String Processing Workshop

String Processing Overview

- What is string processing?
  - String processing refers to any algorithm that works with data stored in strings.
  - We will cover two vital areas in string processing
    - String representation
    - Pattern matching

Strings

- What is a string?
  - The word 'string' is commonly used to refer to any chunk of text. However this can also be extended to mean large chunks of binary data.
  - There are lots issues that arise with strings in the real world:
- Examples of strings:
  - The quick brown fox jumped over the lazy cow.
  - 那只敏捷的棕色狐狸跳过了懒牛
- We typically only see the first example.

String Operations

- Concatenation:
  - “Hello” + “World” = “HelloWorld”
- Indexing:
  - “HelloWorld”[5] = 'W'
- Iteration:
  - “HelloWorld” = 'H', 'e', 'l', 'l', 'o', 'W', 'o', 'r', 'l', 'd'
- Substring:
  - “HelloWorld”.substring(3, 3) = “loW”

String Representations

- The representation used for a string can be an important factor in efficiency.
- There are two main representations that we will discuss:
  - Variable-length arrays
  - Ropes

String Representations

- Variable-length arrays
  - Each character is stored sequentially in memory.
  - Various implementations:
    - A terminating character marks the end of the string. Often called a null or '\0' - CStrings
    - The length is encoded into the initial few bytes of the string. - PStrings
  - Implemented for you in std::string and via char* arrays.

String Representations

- Concatenation(of N and M length strings):
  - O(N + M) *** Why not O(M)?
- Indexing:
  - O(1)
- Iteration(of N length strings):
  - O(N) – always O(1) per element
- Substring(of length S in string of length N):
  - O(S)

String Representations

- Example code - string

```
string a = “Hello”; // a is now “Hello”
string b(“World”); // b is now “World”
string c; // c is now “”
string d = a + “ “ + b + c; // d is “Hello World”

string e = 'Hello'; // ERROR - ' ' is for characters
string f  = “Hello\0World”; // Take a guess...
```

String Representations

- What about char arrays?
  - Really horrible to use in a lot of cases.
  - char* isn't a class so no operators overloaded.
    - Horrible functions need to be used!
  - Sometimes they are necessary:
    - Some functions only take (const) char* arrays. We can convert from a string to char array via string.c_str()!
  - They have their benefits though.
    - Argument is basically: array vs vector

String Representations

- Ropes
  - A heavy duty string. Characters are stored in a concatenation tree.
    - Internal tree nodes mean concatenate the left and right children.
    - Leaf nodes hold the data of the string.
  - Not an easy thing to do in a competition. Also have to worry about balancing issues.
  - Implemented for you in __gnu_cxx:crope
    - Pretty much the same API as std::string with some notable exceptions.

String Representations

- Concatenation(of N and M length strings):
  - O(1) or O(logN)
- Indexing:
  - O(logN)
- Iteration(of N length strings):
  - O(N) or O(NlogN)
- Substring(of length S in string of length N):
  - O(logN)

String Representations

- Interesting facts about Ropes:
  - A very functional data structure.
    - When a substring is requested, very little memory is required.
    - Insertions do not require a significant of amount.
  - There are some languages which have Ropes as their string data structure of choice.
    - Cedar

String Representations

- There are many caveats with Ropes:
  - A much higher constant factor on all algorithms.
  - Iterators and Indexing
    - crope::iterator vs crope[i]
    - ++iterator vs iterator++
  - Consecutive characters might not be consecutive in memory.
  - They work better for algorithms which do not require random access.

String Matching

- What is String Matching?
  - String Matching is the process of determining whether a given string is a substring of another string.
  - String here often refers to texts of characters, but could also apply to other things such as sequences of numbers.

String Matching

- We are given two strings
- The haystack
  - The string in which we are searching.
- The needle
  - The string for which we are searching.
- Example
  - Haystack: ABRACADBRANANAFOOBRA
  - Needle: BRA

String Matching

- Many algorithms exist for solving this problem.
  - Naïve
    - Brute Force String Matching
  - Needle Optimisation
    - Boyer-Moore's Algorithm
    - Horspool's Algorithm
    - Rabin Karp Algorithm
    - Knuth-Morris-Pratt Algorithm
  - Haystack Optimisation
    - Suffix Trees
    - Suffix Arrays

String Matching

- Most of these algorithms are too complicated for the current IOI syllabus.
- We shall discuss four algorithms:
  - Brute Force String Matching
  - Rabin-Karp String Matching
  - Boyer-Moore String Matching
  - Knuth-Morris-Pratt String Matching

Brute Force String Matching

- Brute Force String Matching is the 'just do it' solution.
  - Place the needle at each valid position in the haystack.
  - If all corresponding positions in both strings match, then we have found a match.
  - If a single character does not match we have a mismatch.

Brute Force String Matching

- The good
  - It is conceptually simple and can be modified easily.
  - It is simple to write. In C++ it's only a few lines.
  - It doesn't necessarily perform slowly. It has an average case performance of O(N + M).
- The bad
  - It has a poor worst case of O(NM).

Brute Force String Matching

- Example
  - Haystack: AAAAAAA
  - Needle: AAB

```
AAAAAAA
AAB
 AAB
  AAB
   AAB
    AAB
```

Brute Force String Matching

- This method is very easily modifiable.
  - Approximate String Matching is only a one or two line change to the code.
  - Can be potentially sped up by doing hacks.
    - Jumping by the length of the needle testing for a match of all characters.
    - Testing multiple characters a time.
    - ...
- The law of diminishing returns applies.
- You still have a terrible worst-case performance.

Rabin-Karp String Matching

- Rabin-Karp String Matching uses hashing to reduce needless matching.
  - Similar to the Brute Force Algorithm.
  - Relies on the notion of a rolling hash function of strings.
  - It computes the hash of the needle and stores it.
  - It then computes the hash of each successive substring of the haystack.
  - When the two are equal, we have a potential match.

Rabin-Karp String Matching

- What is a rolling hash function?
  - These allow efficient computation of hash functions of consecutive substrings.
- Two fast operations need to be supported:
  - Hash(s):
    - Compute the hash of a string s.
  - Update(h, a, b):
    - Update the hash value, h, by deleting the first character a and adding the last character b.

Rabin-Karp String Matching

- Example
  - Haystack: AGCDDE
  - Needle: DD
- Hashes:
  - H = hash(DD)
  - H1 = hash(AG)
  - H2 = hash(GC) = update(H1, A, C)
  - H3 = hash(CD) = update(H2, G, D)
  - H4 = hash(DD) = update(H3, C, D)
  - H5 = hash(DE) = update(H4, D, E)

Rabin-Karp String Matching

- Concrete Example
  - hash(s) is the sum of all ASCII characters in s.
  - update(h, a, b) is then h - a + b
- Rabin-Karp:
  - H = hash(DD) = 136
  - H1 = hash(AG) = 136
  - H2 = hash(GC) = update(H1, A, C) = 133
  - H3 = hash(CD) = update(H2, B, D) = 135
  - H4 = hash(DD) = update(H3, C, D) = 136
  - H5 = hash(DE) = update(H4, D, E) = 137

Rabin-Karp String Matching

- Examples of rolling hash functions
  - Sum of all characters in the string.
  - Product of all characters in the string modulo n.
- These are easy to implement, however they do not have good properties as hash functions.
- A better function:
  - Choose two relatively prime numbers a and n.
  - Let the hash value be the sum of a power series increasing in a, with the characters as coefficients, modulo n.

Rabin-Karp String Matching

- The good
  - It is still pretty easy conceptually.
  - It is still pretty simple to write. In C++ it's only a few lines in a few functions
  - It will almost always outperform Brute Force String Matching.
- The bad
  - Still a rare worst case performance of O(NM).
  - Not easy to modify.

Rabin-Karp String Matching

- There are some interesting modifications that can be made to this algorithm
  - Using a hashtable we can test for multiple needles at the same time.
    - Store each needle hash in the table.
    - We simply check the hashtable to see which needles are matched.
    - Much better performance than Brute Force.
- Other modifications are not so easy
  - Approximate string matching?

Boyer-Moore String Matching

- Boyer-Moore String Matching is the smart solution.
  - It is optimal in that there is no asymptotically better algorithm.
- Modifications to the Brute Force algorithm:
  - Use tables to tell us how far we can jump ahead.
  - Try all matches from back to front.
    - Why?

Boyer-Moore String Matching

- From the Needle two tables are constructed:
  - Bad character shift table:
    - This table says how far you can safely jump if you mismatch at a particular character in the needle.
    - This table is the size of the alphabet.
  - Good suffix shift table:
    - This table says how far you can safely jump if you mismatch at a particular point in the needle.
    - This table is the size of the needle.
  - If we get a mismatch we jump the maximum.

Boyer-Moore String Matching

- Bad character shift table:
  - If a character does not occur in the needle, we can jump the length of the needle.
  - Loop through the needle from the first character to the last:
    - We set the character's value to it's distance to the end.
  - If it does occur, we calculate the distance required to
  - When you get a mismatch you use the character in the haystack to determine the jump.

Boyer-Moore String Matching

- Good suffix shift table:
  - Notes how much you can skip based on repeated the suffices in the needle.
  - Loop through the needle from the last character to the first:
    - Determine the minimum amount to shift to align suffices.
- This can be done created in linear time using a complicated algorithm.
- A naïve algorithm can be used which is quadratic in the size of the needle.

Boyer-Moore String Matching

- Example

Boyer-Moore String Matching

- The good
  - Incredibly fast. No more than 3N comparisons are needed in the worst case.
  - Incredibly fast. No more than 3N comparisons are needed in the worst case.
  - Incredibly fast. No more than 3N comparisons are needed in the worst case.
- The bad
  - Very complicated, both conceptually and in code.
  - Again, not easy to modify.

Boyer-Moore String Matching

- Boyer-Moore is incredibly fast.
  - The algorithm can be as fast as O(N/M)
  - It is still not as complicated as other optimal string searching algorithms.
    - See Knuth-Morris-Pratt String matching.
  - Leaving out the complicated good suffix table gives a variant called Boyer-Moore-Horspool.
    - Worst case O(NM)
    - Store each needle hash in the table.
- Other modifications are not so easy
  - Approximate string matching?

Conclusion

- Many algorithms for string matching.
  - We have looked at:
    - Brute force
    - Rabin-Karp
    - Boyer-Moore
    - Knuth-Morris-Pratt
  - All of these preprocess only the needle.
  - There are algorithms which preprocess the haystack.
    - Suffix Trees
    - Suffix Arrays

Conclusion

- Choose wisely
  - Each algorithm is a trade-off between coding complexity and speed.
  - Not all algorithms support the same modifications.
  - C++ string find is implemented efficiently, so explicit coding may not be necessary.
- Do calculations to see which algorithm you can get away with.