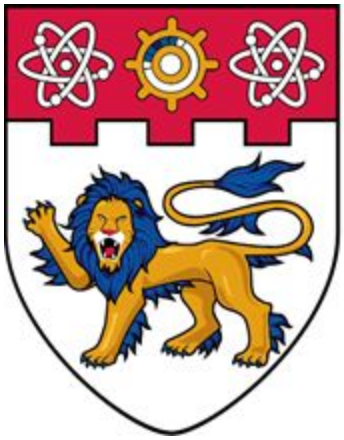


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

CZ2001 Algorithms

Project 1: Searching Algorithms

Lab Group: SS3

Group Members:

GOH CHEN KANG, SEAN
LEE KAI EN
LEE YANG FENG
FAVIAN CHAN JUN WEI
LIM QI WEI

Introduction

The aim of this project is to implement algorithms to search for exact occurrences of a query sequence in a nucleic acid sequence. The proposed algorithms will have better time complexities as compared to brute force searching. A set of complete analysis will be performed on all the algorithms implemented in this project. Language used is Python.

Algorithms

Brute Force

Steps:

1. Create an empty list to hold indexes of occurrences.
2. Use a nested for loop, the outer one looping over the DNA sequence string and the inner one looping over the query sequence string.
3. For all indexes(**n**) of the DNA sequence, compare to see if the character at the current index of the DNA sequence matches the first character of the query sequence**,** where **m** is the length of the query sequence string.
    - Upon a mismatch, break out of the inner for loop. Go to the next iteration.
    - Upon matching, continue matching till failure or complete match found.
4. If the current index of query sequence is equal to the length of the query sequence, an occurrence is found. Add the index to the list.
5. Repeat steps 1 - 5 until the end of the DNA sequence is reached.
6. Return the list.

Proposed Algorithm 1 (skipRedundant)

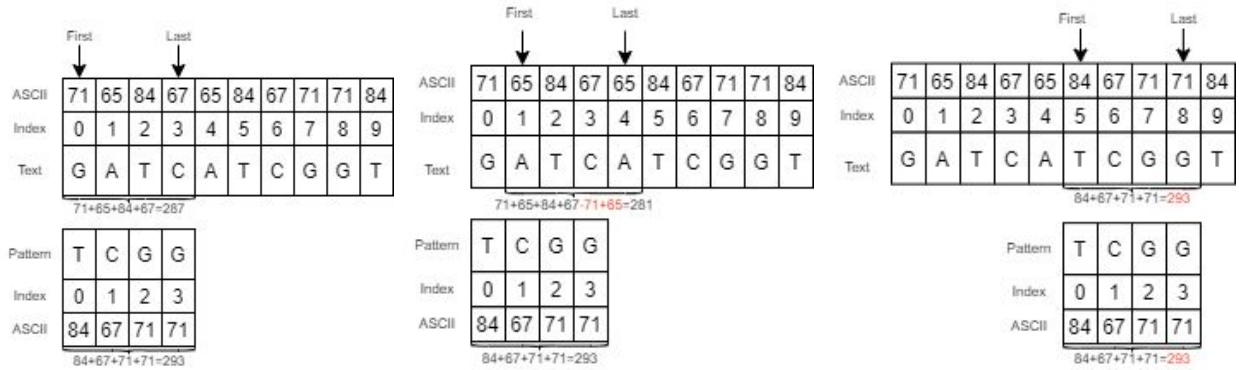
Let **n** be the length of the genome string and **m** be the length of the query string.
Steps:

1. Compare the first index of genome, **i**, with the first index of the query, **j**. Current index of the genome string is moved by adding **j** to **i**.
    - No match: increment **i** and move to the next character.
    - Match: continue by incrementing **j**, until end of string is reached or match fails
2. If **j** reaches **m-1** and it matches the character from the genome string, indicated by value **j** (set to -1 to indicate a break), an occurrence is found and index **i** is added to the initially empty list generated at the start.
3. Regardless of the matching outcome, before resetting **j** to its initial value of **0** at the end of the else block, the compared subsection of the genome in step 1 and 2 will have already been searched for the first occurrence of the first character of the query string, indicated by firstRepeatedPos and repeatedFound.
    - If found, increment **i** by firstRepeatedPos which holds the position of the occurrence.

- If not found, **i** increments by **1**.

4. Reset **j** to bring it back to the first character of the query string.
5. Repeat steps 1 - 4 until the remaining part of the genome string is shorter than the query string which means that there will no longer be any more occurrences found.

Proposed Algorithm 2 (compareSUM1)



Let query string be P[] and genome string be T[]
Let m be the length of P[] and n be length of T[]

Goal
Search for P[] in given T[]

Pre-processing

1. Using the ASCII value of the letters
2. Store the sum of P[] into Psum and sum of T[0] to T[m-1] into Tsum
3. Set a firstIndex on T[0] and lastIndex on T[m-1]

Searching

loop

1. If Psum == Tsum
   check
   a. Check from P[0] == T[firstIndex] … to P[m-1] == T[lastIndex]
      i. If any of them are a mismatch, do ***update***
      ii. If all are a match, return firstIndex and do ***update***
2. If Psum != Tsum
   update
   a. Subtract T[firstIndex] from Tsum
   b. Update firstIndex position + 1
   c. Update lastIndex position +1
   d. Add T[lastIndex] to Tsum, do ***loop***

Analysis of Algorithms

Brute Force Time Complexity

Best case: No match at start of every index or the query is one character only, present at every index

- Search **n - m + 1** indexes 1 time each
- Time complexity = **(n - m + 1)** x 1 = **n - m + 1** => **O (n - m)** => **O(n)**

Worst case: All indexes have a match and will search all **n - m + 1** indexes for a length of **m** for each index

- Operations in the inner loop: c_2
- Time complexity = (**m** x c_2) x (**n - m + 1**) + 1 = c_2 x **(m(n - m + 1))** + 1
  => **O(m(n-m))** => **O(nm)**

Average case: For the inner loop, assume on average that it takes about a few checks to break the loop, so we can assume that the inner loop runs on constant time on average, giving O(1). So, if the inner loop is approximately constant (c1) then the complexity is (n-m+1)c1. Then we will get **O(n-m)** which is under **O(n+m)**.

Algorithm 1 (skipRedundant) Time Complexity

Best Case:
There exists no occurrences of the query string, giving **(n - m + 1)** comparisons of `if queryString[j] != genomeString[i]`. Hence, the time complexity would be **(n - m + 1)** => **O(n - m) = O(n)**.

Worst Case:
The genome contains **n** repeating characters and the query contains the same character repeated **m** times, turning the search into a brute force search instead. In 1 loop, there will be **1** comparison of `if queryString[j] != genomeString[i]`. Then, the `else` code block has **2(m - 1) + 2 = 2m** comparisons for each loop, which repeats **(n - m + 1)** times hence, the total would be **2m(n - m + 1)**, giving a time complexity of **O(nm - m^2)** => **O(nm)**.

Eg. genome: AAAAAAAAAAAAAAA, query: AAA

Average Case:

Time cost = $\sum_{i=1}^{n-m+1} \sum_{j=1}^{m-1} \frac{1}{i} \times 2j = 2[\sum_{i=1}^{n-m+1} \frac{1}{i}][2(1+2+..+(m-1))] = 2H_{n-m+1}(2 \times \frac{1}{2}m(m-1))$

= $2H_{n-m+1}(m(m-1))$ => $O(\frac{m^2}{n})$

Algorithm 2 (compareSum1) Time Complexity

Pre-processing:
Store the sum of P[] into Psum **(m)** and sum of T[0] to T[m-1] into Tsum **(m)**. Therefore time complexity will be **(m+m)** = **O(m)**

Best case:
When Psum == Tsum occurs only once and algorithm have to ***update*** at every mismatch **(n-m+1)** and have to search from P[0] == T[firstIndex] to P[m-1] == T[lastIndex] only once **(m)**. Therefore the time complexity will be **((n-m+1)+m)** = **O(n)**.

Worst case:
When Psum == Tsum and algorithm have to search from P[0] == T[firstIndex] to P[m-1] == T[lastIndex] **(m)** after every ***update*** **(n-m+1)**. Therefore the time complexity will be **(m*(n-m+1))** = **O(mn)**.

Average case:
(no. of comparison) * (avg no. of comparison for each index in inner loop)

$= (n - m + 1) * \frac{1}{m} \sum_{i=1}^{m} i$

$= (n - m + 1) * \frac{1}{m} * \frac{m}{2}(1 + m)$

$= (n - m + 1) * \frac{1+m}{2}$

=> **O(nm)**

Statements of Contribution

Sean:
- Analysis of algorithms and report

Kai En:
- Came up with Brute Force and Algorithm 1 (skipRedundant). Did up the report descriptions and analysis of both algorithms.

Yang Feng:
- Research on various searching algorithm and attempt on analysis

Favian:
- Came up with Algorithm 2 (compareSum1), report

Qi Wei:
- improvements for algorithm for report, diagrams and presentation slides