Survey of Scientific Computing

(SciComp 301)

Dave Biersach
Brookhaven National Laboratory
dbiersach@bnl.gov

Session 14

Cryptanalysis, Anagrams

Session Goals

- Manage a C++ **string** as a **vector<char>**
  - Understand **ASCII** as an encoding mechanism
  - Read an ASCII **text file** stored on disk into a memory buffer
  - Generate a **histogram of character frequencies** within a file
- Encrypt and decrypt files using "**Caesar Shift**"
  - Perform **bigram analysis** on unreadable ciphertext to determine the author's native language
- Generate and discover simple and compound **anagrams**
  - Avoid combinatorial (exponential) explosion in search space

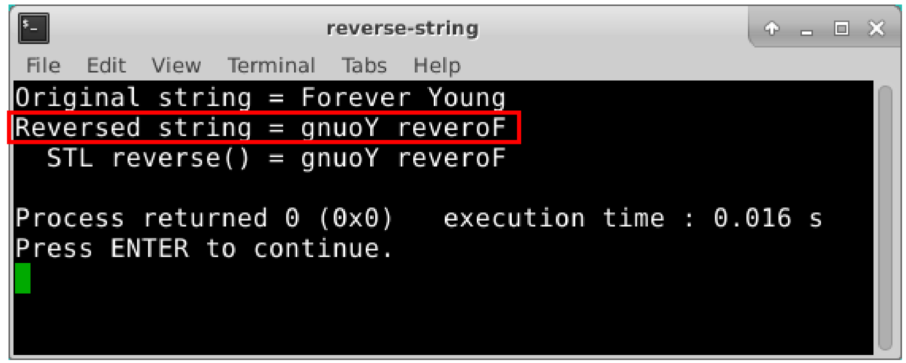
C++ strings

- A C++ string is mostly equivalent to a **vector<char>**
  - A C++ **char** data type holds one “character”
  - There may be a difference between the length of a string (the number of "characters" in the string) and the actual number of bytes required to store it in memory or on disk
  - The memory size (number of bits) of a character can vary by encoding scheme and platform (Windows vs. Linux)
- A string has **.size()** or **.length()** methods to get the number of *characters* in the string
- We can access individual characters using the **.at()** method

Reverse a String

i	s.at()
0	F
1	o
2	r
3	e
4	v
5	e
6	r
7	*(space)*
8	Y
9	o
10	u
11	n
12	g

s.length()==13



We could try to swap each letter around the middle element, but that would require complicated code

Open Lab 1 **Reverse String**

main.cpp

```cpp
#include "stdafx.h"

using namespace std;

string ReverseString(string a)
{
    string b;

    // Implement your algorithm here


    return b;
}

int main()
{
    string s = "Forever Young";
    string r = ReverseString(s);

    cout << "Original string = "
         << s << endl;

    cout << "Reversed string = "
         << r << endl;

    reverse(s.begin(),s.end());

    cout << "  STL reverse() = "
         << s << endl;

    return 0;
}
```

Work backwards from the *end* of string **a** appending each character to string **b**

Edit Lab 1 – Reverse String

- Only add code to the **ReverseString()** function
  - A string is an vector – you can access individual elements (characters) using the **.at()** method
  - Use **.length()** to get the number of characters in the string
  - You can append characters to the end of a string using the **+** operator
- On return, **b** should be the string **a** in reverse char order

```
string ReverseString(string a)
{
    string b;

    // Implement your algorithm here


    return b;
}
```

Edit Lab 1 – Reverse String

- Only add code to the **ReverseString()** function
  - A **string** is an vector – you can access individual elements (characters) using the **.at()** method
  - Use **.length()** to get the number of characters in the string
  - You can append characters to end of a string using the **+** operator
- On return, **b** should be the string **a** in reverse char order

```
string ReverseString(string a)
{
    string b;

    for(int i = a.length() - 1; i >= 0; --i)
        b += a.at(i);

    return b;
}
```

Check Lab 1 Reverse String

main.cpp

```
#include "stdafx.h"

using namespace std;

string ReverseString(string a)
{
    string b;

    for(int i = a.length() - 1; i >= 0; --i)
        b += a.at(i);

    return b;
}

int main()
{
    string s = "Forever Young";
    string r = ReverseString(s);

    cout << "Original string = "
         << s << endl;

    cout << "Reversed string = "
         << r << endl;

    reverse(s.begin(),s.end());

    cout << "  STL reverse() = "
         << s << endl;

    return 0;
}
```

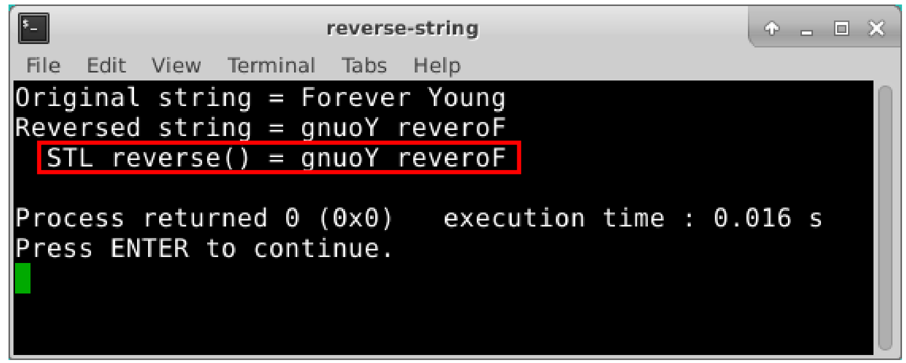
Walk backwards through string **a** while dynamically building string **b** one character at a time

The C++ Standard Template Library **(STL)** has a built-in function to reverse the order of the elements in **any** vector

Run Lab 1 – Reverse String

i	s.at()
0	F
1	o
2	r
3	e
4	v
5	e
6	r
7	*(space)*
8	Y
9	o
10	u
11	n
12	g

s.length()==13



STL = **S**tandard **T**emplate **L**ibrary

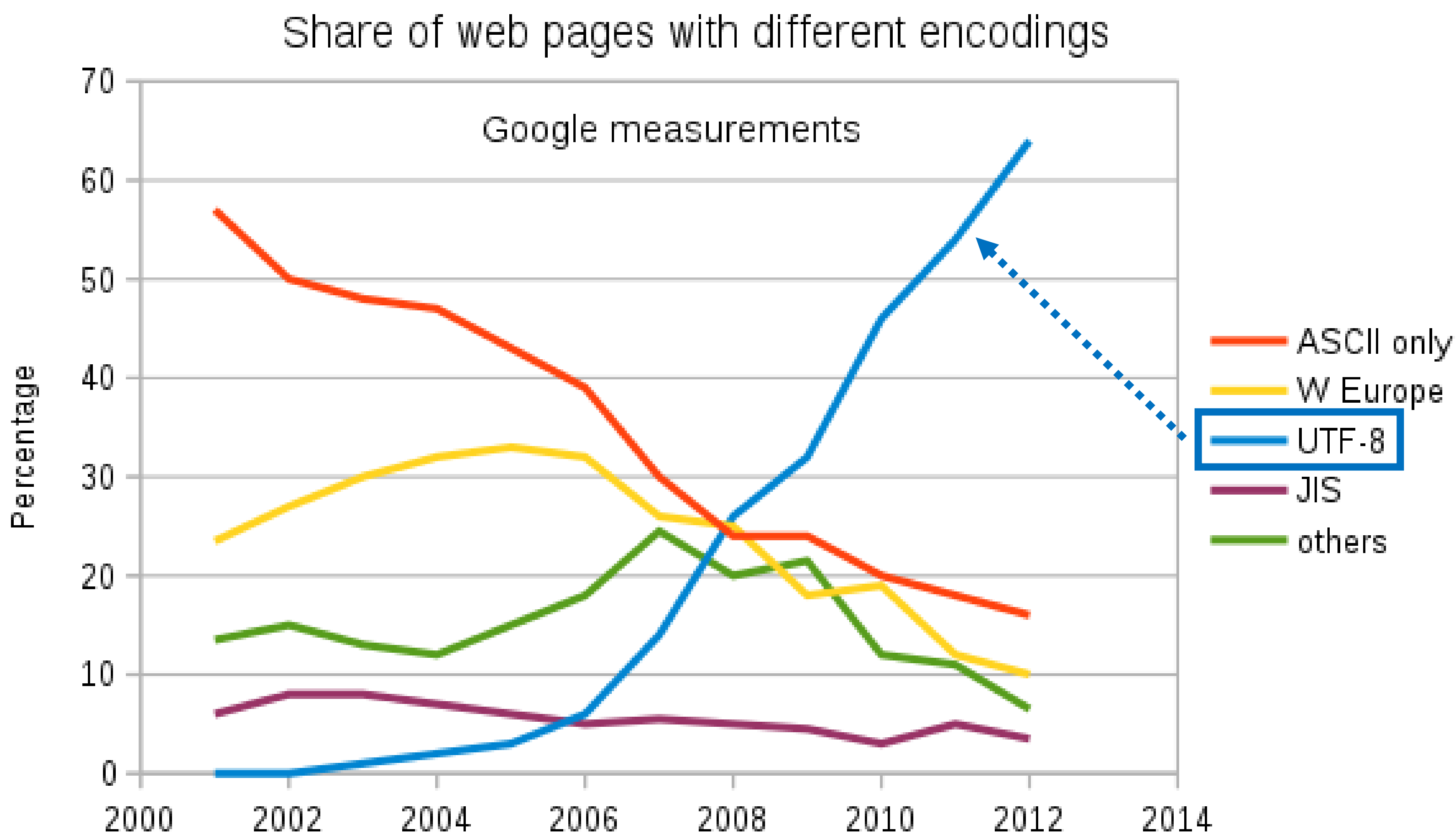
The **STL** is a set of free & open-source functions and classes to ***reduce*** the amount of code C++ programmers must write to solve <u>common problems</u>

ASCII ("as-key")

- **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange
  - ASCII was the most common legacy International standard used across the Internet until 2007
  - Since 2008 ASCII has been surpassed by **UTF-8** (Universal Transformation Format), which includes ASCII as a subset
- ASCII is an 8-bit (**one byte**) character encoding scheme
  - ASCII maps most of the characters in the (Western) languages descending from Latin to a specific integer value
  - In ASCII there is a 1:1 correspondence between a letter and a number and every character is **always** one byte long
  - Inside a computer, **all** letters, punctuation marks, even numbers (when treated as strings) are encoded as either ASCII or UTF

Learn about **UTF-8**

https://en.wikipedia.org/wiki/UTF-8



ASCII range for common English characters

Dec	Char
32	(space)
33	!
34	"
35	#
36	$
37	%
38	&
39	'
40	(
41	)
42	*
43	+
44	,
45	-
46	.
47	/
48	0
49	1
50	2

Dec	Char
51	3
52	4
53	5
54	6
55	7
56	8
57	9
58	:
59	;
60	<
61	=
62	>
63	?
64	@
65	A
66	B
67	C
68	D
69	E

Dec	Char
70	F
71	G
72	H
73	I
74	J
75	K
76	L
77	M
78	N
79	O
80	P
81	Q
82	R
83	S
84	T
85	U
86	V
87	W
88	X

Dec	Char
89	Y
90	Z
91	[
92	\
93	]
94	^
95	_
96	`
97	a
98	b
99	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k

Dec	Char
108	l
109	m
110	n
111	o
112	p
113	q
114	r
115	s
116	t
117	u
118	v
119	w
120	x
121	y
122	z
123	{
124	|
125	}
126	~

Creating a Frequency Histogram

- Consider Lincoln's Gettysburg Address:

  Four score and seven years ago our fathers brought forth on this continent a new nation, conceived in liberty, and dedicated to the proposition that all men are created equal.

  Now we are engaged in a great civil war, testing whether that nation, or any nation so conceived and so dedicated, can long endure…

- We want to perform a **histogram analysis** of Lincoln's speech
  - What **letter** do you think occurs most frequently in **English**?
  - Spaces (**ASCII value 32**) usually occur most often because we use spaces as a word breaker

Letter Frequencies in the English Language

Letter	Relative frequency in the English language
a	8.167%
b	1.492%
c	2.782%
d	4.253%
e	12.702%
f	2.228%
g	2.015%
h	6.094%
i	6.966%
j	0.153%
k	0.772%
l	4.025%
m	2.406%

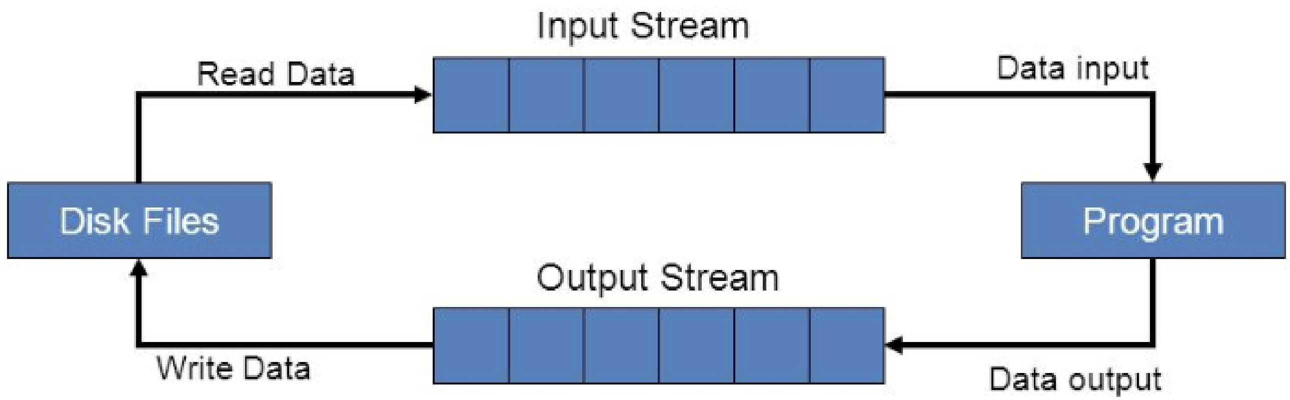
Letter	Relative frequency in the English language
n	6.749%
o	7.507%
p	1.929%
q	0.095%
r	5.987%
s	6.327%
t	9.056%
u	2.758%
v	0.978%
w	2.361%
x	0.150%
y	1.974%
z	0.074%

ETAOIN SHRDLU

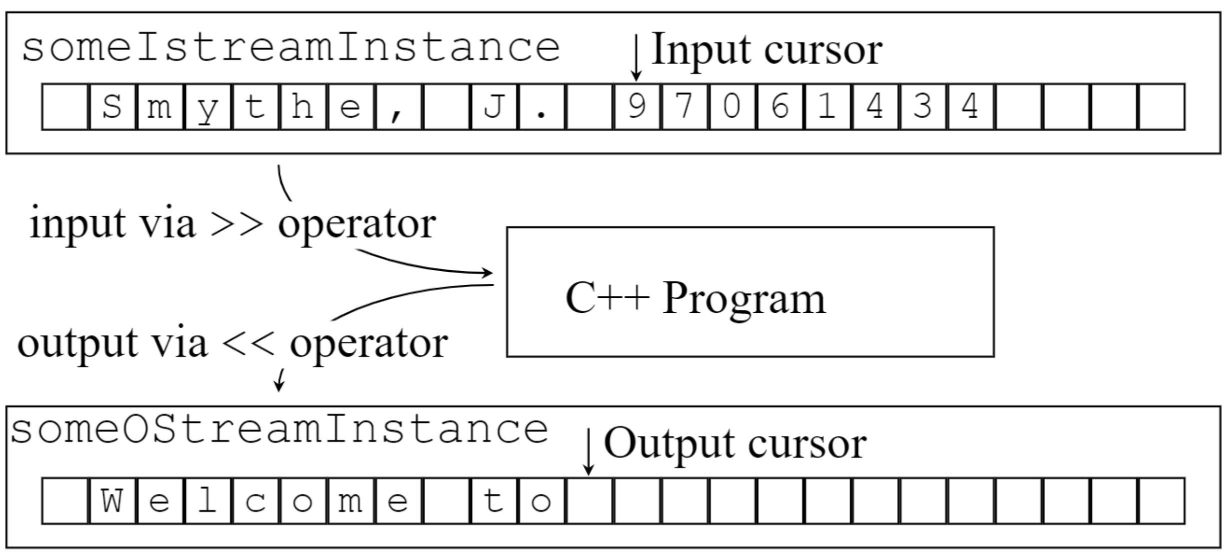
ASCII Text Files – A “stream”

- A file on disk is essentially just a `byte` **vector**
  - A `byte` is an 8-bit *unsigned integer* between 0 and 255 (`uint8_t`)
  - We can declare a byte vector and load it with the contents of a file
  - A **stream** of file bytes in memory is called a **buffer**
- We can then access any individual character within the file
  - Use the normal **.at() method** on this vector of **"fileBytes"**
  - We only need to specify an **index value** to get a specific byte

ASCII Text Files – A “stream”

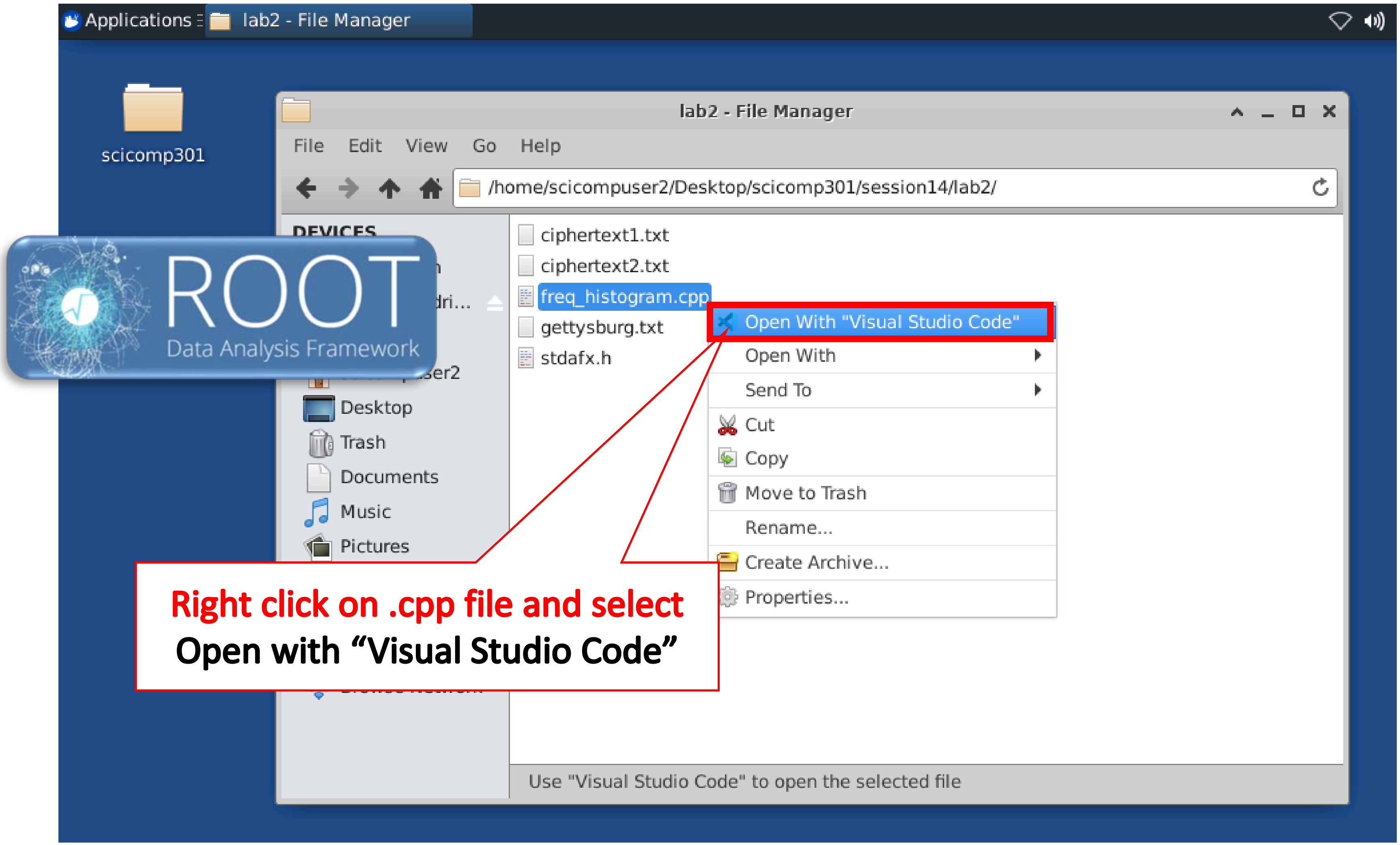


cin >> n;



cout << n;

Open Lab 2: freq_histogram.cpp



View Lab 2 Frequency Histogram

```
void freq_histogram(string filename)
{
    // Open file at the end (ios::ate) so the next "get"
    // position will return the actual file size
    ifstream ifs(filename, ios::binary | ios::ate);
    ifstream::pos_type pos = ifs.tellg();

    // Allocate a vector big enough to hold all the file bytes
    vector<unsigned char> fileBytes(pos);

    // Read in the file from the beginning straight into the vector
    ifs.seekg(0, ios::beg);
    ifs.read((char*)(fileBytes.data()), pos);

    // Create a ROOT chart
    string title = "Frequency Analysis";
    TCanvas* c1 = new TCanvas(title.c_str());
    c1->SetTitle(title.c_str());

    // Create a ROOT one dimensional histogram of integers
    TH1I* h1 = new TH1I(nullptr, title.c_str(), 256, 0, 257);
    h1->SetStats(kFALSE);

    TAxis* ya = h1->GetYaxis();
    ya->SetTitle("Count");
    ya->CenterTitle();

    TAxis* xa = h1->GetXaxis();
    xa->SetTitle("ASCII Value");
    xa->CenterTitle();
    xa->SetTickSize(0);
```

A file on disk is just a vector of integers (unsigned char = byte)

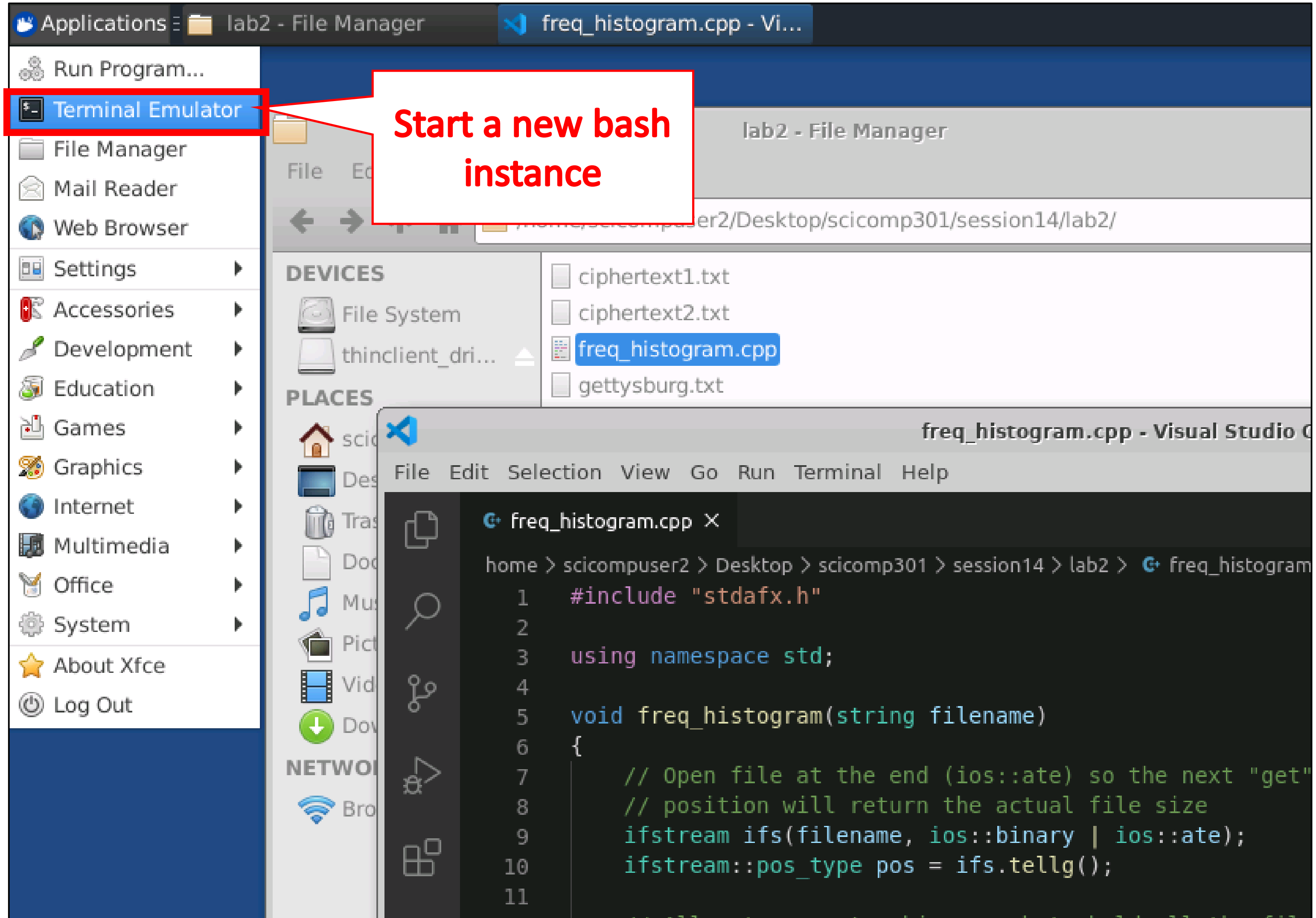
View Lab 2 Frequency Histogram

```cpp
    // Create a ROOT chart
    string title = "Frequency Analysis";
    TCanvas* c1 = new TCanvas(title.c_str());
    c1->SetTitle(title.c_str());

    // Create a ROOT one dimensional histogram of integers
    TH1I* h1 = new TH1I(nullptr, title.c_str(), 256, 0, 257);
    h1->SetStats(kFALSE);

    TAxis* ya = h1->GetYaxis();
    ya->SetTitle("Count");
    ya->CenterTitle();

    TAxis* xa = h1->GetXaxis();
    xa->SetTitle("ASCII Value");
    xa->CenterTitle();
    xa->SetTickSize(0);

    // Fill the histogram using the bytes in the file
    for (auto item : fileBytes)
        h1->Fill((int)item);

    // Label any bin with the ASCII value
    // if the bin count is > 6% of the file size,
    // as these would be noteworthy occurrences
    for (int i{1}; i < xa->GetNbins(); ++i)
        if (h1->GetBinContent(i) > fileBytes.size() * 0.06)
            xa->SetBinLabel(i, to_string(i).c_str());

    h1->Draw();
}
```

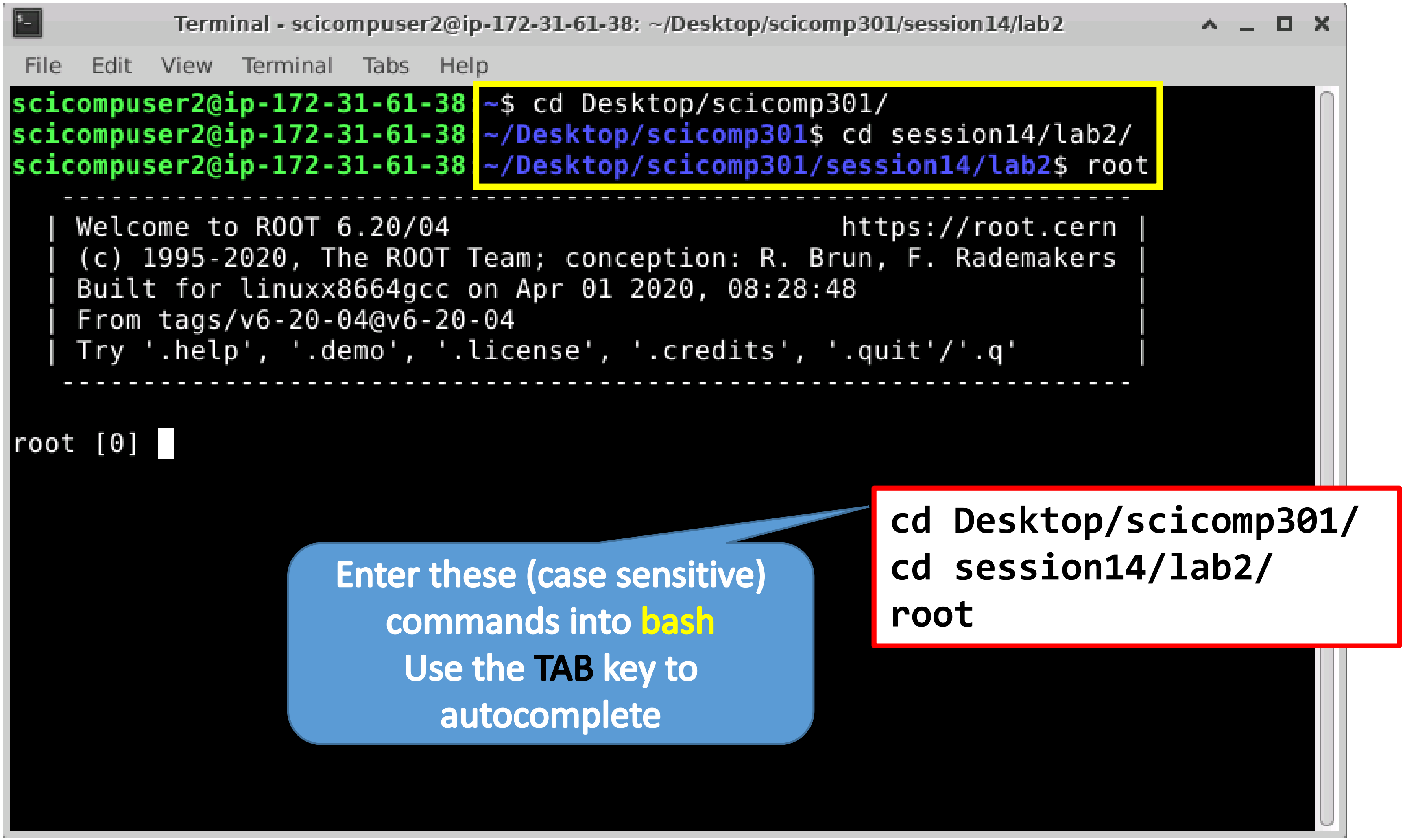
A **ROOT** histogram can automatically tally every occurrence in a vector

Only label ASCII values that occur more than 6% in the text file

Run Lab 2: freq_histogram.cpp



Run Lab 2: freq_histogram.cpp



Run Lab 2: freq_histogram.cpp

Terminal - scicompuser2@ip-172-31-61-38: ~/Desktop/scicomp301/session14/lab2

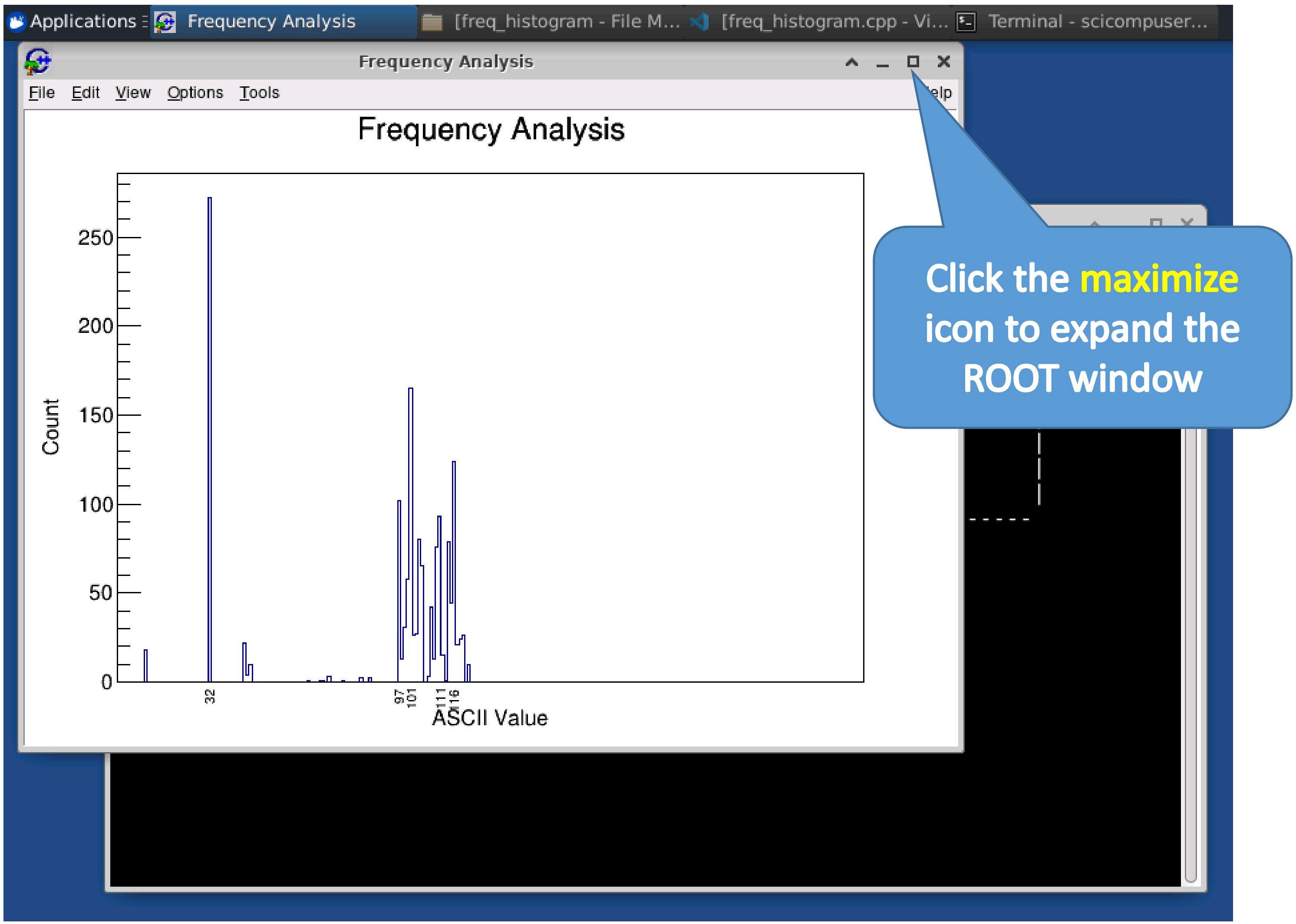
File Edit View Terminal Tabs Help

```
scicompuser2@ip-172-31-61-38:~$ cd Desktop/scicomp301/
scicompuser2@ip-172-31-61-38:~/Desktop/scicomp301$ cd session14/lab2/
scicompuser2@ip-172-31-61-38:~/Desktop/scicomp301/session14/lab2$ root
   ------------------------------------------------------------------
  | Welcome to ROOT 6.20/04                        https://root.cern |
  | (c) 1995-2020, The ROOT Team; conception: R. Brun, F. Rademakers |
  | Built for linuxx8664gcc on Apr 01 2020, 08:28:48                 |
  | From tags/v6-20-04@v6-20-04                                      |
  | Try '.help', '.demo', '.license', '.credits', '.quit'/'.q'       |
   ------------------------------------------------------------------

root [0] .x freq_histogram.cpp("gettysburg.txt")
root [1]
```

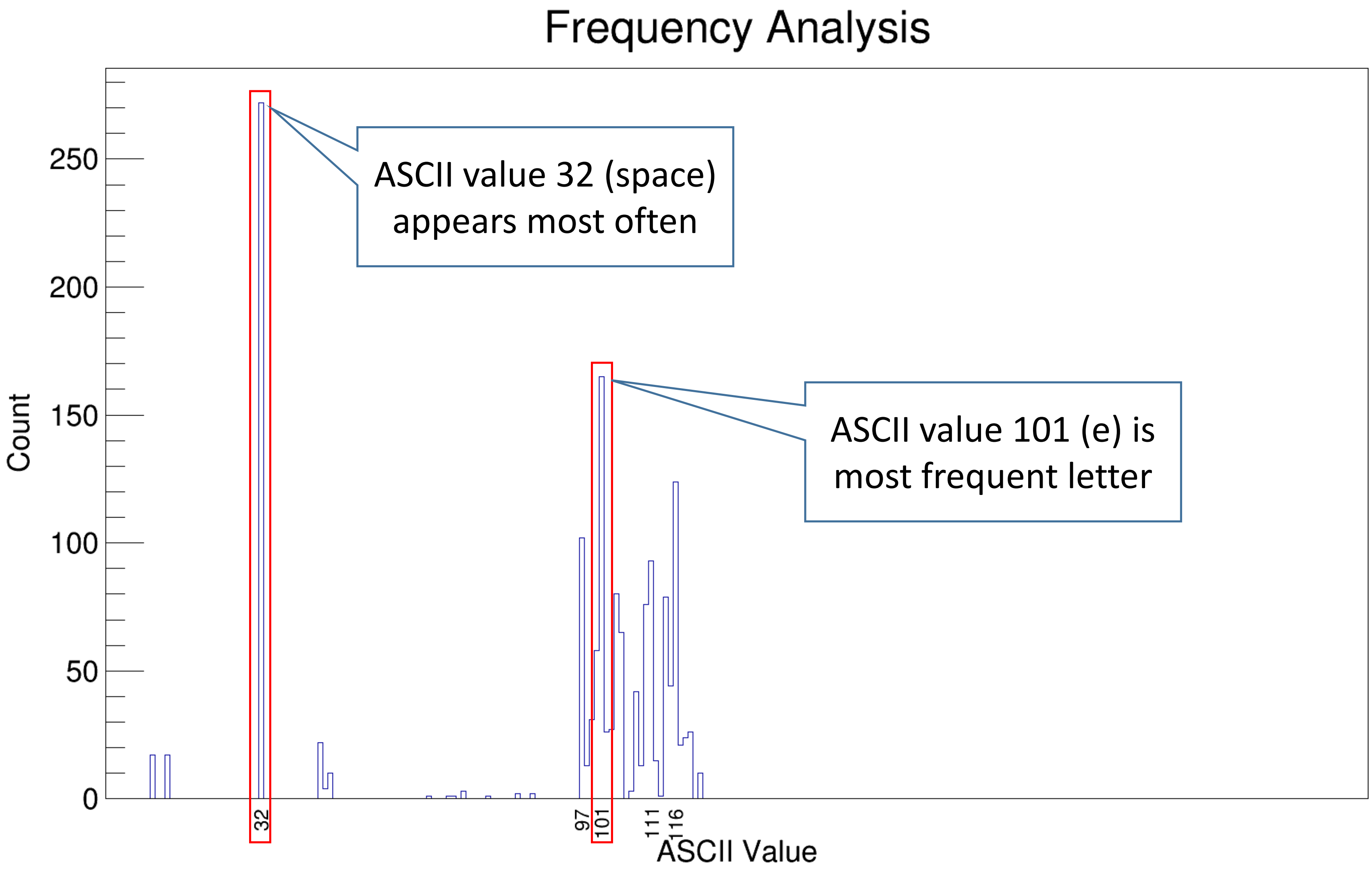
```
.x freq_histogram("gettysburg.txt")
```

Enter this (case sensitive) command into ROOT

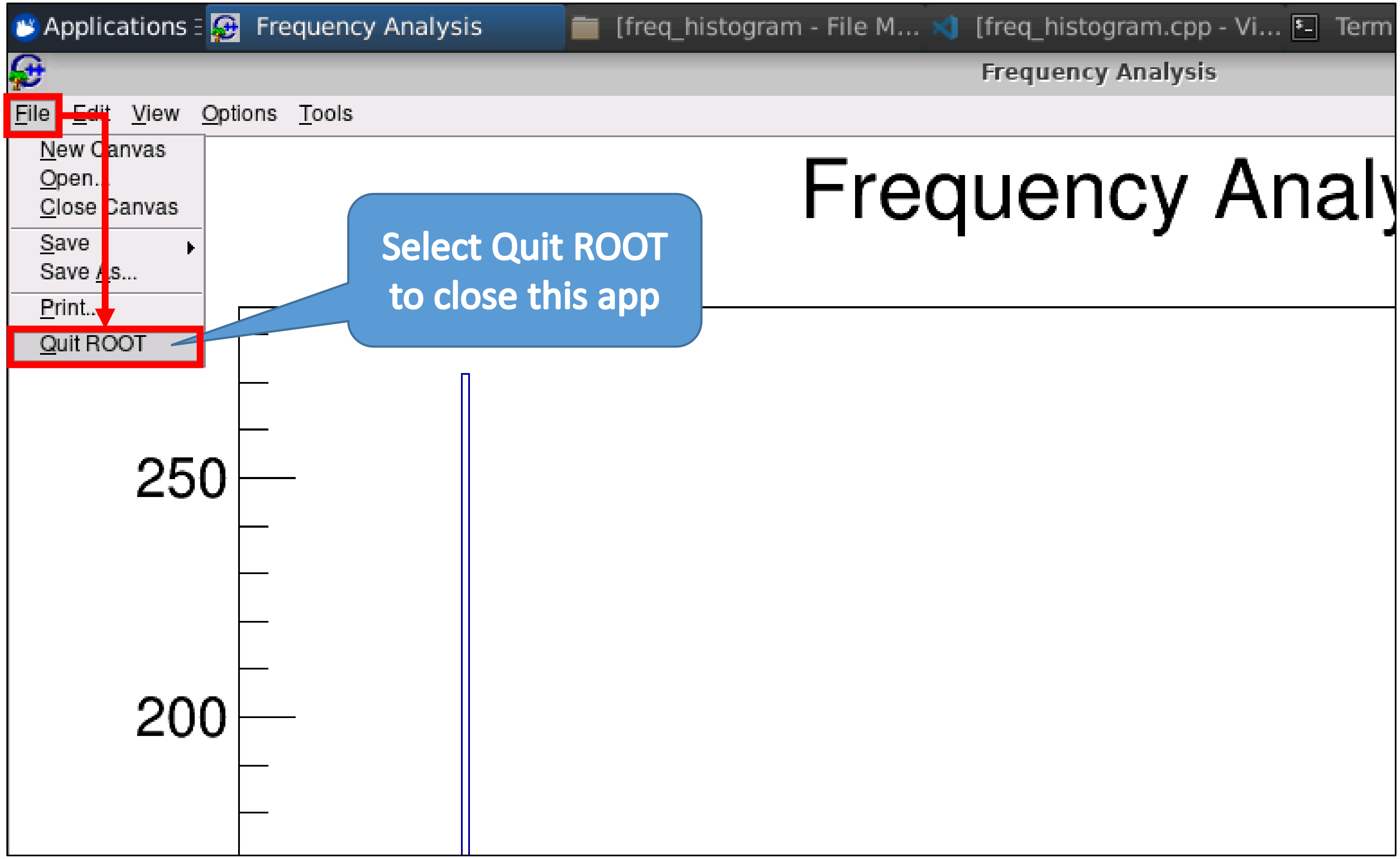
Check Lab 2: freq_histogram.cpp



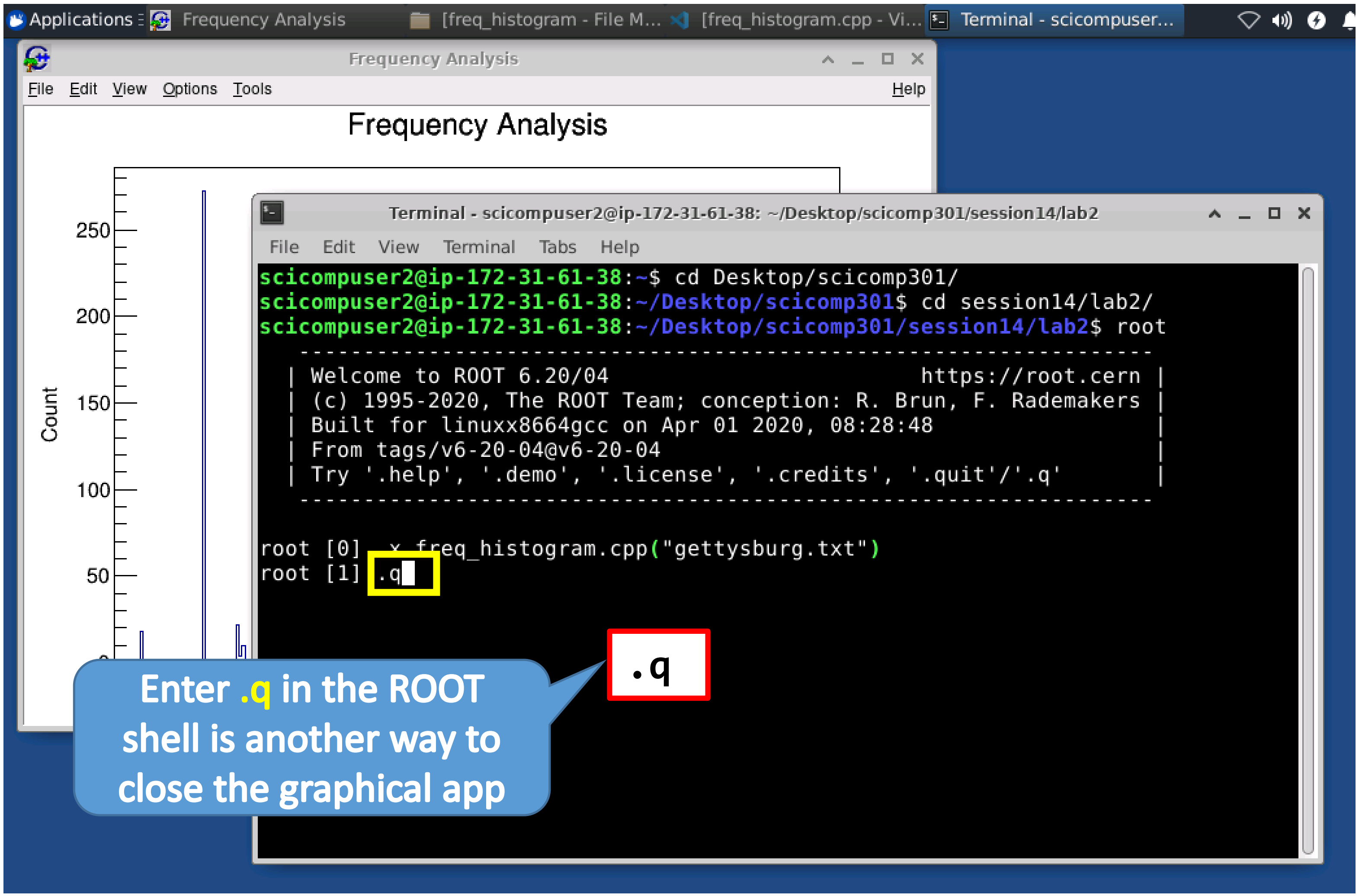
Histogram of Lincoln's Gettysburg Address



Stop Lab 2: **freq_histogram.cpp**

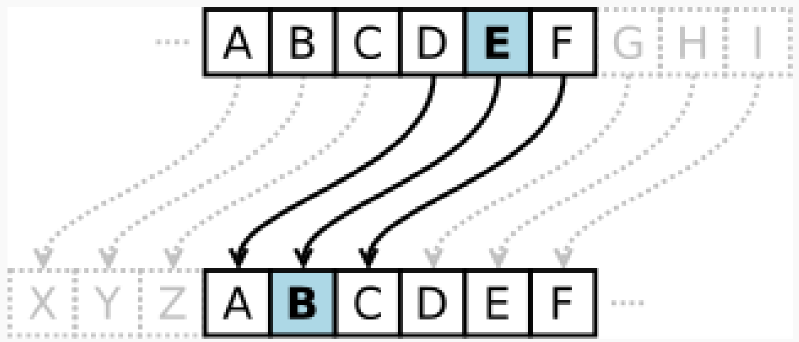


Stop Lab 2: freq_histogram.cpp



The Caesar Shift Cipher

- Roman Emperor **Julius Caesar** used a simple (but effective for its time) encryption scheme for his private correspondence
- To create "**cipher text**" from "**plain text**" simply shift the original letters **forward** (***or backward***) a given number of letters in the alphabet
- To decrypt the message, simply **reverse the sign of the shift**



The action of a Caesar cipher is to replace each plaintext letter with a different one a fixed number of places down the alphabet. The cipher illustrated here uses a left shift of three, so that (for example) each occurrence of E in the plaintext becomes B in the ciphertext.

The Caesar Shift Cipher

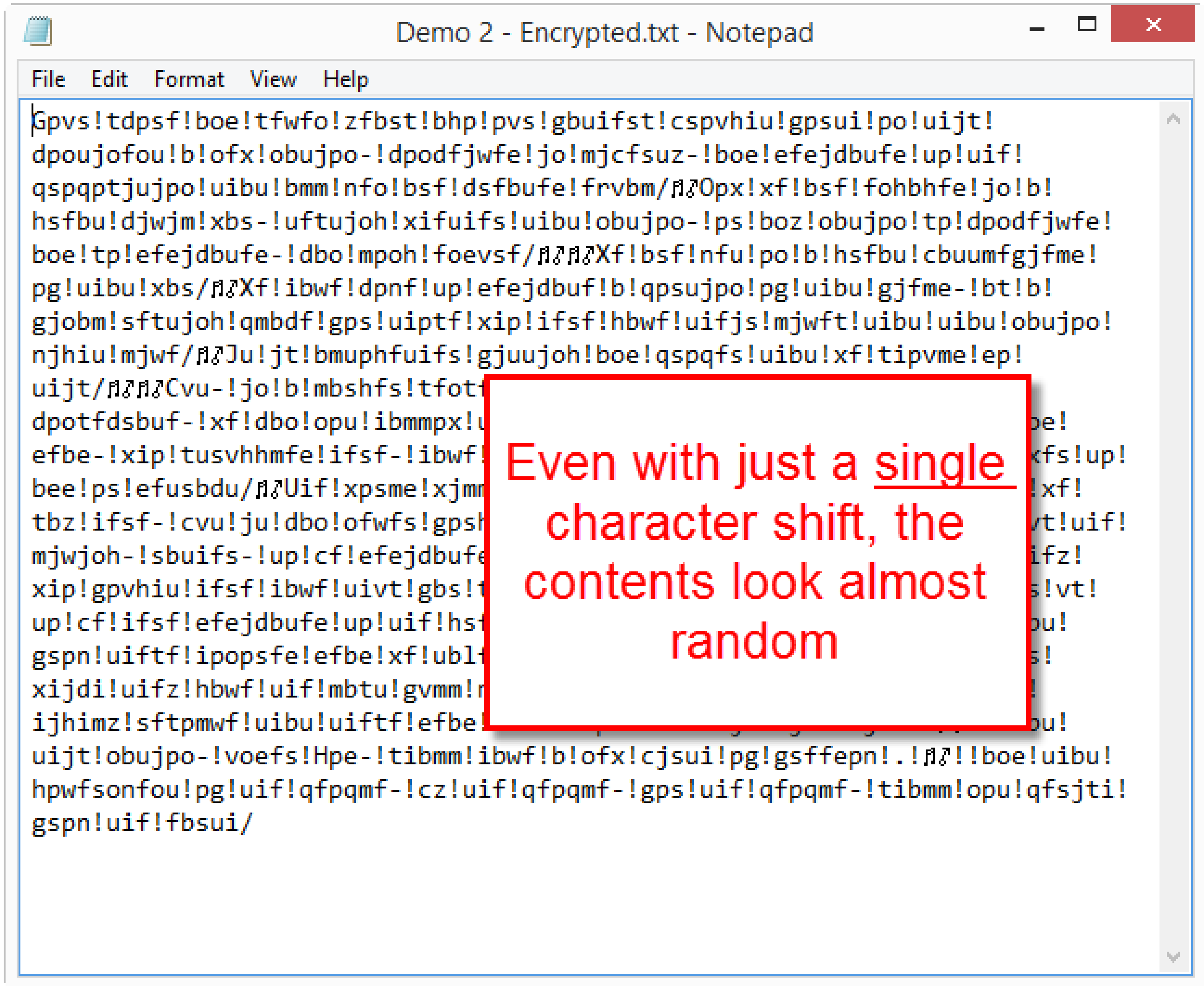
- Consider Lincoln's Gettysburg Address:

  Four score and seven years ago our fathers brought forth on this continent a new nation, conceived in liberty, and dedicated to the proposition that all men are created equal.

  Now we are engaged in a great civil war, testing whether that nation, or any nation so conceived and so dedicated, can long endure…

- How effective is the Caesar Shift at encrypting **plaintext**?
- What if we shifted each letter by just **one** position?
  - Simply add **+1** to the ASCII value of each plaintext character
  - Save the shifted values to a new **ciphertext** file

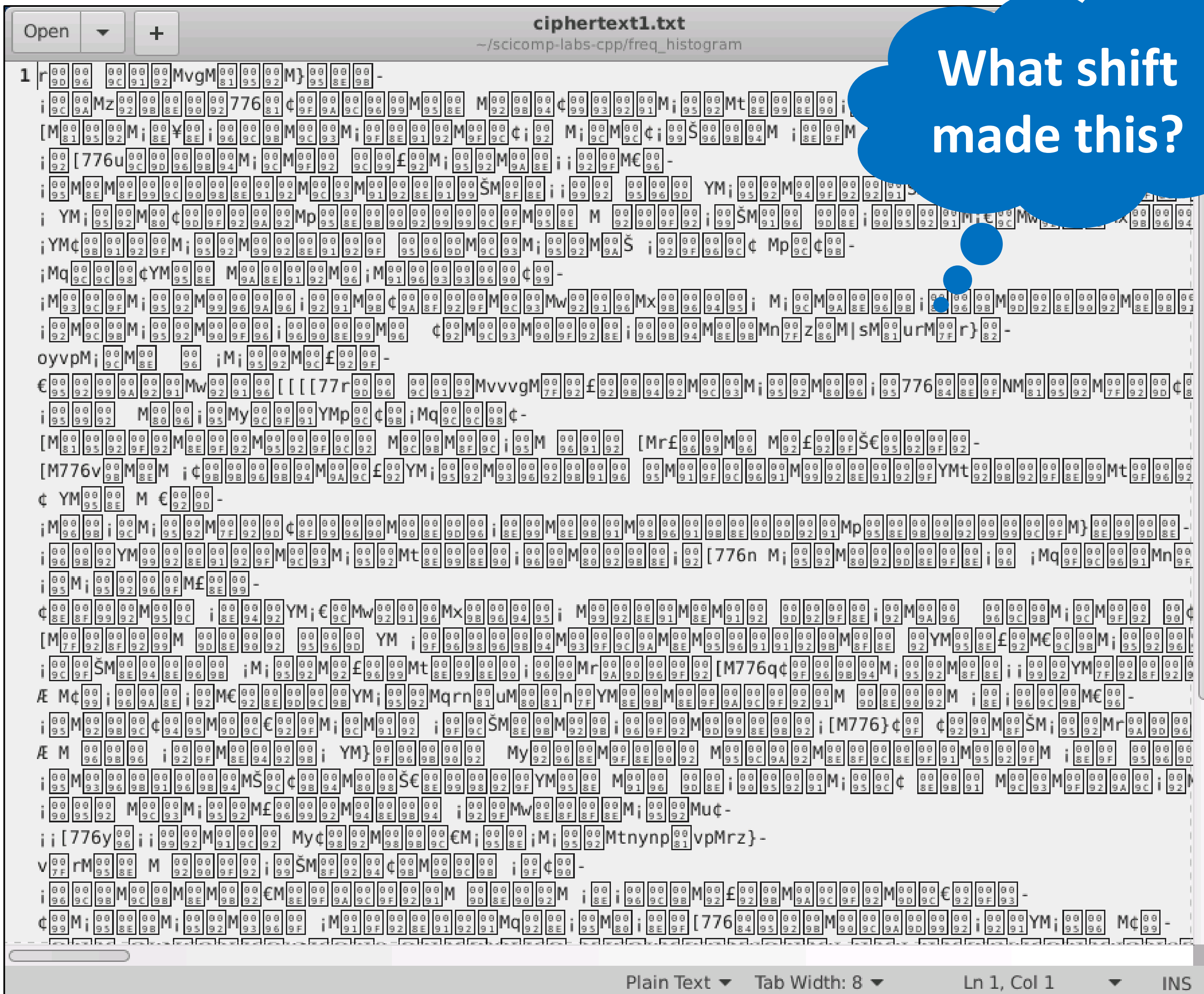
The Caesar Shift Cipher



The Caesar Shift Cipher

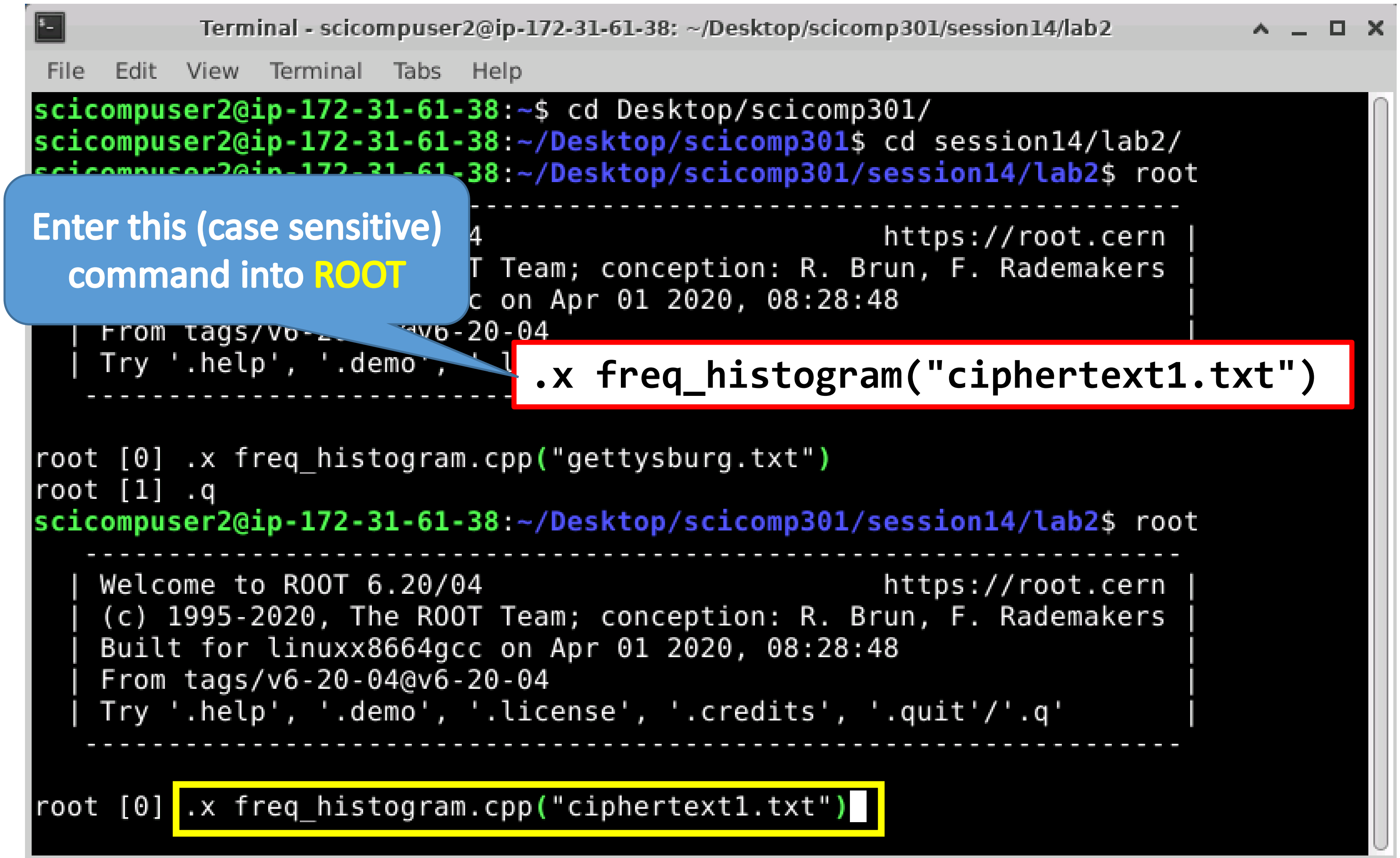
- If we are given a text file written in **English** and encrypted by a **Caesar Shift**, can we figure out **what shift value** was used?
- We could use "**brute force**" and try every possible value to see what shift produces legible prose…
  - **But we'd need to possibly try every shift value from +1 to +255**
  - It would take a long time to sift through all the decrypted files because all incorrect **shift** values generate more *gibberish*
- Can we gleam any insight from analyzing the **character histogram** of the encrypted file?

ciphertext1.txt

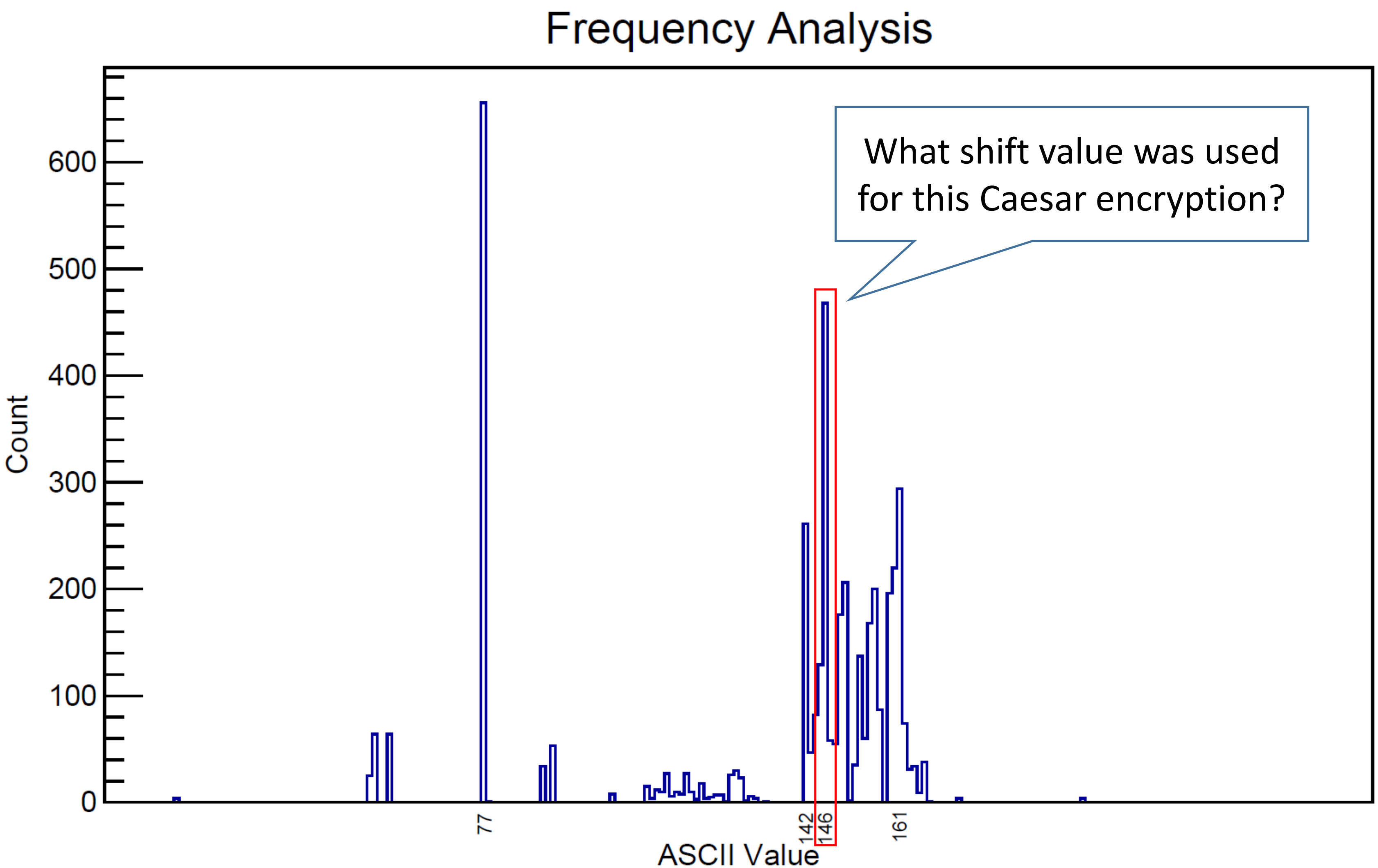


What shift made this?

Run Lab 2: freq_histogram.cpp

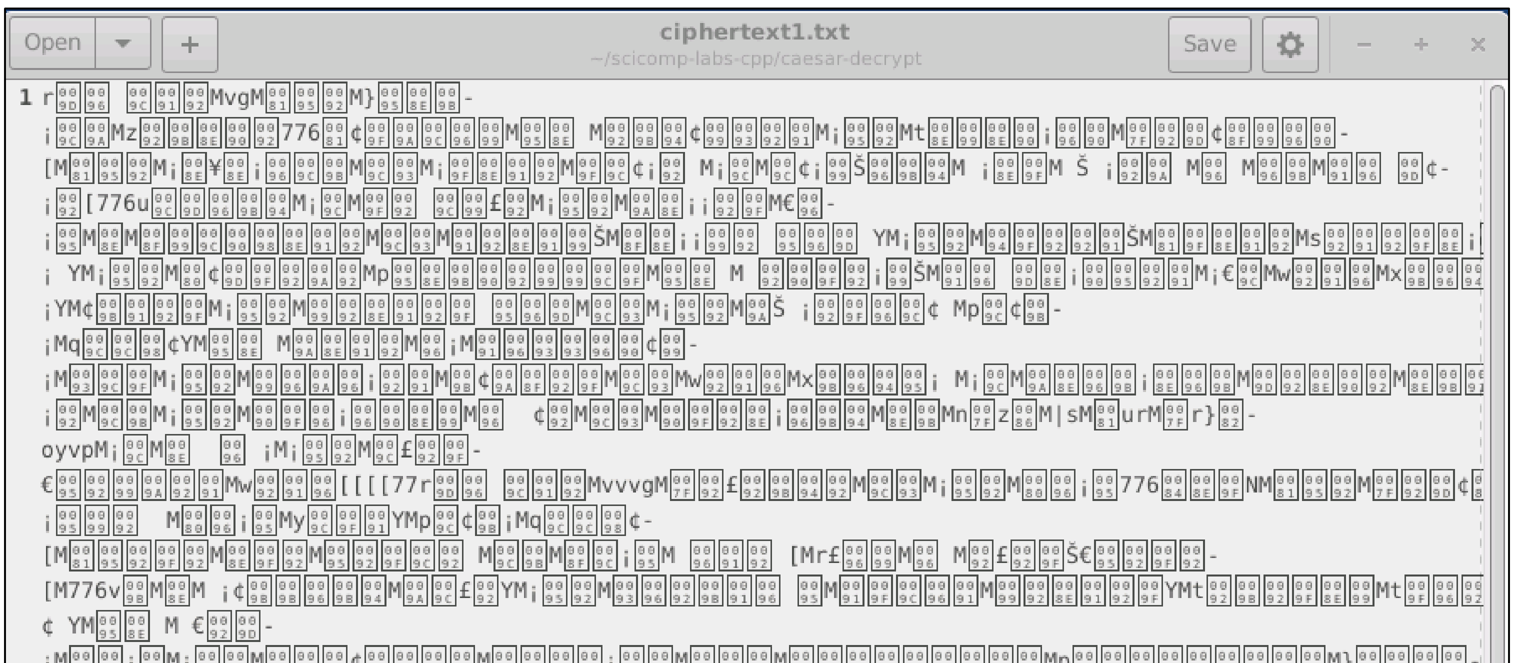


Histogram of Ciphertext #1



Open Lab 3 – Caesar Decrypt

- **Your mission is to decrypt the ciphertext1.txt file**
- What if the survival of your country depended upon your ability to crack the encryption?



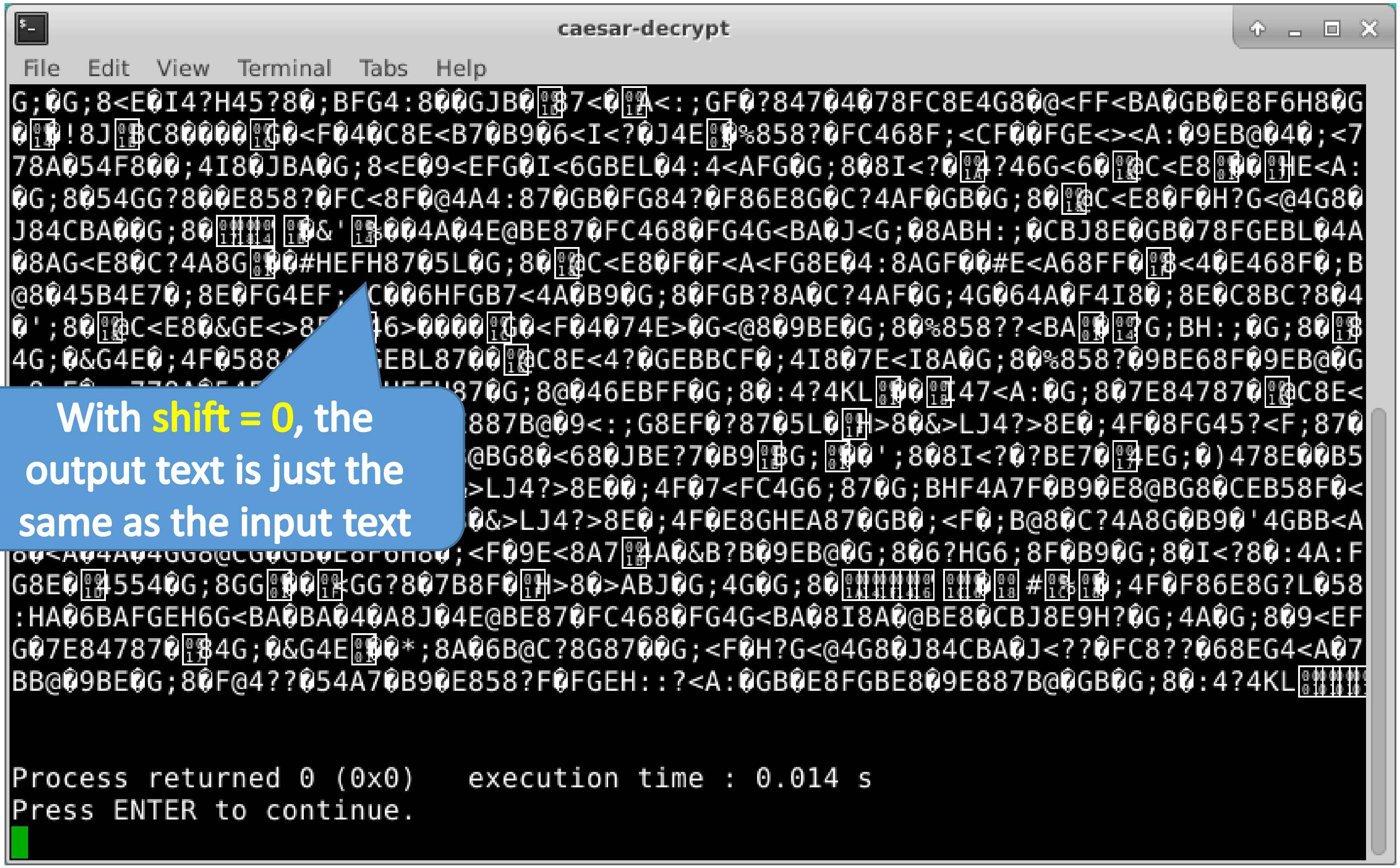
View Lab 3 – Caesar Decrypt

main.cpp

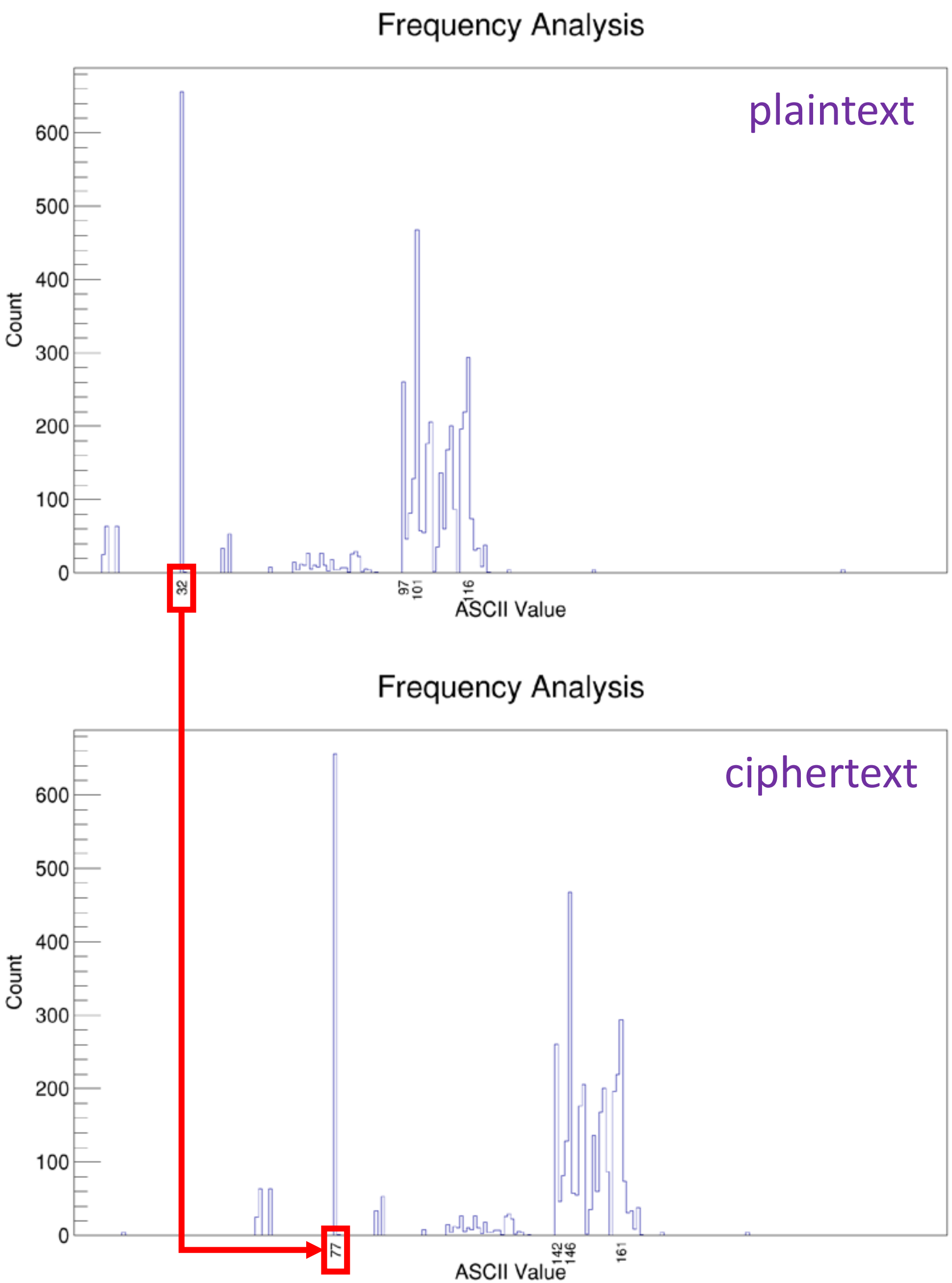
```cpp
#include "stdafx.h"

using namespace std;

int main()
{
    // Open the ciphertext file
    ifstream ifs("ciphertext1.txt", ios::binary | ios::ate);

    // Read the input file straight into the buffer
    ifstream::pos_type pos = ifs.tellg();
    vector<unsigned char> buff(pos);
    ifs.seekg(0, ios::beg);
    ifs.read((char*)(buff.data()), pos);
    ifs.close();

    int shift = 0;

    // Shift every character in ciph
    for (auto c : buff)
        cout << (char)(c + shift);

    cout << endl << endl;

    return 0;
}
```

What will the output be if we leave **shift = 0** ?

Run Lab 3 – Caesar Decrypt



With **shift = 0**, the output text is just the same as the input text



Notice the ***relative*** letter frequencies (individual bar heights) are the same before and after encryption

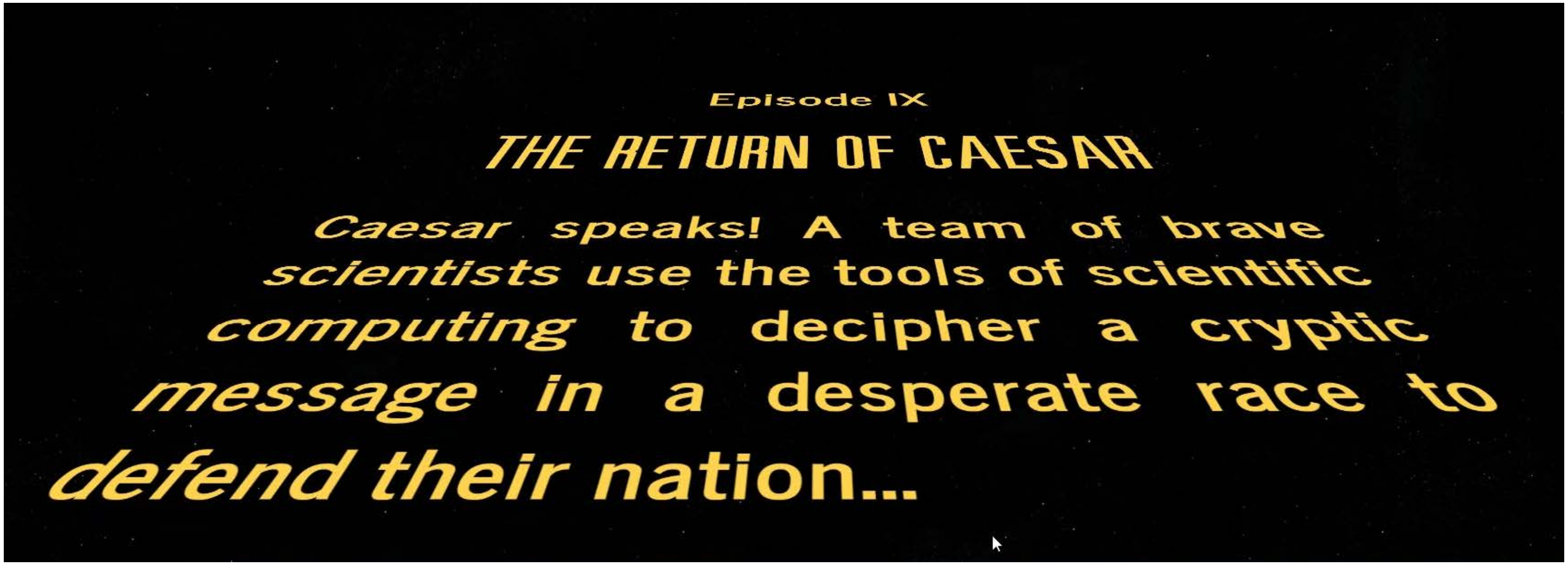
Edit Lab 3 – Caesar Decrypt

main.cpp

```cpp
#include "stdafx.h"

using namespace std;

int main()
{
    // Open the ciphertext file
    ifstream ifs("ciphertext1.txt", ios::binary | ios::ate);

    // Read the input file straight into the buffer
    ifstream::pos_type pos = ifs.tellg();
    vector<unsigned char> buff(pos);
    ifs.seekg(0, ios::beg);
    ifs.read((char*)(buff.data()), pos);
    ifs.close();

    int shift = 0;

    // Shift every character in ciph
    for (auto c : buff)
        cout << (char)(c + shift);

    cout << endl << endl;

    return 0;
}
```

What Caesar shift value will ***reverse*** the encryption that produced **ciphertext1.txt** ?

Check Lab 3 – Caesar Decrypt



The Caesar Shift Cipher

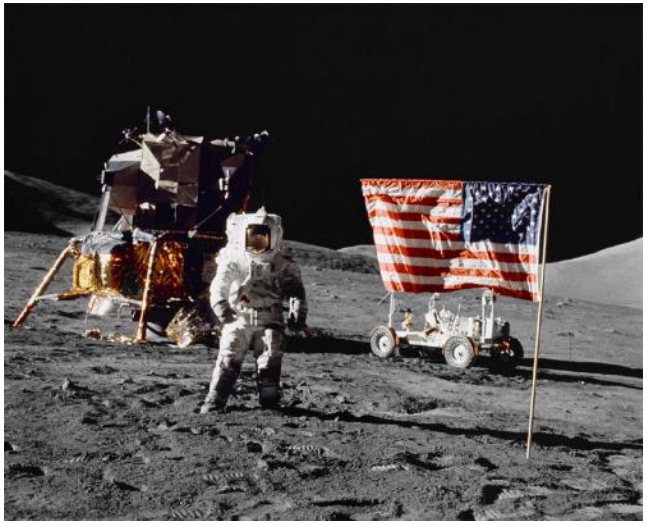
Because the **Caesar Shift** is a **monoalphabetic substitution cipher**, it is susceptible to **cryptanalysis** (breaking) by **frequency analysis**

Bigram Analysis

- Most Western (Latin influenced) languages have a unique fingerprint from their most frequent **bigrams** (two-letter pair)
  - In **English** the bigrams **TH** and **HE** occur most often, since “the” is the most frequent word in English
  - All languages have definite articles like “the” but each language spells theirs differently, and this helps establish a **distinct statistical profile** for each language
- Linguists and statisticians have compiled lists of the most frequent **bigrams** *per* language
- We will analyze the bigrams in **President Kennedy’s Rice University Speech** - where he set the goal in 1962 for the USA to reach the moon before 1970!

Kennedy's Moon Speech in 1962

"We choose to go to the moon in this decade and do the other things, not because they are easy, *but <u>because</u> they are hard*, because that goal will serve to organize and measure **the best of our energies and skills**, because that challenge is one that we are willing to accept, one we are unwilling to postpone, and one which we intend to win."



Bigram Analysis of Kennedy's Moon Speech (English)

freq-bigrams

File Edit View Terminal Tabs Help

```
Most recurring bigrams in file:
  Kennedy Moon English.txt

ASCII            CHARS    FREQ
(65, 76)         AL       1.13
(65, 78)         AN       1.79
(65, 83)         AS       1.13
(65, 84)         AT       1.59
(69, 65)         EA       1.53
(69, 78)         EN       1.53
(69, 82)         ER       1.66
(69, 83)         ES       1.46
(69, 84)         ET       1.72
(72, 65)         HA       1.33
(72, 69)         HE       2.32
(73, 78)         IN       1.53
(76, 69)         LE       0.99
(76, 76)         LL       0.99
(78, 68)         ND       1.13
(78, 69)         NE       0.99
(78, 71)         NG       1.33
(78, 84)         NT       0.99
(79, 78)         ON       1.33
(82, 69)         RE       1.53
(83, 84)         ST       1.53
(84, 69)         TE       0.99
(84, 72)         TH       3.38
(84, 79)         TO       1.66

Process returned 0 (0x0)   execution time : 0.023 s
Press ENTER to continue.
```

Kennedy’s Moon Speech Translated

Nous choisissons d'aller sur la lune. Nous choisissons d'aller sur la lune dans cette décennie et de faire d'autres choses, non pas parce qu'ils sont faciles, mais parce qu'ils sont difficiles, parce que ce but servira à organiser et mesurer le meilleur de nos énergies et de compét
parce que ce défi est l'un
sommes prêts à accepter,
ne sommes pas disposés à
et celui qui nous avons l'i
gagner.

Elegimos ir a la Luna. Elegimos ir a la Luna en esta década y hacer las otras cosas, no porque sean fáciles, sino porque son difíciles, porque esa meta servirá para organizar y medir lo mejor de nuestras energías y habilidades, porque ese desafío es uno que estamos dispuestos a
dispuestos
nemos la

Wir wählen, um zum Mond zu fliegen. Wir wählen, um zum Mond in diesem Jahrzehnt zu gehen und die anderen Dinge, nicht weil sie leicht sind, sondern weil sie hart sind, denn das Ziel wird dazu dienen, zu organisieren und zu messen, das Beste aus unserer Energien und Fähigkeiten, denn das ist eine Herausforderung dass wir bereit sind zu akzeptieren, das wir nicht bereit sind, zu verschieben, und eine, die wir beabsichtigen, zu gewinnen.

Bigram Statistics by Language

Bigrams - Kennedy Speech

Color	Meaning
Yellow	= Unique Indicators (for each language)
Green	= Relative Indicator (see German)

English		Speech
TH	2.71	3.38
HE	2.33	2.32
IN	2.03	1.53
ER	1.78	1.66
AN	1.61	1.79
RE	1.41	
ES	1.32	
ON	1.32	
ST	1.25	
NT	1.17	
EN	1.13	
AT	1.12	
Top 5:	**10.46**	**10.68**

Spanish		Speech
DE	2.57	2.41
ES	2.31	2.84
EN	2.27	1.75
EL	2.01	1.57
LA	1.80	1.69
OS	1.79	
ON	1.61	
AS	1.56	
ER	1.52	
RA	1.47	
AD	1.43	
AR	1.43	
Top 5:	**10.96**	**10.26**

French		Speech
ES	2.91	2.17
LE	2.08	2.17
DE	2.02	2.11
EN	1.97	1.61
ON	1.70	2.00
NT	1.69	
RE	1.62	
AN	1.28	
LA	1.25	
ER	1.21	
TE	1.19	
EL	1.15	
Top 5:	**10.68**	**10.06**

German		Speech
ER	3.90	3.29
EN	3.61	4.44
CH	2.36	1.67
DE	2.31	1.90
EI	1.98	1.73
TE	1.98	
IN	1.71	
ND	1.68	
IE	1.48	
GE	1.45	
ST	1.21	
NE	1.19	
Top 5:	**14.16**	$\boxed{\textbf{13.03}}$

German is the most consistent language for bigrams

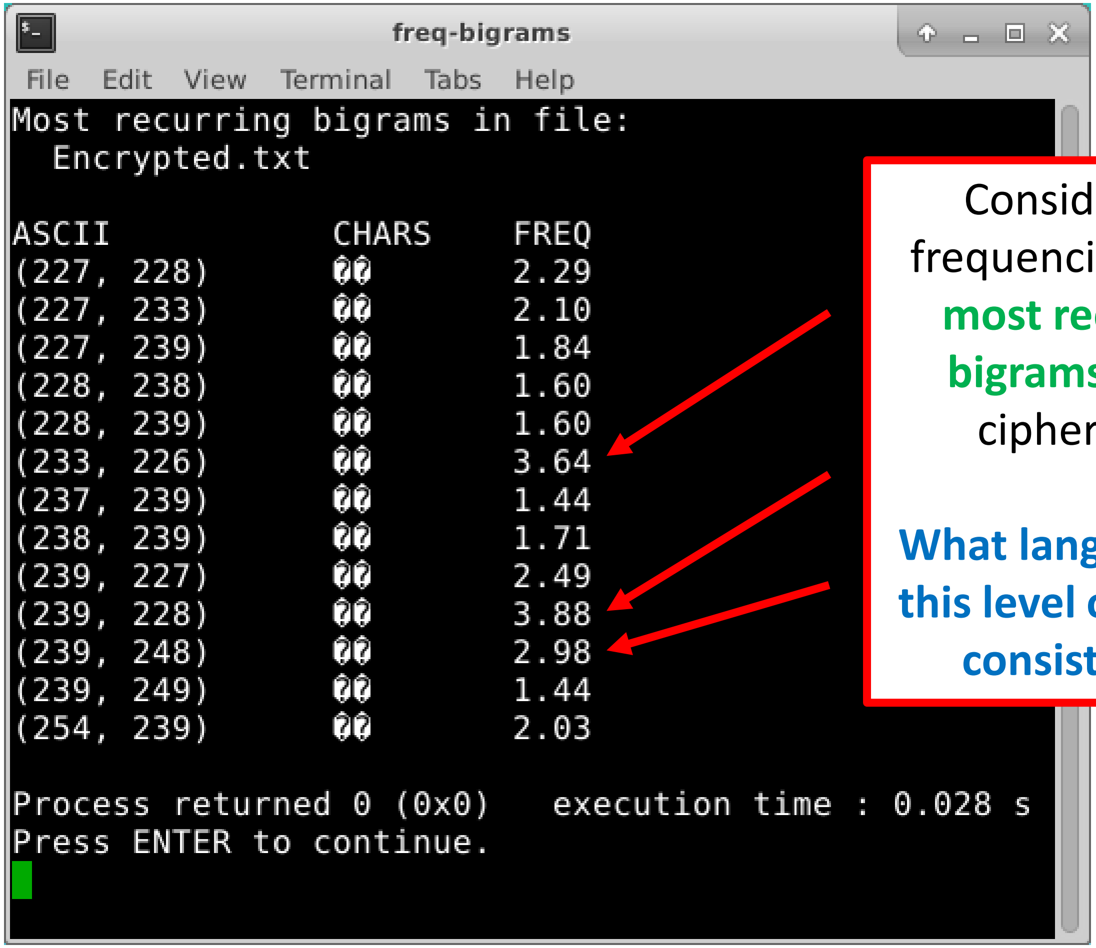
Lab 4 – Frequency of Bigrams

- You have been given another encrypted message
- It is encrypted with a **monoalphabetic substitution cipher**
  - Frequency analysis **does not** suggest a simple Caesar shift was used
  - Rather, a ***different shift* value** was used for each plaintext letter!
- Even if you are ***unable*** to break the encryption, **can you tell what language was used** when writing the plaintext?
  - Q: **Are you serious**? How can you possibly discern the author's language if you **cannot even read the contents** in the first place?
  - A: Enigma was also **unbreakable**

View Lab 4 – Encrypted.txt

What was the source language?

Run Lab 4 – Frequency of Bigrams



```
freq-bigrams
File  Edit  View  Terminal  Tabs  Help
Most recurring bigrams in file:
  Encrypted.txt

ASCII              CHARS     FREQ
(227, 228)         ��        2.29
(227, 233)         ��        2.10
(227, 239)         ��        1.84
(228, 238)         ��        1.60
(228, 239)         ��        1.60
(233, 226)         ��        3.64
(237, 239)         ��        1.44
(238, 239)         ��        1.71
(239, 227)         ��        2.49
(239, 228)         ��        3.88
(239, 248)         ��        2.98
(239, 249)         ��        1.44
(254, 239)         ��        2.03

Process returned 0 (0x0)   execution time : 0.028 s
Press ENTER to continue.
```

Consider the frequencies of the **most recurring bigrams** in the cipher text.

What language has this level of bigram consistency?

Lab 4 – Plaintext

Bonus - Decrypted - Twilight Chapter One - German.txt - Notepad

File Edit Format View Help

KAPITEL EINS - ERSTEN BLICK

Meine Mutter fuhr mich zum Flughafen m
am Himmel ein perfekter, wolkenlosen b
Ich wurde als Abschiedsgeste tragen. M

In der olympischen Halbinsel im Nordwe
existiert unter einem nahezu konstante
jeder andere Ort in den Vereinigten St
allgegenwärtigen Schatten, die meine M
war. Es war in dieser Stadt, die ich g
vierzehn war. Das war das Jahr habe ic
mein Vater, Charlie, Urlaub gemacht mi

Es war nach
verabscheute

Ich liebte P
Stadt.

"Bella", sag
müssen nicht

Meine Mutter
Panik, als i
für sich sel
bezahlt, gäb
verloren, ab

"Ich will ge
Lüge so oft

Bonus - Translated - Twilight Chapter One - English.txt - Notepad

File Edit Format View Help

CHAPTER ONE – FIRST SIGHT

My mother drove me to the airport with the windows rolled down. It was seventy-five degrees in Phoenix, the sky a perfect, cloudless blue. I was wearing my favorite shirt – sleeveless, white eyelet lace; I was wearing it as a farewell gesture. My carry-on item was a parka.

In the Olympic Peninsula of northwest Washington State, a small town named Forks exists under a near-constant cover of clouds. It rains on this inconsequential town more than any other place in the United States of America. It was from this town and its gloomy, omnipresent shade that my mother escaped with me when I was only a few months old. It was in this town that I'd been compelled to spend
y summer until I was fourteen. That was the year I finally put my foot down; these past
, my dad, Charlie, vacationed with me in California for two weeks instead.

rks that I now exiled myself – an action that I took with great horror. I detested Forks.

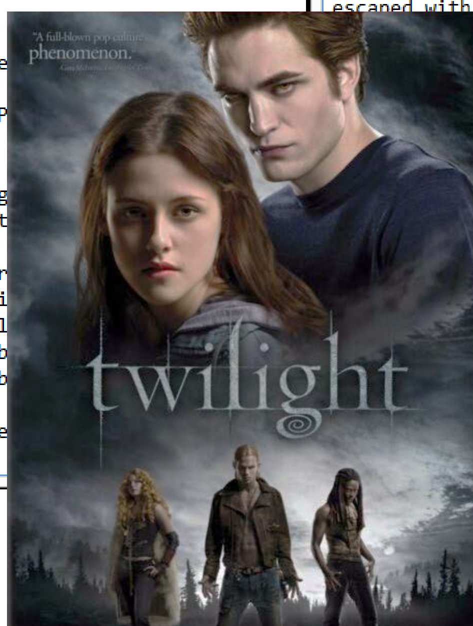
nix. I loved the sun and the blistering heat. I loved the vigorous, sprawling city.

mom said to me – the last of a thousand times – before I got on the plane. "You don't have

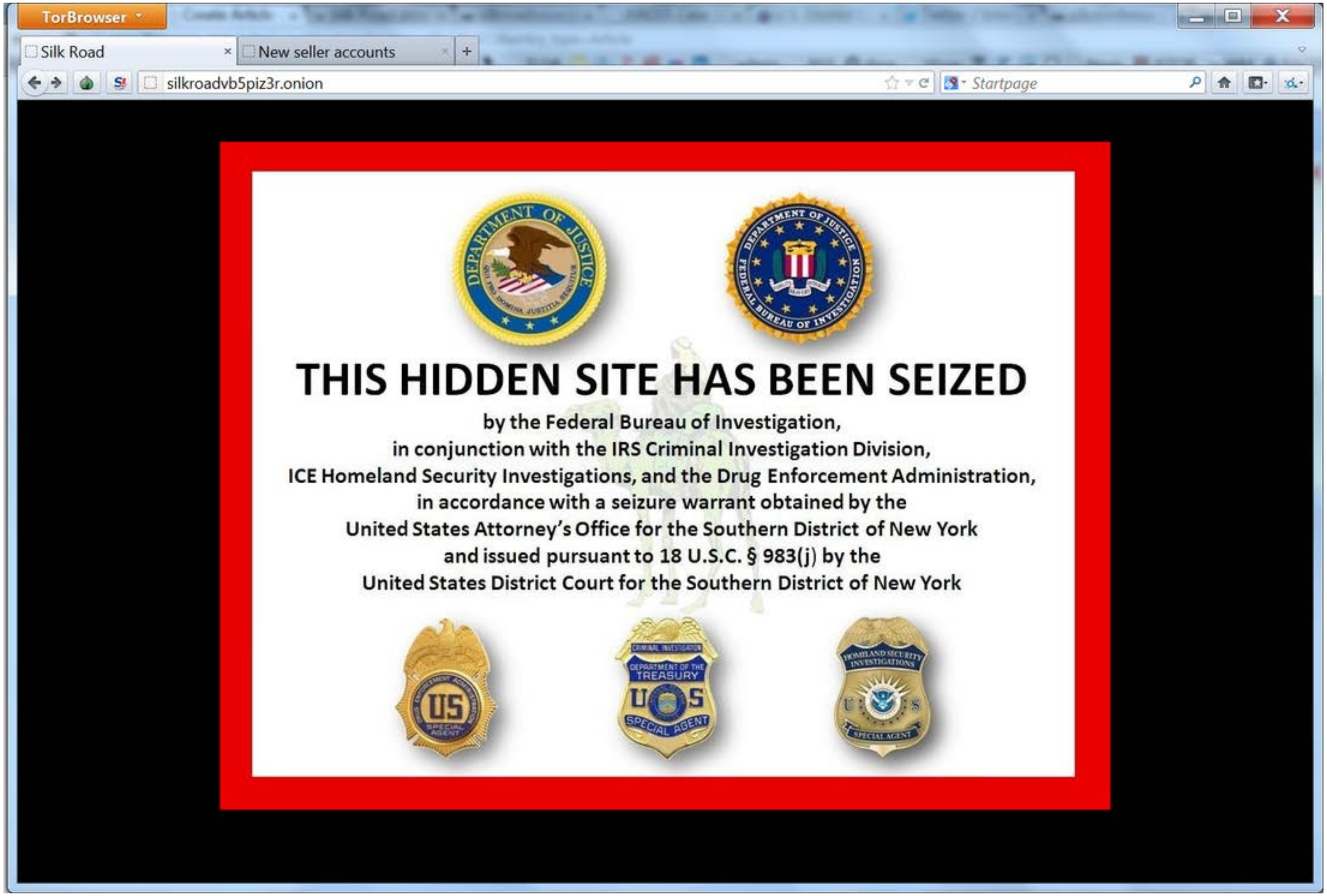
like me, except with short hair and laugh lines. I felt a spasm of panic as I stared at
ildlike eyes. How could I leave my loving, erratic, harebrained mother to fend for
course she had Phil now, so the bills would probably get paid, there would be food in the
gas in her car, and someone to call when she got lost, but still…

," I lied. I'd always been a bad liar, but I'd been saying this lie so frequently lately
ded almost convincing now.

I said hi."



The Fallacy of the Smarter Criminal



Anagrams

- Different words all spelled with the same set of letters are called **anagrams**
- Given an **English dictionary**, how could you **find all** the anagrams of a word?
- What algorithm would you use? **Trial and error**?

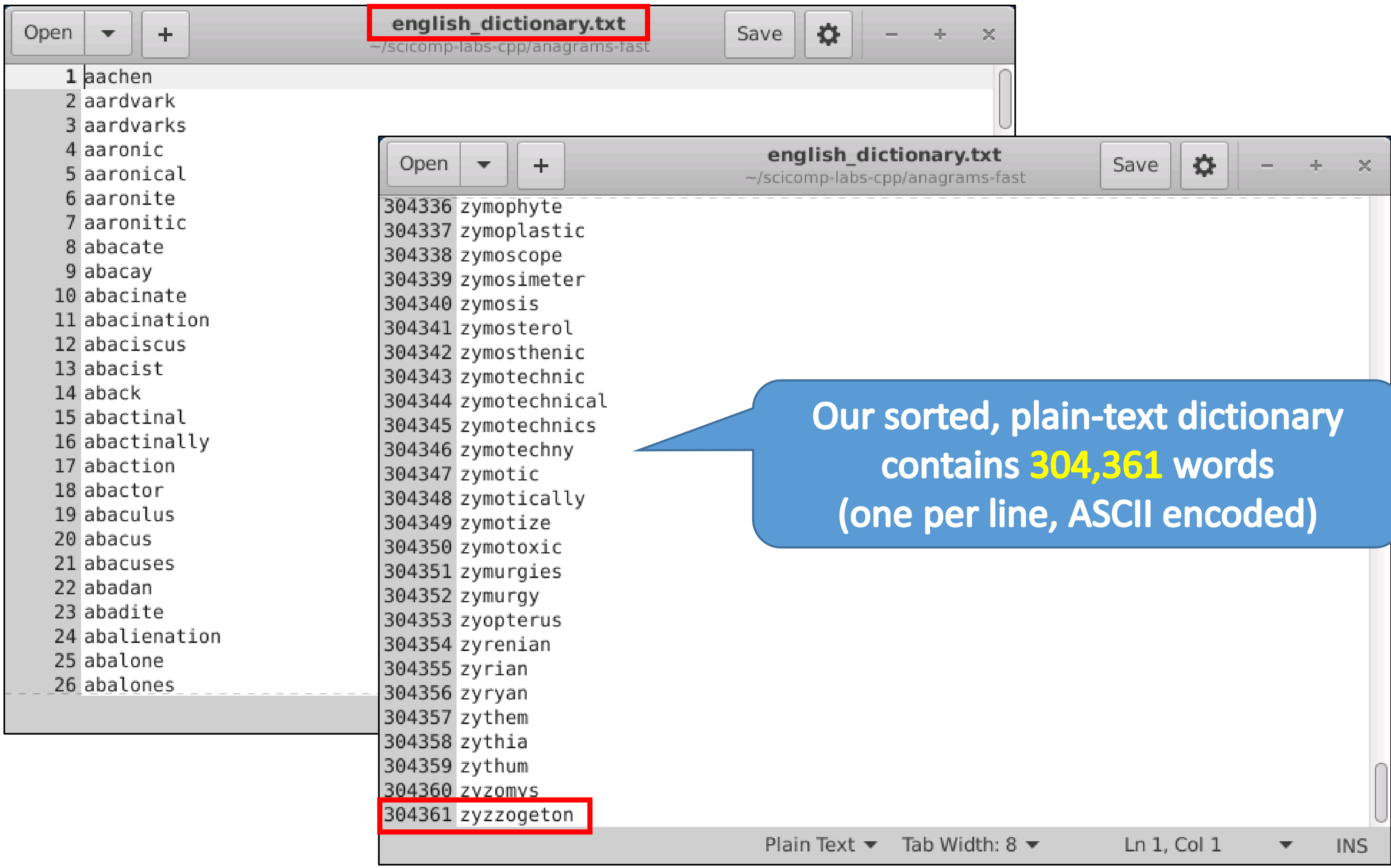
Word	Letters	Anagrams	Permutations
STOP	4	6	24
LEAST	5	10	120
TRACES	6	9	720
PLAYERS	7	7	5,040
RESTRAIN	8	6	40,320
MASTERING	9	4	362,880
SUPERSONIC	10	3	3,628,800

Anagrams

- stop = post, pots, spot, tops
- least = slate, stale, steal, tales
- traces = ???
- players = ???
- restrain = ???
- mastering = ???
- supersonic = ???

Naive Approach = Try every possible permutation of all given letters, checking in dictionary file to see if that permutation it is a valid English word

english_dictionary.txt



Open Lab 5 – Slow Anagrams

main.cpp

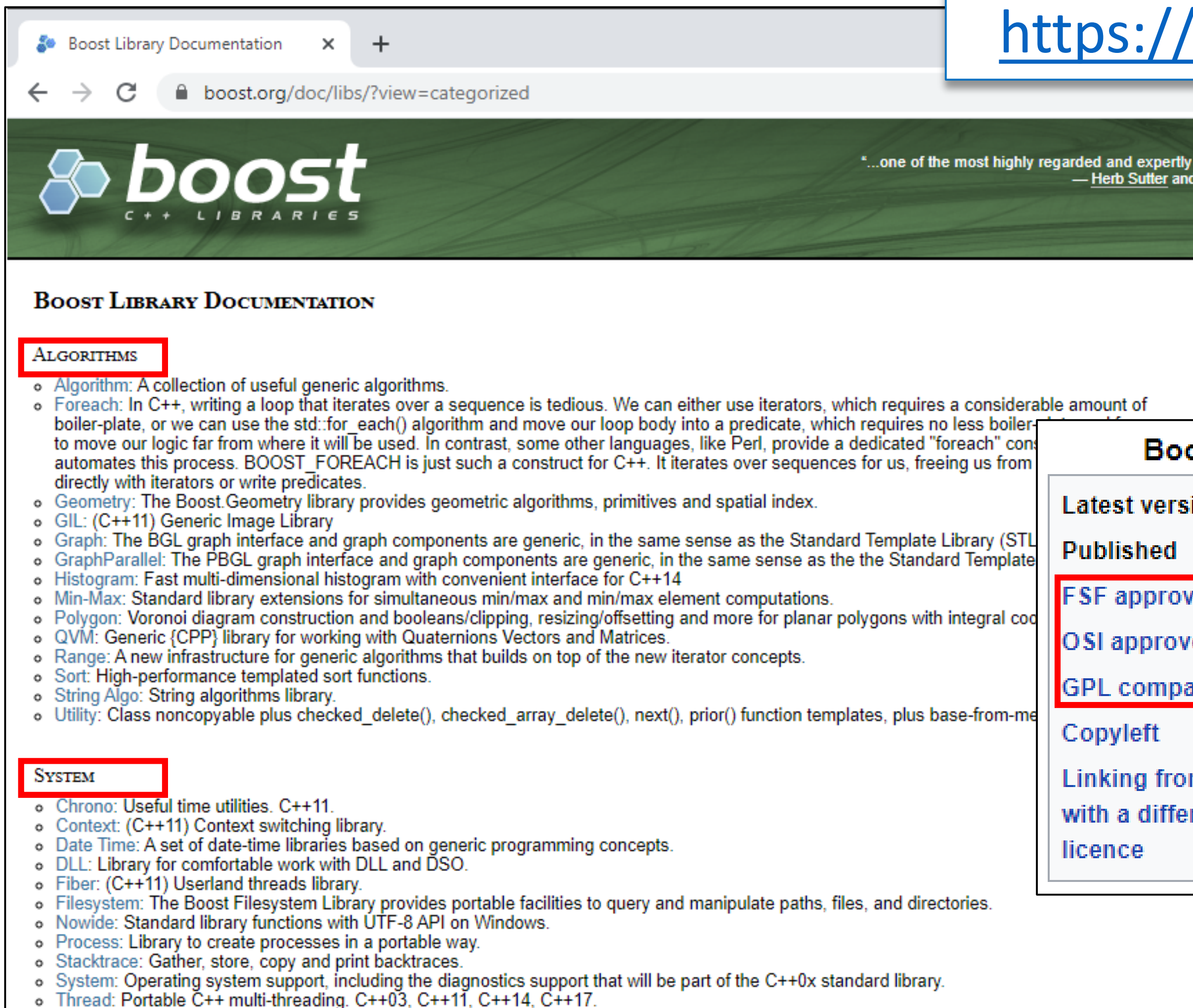
```
#include "stdafx.h"

using namespace std;

vector<string> phrases
{
    "Stop", "Least","Traces", "Players", "Restrain"
};

vec
vec
```

```
cout << "Total search time = "
     << fixed << setprecision(3)
     << timer.elapsed() << " s" << endl;
```

```
int main()
{
    // Read in the dictionary file
    ifstream inputFile("english_dictionary.txt");
    string line;
    while (getline(inputFile, line))
    {
        boost::trim(line);
        if (line.length() > 0)
            dictionary.push_back(line);
    }

    // Start a timer
    boost::timer timer;
```

BOOST Libraries for C++

https://www.boost.org



Boost Library Documentation

boost.org/doc/libs/?view=categorized

boost

C++ LIBRARIES

"...one of the most highly regarded and expertly c

— Herb Sutter and

Boost Library Documentation

Algorithms

- Algorithm: A collection of useful generic algorithms.
- Foreach: In C++, writing a loop that iterates over a sequence is tedious. We can either use iterators, which requires a considerable amount of boiler-plate, or we can use the std::for_each() algorithm and move our loop body into a predicate, which requires no less boiler-to move our logic far from where it will be used. In contrast, some other languages, like Perl, provide a dedicated "foreach" con automates this process. BOOST_FOREACH is just such a construct for C++. It iterates over sequences for us, freeing us from directly with iterators or write predicates.
- Geometry: The Boost.Geometry library provides geometric algorithms, primitives and spatial index.
- GIL: (C++11) Generic Image Library
- Graph: The BGL graph interface and graph components are generic, in the same sense as the Standard Template Library (STL
- GraphParallel: The PBGL graph interface and graph components are generic, in the same sense as the the Standard Template
- Histogram: Fast multi-dimensional histogram with convenient interface for C++14
- Min-Max: Standard library extensions for simultaneous min/max and min/max element computations.
- Polygon: Voronoi diagram construction and booleans/clipping, resizing/offsetting and more for planar polygons with integral coo
- QVM: Generic {CPP} library for working with Quaternions Vectors and Matrices.
- Range: A new infrastructure for generic algorithms that builds on top of the new iterator concepts.
- Sort: High-performance templated sort functions.
- String Algo: String algorithms library.
- Utility: Class noncopyable plus checked_delete(), checked_array_delete(), next(), prior() function templates, plus base-from-me

System

- Chrono: Useful time utilities. C++11.
- Context: (C++11) Context switching library.
- Date Time: A set of date-time libraries based on generic programming concepts.
- DLL: Library for comfortable work with DLL and DSO.
- Fiber: (C++11) Userland threads library.
- Filesystem: The Boost Filesystem Library provides portable facilities to query and manipulate paths, files, and directories.
- Nowide: Standard library functions with UTF-8 API on Windows.
- Process: Library to create processes in a portable way.
- Stacktrace: Gather, store, copy and print backtraces.
- System: Operating system support, including the diagnostics support that will be part of the C++0x standard library.
- Thread: Portable C++ multi-threading. C++03, C++11, C++14, C++17.

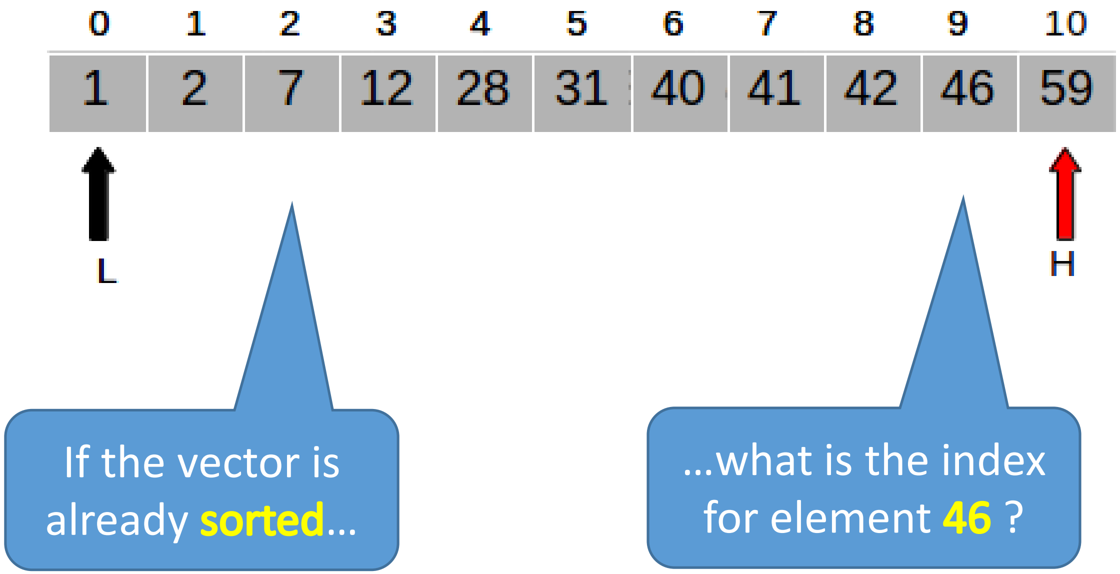
Boost Software License

Latest version	1.0
Published	17 August 2003
FSF approved	Yes
OSI approved	Yes
GPL compatible	Yes
Copyleft	No
Linking from code with a different licence	Yes

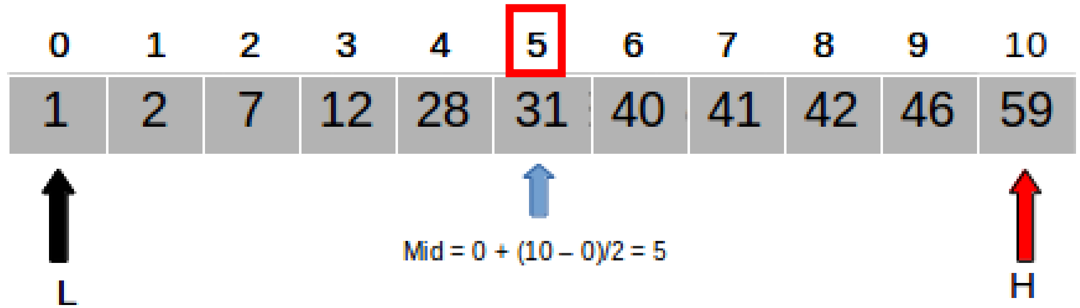
View Lab 5 **Slow Anagrams**

```
// Find any anagrams for every requested phrase
for (auto& phrase : phrases) {

    // Convert phrase to all lowercase
    boost::to_lower(phrase);

    // Create vector of individual characters
    vector<char> letters;
    for (auto s : phrase)
        letters.push_back(s);

    // Add all permutations of letters to words vector
    words.clear();
    Permute<char>(&letters, letters.size());

    // Remove redundant permutations caused
    // by a phrase having duplicated letters
    sort(words.begin(), words.end());
    auto last = unique(words.begin(), words.end());
    words.erase(last, words.end());

    // Display only words that are found in the dictionary
    for (const auto& word : words)
        if (binary_search(dictionary.begin(),
                          dictionary.end(), word))
            cout << word << endl;

    cout << endl;
}
```

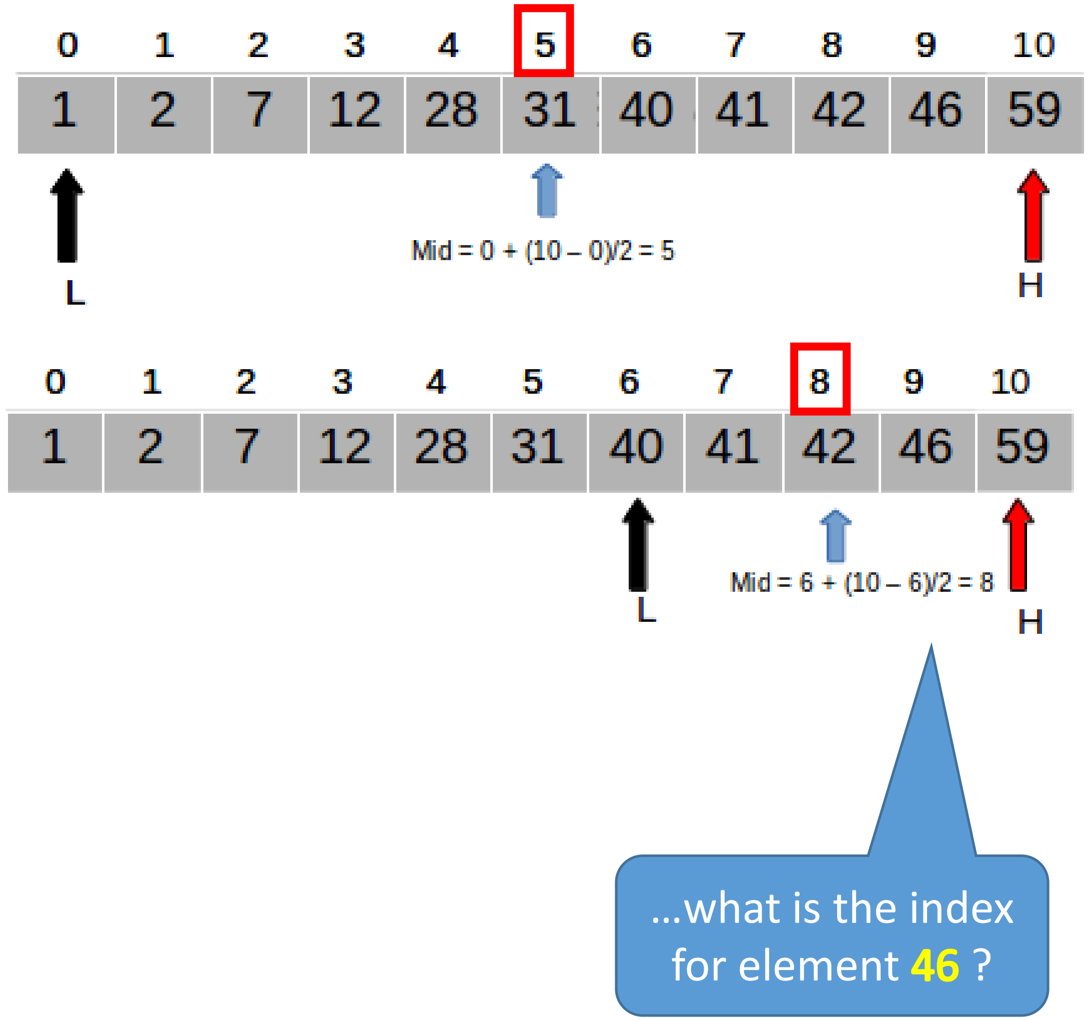
Binary Search



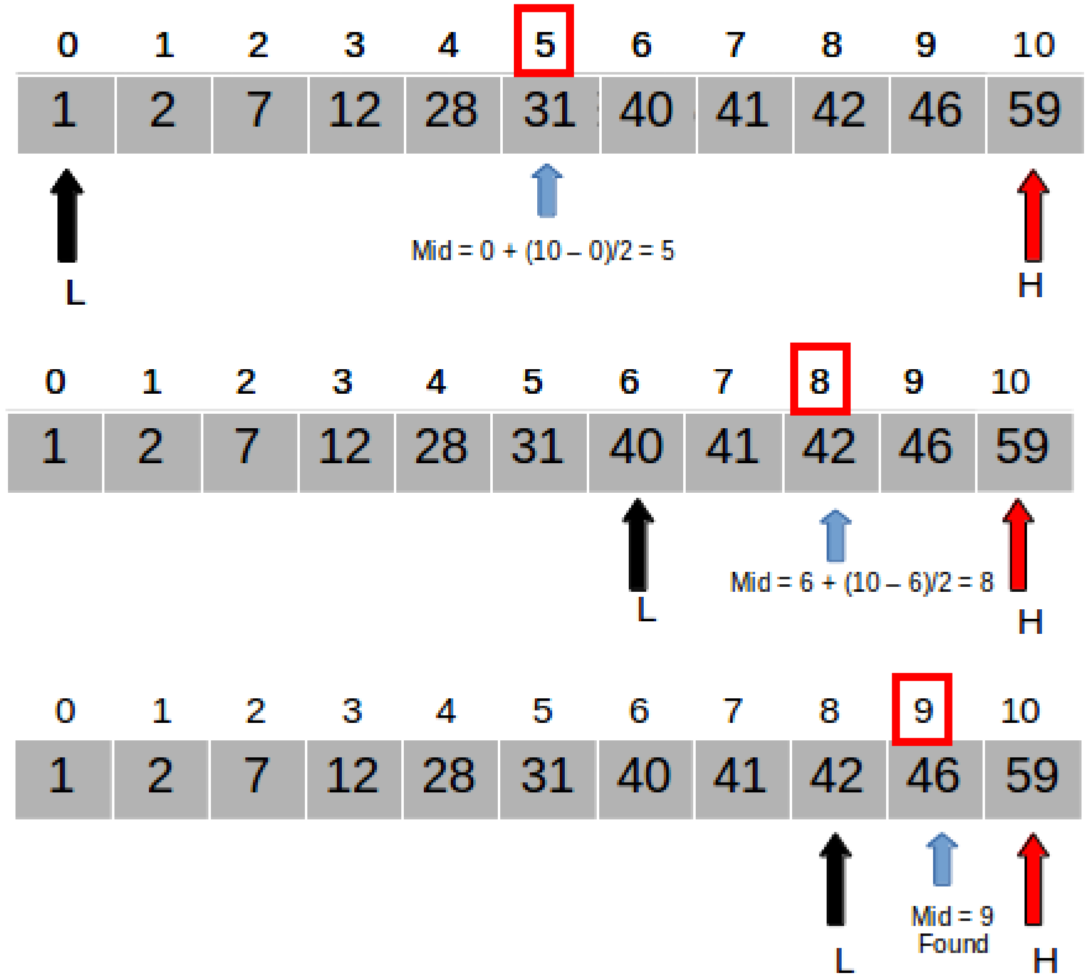
Binary Search



Binary Search



Binary Search

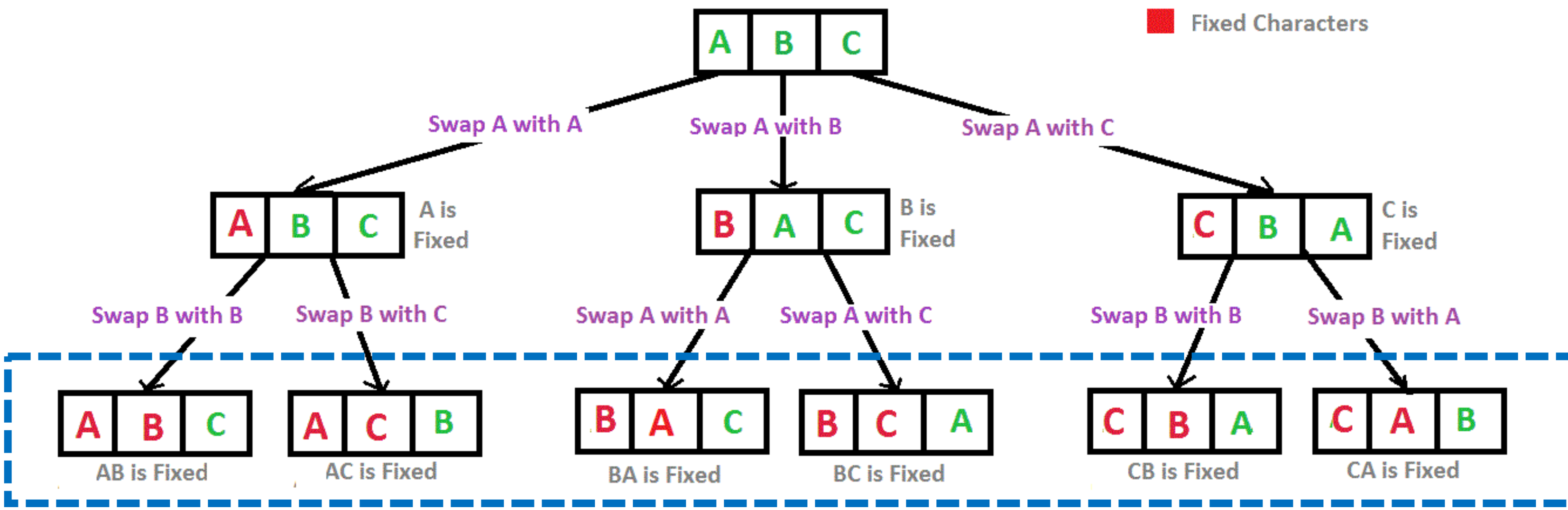


We had to check only **three** elements to find the **46**

View Lab 5 Slow Anagrams

```
template <typename T>
string Concat(vector<T>* set)
{
    string c{};
    for (const auto& item : *set)
        c += item;
    return c;
}

template <typename T>
void Swap(vector<T>* set, int a, int b)
{
    T tmp = set->at(a);
    set->at(a) = set->at(b);
    set->at(b) = tmp;
}

template <typename T>
void Permute(vector<T>* set, int level)
{
    // Heap's Algorithm
    if (level == 0) {
        // At this point, set contains a new permutation
        words.push_back(Concat(set));
    } else {
        for (int i{ 0 }; i < level; ++i) {
            Permute(set, level - 1);
            Swap(set, level % 2 == 1 ? 0 : i, level - 1);
        }
    }
}
```

Permutations



Recursion Tree for Permutations of String "ABC"

Permutation

Permutation of Full Set

$$ {}_nP_k = \frac{n!}{(n-k)!} $$

Size of Set

\# Selected

Permutation of Left Behind Set

$$ {}_3P_3 = \frac{3!}{(3-3)!} = \frac{3!}{0!} = \frac{3 \times 2 \times 1}{1} \boxed{= 6} $$

Permutations

Permutations by interchanges

By B. R. Heap

Methods for obtaining all possible permutations of a number of objects, in which each permutation differs from its predecessor only by the interchange of two of the objects, are discussed. Details of two programs which produce these permutations are given, one allowing a specified position to be filled by each of the objects in a predetermined order, the other needing the minimum of storage space in a computer.

In programs of a combinatorial nature, it is often required to produce all possible permutations of N objects. Many methods can be used for this purpose and a general review of them has been given by D. H. Lehmer in *Proceedings of Symposia in Applied Mathematics* (American Mathematical Society), Vol. 10, p. 179. In this note we shall describe methods for obtaining the permutations in which each permutation is obtained from its predecessor by means of the interchange of two of the objects. Thus $(N-2)$ of the N objects are undisturbed in going from one permutation to the next.

We shall consider values of N up to $N = 12$, since the

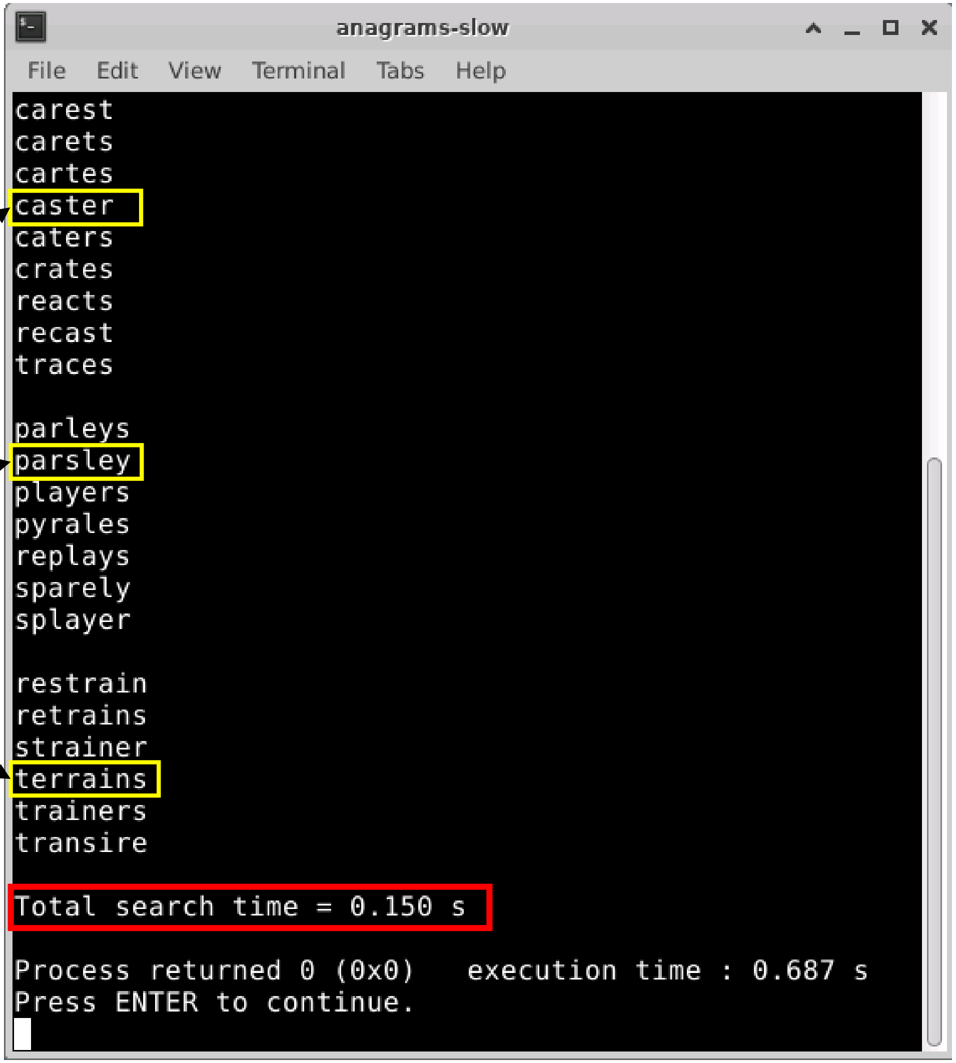
of the first $(n-1)$ objects and again permute the first $(n-1)$ objects. Again interchange the nth object with one of the first $(n-1)$ objects, making sure that this object has not previously occupied the nth cell. Now repeat the process until each of the objects has filled the nth position while the other $(n-1)$ have been permuted, and clearly all $n!$ permutations have been found. Finally, it is clear that two objects can be permuted by a simple interchange, and so N objects can be so permuted. To achieve this one only needs to specify a total of

$$1 + 2 + 3 + \ldots + (N-1) = \tfrac{1}{2}N(N-1)$$

1963

Run Lab 5 – Slow Anagrams

- **Stop**
- **Least**
- **Traces**
- **Players**
- **Restrain**



Anagrams

- stop = post, pots, spot, tops
- least = slate, stale, steal, tales
- traces = carets, caster, caters, crates, reacts, recast
- players = parsley, parleys, replays, sparely
- restrain = retrains, strainer, terrains, trainers
- mastering = ???
- supersonic = ???

10 letters $\rightarrow 10! = 3{,}628{,}800$ **permutations!**

Novel Approach - Think in reverse! First, **SORT** each word in the given dictionary by **letter order**, then **SEARCH** for all words having matching letter orders – those words must be valid anagrams!

A way of finding anagrams that much faster than by trying every permutation!

	Word in Letter Order							
POSSUM	M	O	P	S	S	U		MOPSSU
POST	O	P	S	T				OPST
POSTAGE	A	E	G	O	P	S	T	AEGOPST
POTION	I	N	O	O	P	T		INOOPT
POTS	O	P	S	T				OPST
POUCH	C	H	O	P	U			CHOPU
SPOT	O	P	S	T				OPST
SPOUSE	E	O	P	S	S	U		EOPSSU
STOP	O	P	S	T				OPST
TOPICS	C	I	O	P	S	T		CIOPST
TOPS	O	P	S	T				OPST
TORCH	C	H	O	R	T			CHORT

Letter Order	Actual Word
MOPSSU	POSSUM
OPST	POST
AEGOPST	POSTAGE
INOOPT	POTION
OPST	POTS
CHOPU	POUCH
OPST	SPOT
EOPSSU	SPOUSE
OPST	STOP
CIOPST	TOPICS
OPST	TOPS
CHORT	TORCH

Sort list by the **first** column

AEGOPST	POSTAGE
CHOPU	POUCH
CHORT	TORCH
CIOPST	TOPICS
EOPSSU	SPOUSE
INOOPT	POTION
MOPSSU	POSSUM
OPST	POST
OPST	POTS
OPST	SPOT
OPST	STOP
OPST	TOPS

Matching word orders are all **anagrams**

Open Lab 6 – Fast Anagrams

main.cpp

```cpp
#include "stdafx.h"

using namespace std;

class Anagram
{
public:
    Anagram(string word);
    string word;
    string letters;
};

Anagram::Anagram(string word)
{
    boost::to_lower(word);
    this->word = word;
    sort(word.begin(), word.end());
    this->letters = word;
}

auto compare_lambda = []
                      (const Anagram& a, const Anagram& b) -> bool
                      {
                          return a.letters < b.letters;
                      };

vector<string> phrases
```

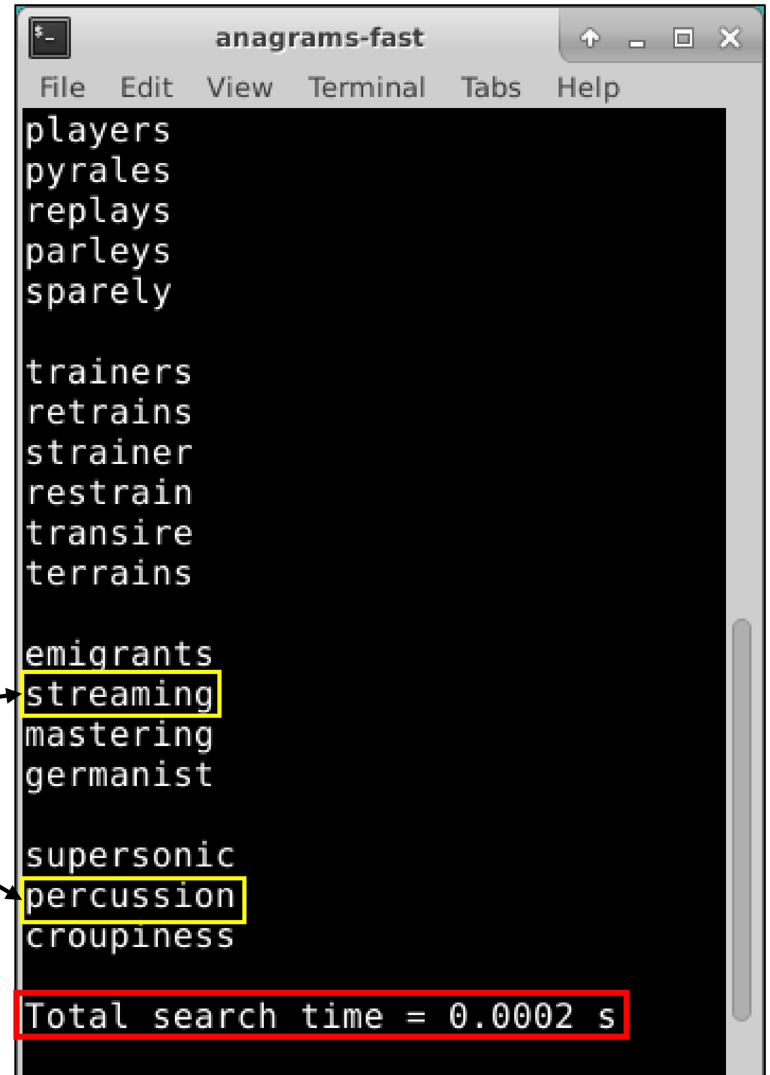
A **lambda** expression is a *closure*

```cpp
int main()
{
    // Load dictionary into the anagrams vector
    string line;
    ifstream inputFile("english_dictionary.txt");
    while (getline(inputFile, line))
    {
        boost::trim(line);
        if (line.length() > 0 )
            anagrams.push_back(Anagram(line));
    }

    // Sort the anagrams by their sorted letters
    sort(anagrams.begin(), anagrams.end(),compare_lambda)

    // Start a timer
    boost::timer timer;

    for (const auto& phrase : phrases)
    {
        Anagram input{ phrase };

        // Find *first* word in dictionary that has sorted letters
        // matching the current phrase also sorted by letters
        auto lower = lower_bound(anagrams.begin(),
                                 anagrams.end(), input, compare_lambda);

        // Find *last* word in dictionary that has sorted letters
        // matching the current phrase also sorted by letters
        auto upper = upper_bound(lower, anagrams.end(),
                                 input,compare_lambda);

        // Display all dictionary words matching the phrase's anagram
        for(auto& a{lower}; a < upper; ++a)
            cout << a->word << endl;

        cout << endl;
    }

    cout << "Total search time = "
         << fixed << setprecision(4)
         << timer.elapsed() << " s" << endl;

    return 0;
}
```

View Lab 6 **Fast Anagrams**

Run Lab 6 – Fast Anagrams

- Stop
- Least
- Traces
- Players
- Restrain
- Mastering
- Supersonic



Slow vs. Fast Anagrams

- The Slow Anagram algorithm took **150** ms while the Fast Anagram algorithm took **0.2ms** – that is a **750X speedup** despite including **3 additional long words** in the search!
  - Even **binary searching** the dictionary cannot overcome the penalty of enumerating permutations which **could never be valid English words**
  - Wasted effort is the inherent problem with the Slow Anagram
- Don't expand the search space by testing **unconstrained permutations** – that leads to **combinational explosion**
  - The ***dictionary*** naturally constrains the **search space**
  - Mine the "answer key" for *any* information it possibly contains
  - That which constrains you often has hidden value

Open Lab 7 **Compound Anagrams**

```
class Anagram2
{
public:
    Anagram2(string word);
    Anagram2(string word1, string word2);
    string word1;
    string word2;
    string letters;
};

Anagram2::Anagram2(string word)
{
    sort(word.begin(), word.end());
    this->letters = word;
}

Anagram2::Anagram2(string word1, string word2)
{
    this->word1 = word1;
    this->word2 = word2;
    string word = word1 + word2;
    sort(word.begin(), word.end());
    this->letters = word;
}

auto compare_lambda = [](const Anagram2& a, const Anagram2& b) ->bool {
    return a.letters < b.letters; };

bool contained(string a, string b)
{
    // Is a fully & uniquely contined in b?
    if (a.length() > b.length())
        return false;

    for (size_t i{}; i < a.length(); i++)
    {
        auto pos = b.find(a[i], i);
        if (pos == string::npos)
            return false;
        b[pos] = ' ';
    }

    return true;
}
```

moor (room) ≈ **dimoorrty** (dormitory)

What **two** smaller words can be made out of the letters of just **one** word?

View Lab 7 **Compound Anagrams**

```
// Read in the dictionary file
ifstream inputFile("english_dictionary.txt");
string line;
while (getline(inputFile, line)) {
    boost::trim(line);
    if (line.length() > 0) {
        Anagram2 word(line);
        // Only add words from dictionary that could
        // possibly be in the anagram of the given phrase
        if (contained(word.letters, input.letters))
            dictionary.push_back(line);
    }
}

// Create compound anagram from every
// successive two words in the dictionary
for (size_t i{}; i < dictionary.size() - 1; i++)
    for (size_t j{ i + 1 }; j < dictionary.size(); j++)
        anagrams.push_back(
            Anagram2(dictionary.at(i),
                dictionary.at(j)));

// Sort the anagrams by their sorted letters
sort(anagrams.begin(), anagrams.end(), compare_lambda);

// Find *first* word in dictionary that has sorted letters
// matching the current phrase also sorted by letters
auto lower = lower_bound(anagrams.begin(), anagrams.end(),
    input, compare_lambda);

// Find *last* word in dictionary that has sorted letters
// matching the current phrase also sorted by letters
auto upper = upper_bound(lower, anagrams.end(),
    input, compare_lambda);

// Create a vector concatenating both words of each anagram
vector<string> phrases;
for (auto& a = lower; a < upper; a++)
    phrases.push_back(a->word1 + " " + a->word2);

// Sort & display the vector of the compound anagrams
sort(phrases.begin(), phrases.end());
for (auto& s : phrases)
    cout << s << endl;
```

dimoorrty (dormitory) = dimoorrty (dirty room)

Run Lab 7 - Compound Anagrams

anagrams-compound

File Edit View Terminal Tabs Help

```
Dormitory
dirt moory
dirt roomy
dirty moor
dirty moro
dirty room
dom riotry
domy trior
dormy riot
dormy roit
dormy tiro
dormy trio
dorty miro
dorty mori
dory timor
dryrot imo
dryrot moi
ir torydom
mird rooty
mod riotry
moody tirr
motor yird
motory rid
moy torrid
myoid torr
ri torydom
torrid yom

Process returned 0 (0x0)   execution time : 0.262 s
Press ENTER to continue.
```

Edit Lab 7 - Compound Anagrams

Uncomment the other lines to reveal lurking compound anagrams.. ☺

```
vector<string> dictionary;
vector<Anagram2> anagrams;

int main()
{
    string phrase{ "Dormitory" };
    //string phrase{ "Software" };
    //string phrase{ "Mother-In-Law" };

```

Run Lab 7 - Compound Anagrams

```cpp
int main()
{
    //string phrase{ "Dormitory" };
    string phrase{ "Software" };
    //string phrase{ "Mother-In-Law" };
```

```
anagrams-compound
File  Edit  View  Terminal  Tabs  Help
of waters
ofer sawt
ofer staw
ofer swat
ofer taws
ofer twas
ofer wast
ofer wats
oft resaw
oft sawer
oft seraw
oft sware
oft swear
oft wares
oft warse
oft wears
ofter saw
ofter swa
ofter was
ora wefts
ore wafts
ores waft
orf sweat
orf tawse
```

```cpp
int main()
{
    //string phrase{ "Dormitory" };
    //string phrase{ "Software" };
    string phrase{ "Mother-In-Law" };
```

```
anagrams-compound
File  Edit  View  Terminal  Tabs  Help
hirta lowmen
hitler woman
hitman lower
hitman owler
hitman rowel
hl tirewoman
hmo wrentail
holeman writ
holm tawnier
holm tinware
holm wintera
holman twire
holman write
holt wireman
homer talwin
hominal wert
horim tawnle
horme talwin
horntail mew
horntail wem
hotelman wir
hotline warm
how terminal
how tramline
```

NOTICE: You didn't hear that from me! ☺

Who knew?

Listen = Silent

The Morse Code = Here come dots

The meaning of life = The fine game of nil

Statue of Liberty = Built to stay free

Now you know…

- C++ **strings** are essentially a vector of type **char**
  - **ASCII** is an international standard for encoding most Western language characters into a <u>single</u> byte
- Character histograms enable **frequency analysis**
  - Caesar-Shift ciphers are <u>not</u> very secure
  - All **monoalphabetic substitution ciphers** can be broken with **bigram analysis**
  - Using “brute force” to crack a cipher is often **intractable** – get statistics on your side to help you out!
- **Heap’s Algorithm** will generate all **permutations** of a set
- Consider problems *backwards*: don’t expand search spaces