COMP 364: Computer Tools for Life Sciences
Algorithm design: Linear and Binary Search

Mathieu Blanchette, based on material from Christopher J.F. Cameron and Carlos G. Oliver
Algorithms

An **algorithm** is a predetermined series of instructions for carrying out a task in a finite number of steps

- or a recipe

Input → algorithm → output
Example algorithm: baking a cake

What is the input? algorithm? output?
Pseudocode is a universal and informal language to describe algorithms from humans to humans.

It is not a programming language (it can’t be executed by a computer), but it can easily be translated by a programmer to any programming language.

It uses variables, control-flow operators (while, do, for, if, else, etc.).
Example Python statements

```python
students = ["Kris", "David", "JC", "Emmanuel"]
grades = [75, 90, 45, 100]
for student, grade in zip(students, grades):
    if grade >= 60:
        print(student, "has passed")
    else:
        print(student, "has failed")

#output:
#Kris has passed
#David has passed
#JC has failed
#Emmanuel has passed
```
Example pseudocode

Algorithm 1 Student assessment
1:   for each student do
2:      if student’s grade ≥ 60 then
3:         print ‘student has passed’
4:      else
5:         print ‘student has failed’
6:      end if
7:   end for
Search algorithms

**Search** algorithms locate an item in a data structure

**Input:** a list of (un)sorted items and value of item to be searched

**Algorithms:** linear and binary search algorithms will be covered

- images if search algorithms taken from: http://www.tutorialspoint.com/data_structures_algorithms/

**Output:** if value is found in the list, return index of item

**Example:**

- search ( key = 5, list = [ 3, 7, 6, 2, 5, 2, 8, 9, 2 ] ) should return 4.
- search ( key = 1, list = [ 3, 7, 6, 2, 5, 2, 8, 9, 2 ] ) should return nothing.
Linear search

A very simple search algorithm

- a sequential search is made over all items one by one
- every item is checked
- if a match is found, then index is returned
- otherwise the search continues until the end of the sequence

Example: search for the item with value 33

10 14 19 26 27 31 33 35 42 44
Linear search #2

Starting with the first item in the sequence:

Then the next:
Linear search #3

And so on and so on...
Linear search #4

Until an item with a matching value is found:

If no item has a matching value, the search continues until the end of the sequence
Algorithm 2 Linear search

1: procedure LINEAR_SEARCH(sequence, key)  
2:     for index = 0 to length(sequence) do  
3:         if sequence[index] == key then  
4:             return index  
5:         end if  
6:     end for  
7:     return None  
8: end procedure
def linear_search(sequence, key):
    for index in range(0, len(sequence)):
        if sequence[index] == key:
            return index
    return None
Issues with linear search

Running time: If the sequence to be searched is very long, the function will run for a long time.

Example: The list of all medical records in Quebec contains more than 8 Million elements!

Much of computer science is about designing *efficient* algorithms, that are able to yield a solution quickly even on large data sets.

See experimentation on Wing...
Binary search

A fast search algorithm (compared to linear)
- the sequence of items must be sorted
- works on the principle of ‘divide and conquer’

**Analogy:** Searching for a word (called the key) in an English dictionary.

To look for a particular word:
- Compare the word in the middle of the dictionary to the key
- If they match, you’ve found the word! Stop.
- If the middle word is greater than the key, then the key is searched for in the left half of the dictionary
- Otherwise, the key is searched for in the right half of the dictionary
- This repeated halves the portion of the dictionary that needs to be considered, until either the word is found, or we’ve narrowed it down to a portion that contains zero word, and we conclude that the key is not in the dictionary
Example: let’s search for the value 31 in the following sorted sequence

First, we need to determine the middle item:

```python
sequence = [10, 14, 19, 26, 27, 31, 33, 35, 42, 44]
low = 0
high = len(sequence) - 1
mid = low + (high-low)//2  # integer division
print (mid)  # prints: 4
```
Binary search #3

Since \( index = 4 \) is the midpoint of the sequence

- we compare the value stored (27)
- against the value being searched (31)

The value at index 4 is 27, which is not a match

- the value being search is greater than 27
- since we have a sorted array, we know that the target value can only be in the upper portion of the list
Binary search #4

*low* is changed to *mid + 1*

Now, we find the new *mid*

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>14</td>
<td>19</td>
<td>26</td>
<td>27</td>
<td>31</td>
<td>33</td>
<td>35</td>
<td>42</td>
</tr>
</tbody>
</table>

low = mid + 1  # 5
mid = low + (high-low)//2  # integer division
print (mid)  # prints: 7
Binary search #4

*mid* is 7 now

- compare the value stored at index 7 with our value being searched (31)

![Sorted List](image)

The value stored at location 7 is not a match

- 35 is greater than 31
- since it’s a sorted list, the value must be in the lower half
- set *high* to *mid* - 1
Binary search #5

Calculate the mid again

▶ $mid$ is now equal to 5

We compare the value stored at index 5 with our value being searched (31)

▶ It is a match!
Binary search #6

Remember,

▶ binary search halves the searchable items
▶ improves upon linear search, but...
▶ requires a sorted collection

Useful links

bisect - Python module that implements binary search
▶ https://docs.python.org/2/library/bisect.html

Visualization of binary search
▶ http://interactivepython.org/runestone/static/pythonds/SortSearch/TheBinarySearch.html
Algorithm 3  Binary search

1: \textbf{procedure} \textsc{Binary\_Search}(sequence, key)
2: \hspace{1em} low = 0, high = \text{length}(sequence) \hspace{1em} - \hspace{1em} 1
3: \hspace{1em} \textbf{while} low \leq high \hspace{1em} \textbf{do}
4: \hspace{2em} mid = (low + high) / 2
5: \hspace{2em} \textbf{if} sequence[mid] > key \hspace{1em} \textbf{then}
6: \hspace{3em} high = mid - 1
7: \hspace{2em} \textbf{else if} sequence[mid] < key \hspace{1em} \textbf{then}
8: \hspace{3em} low = mid + 1
9: \hspace{2em} \textbf{else}
10: \hspace{3em} \textbf{return} mid
11: \hspace{2em} \textbf{end if}
12: \hspace{2em} \textbf{end while}
13: \hspace{1em} \textbf{return} ‘Not found’
14: \textbf{end procedure}
Binary search: Python implementation

```python
def binary_search(sequence, key):
    low = 0
    high = len(sequence) - 1
    while low <= high:
        mid = (low + high) // 2
        if sequence[mid] > key:
            high = mid - 1
        elif sequence[mid] < key:
            low = mid + 1
        else:
            return mid
    return None
```
Linear vs Binary search efficiency

Try linear_and_binary_search.py to see for yourself the difference in running time for large lists!

For a list of 100 Million elements, linear search takes about 3 seconds, and binary search takes about 0.001 seconds binary search is more than 3,000 times faster than linear search.

In general,

- the running time of linear search is proportional to the length of the list being searched.
- the running time of linear search is proportional to the logarithm of the length of the list being searched.