Who am I?

Brad Solomon
Teaching Assistant Professor, Computer Science
2233 Siebel Center for Computer Science
Email: bradsol@illinois.edu

Office Hours:
Thursdays, 11:00 AM - 12:00 PM
... or by appointment

https://courses.engr.illinois.edu/cs225/sp2023/info/office-hours/
Who are you?

Take a moment to introduce yourself!

(Your name, a hobby you enjoy, and one thing you hope to get out of this class)

https://piazza.com/illinois/spring2023/cs199225/home
What is this class about?

String Algorithms and Data Structures

Exact string matching

Compressed self-indexes

Inexact pattern matching

Query: 161 atatcaccacgtcaaaggtgactccaactcca---ccactccattttgtt

Sbjct: 481 atatcaccacgtcaaaggtgactccaact-tattgatagtgtttatatgtt
What will you get out of this class?

Understand fundamental string algorithms

Experience applying data structures, algorithms, and algorithm design principles to real world problems

Justify implementation choices based on theoretical or practical considerations

Build a foundation for future data science projects
Course Webpage

https://courses.engr.illinois.edu/cs225/sp2023/pages/honors.html

All course information and links can be found here!

Mediaspace recordings

Piazza

Syllabus
Syllabus

Please read — many important topics:

Course Goals & Topics

Course Expectations

Grading

Commitments to Diversity, Equity, Inclusion

Commitments to Mental Health

Ethics and Academic Integrity Policies
Course Expectations

Weekly assignments (11 total):

Small assignments (~ 2-3 hours / week)

Must pass at least 10 of them (80% is passing)

Must submit your own work

One week extensions for 80% credit
Course Expectations

Class participation:

No attendance grades

Ask questions (synchronously or asynchronously)

Participate in breakout rooms and polls
Mental Health

This class should be low-stress, light work-load.

UIUC offers a variety of confidential services:

**Counseling Center:** 217-333-3704
610 East John Street Champaign, IL 61820

**McKinley Health Center:** 217-333-2700
1109 South Lincoln Avenue, Urbana, Illinois 61801
Diversity, Equity, and Inclusion

“If you witness or experience racism, discrimination, micro-aggressions, or other offensive behavior, you are encouraged to bring this to the attention of…”

Staff (CAs and TAs for CS 225)

Faculty (Myself or Carl)

Campus Belonging Office (Link)

The Office of Student Conflict Resolution (Link)

CS CARES (Link)
Learning Objectives

Review fundamentals of strings

Introduce exact pattern matching problem
What is a string?

A string $S$ is a finite sequence of characters.

Characters are drawn from alphabet $\Sigma$, usually assumed finite.

- Nucleic acid alphabet: $\{ A, C, G, T \}$

What are some other alphabets we could use?
What is a string… in C++?

**char**: 1-byte (8-bit) character encoding [ASCII 256]

**std::string**: uses char alphabet (by default), has significant operation support

```cpp
#include <string>
#include <iostream>

int main() {
    char c[] = "Hello World";
    std::string str = "Hello World";
    return 0;
}
```

<table>
<thead>
<tr>
<th>Math</th>
<th>Strings</th>
</tr>
</thead>
</table>

Fundamental operations
Fundamental string operations

“How efficient is my algorithm at searching for a given pattern $P$?”

“How much memory do I need to allocate for this text file?”
Fundamental string operations

**Size** of $S$, $|S|$: The number of characters in $S$.

$S = \text{“How big?”}$

$|S| = \, ?$
**Fundamental string operations**

**Size** of $S$, $|S|$: The number of characters in $S$.

$S = \text{“How big?”}$

$|S| = 8$
Fundamental string operations

**Size** of $S$, $|S|$: The length of $S$ (in terms of bytes).

$S\. \text{length}()$
Fundamental string operations

“Is this book about data structures?”

“Is this student enrolled at UIUC?”
Fundamental string operations

S equals T if each character, in order, is the same

S == T

```cpp
#include <string>
#include <iostream>

int main() {
    std::string S = "Thing 1";
    std::string T = "Thing 1";

    if (S == T) {
        std::cout << "S == T" << std::endl;
    } else {
        std::cout << "S != T" << std::endl;
    }

    return 0;
}
```
Fundamental string operations

**S equals T** if each character, in order, is the same

\[
S = T
\]

cchar Equals.cpp

```cpp
#include <string>
#include <iostream>

int main() {
    char S[] = "Thing 1";
    char T[] = "Thing 1";

    if (S == T) {
        std::cout << "S == T" << std::endl;
    } else {
        std::cout << "S != T" << std::endl;
    }

    return 0;
}
```
Fundamental string operations

Reads

<table>
<thead>
<tr>
<th>GTATGCACGCGATAG</th>
<th>TATGTCGCAGTATCT</th>
<th>CACCCTATGTCGCAG</th>
<th>GAGACGCTGGAGCCG</th>
</tr>
</thead>
<tbody>
<tr>
<td>TAGCATTGCGAGACG</td>
<td>GGTATGCAACGCCTA</td>
<td>TGGAGCCGGAGACACC</td>
<td>CGCTGGAGCCGGAGC</td>
</tr>
<tr>
<td>TGTCTTTGATTTCTG</td>
<td>CGCGATAGCATTCGG</td>
<td>GCATTGCGAGACGCT</td>
<td>CCTATGTCGAGATAT</td>
</tr>
<tr>
<td>GACGCTGGAGCCCGGA</td>
<td>GCACCCTATGTCGCA</td>
<td>GTATCTGTCTTTTGT</td>
<td>CCTCATCCTATTATT</td>
</tr>
<tr>
<td>TATCAGCCTACGTTT</td>
<td>CAATATTCGATCATG</td>
<td>GATCACAGGTCTATC</td>
<td>ACCCTATTAACACACT</td>
</tr>
<tr>
<td>CACGGGAGCTCTCCA</td>
<td>TGCACTTTGGTAGTTT</td>
<td>CGTCTGGGGGGGATAG</td>
<td>CACGCGATAGCATTG</td>
</tr>
<tr>
<td>GTATGCACCCGATAG</td>
<td>ACCTACGCATTAATAT</td>
<td>TATTTATGCACCTA</td>
<td>CCACCTACGGGAGCT</td>
</tr>
<tr>
<td>GCGAGACGCTGGAGC</td>
<td>CATACCCCTATTTA</td>
<td>CTGTCTTTGGACTCT</td>
<td>ACTCACCGGAGCTCT</td>
</tr>
<tr>
<td>CCTACGTCTCAATATT</td>
<td>GCACCTACGTTCAAT</td>
<td>GTCTGGGGGGTATGC</td>
<td>AGCCGGAGCACCCTA</td>
</tr>
<tr>
<td>GACGCTGGAGCCCGGA</td>
<td>GCACCCTATGTCGCA</td>
<td>GTATCTGTCTTTTGT</td>
<td>CCTCATCCTATTATT</td>
</tr>
<tr>
<td>TATCAGCCTACGTTT</td>
<td>CAATATTCGATCATG</td>
<td>GATCACAGGTCTATC</td>
<td>ACCCTATTAACACACT</td>
</tr>
<tr>
<td>CACGGGAGCTCTCCA</td>
<td>TGCATTGGGTATTTT</td>
<td>CGTCTGGGGGGGATAG</td>
<td>CACGCGATAGCATTG</td>
</tr>
</tbody>
</table>

Genome

CGTCTGGGGGGGTATGCACGCGATAGCATTGCACGAGCCTGAGCTGAGCCGGAGCACCCTATGTCGCA GTATCTGTCTTTTGTATCTGCTTTTGATTCCTG
Fundamental string operations

*Concatenation* of $S$ and $T$: characters of $S$ followed by characters of $T$

$S = \text{“Beep”}$ \hspace{1cm} $T = \text{“Boop”}$

What is the string $ST$?

What is the string $T$\$S$?
Fundamental string operations

**Concatenation** of $S$ and $T$: characters of $S$ followed by characters of $T$

$$S + T$$

```cpp
#include <string>
#include <iostream>

int main() {
    std::string S = "Beep";
    std::string T = "Boop";

    std::cout << S + T << std::endl;
    std::cout << T + S << std::endl;
    std::cout << S + '$' + T << std::endl;
    std::cout << T + '$' + S << std::endl;
}
```
Google’s stated mission, “to organize the world’s information and make it universally accessible and useful,” could not better capture the immense ambition of modern society for gathering all kinds of data and putting them to use to improve our lives. We are collecting not only huge amounts of data from the physical world (astronomical, climatological, geographical, biological), but also human-generated data (voice, pictures, music, video, books, news, Web contents, emails, blogs, tweets) and society-based behavioral data (markets, shopping, traffic, clicks, Web navigation, likes, friendship networks). Our hunger for more and more information is flooding our lives with data. Technology is improving and our ability to store data is growing fast, but the data we are collecting also grow fast—in many cases faster than our storage capacities. While our ability to store the data in secondary or perhaps tertiary storage does not yet seem to be compromised, performing the desired processing of these data in the main memory of computers is becoming more and more difficult. Since accessing a datum in main memory is about 10^5 times faster than on disk, operating in main memory is crucial for carrying out many data-processing applications.

In many cases, the problem is not so much the size of the actual data, but that of the data structures that must be built on the data in order to efficiently carry out the desired processing or queries. In some cases the data structures are one or two orders of magnitude larger than the data! For example, the DNA of a human genome, of about 3.3 billion bases, requires slightly less than 800 megabytes if we use only 2 bits per base (A, C, G, T), which fits in the main memory of any desktop PC. However, the suffix tree, a powerful data structure used to efficiently perform sequence analysis on the genome, requires at least 10 bytes per base, that is, more than 30 gigabytes.

The main techniques to cope with the growing size of data over recent years can be classified into three families:
Fundamental string operations

S is a **substring** of T if there exists (possibly empty) strings u and v such that T = uSv

A **substring** is a sequence of characters (a string) contained within another string

*S*: pepper

*T*: I_like_pepperoni_pizza
A **substring** of $S$ is a string occurring inside $S$

$S\.substr$(size_t pos, size_t len)
Fundamental string operations

$S$ is a \textbf{prefix} of $T$ if there exists a string $v$ such that $T = Sv$

A \textbf{prefix} is a substring $T = uSv$ where $u = "\"$

\textbf{T:} \quad \text{GTTATAGCTGAT}

\text{GTTATAGCTGAT}

\begin{align*}
S & \quad \text{v}
\end{align*}
Fundamental string operations

$S$ is a prefix of $T$ if there exists a string $\nu$ such that $T = S\nu$

$T$: GTTATAGCTGAT

$T$: GTTATAGCTGAT
Fundamental string operations

$S$ is a **prefix** of $T$ if there exists a string $v$ such that $T = Sv$

**$T$: Pattern matching**

- **Pattern** ✓
- **matching** ✗
- **Patrick** ✗
Fundamental string operations

*S* is a **suffix** of *T* if there exists a string *u* such that *T* = *uS*

A **suffix** is a substring *T* = *uSv* where *v*="""
Fundamental string operations

S is a **suffix** of T if there exists a string u such that $T = uS$

$T$: GTTATAGCTGAT

$T$: GTTATAGCTGAT
Fundamental string operations

$S$ is a suffix of $T$ if there exists a string $u$ such that $T = uS$
Fundamental string operations

Size, $|S|$  
$S.length()$

Equals, $S == T$  
$S == T$

Concatenation, $ST$  
$S + T$

Substring, $uSv$  
$S.substr(pos, len)$
Exact Pattern Matching

Pattern, $P$  
Text, $T$

Find instances of $P$ in $T$

‘instances’: An exact, full length copy
Exact Pattern Matching

Find places where *pattern* $P$ occurs as a substring of *text* $T$. Each such place is an *occurrence* or *match*.

$P$: word

$T$: There would have been a time for such a word

Alignment 1: word

Alignment 2: word

Not a match! Match!

**Alignment**: a way of putting $P$’s characters opposite $T$’s. May or may not correspond to a match.
Exact Pattern Matching

What’s a simple algorithm for exact matching?

P: word

T: There would have been a time for such a word
word word word word word word word word
word word word word word word word word
word word word word word word word word
word word word word word word word word
word word word word word word word word

Try all possible alignments. For each, check if it matches. This is the \textit{naïve algorithm}.
Assignment 1: a_naive

Learning Objective:

- Conceptualize exact pattern matching w/ naïve search
- Demonstrate understanding of fundamental operations

Think about as you code: is naïve search a good solution?
End-of-class brainstorm

How can we improve the naïve algorithm?

… if you have infinite space?

… if I tell you the pattern ahead of time?

… if I tell you the text ahead of time?