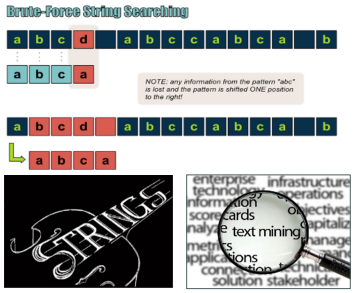
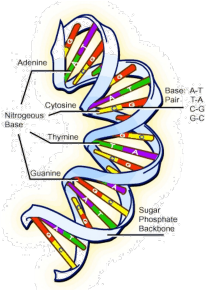
# Lecture 13
# Strings and Sequences




EECS 281: Data Structures & Algorithms

# Strings and Sequences


Data Structures & Algorithms

# Why Study *String* Algorithms?

- Bird's-eye view: **strings are character sequences**
  - Characters taken from an "alphabet"
  - Algorithms on strings are array/sequences algorithms
- What makes those arrays/sequences special?
  - Typical tasks to solve, defined by typical applications
- Applications of interest
  - Human-readable text (ASCII, Unicode)
  - Names and labels (people, files, license plates, etc)
  - DNA analysis
- Out of scope: sequences of objects, doubles



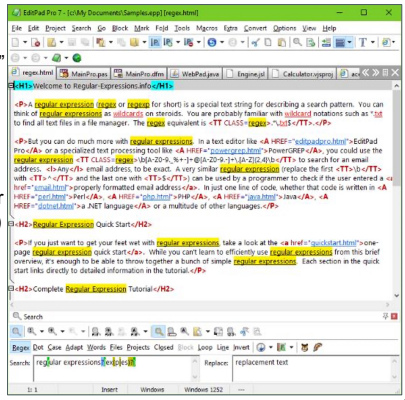
# Working With Strings/Sequences

- Given two strings/sequences
  - Are they the equal? (==)
  - Which would go first in a dictionary? (<=)
  - What do they have in common? (substrings)
  - Find where the strings differ
- Given many strings, order them in the *lexicographic* (dictionary) order
  - Ordering helps with search (fast look-up)




# Working With Strings and Texts

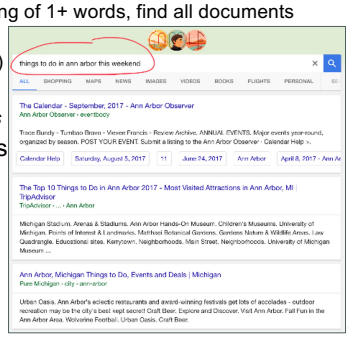
- Given strings *p* and *s*
  - "Needle" and "haystack"
- Find *instances* of *p* in *s*: first, last, next, all
  - Variant: *preprocess s* before *p* is known, optimize search time for many indep. queries (*p*)
- ***Text***: a string with a *separator* character, which delimits *words*
  - Given a word (or a phrase), find instances in text





# Working With Strings and Texts

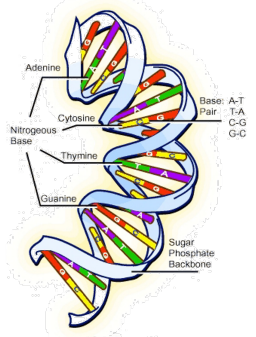
- A ***text corpus***: a collection of documents, each containing a text (+ title and other attributes, such as URL)
  - Given a search string consisting of 1+ words, find all documents containing those words
  - Exact matches (names, ID #s)
  - Approx matches (misspelled)
  - *A huge industry is built on this*
- Given Project 1 submissions
  - Find pairs of similar programs
- Text compression
  - Find repeated words
  - Replace them by numbers






# DNA and Genomic Sequences

- DNA structure
  - Defined by sequences of nucleotide base-pairs
  - **Alphabet**: A, C, G, T
  - In people: 23x2 chromosomes, 3B base-pairs (characters)
  - Genes == subsequences
- String algorithms for
  - Comparing chromosomes
  - Looking up genes
  - Assembling long DNA strands from short sequences





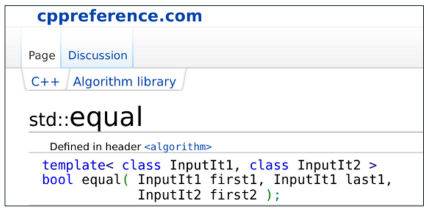

# String-related Data Structures

- Individual strings: C vs C++

| | Null-terminated | Object-oriented |
|---|---|---|
| Overhead | 1 ptr + 1 char | 2+ ptrs: begin/end |
| Complexity of .size() | ***O*(*n*)** | ***O*(1)** |
| Alphabet | ASCII | Configurable |
| Operations, algorithms | pointer arithmetics, stdlibc functions | methods, operators, stdlibc++ functions, STL |

- Sequences (iterator ranges)
  - C++ strings support `.begin()` and `.end()`
- Specialized containers for multiple strings
  - Dictionaries: add, remove, look up words
  - Strings with shared fragments (many words starting with "anti-", "pre-", "over-", "semi-", etc)

# Sequence Equality in STL




- Returns **`true`** if the range `[first1, last1)` is equal to the range `[first2, first2 + (last1 - first1))`, and **`false`** otherwise
- Two ranges are considered equal if for every iterator `i` in the range `[first1, last1)`, `*i == *(first2 + (i - first1))`



# A std::equal() Implementation

```
1  template<class InputIt1, class InputIt2>
2  bool equal(InputIt1 first1, InputIt1 last1,
3             InputIt2 first2) {
4    for ( ; first1 != last1; ++first1, ++first2) {
5      if (!(*first1 == *first2))
6        return false;
7    }  // for
8    return true;
9  }  // equal()
```

From http://en.cppreference.com/w/cpp/algorithm/equal

Job interview question: implement a variant of this function that takes a customizable `object==()` comparator



# Example: Using Sequence Equality

```
1  #include <algorithm>
2  #include <iostream>
3  #include <string>
4
5  bool is_palindrome(const std::string &s) {
6    return std::equal(begin(s), begin(s) + s.size() / 2, rbegin(s));
7  }  // is_palindrome()
8  void test(const std::string &s) {
9    std::cout << '"' << s << '"'
10     << (is_palindrome(s) ? " is" : " is not") << " a palindrome\n";
11 }  // test()
12 int main() {
13   test("radar");
14   test("hello");
15 }  // main()
```



# Strings and Sequences


Data Structures & Algorithms

# Lexicographic Comparison


Data Structures & Algorithms

# Rules of Lex-compare

- Two ranges are compared element by element
- The first mismatching element defines which range is lexicographically *less* or *greater* than the other
- If one range is a prefix of another, the shorter range is lexicographically *less* than the other
- If two ranges have equivalent elements and are of the same length, then the ranges are lexicographically *equal*
- An empty range is lexicographically *less* than any non-empty range
- Two empty ranges are lexicographically *equal*



# Dictionary Order

- **"" < "a" < "ab" < "b" < "ba" < "bc" < "bc0"**
- Overloading comparison operators in C++
  - Don't implement all 6 as independent operators
  - Suffices to implement `<` and `==` (STL uses these)
  - **(a != b)** is the same as **!(a == b)**
  - **(a > b)** is the same as **(b < a)**
  - **(a <= b)** is the same as **!(b < a)**
  - **(a >= b)** is the same as **!(a < b)**
- All other comparisons use **1** operator (plus !)
  - **Bad idea**: implementing `<=` using `<` and `==`



# Applied Strings

- Common implementation trick for `<typename T>`
  - `int compareHelper(const T& x, const T& y)`
  - Returns 0 for `x == y`, -1 for `x < y`, 1 for `x > y`
  - Comp. operators are implemented by post-processing
- Optimizations for strings
  - Short strings (<16B) don't need dynamic memory
  - In C, null-termination simplifies `op<()`: "abc" < "abcd"
  - In C++, strings of different size are `!=`
  - Fertile ground for *vectorized* CPU instructions

# Three-way String Compare

```
1  int compareHelper(const string& s0, const string& s1) {
2    const size_t len0 = s0.length(), len1 = s1.length();
3    for(size_t i = 0; i != std::min(len0, len1); ++i) {
4      if (s0[i] < s1[i])
5        return -1;
6      if (s0[i] > s1[i])
7        return 1;
8    }  // for
9    if (len0 < len1)
10     return -1;
11   if (len0 > len1)
12     return 1;
13   return 0;
14 }  // compareHelper()
```



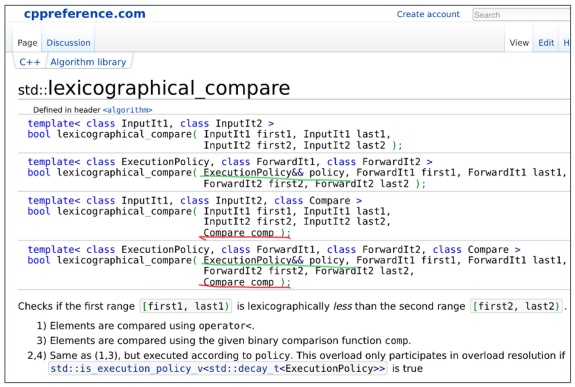
# Runtime Analysis

- ***O*(*n*)** worst-case time, ***O*(*1*)** extra space
  - Cannot do better
- Avg-case complexity for random strings
  - ***O*(*1*)** time: only need the first few characters
- Strings are often not random

  "c:\Program Files\MySQL\MySQL Server 5.0\bin"
  "c:\Program Files\MySQL\MySQL Server 5.5\bin"
  "versunken" "versuchen"
  "befehlen" "empfehlen"
- What if we compare nearby words in a dictionary?



# Lex-comparisons in STL






# Lex-compare: Implementation

```
1  template<class ForIt1, class ForIt2>
2  bool lexicographical_compare(ForIt1 first1, ForIt1 last1,
3                               ForIt2 first2, ForIt2 last2) {
4    while ((first1 != last1) && (first2 != last2)) {
5      if (*first1 < *first2)
6        return true;
7      if (*first2 < *first1)
8        return false;
9      ++first1, ++first2;
10   }  // for
11   return (first1 == last1) && (first2 != last2);
12 }  // lexicographical_compare()
```

From http://en.cppreference.com/w/cpp/algorithm/lexicographical_compare



# Lex-compare: Example

```
1  #include <algorithm>
2  #include <iostream>
3  #include <random>
4  #include <vector>
5  using namespace std;
6  void print_chars(const vector<char> &vec, const string &sep) {
7    for_each(begin(vec), end(vec), [] (auto c) { cout << c << ' '; });
8    cout << sep;
9  }  // print_chars()
10
11 int main() {
12   vector<char> v1 {'a', 'b', 'c', 'd'};
13   vector<char> v2 {v1};
14   mt19937 g{random_device{}()};
15   while (!lexicographical_compare(begin(v1), end(v1), begin(v2), end(v2))) {
16     print_chars(v1, ">= "); print_chars(v2, "\n");
17     shuffle(begin(v1), end(v1), g);
18     shuffle(begin(v2), end(v2), g);
19   }  // while
20   print_chars(v1, "<  "); print_chars(v2, "\n");
21   return 0;
22 }  // main()
```



# Application: Removing Duplicates

- **Given**: a container of string objects
- **Need**: leave only one copy of each string
  - Return all strings, if no duplicates present
- Sort given strings
  - Use STL `std::sort()` from STL with `operator<()`
  - Duplicates will be next to each other
- Compare neighboring strings using `operator==()`
- Copy only the first of duplicate strings
  - Use `std::unique()` from STL with `operator==()`



# Lexicographic Comparison

Click to Watch

Data Structures & Algorithms

# Searching and String Fingerprints

Click to Watch

Data Structures & Algorithms

# Finding a Needle in a Haystack



public member function

**std::string::find** <string>

C++98 C++11

```
string (1)   size_t find (const string& str, size_t pos = 0) const noexcept;
c-string (2) size_t find (const char* s, size_t pos = 0) const;
buffer (3)   size_t find (const char* s, size_t pos, size_type n) const;
character (4) size_t find (char c, size_t pos = 0) const noexcept;
```

**Find content in string**

Searches the string for the first occurrence of the sequence specified by its arguments.

When *pos* is specified, the search only includes characters at or after position *pos*, ignoring any possible occurrences that include characters before *pos*.

Notice that unlike member find_first_of, whenever more than one character is being searched for, it is not enough that just one of these characters match, but the entire sequence must match.

**Parameters**

str
Another string with the subject to search for.

pos
Position of the first character in the string to be considered in the search.
If this is greater than the string length, the function never finds matches.
Note: The first character is denoted by a value of 0 (not 1): A value of 0 means that the entire string is searched.

s
Pointer to an array of characters.
If argument *n* is specified *(3)*, the sequence to match are the first *n* characters in the array.
Otherwise *(2)*, a null-terminated sequence is expected: the length of the sequence to match is determined by the first occurrence of a null character.

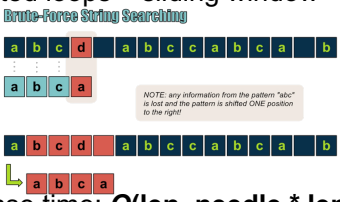
n
Length of sequence of characters to match.

c
Individual character to be searched for.



# Brute-force String Searching

- Two nested loops – sliding window




- Worst-case time: ***O*(len_needle * len_haystack)**
  - Finding "aaaaa**b**" in "aaaaaaaaaaaaaaaaaaaaa**b**a"
- Avg-case: ***O*(len_haystack)**
  - The inner loop performs ***O*(1)** iterations



# Brute-force String Searching

- STL implementations often choose this algorithm
  - Simple, lean implementation
  - Usually fast in practice: ***O*(len_haystack)** time
- However,
  - ***O*(len_needle * len_haystack)** worst-case time
    - Worst cases do appear in realistic strings
  - ***O*(len_needle + len_haystack)** worst-case time possible with pre-processing (*not as fast on most strings*), e.g., the Knuth-Morris-Pratt (KMP) algorithm



# Better Than Brute-force

- Idea: speed up the inner loop for known worst cases
  - Perform **most** eq-comparisons for strings in ***O*(1)** time
  - ***O*(len_needle * len_haystack)** becomes ***O*(len_haystack)**
- Worst-case complexity remains similar, but worst-case inputs are not as obvious and rare in practice
- How do we speed up eq-comparisons?



# Big Idea: **String Fingerprints**

- For each string, compute a number (`int`)
  - When fingerprints differ, strings must differ
  - When fingerprints match, strings rarely differ
  - **The actual numbers don't mean anything**
  - **Many different fingerprint functions exist**
- Ex: simple, but poor fingerprint function, F()
  - Replace each character by its ASCII code
  - Add up all codes

F("tom marvolo riddle ") == F("i am lord voldemort")



# Digression: A Sample Application

- If we only had fingerprint functions with few collisions (that don't show up in practice), we could solve the following problem
- Given *n* strings, find duplicates by comparing fingerprints
  - When FPs match, we must check strings for equality (in case FPs match by luck)
  - If most strings are different, FPs help a lot
  - We can sort FPs to find duplicates



# Rabin Fingerprint

- Instead of adding up character codes, view strings as *decimal numbers*
  - Characters: 'T', 'O', 'M', ' ', 'M', …
  - ASCII codes: 84, 79, 77, 32, 77, …
  - Running fingerprints:
    - **T** **84**
    - TO 10 * 84 + **79**
    - TOM 10 * (10 * 84 + 79) + **77**
    - TOM▮ 10 * (10 * (10 * 84 + 79) + 77) + **32**, …
  - Base 10 is used for illustration only (use larger numbers)
- **Shuffling the chars usually changes result**

# Sliding Rabin Fingerprint

- Calculate fingerprint *fp* for first *m* characters
- Removing a character on the left takes ***O*(1)** time
  - Precompute/store value of $10^{m-1}$ once as *p*
  - Subtract left-most character multiplied by *p* from *fp*
- Adding a character on the right takes ***O*(1)** time
  - Multiply *fp* by 10†
  - Add ASCII code of new right-most character
- Initial calculation of *fp* takes ***O*(*m*)** time
- Each of *n* - *m* slides takes **O(1)** time

† Using 10 for illustration only



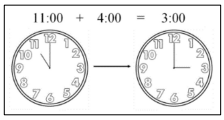
# Rabin-Karp String Search

1. Compute the FP of the *needle* in linear time
2. Check if the *haystack* starts with the needle
   - window = prefix_of_the_haystack
   - If (FP(window) != FP(needle)) go to Step 3
   - If (window == needle), then done
3. Incrementally update the FP of *window*
   - Remove one character on the left
   - Add one new character on the right
4. If (FP(window) == FP(needle)),
   if (window == needle), then done
5. If more chars remain in haystack, go to Step 3



# Technicality: Avoiding Overflows

- For long strings, performing **+**, ***** operations will result in overflows
- A common trick when constructing a fingerprint
  - Replace *conventional arithmetic* (**+**, *****) with *modular arithmetic:*
  - Pick some "largish prime" (eg. 3355439)
  - (x * y) becomes (x * y) % prime
    (x + y) becomes (x + y) % prime
  - Important: conventional arithmetic rules carry over
    (x + y = y + x, x * y = y * x, x * (y + z) = x * y + x * z, etc)






# Rabin-Karp: Time Complexity

- If different substrings never produce equal FPs, runtime is ***O*(len_haystack)**
- Every FP collision incurs ***O*(len_needle)** time to perform equality check of substrings
- In practice, collisions are very rare and don't correspond to meaningful pairs of strings
  - It helps to choose larger primes
- Choose base = $2^k$
  - Coprime with the original prime number
  - Multiplication by base simplifies to a binary shift (faster)



```
1  int rkSearch(const string &needle, const string &haystack, size_t prime) {
2    constexpr int base = 128;
3    const size_t N = needle.length(), H = haystack.length();
4    const size_t z = static_cast<size_t>(pow(base, N - 1)) % prime;
5    int n = 0, h = 0;
6    for (size_t i = 0; i < N; ++i) {
7      n = (base * n + needle[i]) % prime;  // calculate needle fingerprint
8      h = (base * h + haystack[i]) % prime;  // calc. window fingerprint
9    }  // for
10   for (size_t i = 0; i <= H - N; ++i) {
11     if (n == h) {  // check needle fp vs. current window
12       for (size_t j = 0; j < N; ++j) if (haystack[i + j] != needle[j]) break;
13       if (j == N) return i;
14     }  // if
15     if (i < H - N) {  // slide window
16       h = ((h - haystack[i] * z) % prime * base + haystack[i + N]) % prime;
17       if (h < 0) h = (h + prime);
18     }  // if
19   }  // for
20   return -1;  // needle not found
21 }  // rkSearch()
```



# Searching and String Fingerprints


Data Structures & Algorithms