MERRIMACK COLLEGE

CSC6023 - Advanced Algorithms

Amortized Analysis

Patrick Neff

Amortized Analysis

This is a very fast paced course, After this five week topic and we are ready to start the course ending. So, let’s see the amortized analysis, and slightly different point of view.

MERRIMACK COLLEGE

Agenda Of The Presentation

Limitations of Complexity through Asymptotics

- Sort Algorithms Complexity
    a. Big Oh, Theta, and Omega, and yet …

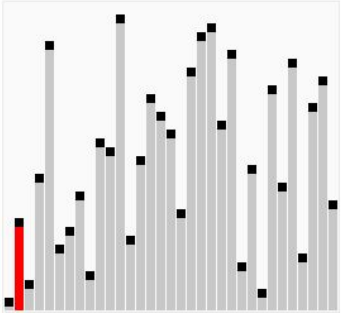
Amortized Analysis

- The Basics
- Examples
    a. Dynamic Arrays
    b. Double Array Queues
- Offline and Online Algorithms
    a. Selection vs Insertion Sort Algorithms
        i. Big Oh
        ii. Amortized Analysis
        iii. Practical Issues

MERRIMACK COLLEGE

Limitations

The Sort Algorithms Comparison



MERRIMACK COLLEGE

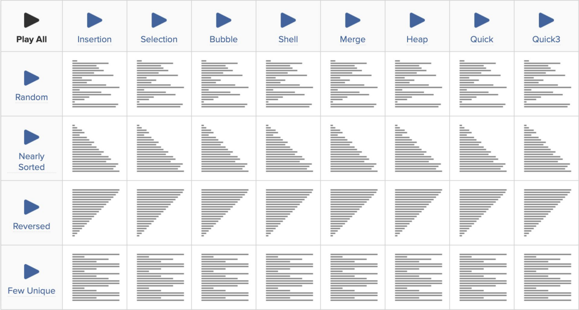
Limitations of Complexity through Asymptotics

- When is an algorithm better than the others?
- For example, the sort algorithms

Sort Algorithms	O	Θ	Ω
Insertion	n^2	n^2	n
Selection	n^2	n^2	n^2
Bubble	n^2	n^2	n
Shell	$n \log n$	$n \log n$	n
Merge	$n \log n$	$n \log n$	$n \log n$
Heap	$n \log n$	$n \log n$	$n \log n$
Quick	n^2	$n \log n$	$n \log n$

Limitations

The Sort Algorithms Comparison



Click the image

MERRIMACK COLLEGE

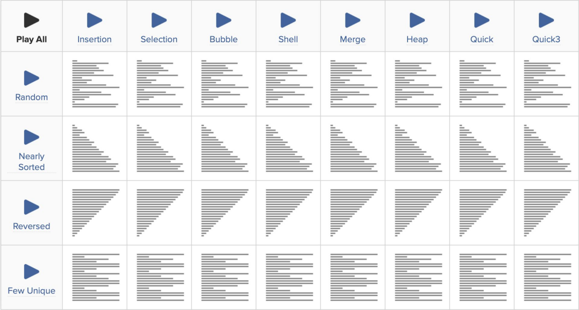
Limitations of Complexity through Asymptotics

- When is an algorithm better than the others?
- For example, the sort algorithms

Sort Algorithms	O	Θ	Ω
Insertion	n^2	n^2	n
Selection	n^2	n^2	n^2
Bubble	n^2	n^2	n
Shell	$n \log n$	$n \log n$	n
Merge	$n \log n$	$n \log n$	$n \log n$
Heap	$n \log n$	$n \log n$	$n \log n$
Quick	n^2	$n \log n$	$n \log n$

Limitations

The Sort Algorithms Comparison



Click the image

MERRIMACK COLLEGE

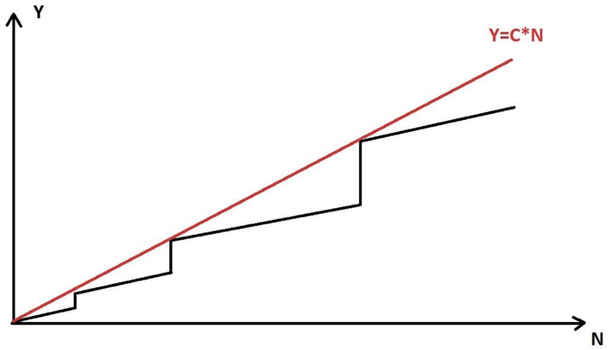
Which Algorithm is the Best One

- We need a different way to estimate...

Sort Algorithms	O	Θ	Ω
Insertion	n^2	n^2	n
Selection	n^2	n^2	n^2
Bubble	n^2	n^2	n
Shell	$n \log n$	$n \log n$	n
Merge	$n \log n$	$n \log n$	$n \log n$
Heap	$n \log n$	$n \log n$	$n \log n$
Quick	n^2	$n \log n$	$n \log n$

Amortized Analysis

What if we compute it differently?



MERRIMACK COLLEGE

The Basics

- The worst case (***Big Oh***) may be too pessimistic
- The best case (***Big Omega***) may be too optimistic
- The average case (***Big Theta***) may be not representative (average and median distant)

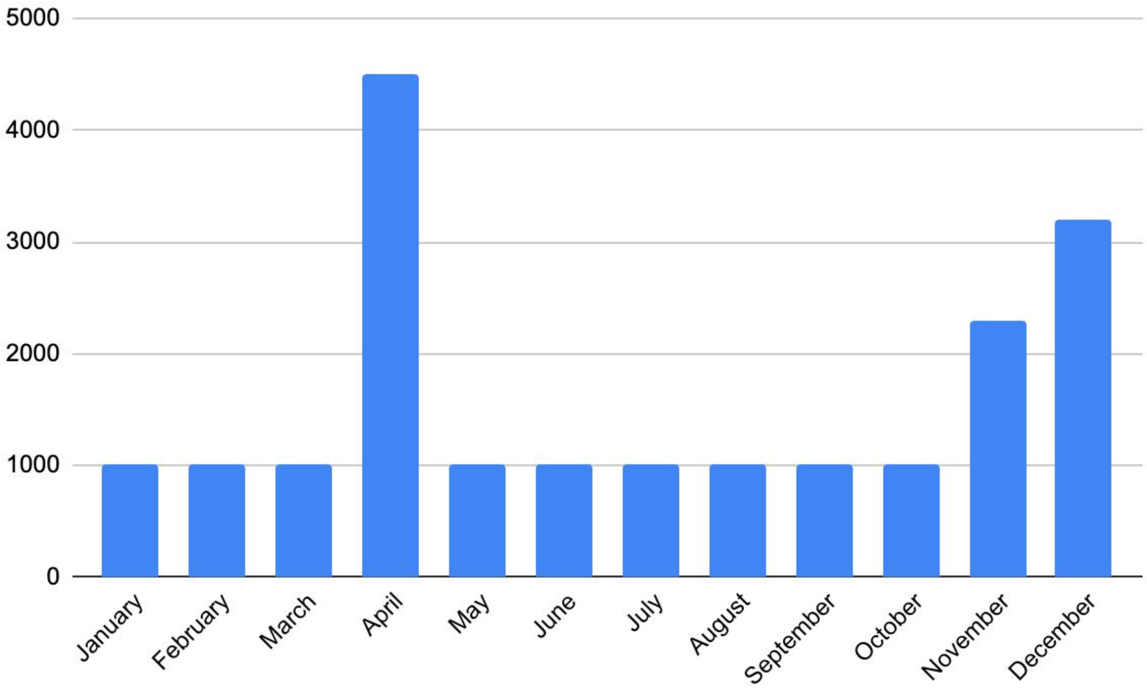
- Amortized Analysis is an alternative method for analysing the algorithm complexity
  - It averages the running times in a sequence

- While usual asymptotics analysis date from the very beginning of Computer Science, Amortized Analysis was formally introduced around mid 80's
  - Aggregate Method
  - Accounting Method
  - Potential Method

Amortized Analysis: Example from real life

Most months have $1000 expense but it is not enough to make $1000/month to live.

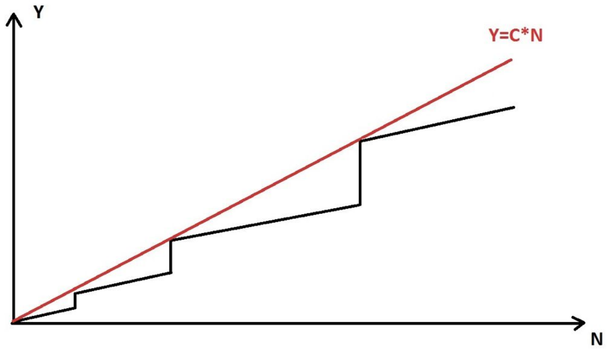
The amortized cost is: $1583.34 so you need that amount so that you don't go broke.



MERRIMACK COLLEGE

Amortized Analysis

What if we compute it differently?



MERRIMACK COLLEGE

The Basics

- Kinds of Amortized Analysis
  - Aggregate Method - Compute the upper bound *T(n)* on total of n operations, then consider *T(n)/n*
  - Accounting Method - Compute the number of operations for each case of the sequence keeping a tab of operation credits
  - Potential Method - Similar to the accounting method, but the credit is expressed as a function, called "potential"
- Typical cases where amortized analysis is useful are those when execution is seldom costly
  - Dynamic Arrays
  - Double Array Queues

Examples

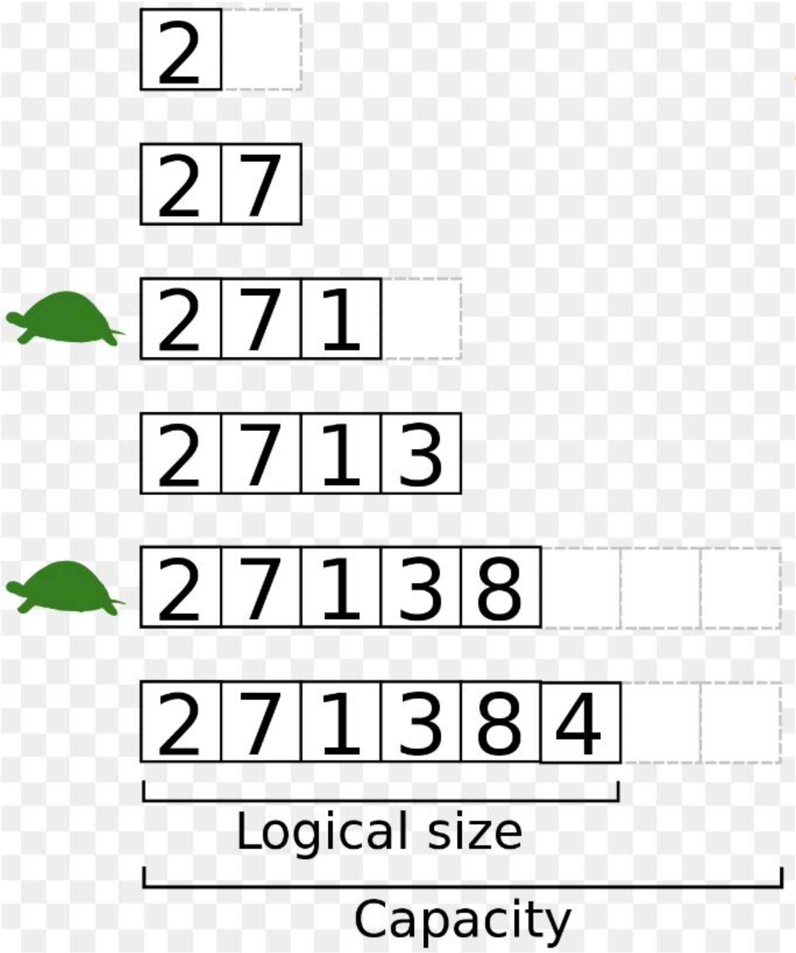
Dynamic Arrays

A Common Case in Many Programming Languages

- Given a fixed sized array, what happens when we append new elements?
- Python implementation (and similar structures in other languages, as STL::vector in C++) of arrays frequently deals with such flexible sized structures
- When you create an array of size *n*, it allocates *n+x* memory holders (usually) for comfortably storing the *n* elements
  - If you remove an element, it just marks it logically (no memory release is actually done)
  - If you append a new element, either it increases the logical number of elements, or it allocates a double sized space and copies the old contents to it

MERRIMACK COLLEGE

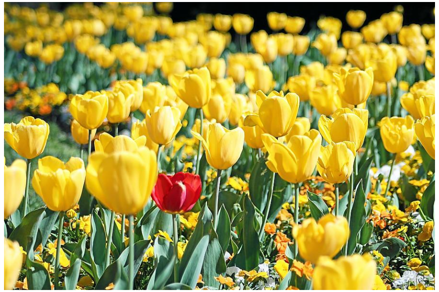
Dynamic Arrays



MERRIMACK COLLEGE

Examples

Dynamic Arrays



MERRIMACK COLLEGE

How much does an append cost?

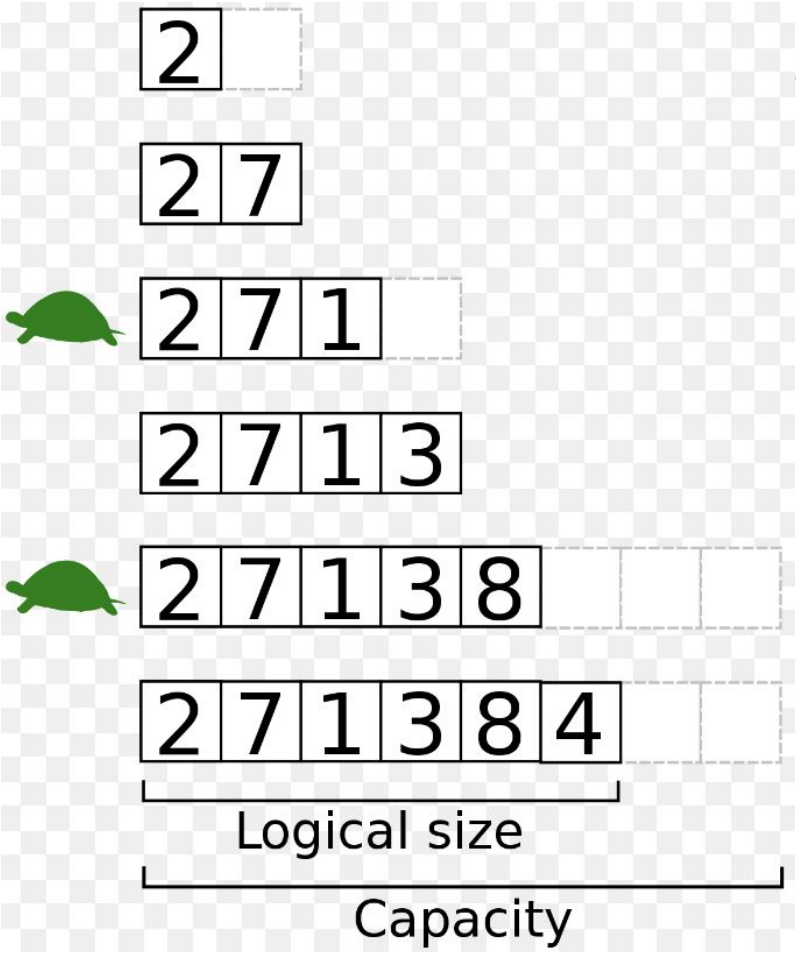
- If the element inclusion is only logical: ***O(1)***
- If it requires new memory allocation: ***O(n)***

```python
def myAppend(a, s, d):
    # a is the array memory allocated
    # s is the logical size of a
    # d is the data to be appended
    if (s < len(a)):
        a[s] = d        # it includes a new item
        s += 1          # it increases the logical size of a
    else:
        a = a + a       # it allocates a new memory twice as big
        a[s] = d        # it includes a new item
        s += 1          # it increases the logical size of a
    return a, s
```

- As the number of executions grows (and so does *n*), the amortized analysis approaches ***O(1)***

Dynamic Arrays

Let’s look at some code.
Dynamic Array



Examples

Dynamic Arrays

- Let's consider the frequency of costly cases
- This simulation code keeps track of the array's logical size (*s*) and the allocated memory size (*m*) and randomly does a remove or an append with different probabilities

How frequent is an append?

```
from random import randrange

def test(initialSize, probRemove):
    accCheap, accCosty = 0, 0
    s = initialSize
    m = 2*s
    for i in range(100000):
        if (randrange(100) < probRemove):
            if (s > 0):
                s -= 1
        else:
            if (s == m):
                m = m*2
                s += 1
                accCosty += 1
            else:
                s += 1
                accCheap += 1
    print("Initial size:", initialSize, "Prob Remove:", probRemove, "out of 100")
    print("Costy: {:7} ({:3.1}%)".format(accCosty,100*accCosty/(accCosty+accCheap)))
    print("Cheap: {:7} ({:3.1}%)".format(accCheap,100*accCheap/(accCosty+accCheap)))
```

```
Initial size: 20 Prob Remove: 1 out of 100
Costy:      12 (0.01%)
Cheap:   98982 (1e+02%)
Initial size: 20 Prob Remove: 5 out of 100
Costy:      12 (0.01%)
Cheap:   94923 (1e+02%)
Initial size: 50 Prob Remove: 1 out of 100
Costy:      10 (0.01%)
Cheap:   99014 (1e+02%)
Initial size: 50 Prob Remove: 5 out of 100
Costy:      10 (0.01%)
Cheap:   94978 (1e+02%)
Initial size: 100 Prob Remove: 1 out of 100
Costy:       9 (0.009%)
Cheap:   98980 (1e+02%)
Initial size: 100 Prob Remove: 5 out of 100
Costy:       9 (0.009%)
Cheap:   95019 (1e+02%)
```

Full code to test available in the content area of module.

Dynamic Array

Task #1 for this week's In-class exercises

- Run Dynamic Array test function trying to find values of initialSize and probRemove that delivers a probability of costly operations of at least 1%

Full code to edit available in content area of module.

Let's look at some code.
Dynamic Array Test

Go to IDLE and edit the code trying to find the desired result (Costly >= 1%)
Save your program in a .py file and submit it in the appropriate delivery room

Deadline: This Friday 11:59 PM EST

New data structure ahead!

Examples

Double Array Queue

- Each enqueue is ***O(1)***
- Each dequeue is either:
  - ***O(1)*** if the *self.a_out* is not empty
  - ***O(n)*** otherwise
- How rare is the costly operation?

MERRIMACK COLLEGE

Let's look at some code.

Implementing a queue with two arrays

- Among several implementations of a queue, there is an option using a two arrays

```
class Queue:
    def __init__(self):
        self.a_in = []
        self.a_out = []
    def enqueue(self, data):
        self.a_in.append(data)
        print(self.a_in)
        print(self.a_out)
    def dequeue(self):
        if (self.a_out == []):
            while len(self.a_in) > 0:
                self.a_out.append(self.a_in.pop())
        print(self.a_in)
        print(self.a_out)
        return self.a_out.pop()
```

Fifth Assignment

Project #5 - this week's Assignment

- Implement a Double Array Queue and test it for a very large case (100,000 randomly decided operations of enqueue or dequeue)
- Your program should compute the number of costly operations and cheap operations
- Your program should also ask the user about the ratio between enqueue and dequeue operations
  - The probability of enqueues and dequeues should never be of less than half the other
  - (67% enqueues - 33% dequeues or vice versa)

Fifth Assignment

Project #5 - this week's Assignment

- This program must be your own, do not use someone else's code
- Any specific questions about it, please bring to the Office hours meeting this Friday or contact me by email
- This is a challenging program to make sure you are mastering your Python programming skills, as well as your asymptotic analysis understanding
- Don't be shy with your questions

Go to IDLE and try to program it
Save your program in a .py file and submit it in the appropriate delivery room

Deadline: Next Tuesday 11:59 PM EST

When Amortized Analysis is Useful?

Let’s look at some code…
And then read this slide again!
Let’s also look at the simulation again.

MERRIMACK COLLEGE

Offline Algorithms

- An offline algorithm receives a full data package and performs the required task
  - All information is known beforehand
    - For example, *selection sort*

Online Algorithms

- An online algorithm receives data portions as the required task is being performed
  - The full information is not known beforehand
    - For example, *insertion sort*

Offline and Online Algorithms

When Amortized Analysis is Useful?

Selection versus Insertion Sort

- Selection sort chooses the smallest element each time and place it in the front (swap)
- Insertion sort compares elements and inserts the element until it is in the right place of the sorted subarray

Sort Algorithms	O	Θ	Ω
Insertion	n^2	n^2	n
Selection	n^2	n^2	n^2

When Amortized Analysis is Useful?

Selection Sort

- Selection sort chooses the smallest element each time and place it in the front (swap)
- It is necessary to know all elements to find the smallest element at each time
- If it is already sorted, nothing to be done

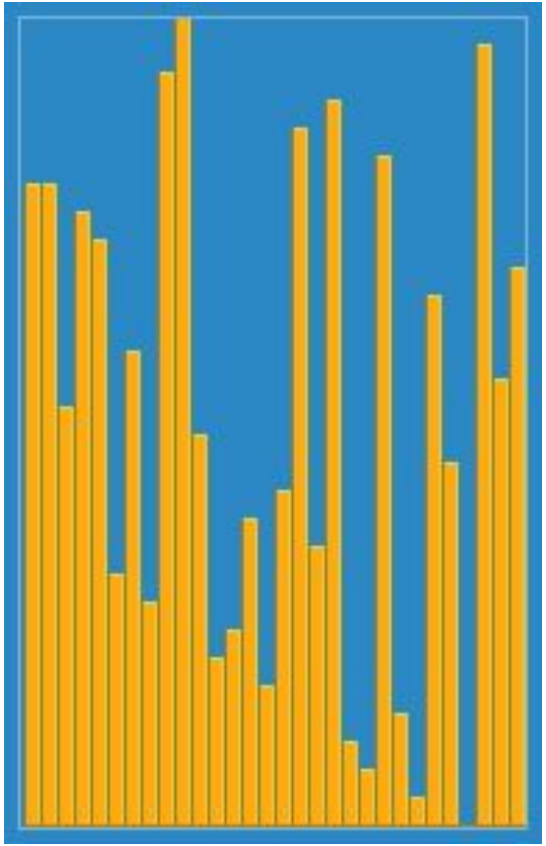
	8
	5
	2
	6
	9
	3
	1
	4
	0
	7

MERRIMACK COLLEGE

When Amortized Analysis is Useful?

Insertion Sort

- Insertion sort compares elements and inserts the element until it is in the right place of the sorted subarray
- It can be applied to each element at time, comparing only with the already sort elements



MERRIMACK COLLEGE

Offline and Online Algorithms

When Amortized Analysis is Useful?

Amortized Analysis of Selection Sort

- Each time that you select the smallest element you perform a single swap
  - To find the smallest element the same cost is always needed
    - ***O(n)***
  - The swap has a constant cost
    - ***O(1)***

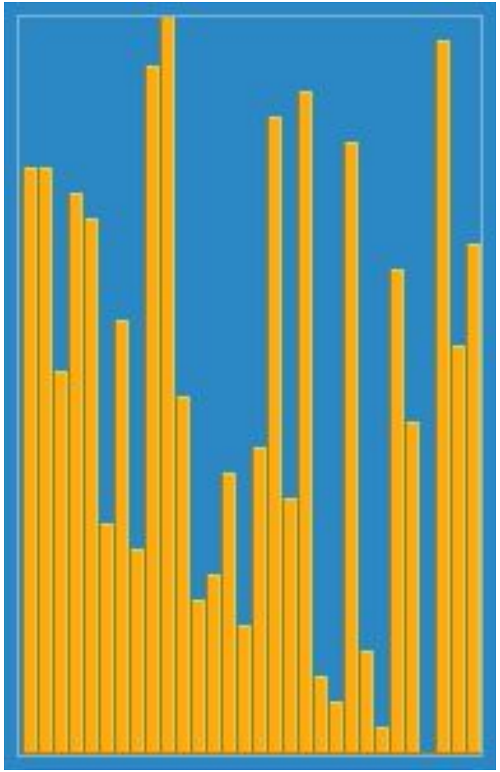
	8
	5
	2
	6
	9
	3
	1
	4
	0
	7

MERRIMACK COLLEGE

When Amortized Analysis is Useful?

Amortized Analysis of Insertion Sort

- Each time that you find the current place of an element you perform a search in the sorted subarray, which can be fast or slow
  - To find the place
    - ***O(1)*** to ***O(n)***
  - For a nearly sorted array several steps will be ***O(1)***, while others will be ***O(n)***



MERRIMACK COLLEGE

Let’s check it out the sorting algorithm graphic again:
https://www.toptal.com/developers/sorting-algorithms

That's all for today folks!

This week's tasks

- Discussion post and comments
- Task #1 for the In-class exercises
- Quiz to be available this Friday
- Project assignment

- Try all exercises seen in class and consult the reference sources, as the more you practice, the easier it gets

Next week

- Linear and Integer Programming

- Don't let work pile up!
- Don't be shy about your questions

MERRIMACK COLLEGE

Have a Great Week!