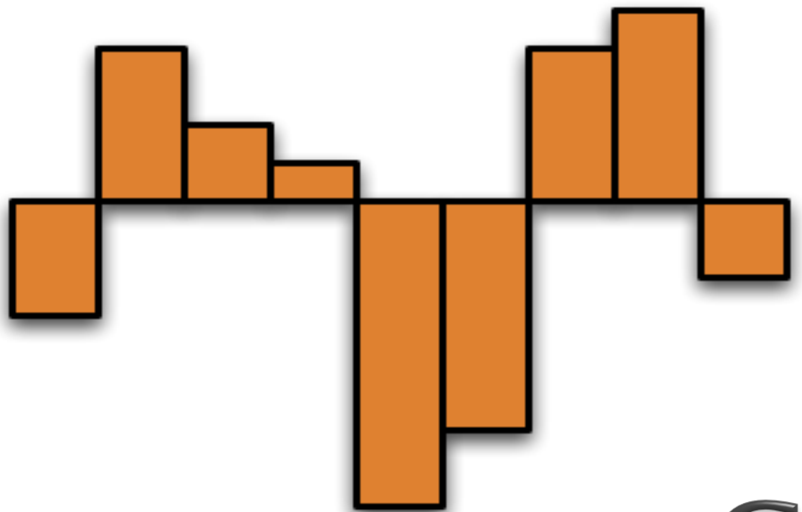


CSSE 230 Day 3

Maximum Contiguous Subsequence Sum

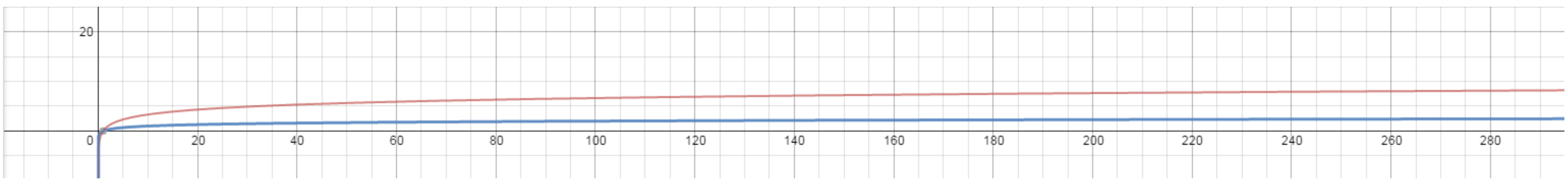
After today’s class you will be able to:

- state and solve the MCSS problem on small arrays by observation
- find the exact runtimes of the naive MCSS algorithms

https://openclipart.org/image/2400px/svg_to_png/169467/bow_tie.png

Homework 1

- Is it true that $\log_a(n)$ is $\theta(\log_b(n))$?
- Complete homework 1 to find out the exciting conclusion!
- Here is the graph for a=2 and b=10:



- Is it true that 3^n is $\theta(2^n)$?

- Also: Reading due next class and short HW 2 posted (skim now).

Limits and Asymptotics

- Consider the limit

$$\lim_{n \to \infty} \frac{f(n)}{g(n)}$$

- What does it say about asymptotic relationship between f and g if this limit is…
  - 0?
  - finite and non-zero?
  - infinite?

Q12, day 2

Apply this limit property to the following pairs of functions

1. n and n^2
2. log n and n (on these questions and solutions ONLY, let log n mean natural log)
3. n log n and n^2
4. $\log_a n$ and $\log_b n$ $(a < b)$
5. n^a and a^n $(a > 1)$
6. a^n and b^n $(a < b)$

Recall l'Hôpital's rule:
under appropriate conditions,

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)}$$

Q13–15

Answers

1. n is $O(n^2)$; n^2 is $\Omega(n)$
2. $\log n$ is $O(n)$; n is $\Omega(\log n)$
3. $n \log n$ is $O(n^2)$; n^2 is $\Omega(n \log n)$. (Use l'Hopital's rule)

4. $\log_a n$ is $\Theta(\log_b n)$ (so $\log_b n$ is $\Theta(\log_a n)$)
   Hint: Rewrite $\log_a n$ as $\log n/\log a$ and $\log_b n$ as $\log n/\log b$. Simplifying, we see that the original limit is a constant: no differentiating is needed here either.

5. n^a is $O(a^n)$ and is $o(a^n)$; a^n is $\Omega(n^a)$ and $\omega(n^a)$
   Hint: use l'Hopital's rule repeatedly until numerator goes to 0.

6. a^n is $O(b^n)$ and is $o(b^n)$; b^n is $\Omega(a^n)$ and $\omega(a^n)$
   Hint: rewrite as $(a/b)^n$. Because $a < b$, $a/b < 1$, and when $x < 1$, x^n approaches 0 as n goes to infinity.

Thoughts on Teaming

Next week's programming assignment is with a partner

Two Key Rules

- No prima donnas
  - Working way ahead, finishing on your own, or changing the team's work without discussion:
    - harms the education of your teammates
- No laggards
  - Coasting by on your team's work:
    - harms your education
- Both extremes
    - are selfish
    - may result in a failing grade for you on the project

Grading of Team Projects

- We'll assign an overall grade to the project
- Grades of individuals will be adjusted up or down based on team members' assessments

- At the end of the project each of you will:
  - Rate each member of the team, including yourself
  - Write a short Performance Evaluation of each team member with evidence that backs up the rating
    - Positives
    - Key negatives

Ratings

Excellent—Consistently did what he/she was supposed to do, very well prepared and cooperative, actively helped teammate to carry fair share of the load

Very good—Consistently did what he/she was supposed to do, very well prepared and cooperative

Satisfactory—Usually did what he/she was supposed to do, acceptably prepared and cooperative

Ordinary—Often did what he/she was supposed to do, minimally prepared and cooperative

Marginal—Sometimes failed to show up or complete tasks, rarely prepared

Deficient—Often failed to show up or complete tasks, rarely prepared

Unsatisfactory—Consistently failed to show up or complete tasks, unprepared

Superficial—Practically no participation

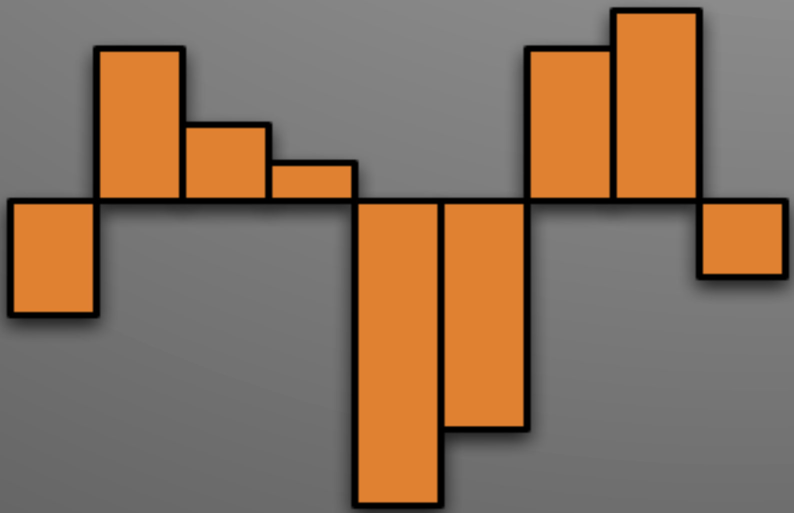
No show—No participation at all

Q3

Maximum Contiguous Subsequence Sum

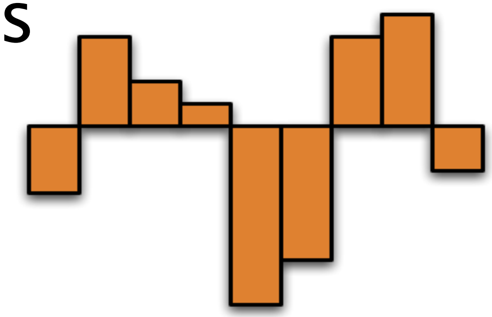
A deceptively deep problem with a surprising solution.

{−3, 4, 2, 1, −8, −6, 4, 5, −2}



A Nice Algorithm Analysis Example

- **Problem**: Given a sequence of numbers, find the maximum sum of a contiguous subsequence.



- Why study?
- Positives and negatives make it interesting. Consider:
  - What if all the numbers were positive?
  - What if they all were negative?
  - What if we left out “contiguous”?
- Analysis of obvious solution is neat
- We can make it more efficient later.

Q4-6

Formal Definition: Maximum Contiguous Subsequence Sum

Problem definition: Given a non-empty sequence of n (possibly negative) integers $A_1, A_2, \ldots, A_n$, find the maximum consecutive subsequence $S_{i,j} = \sum_{k=i}^{j} A_k$, and the corresponding values of i and j.

1-based indexing. But we'll use 0-based indexing

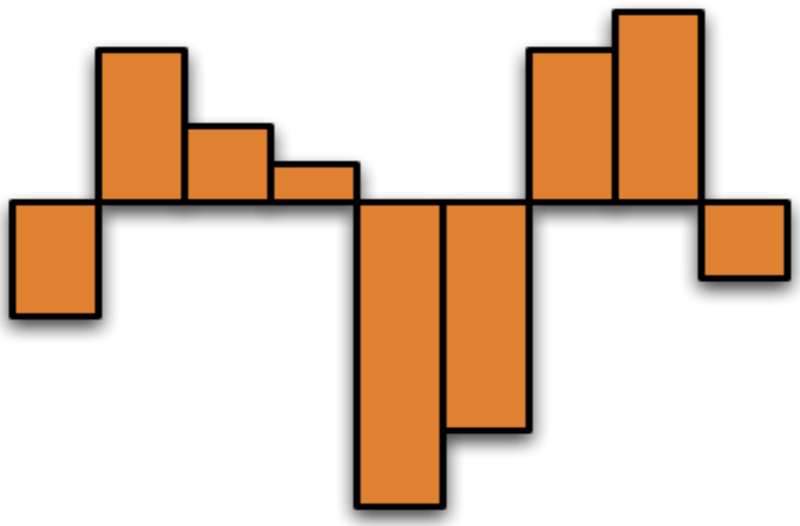
- Quiz questions:
  - In {-2, **11**, **-4**, **13**, -5, 2}, $S_{1,3}$ = ?
  - In {1, -3, 4, -2, -1, 6}, what is MCSS?
  - If every element is negative, what's the MCSS?

Write a simple correct algorithm now

Q7

- Must be easy to explain
- Correctness is KING. Efficiency doesn't matter yet.
- 3 minutes

- Examples to consider:
  - {–3, 4, 2, 1, –8, –6, 4, 5, –2}
  - {5, 6, –3, 2, 8, 4, –12, 7, 2}



First Algorithm

Find the sums of *all* subsequences

```
public final class MaxSubTest {
    private static int seqStart = 0;
    private static int seqEnd = 0;

    /* First maximum contiguous subsequence sum algorithm.
     * seqStart and seqEnd represent the actual best sequence.
     */
    public static int maxSubSum1( int [ ] a ) {
        int maxSum = 0;
            //In the analysis we use "n" as a shorthand for "a.length"
        for( int i = 0; i < a.length; i++ )
            for( int j = i; j < a.length; j++ ) {
                int thisSum = 0;

                for( int k = i; k <= j; k++ )
                    thisSum += a[ k ];

                if( thisSum > maxSum ) {
                    maxSum   = thisSum;
                    seqStart = i;
                    seqEnd   = j;
                }
            }
        return maxSum;
    }
```

i: beginning of subsequence

j: end of subsequence

k: steps through each element of subsequence

Where will this algorithm spend the most time?

How many times (exactly, as a function of N = a.length) will that statement execute?

Analysis of this Algorithm

- What statement is executed the most often?
- How many times?

```
//In the analysis we use "n" as a shorthand for "a.length "
for( int i = 0; i < a.length; i++ )
    for( int j = i; j < a.length; j++ ) {
        int thisSum = 0;

        for( int k = i; k <= j; k++ )
            thisSum += a[ k ];
```

Q8-9

Solution

```
//In the analysis we use "n" as a shorthand for "a.length "
for( int i = 0; i < a.length; i++ )
    for( int j = i; j < a.length; j++ ) {
        int thisSum = 0;

        for( int k = i; k <= j; k++ )
            thisSum += a[ k ];
```

Interlude

- Computer Science is no more about computers than astronomy is about _______.

Donald Knuth

Interlude

- Computer Science is no more about computers than astronomy is about telescopes.

Donald Knuth

Where do we stand?

- We showed MCSS is $O(n^3)$.
  - Showing that a **problem** is O(g(n)) is relatively easy – just analyze a known algorithm.
- Is MCSS $\Omega(n^3)$?
  - Showing that a **problem** is Ω (g(n)) is much tougher. How do you prove that it is impossible to solve a problem more quickly than you already can?
  - Or maybe we can find a faster algorithm?

f(n) is O(g(n)) if $f(n) \leq cg(n)$ for all $n \geq n_0$
- So O gives an upper bound

f(n) is Ω(g(n)) if $f(n) \geq cg(n)$ for all $n \geq n_0$
- So Ω gives a lower bound

f(n) is θ(g(n)) if $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$
- So θ gives a tight bound
- f(n) is θ(g(n)) if it is both O(g(n)) **and** Ω(g(n))

What is the main source of the simple algorithm's inefficiency?

```
        //In the analysis we use "n" as a shorthand for "a.length "
for( int i = 0; i < a.length; i++ )
    for( int j = i; j < a.length; j++ ) {
        int thisSum = 0;

        for( int k = i; k <= j; k++ )
            thisSum += a[ k ];
```

- The performance is bad!

Eliminate the most obvious inefficiency…

```
for( int i = 0; i < a.length; i++ ) {
    int thisSum = 0;
    for( int j = i; j < a.length; j++ ) {
        thisSum += a[ j ];

        if( thisSum > maxSum ) {
            maxSum = thisSum;
            seqStart = i;
            seqEnd   = j;
        }
    }
}
```

This is Θ(?)

MCSS is $O(n^2)$

- Is MCSS $\Omega(n^2)$?
  - Showing that a problem is Ω (g(n)) is much tougher. How do you prove that it is impossible to solve a problem more quickly than you already can?
  - Can we find a yet faster algorithm?

f(n) is O(g(n)) if $f(n) \leq cg(n)$ for all $n \geq n_0$
- So O gives an upper bound

f(n) is Ω(g(n)) if $f(n) \geq cg(n)$ for all $n \geq n_0$
- So Ω gives a lower bound

f(n) is θ(g(n)) if $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$
- So θ gives a tight bound
- f(n) is θ(g(n)) if it is both O(g(n)) **and** Ω(g(n))

Can we do even better?

Tune in next time for the exciting conclusion!