A Topics Based Introduction to Computer Science

Instructor: Benny Chor

Teaching Assistant (and Python Guru): Rani Hod

School of Computer Science
Tel-Aviv University
Fall Semester, 2011/12

as of May 16, 2012, on youtube

http://www.youtube.com/watch?v=Q7bDLG-qTkc&list=PLF6EE69996E72E02C&feature=view_all

Course site: http://tau-cs1001-py.wikidot.com
Intro to CS at TAU has been taught following closely the MIT 6.001 course “Structure and Interpretation of Computer Programs”, with Scheme as the programming language.

The Scheme based course, 6.001, is being phased out in recent years at many places (even at MIT).

Two years ago, following a meeting with 10-12 faculty members, the curriculum committee recommended a new course be designed and run as a pilot.

After two test runs, approved as TAU Intro CS course.
Design Goals, Alternatives, and Decisions, CS1001.py

- Major Alternatives:
  - An intro to programming course vs. an intro to Computer Science course.
Design Goals, Alternatives, and Decisions, CS1001.py

- Major Alternatives:
  - An intro to programming course vs. an intro to Computer Science course.

- Choices and Decisions taken:
  - A topics based introduction to Computer Science.
  - Introducing a wide spectrum of problems (including some not covered at all in undergraduate studies).
  - Restricting to concepts that can be explained in reasonably simple ways for first year students (sometimes giving simplified views).
More Design Goals, Alternatives, and Decisions, CS1001.py

- All topics taught include actual implementation as well as some theoretical background.
- Often demonstrate solutions by running code in class.
- Time complexity examined using both $O$ notation and by actually timing the executions.

- Additional, important goals: the course should be exciting and fun, yet demanding.
More Design Goals, Alternatives, and Decisions, CS1001.py

- All topics taught include actual implementation as well as some theoretical background.
- Often demonstrate solutions by running code in class.
- Time complexity examined using both $O$ notation and by actually timing the executions.

- Additional, important goals: the course should be exciting and fun, yet demanding.

- Chose Python as the programming language “vehicle”. This is an important decision, but not a crucial one.
- Python is interpreted (not compiled). It supports easily developing and modifying code. The price to pay is lower performance – deemed less critical in this context.
Planned Dual Learning Outcome, CS1001.py

- Students are expected to get a broad view of central subjects in Computer Science.

- Students are expected to acquire a good knowledge and proficiency of programming and understanding short programs.
• Students are expected to get a **broad view** of central subjects in Computer Science.

• Students are expected to acquire a good knowledge and **proficiency of programming** and understanding short programs.

• Both goals are **integrally interleaved**: Each topic is accompanied by programming examples and assignments.
Course Structure (from Lecture 1 of New Course)

- We will “sample” 10–14 different topics of interest from a fairly wide range in Computer Science.
- This should hopefully expose you to some of the beautiful topics and ideas in the field.
- Naturally we will not get into any of them in great depth.
- We leave this to the required and elective courses you will take later in your studies.
- Each topic will be accompanied by programming examples (given by us) and tasks (to be done by you).
- Lectures and recitations are coordinated and both form integral parts of the course.
This is not a programming course.

Yet, you will learn and use a specific programming language – Python.

And basic ideas in programming like iteration, control structure, recursion, procedures, objects, basic data structures, etc. etc.

Python is a relatively new programming language, gaining popularity in many applications.

Those of you who heard about Scheme and were hoping to learn it here, should take the other intro CS class.
One More Thing this Course is Not About

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

E.W. Dijkstra
Course Topics
(What we did cover – not a complete list)

- Python programming basics (2–3 meetings)
- Bits, bytes, and representation of numbers in the computer.
- Huge integers, with applications to primality testing and public key cryptography. GCD.
- Representing and manipulating images.
- Text compression (Huffman, Codebook, Ziv-Lempel).
- Simple error correction codes (Hamming, Hadamard, others).
- String matching.
- Hashing and hash functions, including Cuckoo hashing.
- Comparing and aligning sequences, with applications in computational biology and in computer music.
- Numerical computations (Newton–Raphson root finding).
- Secret sharing and visual secret sharing.
- The infrastructure of the net (guest lecture by Eyal Dagan).
More Course Topics  
(What we did cover – not a complete list either)

- Lists slicing and list comprehension. Immutable and mutable objects.
- Recursion and memoization. Higher order functions.
- Defining functions using `lambda` expressions.
- Sets and dictionaries; Iterators and generators.
- Object oriented programming (classes and methods).
- Random number generation; Timing operations; File I/O.
- Faster matrix multiplication (Strassen).
- Mandelbrot’s fractal.
- The hare and turtle method. Application to integer factoring.
Samples from Four Classes

- Large integer arithmetic and primality testing.
- Hashing and Cuckoo hashing.
- Image representation and manipulation.
- Text compression.
Example 1: Large Integer Arithmetic and Primality Testing (week 3 of 13)
Integer Exponentiation: Naive Method

How do we compute $a^b$, where $a, b \geq 2$ are both integers?

The naive method: Compute successive powers $a, a^2, a^3, \ldots, a^b$.

Starting with $a$, this takes $b - 1$ multiplications, which is exponential in the length of $b$.

For example, if $b$ is 20 bits long, say $b = 2^{20} - 17$, such procedure takes $b = 2^{20} - 17 = 1048559$ multiplications.

If $b$ is 1000 bits long, say $b = 2^{1000} - 17$, such procedure takes $b = 2^{1000} - 17$ multiplications.
Integer Exponentiation: Naive Method

How do we compute $a^b$, where $a, b \geq 2$ are both integers?

The naive method: Compute successive powers $a, a^2, a^3, \ldots, a^b$.

Starting with $a$, this takes $b - 1$ multiplications, which is exponential in the length of $b$.

For example, if $b$ is 20 bits long, say $b = 2^{20} - 17$, such procedure takes $b = 2^{20} - 17 = 1048559$ multiplications.

If $b$ is 1000 bits long, say $b = 2^{1000} - 17$, such procedure takes $b = 2^{1000} - 17$ multiplications.

In decimal, $2^{1000} - 17$ is

10715086071862673209484250490600018105614048117055336074437503883703510511249361224931983788156958581275946729175531468251871452856923140435984577574698574803934567774824230985421074605062371141877954182153046474983581941267398767559165543946077062914571196477686542167660429831652624386837205668069359. 

A 1000 bits long input is not very large. Yet such computation is completely infeasible.
Efficient Integer Exponentiation: Iterated Squaring

Let $\ell = \lceil \log_2 b \rceil$. The so called iterated squaring algorithm computes $a^b$ using just $2\ell$ multiplications. Instead of computing all successive powers of $a$, namely $\{a, a^2, a^3, \ldots, a^b\}$, we compute just successive powers of two powers of $a$, namely $\{a^1, a^2, a^4, a^8, \ldots, a^{2\ell}\}$.

To accomplish this, observe that $a^{2i+1} = (a^{2i})^2$. So by starting with $a^1 = a$, and iterated squaring the last outcome, $\ell - 1$ times. Note that squaring is just one multiplication.
Wait a Minute

Using iterated squaring, we can compute $a^b$ for any $a$ and, say, $b = 2^{100} - 17 (= 1267650600228229401496703205359)$. This will take at most 200 multiplications, a piece of cake even for an old, faltering machine.
Wait a Minute

Using iterated squaring, we can compute $a^b$ for any $a$ and, say, $b = 2^{100} - 17 \ (= 1267650600228229401496703205359)$. This will take at most 200 multiplications, a piece of cake even for an old, faltering machine.

A piece of cake? Really? 200 multiplications of what size numbers?

For any integer $a \neq 0, 1$, the result of exponentiation above is over $2^{99}$ bits long. No machine could generate, manipulate, or store such huge numbers.

Can anything be done? Not really!

Unless you are ready to consider a closely related problem:

**Modular exponentiation**: Compute $a^b \mod c$, where $a, b, c \geq 2$ are all integers. This is the remainder of $a^b$ when divided by $c$. In Python, this can be expressed as `pow(a, b, c)`.
Fermat’s Little Theorem

Let $p$ be a prime number, and $a$ any integer in the range $1 \leq a \leq p - 1$.

Then $a^{p-1} \equiv 1 \pmod{p}$. 
By Fermat’s little theorem, if $p$ is a prime and $a$ is in the range $1 \leq a \leq p - 1$, then $a^{p-1} \equiv 1 \pmod{p}$.

Suppose that we are given an integer, $m$, and for some $a$ in in the range $2 \leq a \leq m - 1$, $a^{m-1} \not\equiv 1 \pmod{m}$.

Such $a$ supplies a concrete evidence that $m$ is composite (but says nothing about $m$’s factorization).
Fermat Test: Example

Let us show that the following a 164 digits integer, \( m \), is composite. We will use Fermat test, employing the built ins \texttt{pow} function.

```python
>>> m=57586096570152913699974892898380567793532123114264532903689671329
43152103259505773547621272182134183706006357515644099320875282421708540
9959745236008778839218983091
>>> a=65
>>> \texttt{pow(a,m-1,m)}
28361384576084316965644957136741933367754516545598710311795971496746369
83813383438165679144073738154035607602371547067233363944692503612270610
9766372616458933005882 # does not look like 1 to me
```
Let us show that the following a 164 digits integer, \( m \), is composite. We will use Fermat test, employing the built ins `pow` function.

\[ m=57586096570152913699974892898380567793532123114264532903689671329 \\
43152103259505773547621272182134183706006357515644099320875282421708540 \\
9959745236008778839218983091 \]

\[ a=65 \]

\[ \text{pow}(a,m-1,m) \]

\[
28361384576084316965644957136741933367754516545598710311795971496746369 \\
83813383438165679144073738154035607602371547067233363944692503612270610 \\
9766372616458933005882 \quad \# \text{does not look like 1 to me} \]

This proof gives no clue on \( m \)’s factorization (but I just happened to bring the factorization along with me, tightly packed in my backpack:

\[ m = (2^{271} + 855)(2^{273} + 5) \].
Randomized Primality Testing

- The input is an integer $m$ with $n$ bits ($2^{n-1} < m < 2^n$)
- Repeat 100 times
  - Pick $a$ in the range $1 \leq a \leq m - 1$ at random and independently.
  - Check if $a$ is a witness ($a^{m-1} \neq 1 \mod m$) (Fermat test for $a, m$).
- If one or more $a$ is a witness, output "$m$ is composite".
- If no witness found, output "$m$ is prime".

Remark: This idea, which we term Fermat primality test, is based upon seminal works of Solovay and Strassen in 1977, and Miller and Rabin, in 1980.
Randomized Primality Testing

- The input is an integer \( m \) with \( n \) bits (\( 2^{n-1} < m < 2^n \))
- Repeat 100 times
  - Pick \( a \) in the range \( 1 \leq a \leq m - 1 \) at random and independently.
  - Check if \( a \) is a witness (\( a^{m-1} \neq 1 \mod m \)) (Fermat test for \( a, m \)).
- If one or more \( a \) is a witness, output “\( m \) is composite”.
- If no witness found, output “\( m \) is prime”.

Remark: This idea, which we term Fermat primality test, is based upon seminal works of Solovay and Strassen in 1977, and Miller and Rabin, in 1980.
Properties of Fermat Primality Testing

- **Randomized**: uses coin flips to pick the $a$’s.
- Run time is polynomial in $n$, the length of $m$.
- If $m$ is prime, the algorithm always outputs “$m$ is prime”.


Properties of Fermat Primality Testing

- **Randomized**: uses coin flips to pick the $a$’s.
- Run time is polynomial in $n$, the length of $m$.
- If $m$ is **prime**, the algorithm always outputs “$m$ is prime”.

- If $m$ is **composite**, the algorithm may err and outputs “$m$ is prime”.
- Miller-Rabin showed that if $m$ is composite, then at least $\frac{3}{4}$ of all $a \in \{1, \ldots, m - 1\}$ are witnesses.
- To err, **all** random choices of $a$’s should yield non-witnesses. Therefore,

  \[
  \text{Probability of error} < \left(\frac{1}{4}\right)^{100} \ll 1 .
  \]

- Note: With **much higher probability** the roof will collapse over your heads as you read this line, an atomic bomb will get off within a 1000 miles radius (maybe not such a great example in November 2011 July 2012), etc. etc.
Primality Testing: Simple Python Code

```python
def fermat_test(m):
    """ probabilistic test for m’s compositeness """
    for i in range(0,100):
        a = random.randint(1,m-1)  # randomly chosen int in [1..m-1]
        if pow(a,m-1,m) != 1:  # a is a witness
            print(m," is composite")
            break
    else:  # else is for the termination of for loop
        print(m," is prime")
```

Let us test this on some large numbers:

```python
>>> import random # package for random numbers
>>> fermat_test(3**100+102)
515377520732011331036461129765621272702107522103 is composite
>>> fermat_test(5**100+126)
7888609052210118054117285652827862296732064351090230047702789306640751 is prime
>>> fermat_test(7**80+120+22)
40536215597144386832065866109016673800875222251012083746192454448143 is composite
```
def fermat_test(m):
    """ probabilistic test for m’s compositeness ""
    for i in range(0,100):
        a = random.randint(1,m-1)  # randomly chosen int in [1..m-1]
        if pow(a,m-1,m) != 1:  # a is a witness
            print(m," is composite")
            break
    else:  # else is for the termination of for loop
        print(m," is prime")

Let us test this on some large numbers:

>>> import random  # package for random numbers
>>> fermat_test(3**100+102)
515377520732011331036461129765621272702107522103 is composite
>>> fermat_test(5**100+126)
7888609052210118054117285652827862296732064351090230047702789306640751 is prime
>>> fermat_test(7**80+120+22)
40536215597144386832065866109016673800875222251012083746192454448143 is composite
Example 2: Hash Functions (week 7 of 13)

- The dictionary problem: Insert, delete, find.
- Using hash functions and hash table to solve the dictionary problem in \( O(1) \) time per operation on average.
- Resolving collisions using chaining.
- Measuring maximum load experimentally: Small and large cases.
- Employing hashing to solve string problems on very long strings (e.g. proteomes and/or genomes).
Example 2: Hash Functions (week 7 of 13)

- The dictionary problem: Insert, delete, find.
- Using hash functions and hash table to solve the dictionary problem in $O(1)$ time per operation on average.
- Resolving collisions using chaining.
- Measuring maximum load experimentally: Small and large cases.
- Employing hashing to solve string problems on very long strings (e.g., proteomes and/or genomes).

In our setting, there is a dynamic (changing with time) collection of up to $n$ items, which we want to insert to a hash table with $m$ slots. Every item is an object that is identified by a key. For example, items may be instances of our Student class, and the keys are students' names.
Chaining: An Approaches for Dealing with Collisions

In our example of hashing students’ names, we are going to use chaining. We will implement and analyze chaining on small lists and also on much larger examples.
Using Hashing To Solve Large String Problems

A hash function takes a string and returns a fixed-size integer. This is useful for

1. Avoiding collisions
2. Quick lookups

A hash table is a data structure that uses a hash function to map keys to indices

1. Insertion
2. Lookup
3. Deletion

A hash function should be:

1. Deterministic
2. Uniformly distributed

Common hash functions:

1. MurmurHash
2. SHA-256
3. BLAKE2

A Bloom filter is a probabilistic data structure that can be used to:

1. Store large sets
2. Check membership

A Bloom filter can add false positives but never false negatives.

Example:

```python
def read_proteome(filename):
    for protein in open(filename).read().split('
\n'):
        # Process protein
```

לקריאה וקציפת תוצרי הלויור במנוע חיפוש המבנה.
Homework Problem: Longest Common Subsequences in Bacterial Proteomes

- Download proteome sequences of four bacteria (leprosy, cholera, tuberculosis, salmonella).
- Proteomes are half a million to a few million long (letters are mostly amino acids).
- For each pair of proteomes, find longest common subsequence.
Large String Problems: Naïve vs. Hashing Based Approach

Let us look at this task. We are given two strings $S$ and $T$, both of length $n$. We want to find (contiguous) substrings of maximal length that are exactly shared by $S$ and $T$.

We do not know in advance what this maximal length will be. Let us say we are now looking for substrings of length $\ell$.

The naïve method: If $n \approx 10^6$ and $\ell \approx 10^2$, such task will take about $10^{14}$ operations. This is completely infeasible.
Large String Problems: Naïve vs. Hashing Based Approach

Let us look at this task. We are given two strings $S$ and $T$, both of length $n$. We want to find (contiguous) substrings of maximal length that are exactly shared by $S$ and $T$.

We do not know in advance what this maximal length will be. Let us say we are now looking for substrings of length $\ell$.

The naïve method: If $n \approx 10^6$ and $\ell \approx 10^2$, such task will take about $10^{14}$ operations. This is completely infeasible.

A hashing based approach: Hashing a $\ell$ long substring costs $O(\ell)$ operations. The number of substrings is $n - \ell = O(n)$. The expected length of chains encountered in inserting a single element is $O(1)$. So overall, inserting all substrings to the hash table takes $O(n\ell)$ expected number of operations. The finding phase takes $O(n\ell)$ operations on the average. If $n \approx 10^6$ and $\ell \approx 10^2$, such task will take a small constant times $10^8$ operations. This may cause your PC to cry, but is essentially feasible.
Example 3: Digital Images Representation (week 9 of 13)
Any guesses as to what this image is (or is part of)?
Example 3: Digital Images Representation (week 9 of 13)

Any guesses as to what this image is (or is part of)?

```
[100 112 88 ..., 134 145 166]
[ 80 132 134 ..., 130 184 158]
[ 44  51  56 ..., 132 122  9]
 ..., 
[ 14  17  15 ..., 206 204 184]
[ 21  11  12 ..., 203 176 185]
[ 24  13  16 ..., 200 180 182]
```
Grey Level Images

For the sake of simplicity, the remaining of this presentation will deal with greyscale images only. However, what we will do is applicable to color images as well.

To enable representation on bounded precision, digital devices, real numbers expressing grey levels have to be discretized.

A good quality photograph (that is, good by human visual inspection) has about 256 grey level values (8 bits), where 0 represents black, and 255 represents white (not very intuitive, I agree :-).

We remark that in some applications, such as medical imaging, 4096 grey levels (12 bit) are used.
import numpy
from PIL import Image

Albert0 = Image.open("albert1951.jpg")
print (Albert0.size, Albert0.mode)
Albert=Albert0.convert("L")
print (Albert.size, Albert.mode)

A=numpy.asarray(Albert)
T=A[250:310,120:170].copy()
    # a slice of the original
Tongue=Image.fromarray(numpy.uint8(T))

Tongue.show()
Albert.show()
A Grey Level Image and a 60-by-50 Slice Thereof
Tinkering with a Real Image: Example

import numpy
from PIL import Image

im0 = Image.open("lena_original.jpg")
# a 512-by-512 pixel, grey level photo

Lena = im0.convert("L")  # converts to 8 bit black and white
Lena.show()
The Original Lena
Tinkering with a Real Image: Example

C = numpy.asarray(Lena)

for i in range(100):
    for j in range(100):
        C[i,j]=0  # black square at upper left corner

for i in range(200,300):
    for j in range(200,300):
        C[i,j]=128  # grey square at middle

for i in range(412,512):
    for j in range(412,512):
        C[i,j]=255  # white square at lower right corner
LenaBGW = Image.fromarray(numpy.uint8(C))

LenaBGW.show()
The Modified Lena
Additional Simple Modification: Create Negative

def negate(file):
    ''' negate image '''
    im = open_image_as_array(file)
    v, h = im.shape
    im2 = numpy.zeros(im.shape)
    for i in range(v):
        for j in range(h):
            im2[i, j] = 255 - im[i, j]
    show_array_as_image(im2)
Additional Simple Modification: Create Negative

def negate(file):
    ''' negate image '''
    im = open_image_as_array(file)
    v, h = im.shape
    im2 = numpy.zeros(im.shape)
    for i in range(v):
        for j in range(h):
            im2[i, j] = 255 - im[i, j]
    show_array_as_image(im2)
def negate(file):
    ''' negate image '''
    im = open_image_as_array(file)
    v, h = im.shape
    im2 = numpy.zeros(im.shape)
    for i in range(v):
        for j in range(h):
            im2[i, j] = 255 - im[i, j]
    show_array_as_image(im2)
def stretch(file, ratio=1):
    ''' stretch image ratio-fold horizontally'''
    im = open_image_as_array(file)
    v, h = im.shape
    im2 = numpy.zeros((v, h*ratio))

    for i in range(v):
        for j in range(h):
            for k in range(ratio):
                im2[i, ratio*j+k] = im[i,j]
    show_array_as_image(im2)
Additional Simple Modification: Stretch Vertically
Homework Problem: Reconstructing Shredded Images

- A gray level image, 1000-by-1000 pixels, is shredded to approximately 200 vertical stripes.
- These 200 stripes are given as input.
- The task is to reconstruct the original image.
- Scenes taken from TAU campus. First student to identify the location (correctly) gets a bonus.

- Problem inspired by the (much harder) DARPA Shredder Challenge ($50,000 reward, nearly 9000 teams competing, claimed by the team “All Your Shreds Are Belong To U.S.” on December 2011).
Reconstructing Shredded Images: Challenge 18
Reconstructing Shredded Images: Solution 18
Reconstructing Shredded Images: Challenge 20
Reconstructing Shredded Images: Solution 20
Example 4: Text Compression (week 11 of 13)
Representation of Characters: ASCII (Reminder)

The initial encoding scheme for representing characters is the so-called ASCII (American Standard Code for Information Interchange). It has 128 characters (represented by 7 bits). These include 94 printable characters (English letters, numerals, punctuation marks, math operators), space, and 33 invisible control characters (mostly obsolete).

(table from Wikipedia. 8 rows/16 columns represented in hex, e.g. ‘a’ is 0x61, or 97 in decimal representation)
Huffman Compression, High Level View

Huffman code translates the binary representation of a text string to a binary string, which should

- Makes it possible to translate back to the original.
- Has a shorter length (usually, not always).
- Work on a letter by letter basis.

Huffman code assigns short coding to letters occurring frequently, and longer coding to letters occurring infrequently.

This way, the encoding of a typical string will be shorter, since it contains more frequent letters (where we save length) than rare ones (where we pay extra length). (Note that this approach will not work for random strings.)

The rest is just details.

(but, as you surely know, the devil is in the details).
Frequencies of Letters in Natural Languages

The distribution of single letters in any natural languages’ texts is highly non uniform.

We can compute these frequencies by taking a “representative text”, or corpus, and simply count letters. For example, in English, “e” appears approximately in 12.5% of all letters, whereas “z” accounts for just 0.1%. In Hebrew, “Yud” (ord=1497) appears approximately in 11% of all letters (not counting spaces, digits, etc.) while “Za’in” (ord=1494) appears approximately in just 0.8% of all letters.

(As an aside, this observation enables breaking single letter substitution ciphers.)
A completely different approach was proposed by Yaacov Ziv and Abraham Lempel in a seminal 1977 paper ("A Universal Algorithm for Sequential Data Compression", IEEE transactions on Information Theory).

Unlike Huffman, all variants of LZ compression do not assume any knowledge of character distribution. The algorithm finds redundancies in texts using a different strategy.

We will go through this important compression algorithm in detail.
A completely different approach was proposed by Yaacov Ziv and Abraham Lempel in a seminal 1977 paper (“A Universal Algorithm for Sequential Data Compression”, IEEE transactions on Information Theory).

Their algorithm went through several modifications and adjustments. The one used most these days is by Terry Welch, in 1984, and known today as LZW compression.

Unlike Huffman, all variants of LZ compression do not assume any knowledge of character distribution. The algorithm finds redundancies in texts using a different strategy. We will go through this important compression algorithm in detail.
Ziv-Lempel: Riding on Text Repetitions

The basic idea of the Ziv-Lempel algorithm is to “take advantage” of repetitions in order to produce a shorter encoding of the text. Let $T$ be an $n$ character long text. In Python’s spirit, we will think of it as $T[0]T[1]...T[n-1]$.

Suppose we have a $k$ long repetition ($k > 0$) at positions $j, p = j + m$ ($m > 0$): $T[j]T[j+1]...T[j+k-1]=T[p]T[p+1]...T[p+k-1]$.

**Basic Idea:** Instead of coding $T[p]T[p+1]...T[p+k-1]$ character by character, we can fully specify it by identifying the starting point of the first occurrence, $j$, and the length of the repetition, $k$. 
Huffman vs. Ziv Lempel: Basic Difference

Huffman compression is static. It compute frequencies, based on some standard corpus. These frequencies are used to build compression and decompression dictionaries, which are subsequently employed to compress and decompress any future text.

The statistics (or derived dictionaries) are shared by both sides before communication starts.

By way of contrast, Ziv-Lempel compression(s) are adaptive. There is no precomputed statistics. The basic redundancies employed here are repetitions, which are quite frequent in human generated texts.

There is no need to share any data before transmission commences. In a sense, the decompressed text will serve as the basis for subsequent decompression.
Homework Example: Text Compression

- Use implementations of Huffman coding and Ziv–Lempel compression (developed in class).
- Evaluate compression ratio and run time for
  - Yesterday’s NY Times issue.
  - “The Odyssey”, by Homer (∼700 BC).
  - “Romeo and Juliet”, by William Shakespeare (1564-1616).
  - The four bacterial proteomes: leprosy, cholera, tuberculosis, salmonella – author unknown (yet hundreds millions mya).

- Different contexts, e.g. men made vs. “nature made” texts.
- Universality of letter frequencies in spoken language.
We implemented a very naive version of LZ compression. Running it on an online issue of the *Ny Times*, it was compressed to 37% of original (much better than Huffman’s 66%).

We then used Ziv-Lempel to compress the Cholera proteome, and got only 86.8% of its original size. Huffman, on the other hand, resulted in 78.4% of the Cholera proteome size, or 10 percent better than LZ.
The Cholera proteome is (to the best of our knowledge) *not man made*. So some properties common in human generated text, like repetitions, are not very frequent. Thus the Ziv-Lempel compression ratio is not very impressive here.

But, most of the “cholera text” is over the *amino acid alphabet*, which has *just 20 characters*. The vast majority of the characters in the text can thus be encoded using *under 5 bits* on average. This explains why Huffman compressed better here.
And Now For Something Completely Different: Some Statistics, Feedback, and Reflections
Techniques Employed in Teaching the Course

- Mixture of presentations and blackboard work.
- Presentations e-mailed to students the night before (even if contained errors).
- Running code (prepared beforehand) in class.
- Timing executions.
- Comparing efficiency of own implementations to built-in ones (sorting, modular exponentiation, hashing).
- Active forum (contributions by both staff and students).
- Encouraged experimenting, looking for material online, etc.
First Three Pilot Runs: Attendance

- **First run** (spring 2011): Group of 35 pre-selected students (CS-Math and Bioinfo tracks, plus 3 math students).

- **Second run** (fall 2011/12): Group of 106 students. Choosing the course by bidding (Scheme course was run parallel, and had approx. 90 students).
  - Some of these 106 took this as an elective course (e.g. a few Ph.D. students from life Science).

- **Third run** (spring 2012): Group of 84 students. Choosing the course by bidding (Scheme course was run parallel, and has approx. 93 students).
Feedback by Students: Criticizing

- Very hard for students with no programming background – pace is too fast for them.
- Too many topics covered.
- Home assignment load is too large.
- Usage of English in slides and forum.
- Automatic grading mechanism used is not user friendly.
Feedback by Students: Praising

- Large set of topics
- Wide exposure.
- Choice of programming language.
- Staff accessibility in forum.
- Variety of real life problems and homework assignments.
Reflections

- Other contextual approaches to CS101 focus on single contexts, e.g. biology [Dodds, Libeskind-Hadas, Bush 2010], media [Tew, McCracken, Guzdial 2005], robots [Summet et al. 2009], web [Pearce and M. Nakazawa, 2008].
- We cover a much wider variety of topics, but in less depth.
- Course not stable yet.
- Course likely to change with instructors. But core should stabilize.
Reflections

- Other contextual approaches to CS101 focus on single contexts, e.g. biology [Dodds, Libeskind-Hadas, Bush 2010], media [Tew, McCracken, Guzdial 2005], robots [Summet et al. 2009], web [Pearce and M. Nakazawa, 2008].
- We cover a much wider variety of topics, but in less depth.
- Course not stable yet.
- Course likely to change with instructors. But core should stabilize.
- Overall we think this was a very positive experience (and students seem to agree).
Famous Last Words

You have brains in your head.
You have feet in your shoes.
You can steer yourself
any direction you choose.
You’re on your own. And you know what you know.
And YOU are the guy who’ll decide where to go.