd Customer leaves, 31

Arrival	transaction length
20	5
22	4
23	2
30	3

Queue

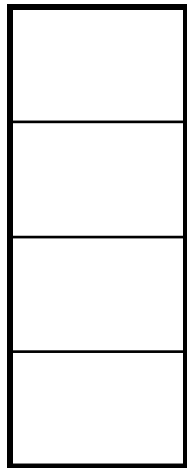
C4

Time	Event

Current Event
4 th Customer leaves, 34

Arrival	transaction length
20	5
22	4
23	2
30	3

Queue



Time	Event

Current Event





Algorithm Analysis: Big O Notation

Objectives

- Determine the running time of simple algorithms in the:
 - Best case
 - Average case
 - Worst case
 - Understand the mathematical basis of O notation
 - Use O notation to measure the running time of algorithms
-

Algorithm Analysis

- It is important to be able to describe the efficiency of algorithms
 - Time efficiency
 - Space efficiency
 - Choosing an appropriate algorithm can make an enormous difference in the usability of a system e.g.
 - Government and corporate databases with many millions of records, which are accessed frequently
 - Online search engines
 - Real time systems (from air traffic control systems to computer games) where near instantaneous response is required
-

Measuring Efficiency of Algorithms

- It is possible to time algorithms
 - `System.currentTimeMillis()` returns the current time so can easily be used to measure the running time of an algorithm
 - More sophisticated timer classes exist
 - It is possible to count the number of operations that an algorithm performs
 - Mathematically calculate the number of operations.
 - Printing the number of times that each line executes (profiling)
-

Timing Algorithms

- It can be very useful to time how long an algorithm takes to run
 - In some cases it may be essential to know how long a particular algorithm takes on a particular system
 - However, it is not a good general method for comparing algorithms
 - Running time is affected by numerous factors
 - How are the algorithms coded?
 - What computer should we use?
 - CPU speed, memory, specialized hardware (e.g. graphics card)
 - Operating system, system configuration (e.g. virtual memory), programming language, algorithm implementation
 - Other tasks (i.e. what other programs are running), timing of system tasks (e.g. memory management)
 - What data should we use?
-

Cost Functions

- Because of the sorts of reasons just discussed for general comparative purposes we will count, rather than time, the **number of operations** that an algorithm performs
 - Note that this does not mean that actual running time should be ignored!

For a linked list of size n

```
Node curr=head;
```

1 assignment

```
while (curr != null) {
```

n+1 comparisons

```
    System.out.println(curr.getItem());
```

n prints

```
    curr = cur.getNext();
```

n assignments

```
}
```

If each assignment, comparison and print operation requires a, c, and p time units then the above code requires $(n+1)*a + (n+1)*c + n*p$ *units of time*.

Cost Functions

- For simplicity we assume that each operation take one unit of time.
 - If algorithm (on some particular input) performs t operations, we will say that it runs in **time t** .
 - Usually running time t depends on the data size (the input length).
 - We express the time t as a **cost function** of the data size n
 - We denote the cost function of an algorithm A as $t_A()$, where $t_A(n)$ is the time required to process the data with algorithm A on input of size n
 - Typical example of the input size: number of nodes in a linked list, number of disks in a Hanoi Tower problem, the size of an array, the number of items in a stack, the length of a string, ...
-

Nested Loop

```
for (i=1 through n) {  
    for (j=1 through i) {  
        for (k=1 through 5) {  
            Perform task T;  
        }  
    }  
}
```

- If task T requires t units of time, the inner most loop requires $5 \cdot t$ time units and the loop on j requires $5 \cdot t \cdot i$ time units.
- Therefore, the outermost loop requires

$$\sum_{i=1}^n (5 \cdot t \cdot i) = 5 \cdot t \cdot (1 + 2 + \dots + n) = 5 \cdot t \cdot n \cdot (n + 1) / 2$$

Best, Average and Worst Case

- The amount of work performed by an algorithm may vary based on its input (not only on its size)
 - This is frequently the case (but not always)
 - Algorithm efficiency is often calculated for three, general, cases of input
 - Best case
 - Average (or “usual”) case
 - Worst case
-

Algorithm Growth Rates.

- We often want to compare the performance of algorithms
 - When doing so we generally want to know how they perform when the problem size (n) is large
 - So it's simpler if we just find out how the algorithms perform as the input size **grows- *the growth rate***.
-

-
- E.g.
 - Algorithm A requires $n^2/5$ time units to solve a problem of size n
 - Algorithm B requires $5*n$ time units to solve a problem of size n
 - It may be difficult to come up with the above conclusions and besides they do not tell us the exact performance of the algorithms A and B.
 - It will be easier to come up with the following conclusion for algorithms A and B
 - Algorithm A requires time proportional to n^2
 - Algorithm B requires time proportional to n
 - From the above you can determine that for large problems B requires significantly less time than A.
-

Algorithm Growth Rates

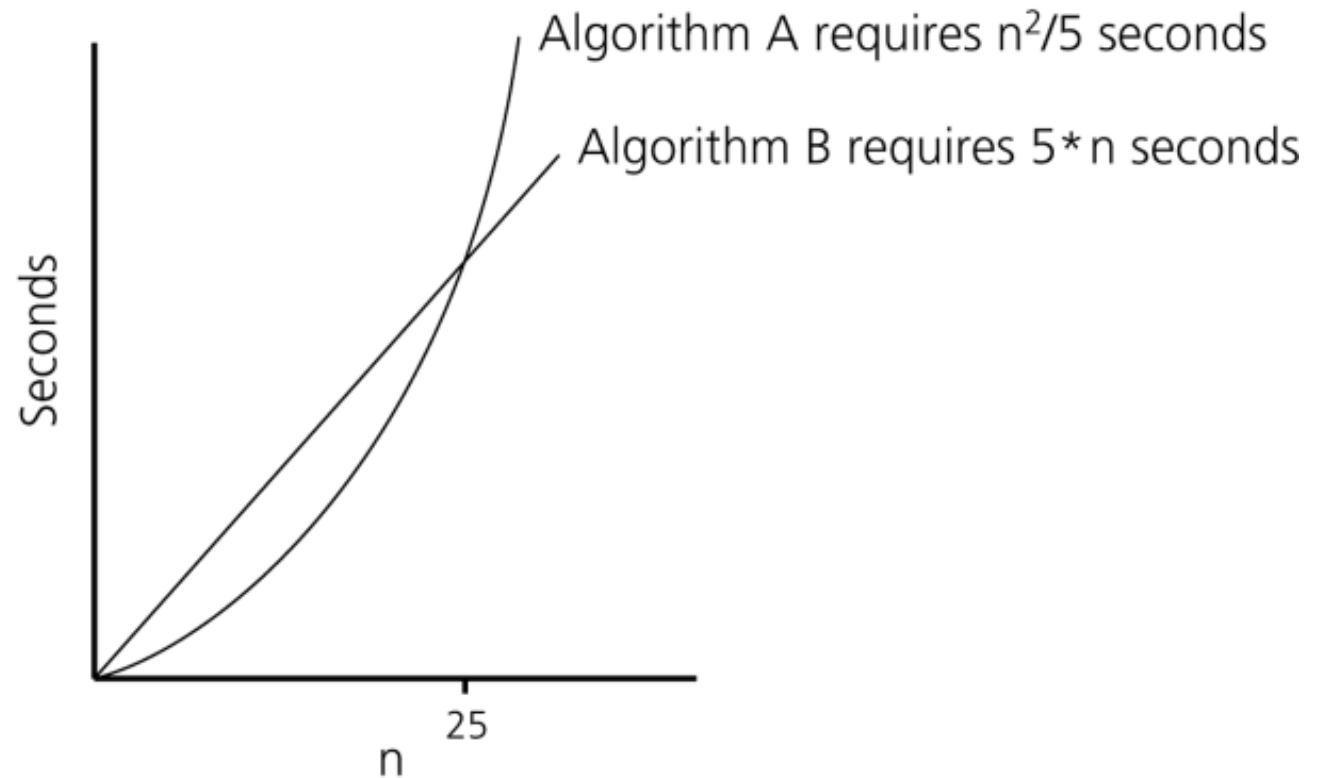


Figure 10-1

Time requirements as a function of the problem size n

-
- Since cost functions are complex, and may be difficult to compute, we approximate them using O notation – O notation determines the growth rate of an algorithm time.
-

Example of a Cost Function

- Cost Function: $t_A(n) = n^2 + 20n + 100$
 - Which term dominates?
 - It depends on the size of n
 - $n = 2$, $t_A(n) = 4 + 40 + \underline{100}$
 - The constant, 100, is the dominating term
 - $n = 10$, $t_A(n) = 100 + \underline{200} + 100$
 - $20n$ is the dominating term
 - $n = 100$, $t_A(n) = \underline{10,000} + 2,000 + 100$
 - n^2 is the dominating term
 - $n = 1000$, $t_A(n) = \underline{1,000,000} + 20,000 + 100$
 - n^2 is the dominating term
-

Big O Notation

- O notation approximates the cost function of an algorithm
 - The approximation is usually good enough, especially when considering the efficiency of algorithm as n gets very large
 - Allows us to estimate rate of function growth
 - Instead of computing the entire cost function we only need to count the number of times that an algorithm executes its *barometer instruction(s)*
 - The instruction that is executed the most number of times in an algorithm (the highest order term)
-

Big O Notation

- Given functions $t_A(n)$ and $g(n)$, we can say that the efficiency of an algorithm is *of order* $g(n)$ if there are positive constants c and m such that
 - $t_A(n) < c.g(n)$ for all $n > m$
 - we write
 - $t_A(n)$ is $O(g(n))$ and we say that
 - $t_A(n)$ is of order $g(n)$
 - e.g. if an algorithm's running time is $3n + 12$ then the algorithm is $O(n)$. If $c=3$ and $m=12$ then $g(n) = n$:
 - $4 * n \geq 3n + 12$ for all $n \geq 12$
-

In English...

- The cost function of an algorithm A, $t_A(n)$, can be approximated by another, simpler, function $g(n)$ which is also a function with only 1 variable, the data size n .
 - The function $g(n)$ is selected such that it represents an *upper bound* on the efficiency of the algorithm A (i.e. an upper bound on the value of $t_A(n)$).
 - This is expressed using the big-O notation: $O(g(n))$.
 - For example, if we consider the time efficiency of algorithm A then “ $t_A(n)$ is $O(g(n))$ ” would mean that
 - A cannot take more “time” than $O(g(n))$ to execute or that (more than $c \cdot g(n)$ for some constant c)
 - the cost function $t_A(n)$ grows *at most as fast as* $g(n)$
-

The general idea is ...

- when using Big-O notation, rather than giving a precise figure of the cost function using a specific data size n
 - express the behaviour of the algorithm as its data size n grows very large
 - so ignore
 - lower order terms and
 - constants
-