Brute Force, Boyer-Moore, Rabin-Karp

1332 Recitation: Week of June 23rd

Announcement

- Exam 2 is scheduled for Tuesday, June 30th
  - The exam will be open from 12am EDT until 11:59pm EDT
  - Topic list + more details can be found on Canvas announcement!
- Homework 7 on Pattern Matching is a **two-week** homework
  - It is due on July 6th at 11:55pm EDT, late submission deadline 2:00am EDT

Pattern Matching Intro

- How can we search for a pattern of (length m) in a text string (of length n)?
- We can adapt the following algorithms to solve two use cases:
  - Find first occurrence of a pattern in text
  - Find all occurrences of a pattern in text
- Most of these algorithms will have two variables:
  - **i**, which keeps track of the index where in the text where the start of the pattern is aligned
  - **j**, which tells us which index of the pattern we are currently comparing
  - This means that we will be looking at **pattern.charAt(j)** and **text.charAt(i + j)**

Brute Force

- Most basic search; no optimizations
- Algorithm
  - Line up index 0 of the pattern with index 0 of the text (this means i = 0)
  - Compare each character of the pattern with each character of the text
  - If they don't match, shift the pattern over by 1
  - On a character match, increment j
  - On a character mismatch, incremented i and reset j to 0
  - If finding all occurrences and a match is found, we increment i and reset j to 0
- Example
  - Text: ghogobghost
  - Pattern: ghost

Brute Force - Efficiency

- Worst case
  - When we have a mismatch at the very last character:
    - I.e. text “aaaaaa” and pattern “aac”: O(mn)
- Best cases
  - If finding just the first occurrence
    - When all characters in the pattern match: O(m)
  - If finding all occurrences
    - If we mismatch at every first character comparison: O(n)
- We will be very specific on exams about first/all occurrences!

Best case	Worst case	Average case
O(m) or O(n)	O(mn)	O(mn)

Boyer-Moore

- We skip past sections of text where a match is impossible!
- Possible because we pre-process the pattern to construct a Last Occurence table for all unique characters in the pattern
- When a mismatch occurs, we can align the pattern in a more optimal way to the text to avoid redundant comparisons

Boyer-Moore Last Occurrence Table

- Acts as a mapping from character in the pattern to the last index that character occurs in the pattern
- All characters that don't exist in the pattern have a "phantom mapping" of -1
    - getOrDefault() is useful here

Pattern: shanghai

Index:	0	1	2	3	4	5	6	7
Character:	S	H	A	N	G	H	A	I

Last Occurrence Table:

Key:	S	H	A	N	G	I	*
Value:	0	5	6	3	4	7	-1

Boyer-Moore - Algorithm

- ➔ i, j <- indices (i = alignment of the pattern to the text, j = the characters in the pattern we are comparing)
- ➔ We are looking at jth character in the pattern and the (i+j)th character in the text
- ➔ We initially set i = 0 before the outer loop and j = patternLength - 1 before the inner loop
- Start comparing from the back of the pattern
- If character at pattern and text match:
    - Decrement j and continue comparing characters in the pattern and text until a mismatch or until j == -1
- After the loop:
    - If j == -1, a match has been found starting at index i in the pattern
        - If looking for all occurrences, increment i, then continue the outer loop
    - Otherwise, we had a mismatch: to realign the pattern, we first get the value in the last occurence table corresponding to the mismatched character in the text -> call this value *shift*
        - If *shift* < j, i += j - *shift*; otherwise, i++
    - Attempt to realign the last occurence of this character in the pattern with the mismatched character in the text
        - If the pattern would have moved backwards, increment i by 1 instead
    - If character in text doesn’t exist in pattern, shift the pattern past this index with i += j - (-1)

Boyer-Moore - Example, Efficiencies

- Example!
  - Text: abacacadacaababacabab
  - Pattern: abacaba
- Average case: very text/pattern/alphabet dependent
- Worst case: O(mn) for finding first and all occurrences
  - When mismatches on first character of pattern with a character of the text that exists in the pattern
    - Similar to brute force worst case
- Best case:
  - Finding first occurrences: O(m)
    - O(m) for preprocessing + O(m) for matching on first try
  - Finding all occurrences: O(n/m + m)
    - Pattern always shifts by its length
- Works better when there's less overlap between letters in the pattern and letters in the text; works better with a *large alphabet*

Rabin-Karp

- BIG IDEA: We compute hashes for substrings and compare characters only when the hashes match
- Calculating the hash of a substring
  - Select a base number - hopefully a large prime number to help eliminate hash collisions
  - For each character:
    - Convert char to integer (ASCII values)
    - Compute a power of base - this is to account for the position of the char in the string
  - Add all values together to form the entire hashcode

$$H(\text{“abc”}) = 1 \times 26^2 + 2 \times 26^1 + 3 \times 26^0.$$

- Rolling the text hash (in O(1)!)
  - **newHash = (oldHash - (first char in text substring * baseToTheM-1))*base + new char at end of text substring**
    - Do not compute the hash all over again as that is unnecessary and inefficient

Rabin-Karp - Algorithm

- Calculate the hash of the pattern
- Calculate the initial hash of the text (the first m characters of the text)
  - Calculate the initial text hash and the pattern hash in the same loop starting from index (m - 1) and going to 0
    - This way, you can calculate the power of base needed by starting a variable at 1 and multiplying it by base for each iteration (a RUNNING FACTOR, just like LSD radix)
- Compare the pattern hash to the text hash
  - If they match -> compare the actual characters exactly like brute force
  - Else -> roll the hash and compare again
- Example!

Efficiencies

- Bad hashing function or bad base
    - If we have a bad hashing function, we’ll have lots of places where the hashes are equal, but the characters are different
- Worst case
    - O(mn) - nothing but hash collisions
- Best case:
    - Finding first occurence: O(m)
    - Finding all occurrences: O(m + n)
        - O(m) for computing initial hashes
        - O(n) because no hashes match unless it’s a full match

Exam Q&A

- **Arrays**
- **List ADT**
  - ArrayLists
  - LinkedLists
    - Singly-Linked (Coding Possible, HW1)
    - Doubly-Linked
    - Circularly-Linked
- **Linear ADTs**
  - Stacks (Coding Possible, HW2)
  - Queues (Coding Possible, HW2)
  - Deques
- **Trees**
  - BSTs (Coding Possible, HW3)
    - Traversals
  - Heaps
    - Including BuildHeap
  - AVLs (Coding Possible, HW5)
    - Rotations
  - 2-4 Trees
    - Overflow Handling (Promotion)
    - Underflow Handling (Transfer, Fusion)
  - Properties and Operations for all trees above
- **HashMaps**
  - External Chaining (Coding Possible, HW4)
  - Linear Probing
  - Quadratic Probing
- **SkipLists**
  - Additions
  - Removals
  - Traversals
- **Sorting Algorithms**
  - Bubble Sort
  - Cocktail Shaker Sort
  - Insertion Sort
  - Selection Sort
  - Merge Sort
  - LSD Radix Sort
  - Quick Sort
  - Quick Select / kth Select
  - Properties of all sorting algorithms above
    - Stability
    - Adaptability
    - In-Place vs. Out-of-Place
- **Misc.**
  - Big O for all of the above topics
  - Generics
  - Recursion