

Lecture 8

August 16, 2016

Algorithms for Text Processing



Reminder: start the recording

Announcements (1/2)

- Project 2 due tonight
- Assignment 3 due Thursday before class
- Optional 'self-quiz' posted (formerly known as Assignment 4)
 - Answers reviewed next Tuesday
- Project 3 (Thai FST) due next Tuesday 8/23 at 11:45 p.m.
- Project 4 (Search Human Genome) posted, due Thursday Sept. 1st

Project 3 – Java

- Issues mentioned in last week's lecture
 - Issue 1 (source file encoding) problems:
 `javac -encoding UTF-8 project3.java`
 - Issue 4 (runtime encoding) problems:

```
Scanner input = new Scanner(new File("fsm-input.utf8.txt"), "UTF-8");
```

```
FileWriter fw = new FileWriter("student.html");  
fw.write(s);  
fw.close();
```

```
OutputStreamWriter osw = new OutputStreamWriter(new FileOutputStream("student.html"), "UTF-8");  
osw.write(s);  
osw.close();
```

Announcements (2/2)

- Project 4: now posted
 - Write a prefix trie and locate DNA targets in the complete human genome
 - Due at 11:45 p.m. Thursday Sept 1st
 - We will go over how to do it in today's lecture
- Today: More programming & algorithms
- This Thursday
 - applying Bayes' theorem to part-of-speech tagging



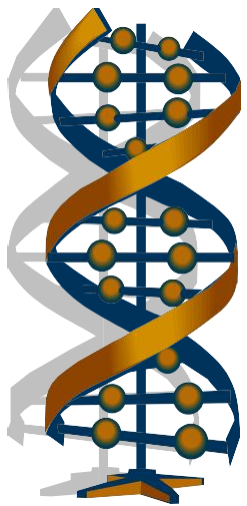
Review: Algorithmic Complexity

- Algorithms can be characterized according to how their use of a resource scales with the size of the input(s).
- We typically look at two resources
 - space (memory consumption)
 - time (execution time)



Technically, these are independent per algorithm, but we often think of a family of algorithms that represent a spectrum of trade-offs between space and time

Example: space/time trade-off



$$f(x) = \begin{cases} 0 & \text{iff } x = 'T' \\ 1 & \text{iff } x = 'C' \\ 2 & \text{iff } x = 'A' \\ 3 & \text{iff } x = 'G' \\ -1 & \text{otherwise} \end{cases}$$

Example: space/time trade-off

4 time steps
0 memory

```
int f(char ch)
{
→   if (ch == 'T')
        return 0;
→   if (ch == 'C')
        return 1;
→   if (ch == 'A')
        return 2;
→   if (ch == 'G')
        return 3;
    return -1;
}
```

1 time step
256b “extra” memory

```
// one-time initialization:
int[] map = Enumerable.Repeat<int>(-1, 256).ToArray();
map['T'] = 0;
map['C'] = 1;
map['A'] = 2;
map['G'] = 3;
// ...

int f(char ch)
{
→   return map[ch];
}
```

generates a sequence of the integer -1 repeated 256 times

renders any sequence to an array

Best/average/worst case

- If an algorithm's performance depends on more than just the size of the input, what is the possible range of complexity for a given input?
- Examples:
 - a **sort** algorithm may perform differently depending on the input ordering
 - intuitively, shouldn't sorting have complexity that's inversely proportional to the **entropy** of the input?
 - Xiaoming Li et al. (2007) *Optimizing Sorting with Machine Learning Algorithms*. IEEE
 - inserting into a **balanced tree** may perform differently depending on the current geometry

Asymptotic notations

- Express resource use in terms of input size
- Discard factors that become irrelevant in the limit

Name	Notation	Expresses
Big-O	$f(n) = O(g(n))$	Upper bound
Big Omega	$f(n) = \Omega(g(n))$	Lower bound
Big Theta	$f(n) = \Theta(g(n))$	Upper and lower bound

- Usually when people say $O(n^2)$ “O of n-squared” etc. they actually mean $\Theta(n^2)$ “Theta of n-squared”
 - because usually, analysis of best case, worst case, and average case must be done separately
 - I will continue this tradition of abuse (i.e. on the next slide)

Complexity Classes

Notation	Name	Example algorithm
$O(k)$	constant	hash table
$O(\log n)$	logarithmic	binary search
$O(n)$	linear	naïve search (best and worst case)
$O(n \log n)$	log-linear	quicksort (best case)
$O(n^2)$	quadratic	naïve sort (best and worst case)
$O(n^3)$	cubic	parsing context-free-grammar (worst case)
$O(n^k)$	polynomial	2-SAT (propositional satisfiability)
$O(k^n)$	exponential	traveling salesman DP
$O(n!)$	factorial	naïve traveling salesman

Complexity scaling

- Consider an algorithm containing 8 million instructions
- State-of-the-art processor 2011 (1.59×10^{11} instructions per second)

$nn = 1111$ → $nn = 333333$

$OO(1)$	$54 \mu\text{s.}$
$OO(\log nn)$	$58 \mu\text{s.}$
$OO(nn)$	$649 \mu\text{s.}$
$OO(nn \log nn)$	$700 \mu\text{s.}$
$OO(nn^2)$	7.8 ms.
$OO(nn^3)$	93 ms.
$OO(nn^{kk}), kk = 8$	23 sec.
$OO(kk^{nn}), kk = 8$	1032 hours
$OO(nn!)$	$2,231,000 \text{ centuries}$

$OO(1)$	$54 \mu\text{s.}$
$OO(\log nn)$	$137 \mu\text{s.}$
$OO(nn)$	19 ms.
$OO(nn \log nn)$	48 ms.
$OO(nn^2)$	6.6 sec.
$OO(nn^3)$	38.6 min.
$OO(nn^{kk})$	$387 \text{ million years}$
$OO(kk^{nn})$	$\text{age of universe} \times 10^{294}$
$OO(nn!)$	$\text{age of universe} \times 10^{718}$

Complexity classes

- Characterize the inherent difficulty of programming problems
- PP : class of decision problems that can be deterministically **solved** in polynomial time
- $NNPP$: class of decision problems that can be deterministically **verified** in polynomial time

Estimating Complexity

- Only the highest-growth term matters
- Constants are ignored

$$nn^2 + 6.02 \times 10^{23}nn$$

$$\rightarrow \text{O}(nn^2)$$

$$6.02 \times 10^{23}nn$$

$$\rightarrow \text{O}(nn)$$

$$5nn^3$$

$$\rightarrow \text{O}(nn^3)$$

Estimating complexity

- What is the time complexity of this program?
- What is its space complexity?

```
Stopwatch stopw = Stopwatch.StartNew();  
int a = 0;  
for (int i = 0; i < n; i++)  
    for (int j = 0; j < n; j++)  
        a++;
```

Output (and running time) of this program, $n=100000$:
32425 (about 32 seconds)

- What about this one?

```
int a = 0;
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        for (int k = 0; k < n; k++)
            a++;
```

- What is the expected output (and running time) of this program ($n=100000$)?

$$\sqrt[3]{32425} \cong 5.8 \times 10^6 \cong 1\text{hr. } 37\text{min.}$$

- What about these?

```
StringBuilder sb = new StringBuilder();  
for (int i = 0; i < n; i++)  
    for (int j = 0; j < n; j++)  
        sb.Append(' ');
```

```
String s = new String(Enumerable.Repeat<Char>(' ', n * n).ToArray());
```

```
int cb = n*n;  
char *p = (char *)malloc(cb);  
_asm mov edi, p    // destination  
_asm mov eax, 0xFFFFFFFF // value  
_asm mov ecx, cb    // number of bytes to store  
_asm rep stosb      // store by byte
```

```
int cb = n*n;  
char *p = (char *)malloc(cb);  
_asm mov edi, p    // destination  
_asm mov eax, 0xFFFFFFFF // value  
_asm mov ecx, cb    // number of bytes to store  
_asm shr ecx, 2     // divide by 4 bytes per dword  
_asm rep stosd      // store by dword
```


Hashing

- Hashing is crucial for indexing very large datasets
- HashSet: $O(1)$ test for membership
- “Dictionary” or “HashMap” or “Associative array” etc. $O(1)$ lookup
 - directly access one item based on the value of another
- All modern programming languages have hashing support
 - you usually won’t need to write your own hash functions

Workhorse data structures

All modern languages have a number of off-the-shelf collection objects. For example C# offers the following, all strongly-typed:

<code>T[] (Array<T>)</code>	Fixed-size, ordered list of objects or values of type T
<code>List<T></code>	Dynamically-sized, ordered list of objects or values of type T
<code>Stack<T></code>	LIFO stack of objects or values of type T
<code>Queue<T></code>	FIFO queue of objects or values of type T
<code>LinkedList<T></code>	Doubly-linked list of objects or values of type T
<code>HashSet<T></code>	Hash table of objects or values of type T
<code>Dictionary<TKey,TValue></code>	Table of objects of type TValue hashed by objects of type TKey
<code>SortedDictionary<TKey,TValue></code>	Sorted table of objects of type TValue hashed by objects of type TKey
<code>ILookup<Tkey,Telement></code>	Multimap (an immutable dictionary that allows duplicate keys)

C# List<T>

```
List<double> radii = new List<double>();
```

```
radii.Add(12.2);
```

```
double q = Math.PI * Math.Pow(radii[0], 2);
```

```
radii.Remove(7.0);           // returns 'false' if not found  
radii.RemoveAt(0);          // remove at index zero
```

```
radii.Sort();
```

C# Dictionary<TKey, TValue>

```
Dictionary<String, int> dict = new Dictionary<String, int>();

dict.Add("number one", 1);    /// add a new value
dict["number four"] = 4;     /// add or change
dict["number four"] = 4444;   /// change existing value
int x = dict["number_one"];   /// retrieve existing value

int y;
if (dict.TryGetValue("number six", out y))
    Console.WriteLine("found it");
else
    Console.WriteLine("did not find it");

if (!dict.ContainsKey("number five"))
    dict.Add("number five", 5);

Console.WriteLine("There are {0} items.", dict.Count);
```

Closures

- Lambda expressions automatically capture local variables that they reference, which are then passed around as part of the lambda variable
 - Caution: languages do this differently with respect to reference (the lambda expression will modify the original value) versus value (the lambda expression has a snapshot of the value)
- This can lead to interesting scoping issues

Lambda expressions

```
// recall Select(ch => ('a' <= ch && ch <= 'z') || ch == '\\' ? ch : ' ');

String s = "Al's 20 fat-ish oxen.";
Func<Char, Char> myfunc = (ch) => ('a' <= ch && ch <= 'z') || ch == '\\' ? ch : ' ';
IEnumerable<Char> iech = s.Select(myfunc);
// iech is now a deferred enumerator for the characters in: " l's    fat ish oxen "

Func<Char, Char> myfunc = (ch) =>
{
    if ('a' <= ch && ch <= 'z')
        return ch;
    if (ch == '\\')
        return ch;
    return ' ';
};

Func<String, int, bool> string_is_longer_than = (s, i) => s.Length > i;

bool b = string_is_longer_than("hello", 3);    // true
```

Closure example

```
int x = 3;
Action a = () =>
{
    Console.WriteLine(x);
};
a();           // prints 3
x = 5;
a();           // prints 5
```

State machine example

```
using System;
using System.Collections.Generic;
using System.Linq;

static class Program
{
    static class MainClass
    {
        enum State { Zero, One, Two };

        static Dictionary<State, Func<Char, State>> machine = new Dictionary<State, Func<Char, State>>
        {
            {
                State.Zero, (ch) => { return State.Two; }
            },
            {
                State.One, (ch) => { return State.One; }
            },
        };

        static void Main(String[] args)
        {
            String s = "the string to parse";
            int i = 0;

            State state = State.Zero;
            while (i < s.Length)
                state = machine[state](s[i++]);

        }
    }
}
```

A dictionary
of lambda
functions

The state machine

Binary trees

- Tree data structures are useful when you'll be doing a lot of inserting and deleting and you want to retain $O(\log n)$ access
 - Don't confuse with **decision tree classifiers**, which you'll get to implement in 572
- In practice, I have found that HashMaps (dictionaries) are better suited for the computational linguistics problems I've encountered
- However, one problem that cannot be solved with everyday data structures is the “all substrings” problem
 - For this, we'll look at a special kind of tree, the prefix trie, or simply “trie”

Trie

- Also known as “prefix trie”
- Pronounced either way: “tree” or “try”
- You will implement a trie for Project 4
- A trie is the most efficient way to search for multiple arbitrary-length substrings in a string

Example

- Identify an arbitrary number of string features in web browser user agent strings

Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:2.0a1pre) Gecko/2008041102 Minefield/4.0a1pre
Mozilla/5.0 (Windows; U; Windows NT 5.1; zh-CN; rv:1.9) Gecko/2008052906 Firefox/3.0
Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1; Trident/4.0; .NET CLR 1.1.4322; .NET CLR 2.0.50727; .NET CLR 3.0.4506.2152; .NET CLR 3.5.30729)
Mozilla/5.0 (compatible; OSS/1.0; Chameleon; Linux) MOT-U9/R6632_G_81.11.29R BER/2.0 Profile/MIDP-2.0 Configuration/CLDC-1.1
SAMSUNG-SGH-E250/1.0 Profile/MIDP-2.0 Configuration/CLDC-1.1 UP.Browser/6.2.3.3.c.1.101 (GUI) MMP/2.0
LG/U8130/v1.0
SonyEricssonC901/R1EA Browser/NetFront/3.4 Profile/MIDP-2.1 Configuration/CLDC-1.1 JavaPlatform/JP-8.4.2
ZTE-V8301/MB6801_V1_Z1_VN_F1BP101 Profile/MIDP-2.0 Configuration/CLDC-1.1 Obigo/Q03C
Mozilla/5.0 (iPhone; U; CPU iPhone OS 3_0 like Mac OS X; en-us) AppleWebKit/420.1 (KHTML, like Gecko) Version/3.0 Mobile/1A542a Safari/419.3
Mozilla/5.0 (iPhone; U; CPU iPhone OS 4_0 like Mac OS X; en-us) AppleWebKit/532.9 (KHTML, like Gecko) Version/4.0.5 Mobile/8A293 Safari/6531.22.7
Mozilla/5.0 (iPod; U; CPU iPhone OS 3_1_1 like Mac OS X; en-us) AppleWebKit/528.18 (KHTML, like Gecko) Mobile/7C145
Mozilla/5.0 (iPad; U; CPU OS 3_2 like Mac OS X; en-us) AppleWebKit/531.21.10 (KHTML, like Gecko) Version/4.0.4 Mobile/7B367 Safari/531.21.10
SIE-S68/36 UP.Browser/7.1.0.e.18 (GUI) MMP/2.0 Profile/MIDP-2.0 Configuration/CLDC-1.1
SIE-EF81/58 UP.Browser/7.0.0.1.181 (GUI) MMP/2.0 Profile/MIDP-2.0 Configuration/CLDC-1.1
9700 Bold: BlackBerry9700/5.0.0.423 Profile/MIDP-2.1 Configuration/CLDC-1.1 VendorID/100

Problem statement

```
String[] targets =  
{ "Mozilla/5.0", "Trident/4.0", "Firefox/3",  
  "iPhone", "Mac OS X", "BlackBerry", "Safari" };  
  
String ua = "Mozilla/5.0 (iPod; U; CPU iPhone OS 3_1_1 like Mac OS X; en-us) AppleWebKit/528.18 (KHTML,  
foreach (String feat in AllSubstrings(ua, targets))  
    Console.WriteLine(feat);  
  
//...  
  
IEnumerable<String> AllSubstrings(String s_in, IEnumerable<String> targets)  
{  
    yield return "hmm...";  
}
```

$O(n^2)$?

Naïve solution

- This is the first solution that should come to mind
 - It is appropriate for small inputs
 - It is easy to understand and trivial to implement
 - Test this on the input set. Maybe it's adequate
 - It provides a baseline for optimization work

```
IEnumerable<String> AllSubstrings(String s_in, IEnumerable<String> targets)
{
    for (int i0 = 0; i0 < s_in.Length; i0++)
        foreach (String t in targets)
            if (i0 + t.Length <= s_in.Length && s_in.Substring(i0, t.Length) == t)
                yield return t;
}
```

- Ok, what if it's not adequate?

Using a trie for the all substrings problem

1. Build a trie from the target strings
 - The targets are all now stored implicitly in the trie, you can release them
2. Scan through the corpus once, using the trie to simultaneously check for all targets

Trie example

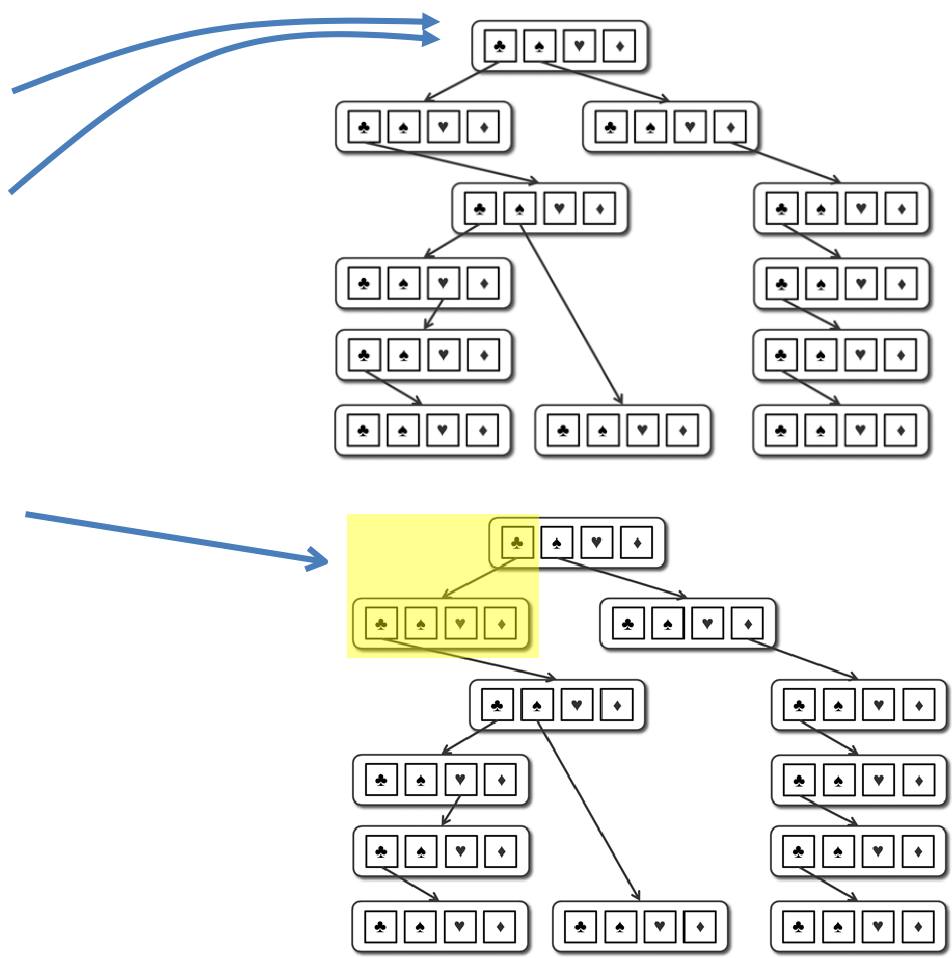
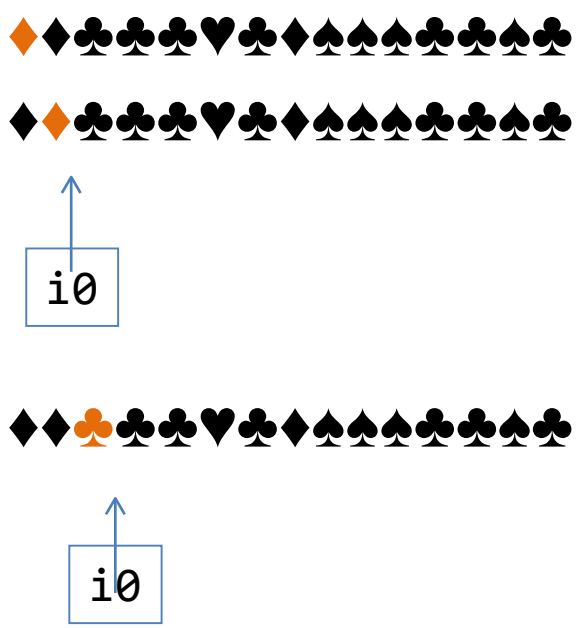
```
String[] targets = { "♣♣♠", "♣♣♣♥♣", "♠♦♣♣♣" };
```

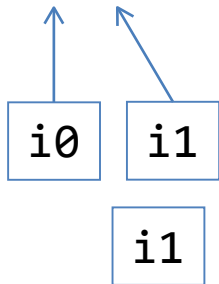
```
String corpus = "♦♦♣♣♣♥♣♦♠♠♠♣♣♠♣";  
//           0           1  
//           012345678901234
```



- target “♣♣♠” found at position 11
- target “♣♣♣♥♣” found at position 2

Trie loaded with the target string

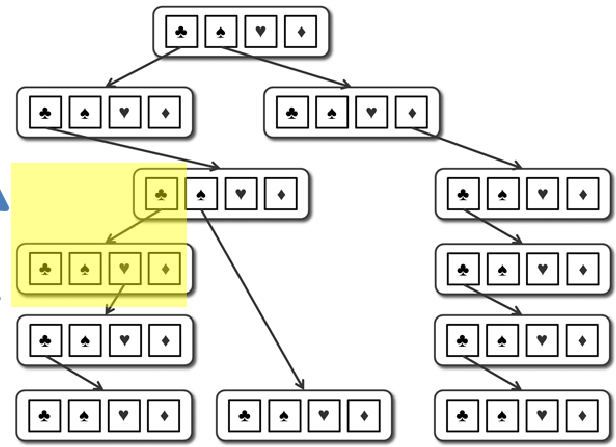
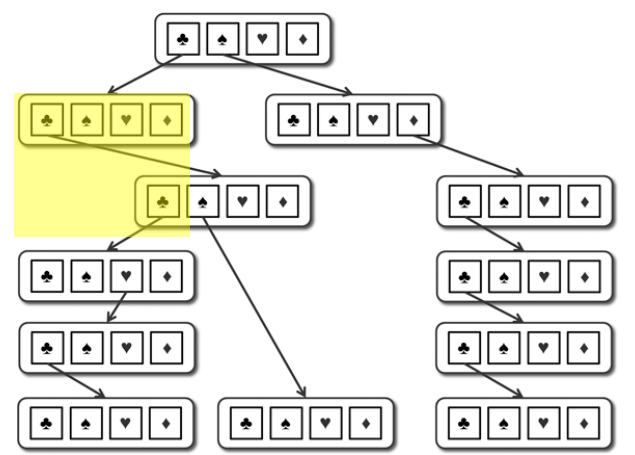




At this point, we're simultaneously checking two active targets, $\clubsuit\clubsuit\spadesuit$ and $\clubsuit\clubsuit\clubsuit\heartsuit\clubsuit$



Found: $\text{corpus.Substring}(i0, i1 - i0 + 1)$
Output and continue from $i0 + 1$



Walking through a trie

- To use a trie for finding all substrings, for each character c_{ii} :
 - attempt to navigate the trie while moving forward $c_{jj+1}...$, returning substrings, as they are found in the trie
 - After reaching the end of the trie, advance to next character c_{ii+1} and repeat
- Recursion is not necessary because we always progress downwards from the trie root
 - You only need a reference to the “current” trie node
 - For this same reason, a trie node does not need to have a back-pointer to its parent node

Trie applications

- A trie guarantees that there is exactly one unique node for every possible prefix of every target
- If several targets share the same prefix, they share those trie nodes

Q: Considering this, what characteristics would a set of target strings have, such that a trie is a good candidate data structure?

Immutable strings

- For modern programming languages, a key performance bottleneck is garbage collection
- Immutable strings provide many benefits, but garbage collector (GC) activity can soar if you're not careful

```
String s = File.ReadAllText("chr1.dna");  
s = s.ToUpper();
```

- If you're going to be modifying a string, use mutable character arrays or special mutable “StringBuilder” objects instead

Project 4

- Due Thursday Sept 1st
- This is a problem where careful selection of data structure (i.e. optimization) matters
- Using grep to search for **one** of the targets took 3min. 42s. on a fast machine
 - Searching for 5,000 targets this way could take about 300 hours.
 - On the same machine, the trie solution takes 4 minutes to find 5,000 targets

Project 4

- Write a trie class. It can be specialized for processing a 4-symbol alphabet, or more general purpose (if you have time)
- Load the trie with the 4,965 target DNA sequences
- In one pass through the human genome corpus (2.8 GB), use the trie to locate all occurrences of any of the target sequences
- Print out the matches: chromosome file number and offset within that file
- The challenge of this project is to correctly build the trie and parse with it. Both of these tasks require keeping a position within a string coordinated with a position in the trie

Next time

- POS tagging
 - If you have access to Jurafsky and Martin 2nd ed., you can review section 5.5 – 5.5.1 (inclusive) before class
- Language modeling

C# Tutorial (continued...)

Interfaces

- `IEnumerable<T>` is one of many system-defined **interfaces** that a class can elect to implement

An **interface** is a named set of zero or more function signatures with no implementation(s)

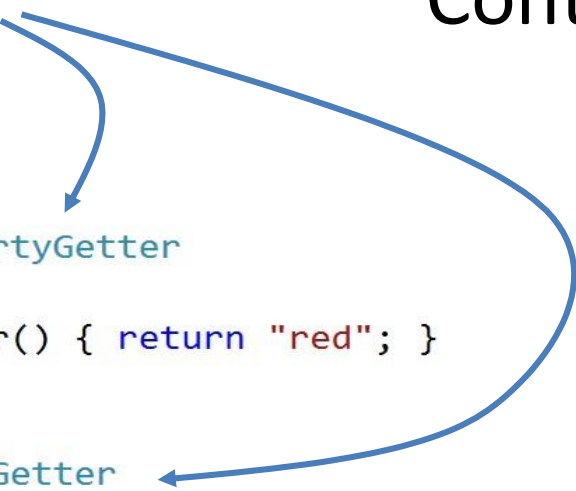
- To implement an interface, a class defines a matching implementation for every function in the interface
- Interfaces are sometimes described as contracts
- You can define and use a reference to an interface just like any other object reference

Contrived Example

```
interface IPropertyGetter
{
    String GetColor();
}

class Strawberry : IPropertyGetter
{
    public String GetColor() { return "red"; }
}

class Ferrari : IPropertyGetter
{
    public String GetColor() { return "yellow"; }
}
```



- This looks like C++ class inheritance
 - yes, but it's more ad-hoc
 - C# classes can have **single inheritance** of other classes, and **multiple inheritance** of interfaces
 - Interfaces can inherit from other interfaces (not shown)

IEnumerable<T>

- This is one of the simplest interfaces defined in the BCL (base class libraries)
- This interface provides just one thing: a way to iterate over elements of type T
- All of the system arrays, collections, dictionaries, hash sets, etc. implement IEnumerable<T>
 - Implementing IEnumerable<T> on your own classes can be very useful, but you don't need to worry about that
 - For now, what's important is that you get to use it, because it's available on all of the system collections

IEnumerator<T>

- **IEnumerable<T>** has only one function, which allows a caller or caller(s) to obtain an *enumerator* object which is able to iterate over elements
 - The actual enumerator object is an object that implements a different interface, called **IEnumerator<T>**
 - This “factory” design allows a caller to initiate and maintain several simultaneous iterations if needed
 - The enumerator object, **IEnumerator<T>** can only:
 - Get the current element
 - Move to the next element
 - Tell you if you’ve reached the end
 - Note: There’s no count
 - ICollection inherits from IEnumerable to provide this

Interfaces as function arguments

- Using interfaces as function arguments allows you to require the absolute minimum functionality the function actually needs
- In this way, the ad-hoc nature of interfaces allows us to comply with the maxim

```
void ProcessSomeStrings(IEnumerable<String> the_strings)
{
    foreach (String s in the_strings)
        Console.WriteLine(s);
}
```

Now, this function is exposing the **weakest (most general) requirement** possible for the processing it has to do. This provides more flexibility to callers since they can choose whatever level of specificity is convenient. The function can be used in the widest possible variety of situations.

Example

```
String[] d1 = { "able", "bodied", "cows", "don't", "eat", "fish" };  
ProcessSomeStrings(d1);
```


```
List<String> d2 = new List<String> { "clifford", "the", "big", "red", "dog" };  
ProcessSomeStrings(d2);
```

```
HashSet<String> d3 = new HashSet<String> { "these", "must", "be", "distinct" };  
ProcessSomeStrings(d3);
```

```
Dictionary<String,int> d4 =  
    new Dictionary<String, int> { { "the", 334596 }, { "in", 153024 } };  
ProcessSomeStrings(d4.Keys);
```

```
void ProcessSomeStrings(IEnumerable<String> the_strings)  
{  
    foreach (String s in the_strings)  
        Console.WriteLine(s);  
}
```

Python users might not be impressed, but the difference is that this is all 100% strongly typed



Iteration is efficient

- That's cool, `IEnumerable<T>` lets a function **not care** about where a sequence of elements is coming from
 - We don't copy the elements around
 - Iterators let us access elements right from their source
- All of those examples iterate over elements that **already exist** somewhere
- Is there a way to iterate over data that's generated on-the-fly, doesn't exist yet, or is never persisted at all?
- Yes!

Iterating over on-the-fly data

```
IEnumerable<String> GetNewsStories(int desired_count)
{
    for (int i = 0; i < desired_count; i++)
        yield return RealtimeNewswireSource.GetLatestStory();
}
```

see next slide

```
// ...
IEnumerable<String> d5 = GetNewsStories(7);
ProcessSomeStrings(d5);
// ...
```

This is exactly the same as before, but this time there's no "collection" of elements sitting anywhere

```
void ProcessSomeStrings(IEnumerable<String> the_strings)
{
    foreach (String s in the_strings)
        Console.WriteLine(s);
}
```

This function doesn't care. In fact, it can't even tell.

yield keyword

- The **yield** keyword makes it easy to define your own custom iterator functions
- Any function that contains the **yield** keyword becomes special
 - It must be declared as returning an `IEnumerable<T>`
 - Deferred execution means that the function's body is not necessarily invoked when you "call" it
 - It must deliver zero or more elements of type `T` using:
`yield return t;`
 - Sometime later, control may continue immediately after this statement to allow you to yield additional elements
 - It may signal the end of the sequence by using:
`yield break;`

Custom iterator function example

```
IEnumerable<String> GetNewsStories(int desired_count)
{
    for (int i = 0; i < desired_count; i++)
        yield return RealtimeNewswireSource.GetLatestStory();
}
```

code from this custom iterator function is *not* executed at this point.

```
// ...
IEnumerable<String> d5 = GetNewsStories(7);
ProcessSomeStrings(d5);
// ...
```

d5 refers to an iterator that “knows how” to get a certain sequence of strings when asked

```
void ProcessSomeStrings(IEnumerable<String> the_strings)
{
    foreach (String s in the_strings)
        Console.WriteLine(s);
}
```

This finally demands the strings, causing our custom iterator function to execute—interleaved with this loop!

LINQ in C#

- Sequences: `IEnumerable<T>`
- Deferred execution
- Type inference
- Strong typing
 - despite the 'var' keyword
 - (C# now has the 'dynamic' keyword, which allows runtime dynamic typing where desired)

LINQ operators

- Filter/Quantify:
Where, ElementAt, First, Last, OfType
- Aggregate:
Count, Any, All, Sum, Min, Max
- Partition/Concatenate:
Take, Skip, Concat
- Project/Generate:
Select, SelectMany, Empty, Range, Repeat
- Set:
Union, Intersect, Except, Distinct
- Sort/Ordering:
OrderBy, ThenBy, OrderByDescending, Reverse
- Convert/Render:
Cast, ToArray, ToList, ToDictionary

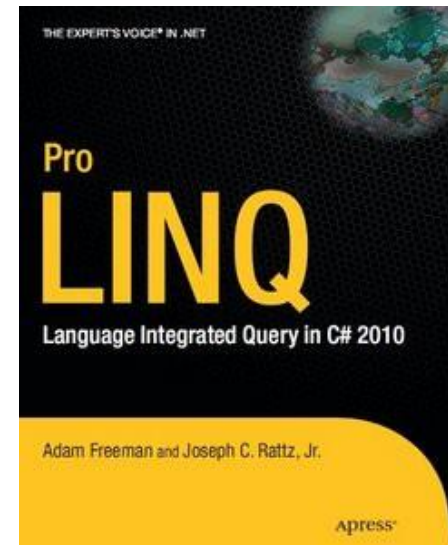
Example

```
Dictionary<String, int> pet_sym_map =  
    File.ReadAllLines("/programming/analytical-grammar/erg-funcs/pet-symbol-key.txt")  
    .Select(s => s.Split(sc7, StringSplitOptions.RemoveEmptyEntries))  
    .Where(rgs => rgs.Length == 3)  
    .Select(rgs => new { id = int.Parse(rgs[0]), sym = rgs[1].Trim().ToLower() })  
    .GroupBy(a => a.sym)  
    .Select(grp => grp.ArgMin(a => a.id))  
    .ToDictionary(a => a.sym, a => a.id);
```

C# LINQ (Language Integrated Query)

- Declarative operations on sequences
- Recommendation:

Joseph C. Rattz, Jr. (2007) *Pro LINQ: Language Integrated Query in C# 2008*. Apress.




Programming Paradigms: Procedural

- Procedural (“imperative”) programming
 - FORTRAN (1954) grew out of hardware assembly languages, which are necessarily procedural
 - We explicitly specify the (synchronous) steps for doing something (i.e. an algorithm)

```
int Factorial(int n)
{
    int f = 1;
    for (int i = n; i > 1; i--)
        f *= i;
    return f;
}
```


Functional Programming

- A type of declarative programming
 -  Constraint-based syntax formalisms such as unification grammars (LFG, HPSG, ...) are also declarative
- Like function definitions in math, the “program” describes asynchronous relationships
- Functions are stateless and should have no side-effects
- Immutable values
 - As a bonus, this really facilitates concurrent programming
- Scheme, Haskell, F#

F# Example

- F# interactive on patas:

```
$ mono /opt/fsharp/bin/fsi.exe --gui-  
      (you must use an ANSI terminal)
```

```
> let rec factorial = function  
    | 0 -> 1  
    | n -> n * factorial(n -  
1);; val factorial : int -> int  
> factorial 5;;  
val it : int =  
120  
> #quit;;
```