CMPT 120

Topic: Searching – Part 1
and Intro to Time Efficiency (Algorithm Analysis)
Last Lecture

• Binary Encoding
Learning Outcome

• At the end of this course, a student is expected to:
  • Create (design), analyze, and explain the behaviour of simple algorithms:
    • ...
    • Describe and illustrate the operation of **linear search**, **binary search**, and \(O(n^2)\) sorting algorithms
    • Analyze the running time (time efficiency) of simple iterative algorithms
    • Compare the running time (time efficiency) of algorithms; determine if one algorithm is more time efficient than another
  • Create (design) small to medium size programs using Python:
    • ...
    • Create programs that **search** lists and strings
When we want to search for a particular element in our data, we can do so in a variety of ways.

- We have a choice of searching algorithms.

So, we shall have a look at these searching algorithms.

But, which one should we choose?

To help us answer this question, we shall analyze these algorithms, looking at their time efficiency and expressing it using the Big O notation.
Searching

• A common problem: find an element in a sequence of elements (e.g.: a string or a list)
Let’s try!
Is ____ in this sequence?

42  8  12  34  2  67  33  26  89  54
What did we do to find the target?
Linear search algorithm

linearSearch(list, target)

set result to value TARGET_NOT_FOUND
set targetNotFound to value true

if list not empty
    set currentElement to first element of list
while targetNotFound AND have not looked at every element of list
    if currentElement == target
        set result to current element
        set targetNotFound to false
    otherwise
        set currentElement to next element of list

return result
Observations

• Our linear search algorithm:
  • finds the first occurrence of the target
    • What if the target occurs many times in the list
  • returns the target
    • What else could it return?
Let’s try this linear search!

ourList = [0, 6, 9, 2, 5, 3, 7, 1, 2, 4]           target: 2
What other test cases can we use?
Test cases to test our linear search algorithm

1. ourList = [ ]  
   target: 7

2. ourList = [ 0, 6, 9, 2, 5, 3, 7, 1, 4 ]  
   target: 0

3. ourList = [ 0, 6, 9, 2, 5, 3, 7, 1, 4 ]  
   target: 4

4. ourList = [ 0, 6, 9, 2, 5, 3, 7, 1, 4 ]  
   target: 5

5. ourList = [ 0, 6, 9, 2, 5, 3, 7, 1, 4 ]  
   target: 8

6. ourList = [ 0, 1, 2, 3, 4, 5, 6, 7, 9 ]  
   target: 4

7. ourList = [ 6, 6, 6, 6, 6, 6, 6, 6, 6 ]  
   target: 0

8. ourList = [ 6, 6, 6, 6, 6, 6, 6, 6, 6 ]  
   target: 6
Reminder: Generalisation guideline and testing

• When we design algorithms/programs/functions, we want to make sure they solve as many similar problems as possible, i.e., they work with as many different data configurations as possible

-> generalisation guideline

• Therefore, we shall test our algorithms/programs/functions accordingly:
  • For example: Does our search algorithm work successfully with …
    • An empty list? A sorted list?
    • An list containing the same element?
    • A list containing the target once?
    • A list containing the target several times, etc…
Linear search algorithm

- **Advantages**
  - Simple to understand, implement and test

- **Disadvantages**
  - Slow (time inefficient) because it looks at every element
  - Wait a minute! Not always!
    - Linear search “processed” some of our test cases faster than others
Linear search “processes” some of our test cases faster than others?

• How can we figure out whether linear search does indeed “process” some of our test cases faster than others?
• How can we observe and express this fact?
• By analyzing it! -> Algorithm Analysis
  • By looking at the behavior of an algorithm, i.e., looking at the time an algorithm takes to execute
• We do this analysis by counting how many times linear search executes its critical operations when it “processes” data (i.e., a test case)

• Critical operations?
  1. loop iteration -> visiting each element
  2. element == target?
Let’s count # of times we do the critical operations

<table>
<thead>
<tr>
<th>Test Case:</th>
<th>iteration count:</th>
<th>== count:</th>
</tr>
</thead>
<tbody>
<tr>
<td>ourList = [ ]</td>
<td>target: 7</td>
<td></td>
</tr>
<tr>
<td>ourList = [ 0, 6, 9, 2, 5, 3, 7, 1, 2, 4 ]</td>
<td>target: 0</td>
<td></td>
</tr>
<tr>
<td>ourList = [ 0, 6, 9, 2, 5, 3, 7, 1, 2, 4 ]</td>
<td>target: 4</td>
<td></td>
</tr>
<tr>
<td>ourList = [ 0, 6, 9, 2, 5, 3, 7, 1, 2, 4 ]</td>
<td>target: 5</td>
<td></td>
</tr>
<tr>
<td>ourList = [ 0, 6, 9, 2, 5, 3, 7, 1, 2, 4 ]</td>
<td>target: 8</td>
<td></td>
</tr>
</tbody>
</table>
Various scenarios

- Based on our results, we can conclude that the way our data is organized influences the time efficiency of linear search algorithm
- There are 3 scenarios
  1. **Best case scenario**
     - The data is organized such that the algorithm requires minimum amount of time to execute, producing minimum iteration and \( == \) counts
  2. **Average case scenario**
     - The data is organized such that the algorithm requires average amount of time to execute
  3. **Worst case scenario**
     - The data is organized such that the algorithm requires maximum amount of time to execute, producing maximum iteration and \( == \) counts
Time efficiency of an algorithm

• **Question:** How can we express the time efficiency of linear search algorithm? Do we use these counts?

• **Answer:** Not quite, but it is the first step!

• **Procedure:**
  1. **We count** number of times critical operation(s) of an algorithm is/are executed
     • In general, computer scientists use the count produced for the worst case scenario
  2. Express this **count** as a function of the **size of our data** -> n (in a general fashion, i.e., for all our test cases)
  3. Then we match this **count**, now expressed as a function of n, to one of these possible **Big O notations**: O(1), O(log n), O(n), O(n^2), … that is the closest to it:
Synonyms

• How an algorithm behaves
• How much time an algorithm takes (requires) to execute
• Number of times critical operations of an algorithm are executed (counts)
Big O notation

• We express the **time efficiency** of an algorithm using this Big O notation

• The reason why this Big O notation uses \( n \) is because computing scientists are interested in knowing how an algorithm behaves as the size of its data, i.e., \( n \) goes to infinity

• So, this Big O notation actually indicates an upper bound on **counts** as this \( n \) goes to infinity

• In other words, the Big O notation describes how the algorithm behaves as the size of its data increases
Time efficiency of linear search algorithm

- Worst case scenario of linear search algorithm has a time efficiency of

since the time required by the linear search algorithm (under the worst case scenario) to find “target” in a list of length $n$ is directly proportional to the number of elements in the list, i.e., $n$
Explanation and examples of Big O notation

- $O(1)$

- $O(\log_2 n)$

- $O(n)$

- $O(n^2)$
Big O notation! Big deal!

- Once we have found the Big O notation of an algorithm, what do we do with it?
- In other words: What’s the purpose of an algorithm’s time efficiency?

- Imagine we have 4 algorithms that have the same purpose
  - Algorithm 1 executes in $O(1)$
  - Algorithm 2 executes in $O(\log_2 n)$
  - Algorithm 3 executes in $O(n)$
  - Algorithm 4 executes in $O(n^2)$
Big O notation

# of times algorithm executes a "critical operation"

As $n$ goes to infinity

Length/size of data

- $O(n^2)$
- $O(n)$
- $O(\log_2 n)$
- $O(1)$
So?

Remember this slide from Lecture 3?

Problem Solving Process using computers

1. State the problem
   - Figure out what the problem is and make sure we understand it

2. a) Design possible solution(s)
   - Solution is expressed as an algorithm

   b) Identify data
   - Input – identify data needed in order to solve problem
   - Structure the data and represent it in solution to problem
   - Output – identify data produced by solution to problem

3. Select the “best” solution
   - By analyzing algorithms
   - Which one is the most effective/efficient?

4. Implement the selected solution
   - We implement the algorithm into a computer program

5. Testing
   - Does the program execute?
   - Does it solve the problem?
Re-introducing Step 3 in Software development process

• If we want to solve a particular problem and we have a choice of algorithms that will solve the problem, then we can select the most time efficient algorithm by considering their time efficiency i.e., their Big O notation

• And this is why computing scientists analyse the behaviour of algorithms as the size of their data, i.e., $n$ goes to infinity
Summary

• Looked at the first way of searching
  • Linear search algorithm
• Introduced time efficiency analysis of algorithms using the Big O notation
Next Lecture

• Another way of searching
• Compare and contrast these 2 searching algorithms