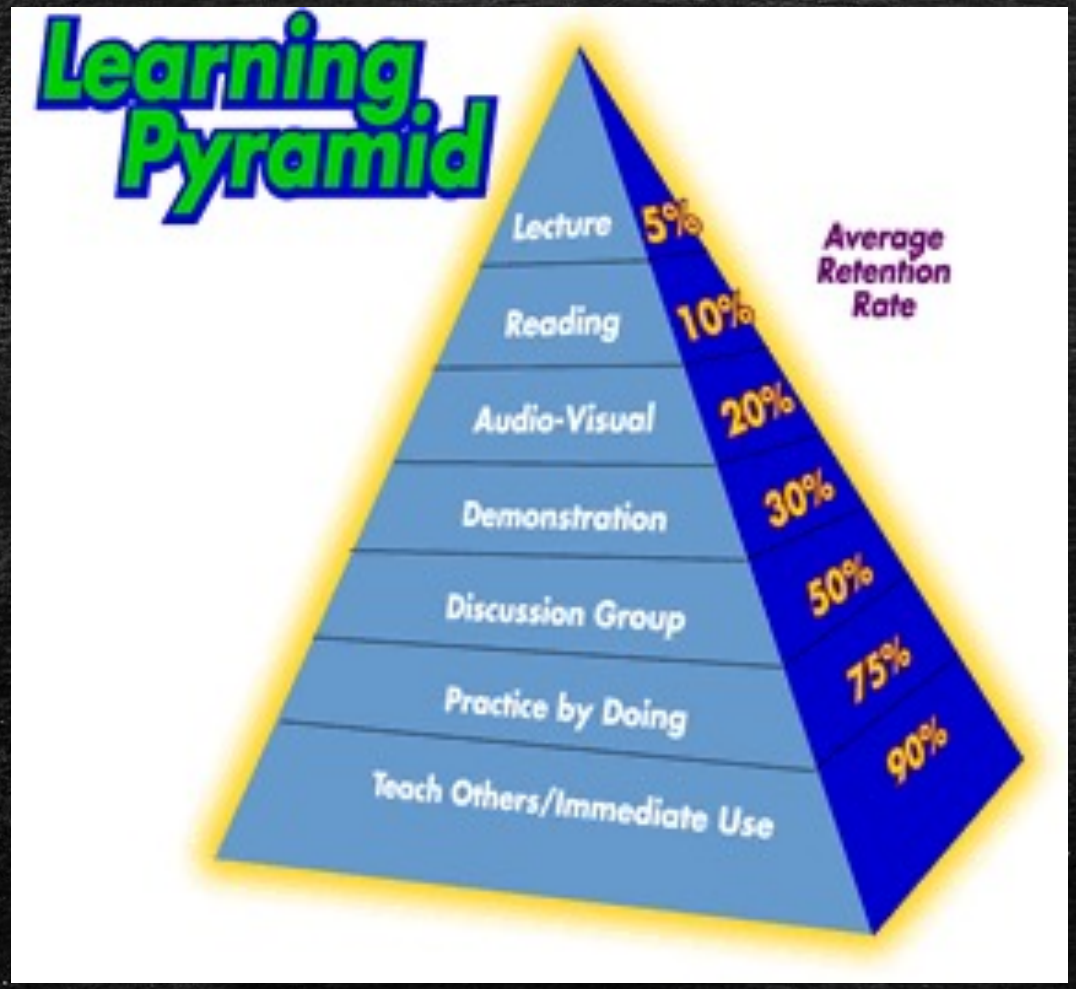
Data Structures and Algorithms

Lecture 6

Learning and Forgetting



Chinese Proverb

- Tell me & I will forget
- Show me & I may not Remember
- Involve me & I will understand

Introduction

- Data Structures + Algorithms = Programs – by Niklaus Wirth
- Computer science is primarily about organizing data and designing algorithms.
- Data structure
  - a data organization, management and storage format that enables efficient access and modification – Wiki
    1. how to store objects in memory,
    2. what data operations to support,
    3. the algorithms for those operations.

Data Structures Theme

- a set of n objects.
  - customer records, names of people, DNA sequences, music files, or images.
- organize the objects in a "searchable" structure for quick access.
- modify these structures as new objects are added to the set, or old ones deleted.
- Efficient data structures – fast search, and not taking too much memory.

Data Structure Examples

- The Google “search” function is essentially a data structure problem: objects are the documents (web pages) organized by their keywords.
- The operating system of a computer maintains a data structure that lets it quickly find the memory location of any variable in your program.
- Firewall systems in routers need to decide for each incoming data packet whether to block it or to allow it, etc.
- Every algorithm depends on thoughtful use of appropriate data structures for its efficient design e.g. Dijkstra's shortest paths, Prim's minimum spanning tree, Huffman coding.

Algorithms

- Algorithm: method for solving a problem. A specific set of instructions used to solve some problem. The core of computer science.
- A good algorithm solves the problem quickly, efficiently
- Examples
  - Public key cryptography allows information to be conveyed in secret, makes online commerce possible.
  - Data compression algorithms allow video, music etc to be transmitted and stored.
  - Tracking and estimation algorithms allow planes to fly, satellites to be launched.
  - Internet: Google, Yahoo
  - Communication: mobile phones, email
  - Biology: gene sequencing

Algorithms

- Two main issues related to algorithms
  - How to design algorithms
  - How to analyze algorithm efficiency

- Algorithm design techniques/strategies
  - Brute force
  - Divide and conquer
  - Decrease and conquer
  - Transform and conquer
  - Space and time tradeoffs
  - Greedy approach
  - Dynamic programming
  - Iterative improvement
  - Backtracking
  - Branch and bound

Algorithms

- Analysis of algorithms
  - How good is the algorithm?
    - time efficiency
    - space efficiency
  - Does there exist a better algorithm?
    - lower bounds
    - optimality
- Problem types
  - sorting
  - searching
  - string processing
  - graph problems
  - combinatorial problems
  - geometric problems
  - numerical problems

Intuition

- Are the following operations "fast" or "slow"?

array

behavior	fast/slow
add at front	slow
add at back	fast
get at index	fast
resizing	slow
binary search	(pretty) fast

linked list

behavior	fast/slow
add at front	fast
add at back	slow
get at index	slow
resizing	fast
binary search	(really) slow

stack

behavior	fast/slow
push	fast
pop	fast

queue

behavior	fast/slow
enqueue	fast
dequeue	fast

Complexity

- "Complexity" is a word that has a special meaning in computer science
- **complexity**: the amount of computational resources a block of code requires in order to run
- main computational resources:
  - **time**: how long the code takes to execute
  - **space**: how much computer memory the code consumes
- Often, one of these resources can be traded for the other:
  - e.g.: we can make some code use less memory if we don't mind that it will need more time to finish (and vice-versa)

Time Complexity

- We usually care more about time complexity
  - we want to make our code run fast!
- But we don't merely measure how long a piece of code takes to determine it's time complexity
- That approach would have results strongly skewed by:
  - size/kind of input
  - speed of the computer's hardware
  - other programs running at the same time
  - operating system
  - etc

Time Complexity

- Instead, we care about the growth rate as the input size increase
- First, we have to be able to measure the input size
  - the number of names to sort
  - the number of nodes in a linked list
  - the number of students in a queue
- We usually call the input size "n"
- What happens if we double the input size (n → 2n)?

Time Complexity

- We can learn about this growth rate in two ways:
  - by examining code
  - by running the same code over different input sizes
- Measuring the growth rate is one of the few areas where computer science is like the other sciences
  - here, we actually collect data
- But this data can be misleading
  - modern computers are very complex
  - some features, like cache memory, interfere with our data

Time Complexity: Rule of Thumb

- **rule of thumb**: this often works (but sometimes doesn't)
- rule of thumb for determining time complexity:
  - find the statement executed most often in the code
  - count how often it's executed
- But be careful how to count!
  - counting is hard
- We'll count most "simple" statements as `1`
  - this includes `i = i + 1`, `x = elementData[i]`, etc
  - but not loops! (or methods that contain loops!)

Examples

1

```
x = 4 * 10 / 3 + 2 - 10 * 42;
```

n

```
x = 0;
for i in range(0, 100):
    x +=i
```

n^2

```
size = 12
for i in range(1, size+1):
    for j in range(1, size+1):
        print(f"{str(i*j):>4}", end="")
    print()
```

$n^2 + n + 1$

Optimizing Code

- Many programmers care a lot about efficiency
- But many inexperienced programmers *obsess* about it
  - and the wrong kind of efficiency, at that
- Which one is faster:

```
print("print")

print("me")
```

or:

```
print("print\nme")
```

Who cares? They're both about the same

- If you're going to optimize some code, improve it so that you get a real benefit!

Don Knuth says...

Premature optimization is the root of all evil!

Don Knuth

- Professor Emeritus at Stanford
- "Father" of algorithm analysis

Growth Rates

- We care about n as it gets bigger
  - it's a lot like calculus, with n approaching infinity
- So, when we see something complicated like this:

$$\frac{n^3 - 18n^2 + 385n + 708}{0.005n^4 - 13n^2 + 7384}$$

- We can remove all the annoying terms:

$$\frac{n^3}{n^4}$$

- And as n gets really big, this approaches o

Big O Notation

- We need a way to write a growth rate of a block of code
- Computer scientists use big O ("big oh") notation
  - O(n)
  - $O(n^2)$
- In big O notation, we ignore coefficients that are constants
  - 5n is written as O(n)
  - 100n is also written as O(n)
  - $0.05n^2$ is written as $O(n^2)$ and will eventually outgrow O(n)
- Each O(*[something]*) specifies a different complexity class

Complexity Classes

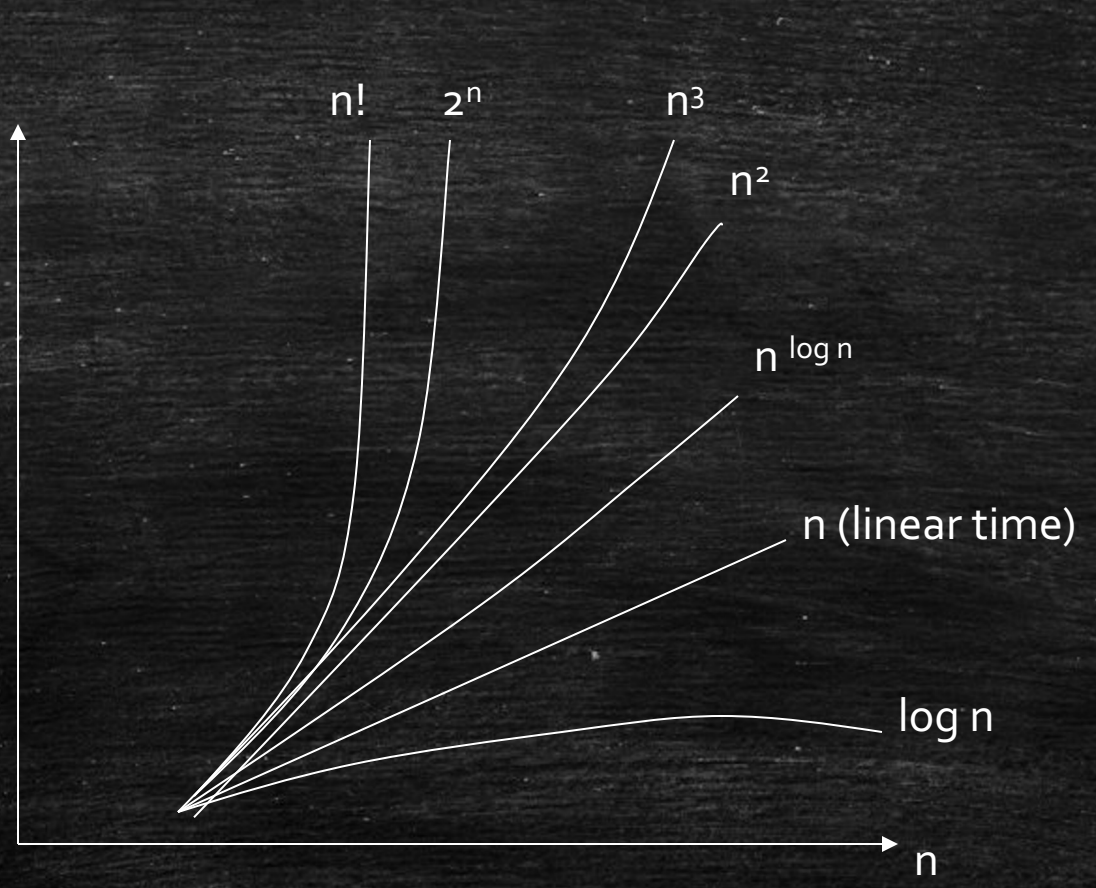
Complexity Class	Name	Example
O(1)	constant time	popping a stack
O(log n)	logarithmic time	binary search on an array
O(n)	linear time	scanning all elements of an array
O(n log n)	log-linear time	binary search on a linked list and good sorting algorithms
$O(n^2)$	quadratic time	poor sorting algorithms (like inserting n items into SortedIntList)
$O(n^3)$	cubic time	Example at the of the slides
$O(2^n)$	exponential time	Really hard problems. These grow so fast that they're impractical

Examples of Each Complexity Class's Growth Rate

- From Reges/Stepp book, page 708:
  - assume that all complexity classes can process an input of size 100 in 100ms

Input Size (n)	O(1)	O(log n)	O(n)	O(n log n)	$O(n^2)$	$O(n^3)$	$O(2^n)$
100	100ms	100ms	100ms	100ms	100ms	100ms	100ms
200	100ms	115ms	200ms	240ms	400ms	800ms	32.7 sec
400	100ms	130ms	400ms	550ms	1.6 sec	6.4 sec	12.4 days
800	100ms	145ms	800ms	1.2 sec	6.4 sec	51.2 sec	36.5 million years
1600	100ms	160ms	1.6 sec	2.7 sec	25.6 sec	6 min 49.6 sec	$4.21 * 10^{24}$ years
3200	100ms	175ms	3.2 sec	6 sec	1 min 42.4 sec	54 min 36 sec	$5.6 * 10^{61}$ years

Complexity classes



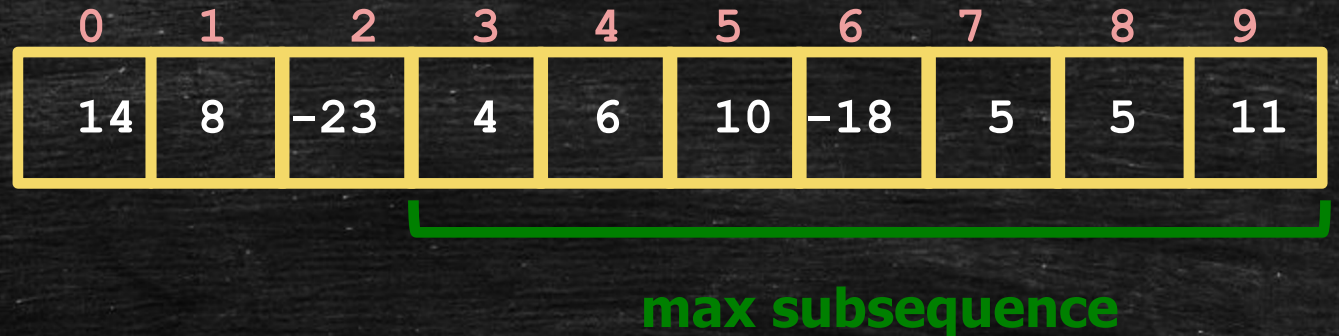
FiGrowth rates of some important complexity classes

Case Study: `maxSum.py`

- Given a list of **`int`**s, find the subsequence with the maximum sum
- Additional information:
  - values in the array can be negative, positive, or zero
  - the subsequence must be contiguous (can't skip elements)
  - you must compute:
    - the value of the sum of this subsequence
    - the starting index (inclusive) of this subsequence
    - the stopping index (inclusive) of this subsequence
- This has been used as a Microsoft interview question!

Case Study: maxSum.py

- For example: suppose you were given the following list:



max sum: 4 + 6 + 10 + -18 + 5 + 5 + 11 = 23

starting index: 3

stopping index: 9

- Notice that we included a negative number (-18)!
  - but this also let us include the 4, 6, and 10

Case Study: maxSum.py

- First, a simple way to solve this: try every subsequence!
- Psuedo-code:

```
// try every start index, from 0 to size - 1
    // try every stop index, from start index to size - 1
        // compute the sum from start index to stop index
```

- Convert to part pseudo-code:

```
for i in range(0, len(lst)):
    for j in range(i, len(lst)):
        // compute the sum from start index to stop index
```

Case Study: maxSum.py

- Now, we just need to convert this pseudo-code:

```
// compute the sum from start index to stop index
```

- …into code. Here's one way:

```
sumz = 0
for i in range(0, len(lst)):
    sumz += lst[i]
```

- And we need to store this sum if it becomes our max sum:

```
if sumz > maxSum:
    maxSum = sumz
```

Case Study: maxSum.py

- Here's our whole algorithm, with some initialization:

```
lst = [14, 8, -23, 4, 6, 10, -18, 5, 5, 11]
maxSum = lst[0]
for i in range(0, len(lst)):
    for j in range(i, len(lst)):
        sumz = 0
        for s in range(j, len(lst)):
            sumz += lst[s]
        if sumz > maxSum:
            maxSum = sumz
print(maxSum)
```

this is the most frequently executed line of code

Case Study: `maxSum.py`

- What complexity class is the previous algorithm?
  - $O(n^3)$ (cubic time)
- This is pretty slow
  - we recalculate the entire sum every time:
    - calculate the entire sum from index 0 to index 0
    - calculate the entire sum from index 0 to index 1
    - …
    - calculate the entire sum from index 0 to index 998
    - calculate the entire sum from index 0 to index 999
- How can we improve it?
  - remember the old sum (values list[start] to list[stop-1])
  - add the single new value (list[stop]) to the old sum

Case Study: maxSum.py

- Improved code, now with a running sum:

```
lst = [14, 8, -23, 4, 6, 10, -18, 5, 5, 11]
maxSum = lst[0]
for i in range(0, len(lst)):
        sumz = 0
        for j in range(i, len(lst)):
                sumz += lst[j]
                if sumz > maxSum:
                        maxSum = sumz
print(maxSum)
```

these are the most frequently executed lines of code

Case Study: `maxSum.py`

- What complexity class is the previous algorithm?
  - $O(n^2)$ (quadratic time)
- This is a *big* improvement over the old code
  - it now runs much faster for large input sizes
- And it wasn't that hard to convert our first version to this improved version
- But we can still do better
  - if only we can figure out how…

Case Study: `maxSum.py`

- There is a better algorithm, but it's harder to understand
  - and I'm not going to formally prove that it always works
- The main idea is that we will find the max subsequence *without* computing all the sums
  - this will eliminate our inner for loop
  - …which means we can find the subsequence with just a single loop over the list
- We need to know when to reset our running sum
  - this will "throw out" all previous values
  - but we have to know for sure that we don't want them!

Case Study: `maxSum.py`

- Suppose we're about to look at an index greater than 0
  - for example index 10
- If we're going to include previous values, we must include the value at the index 9
  - index 9 is immediately before index 10
- We want to use only the best subsequence that ends at 9
- And only if it helps us. When does it help?
  - it helps when the sum of this old subsequence is positive
  - and hurts when the sum of this old subsequence is negative

Case Study: maxSum.py

- Best code:

```
lst = [14, 8, -23, 4, 6, 10, -18, 5, 5, 11]
maxSum = lst[0]
sumz = 0
for i in range(0, len(lst)):
        sumz += lst[i]
        if sumz < 0:
                sumz = 0
        elif (maxSum < sumz):
                maxSum = sumz

print("Maximum Sub Array Sum =", maxSum)
```

these are the most frequently executed lines of code

Case Study: `maxSum.py`

- What complexity class is our best algorithm?
  - O(n) (linear time)
- This is again a big improvement over both other versions
- But let's not just take my word for it
- Let's conduct an experiment on pythontutor.com
  - we'll give an array of **`int`**s of some size to each algorithm
  - …and then give the algorithm an array of *twice* that size
  - …and then give the algorithm an array of *triple* that size
  - …and see how long it takes

maxSum.py

- Output for a list of 10 **`int`**s:
  - Algorithm 1 takes up to 851 steps - in the $O(n^3)$ algorithm
    ~~ time: 8.2969665527343 75e-05 seconds
  - Algorithm 2 takes up to 201 steps - in the $O(n^2)$ algorithm
    ~~ time: 4.410743713378906e-05 seconds
  - Algorithm 3 takes up to 47 steps - in the O(n) algorithm
    ~~ time: 2.3603439331054688e-05 seconds
- What happens as n becomes larger and larger?