COMP 204

Algorithm design: Linear and Binary Search

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

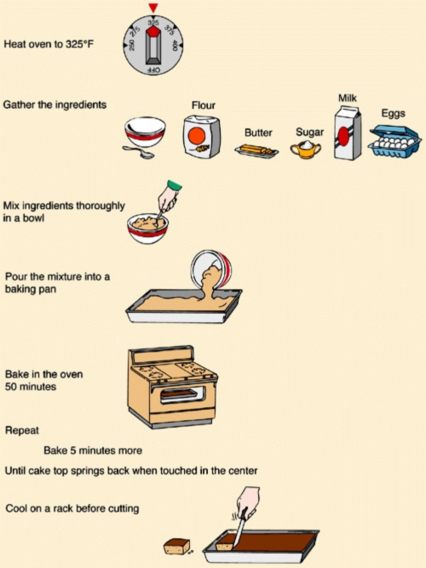
Algorithms

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

- or a recipe

Input $\rightarrow$ algorithm $\rightarrow$ output

Example algorithm: baking a cake



What is the input?

algorithm?

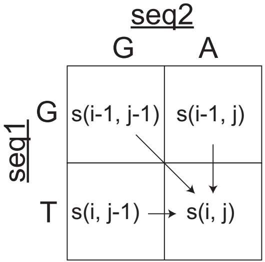
output?

Example algorithm: sequence alignment (A2)

Input: seq1, seq2

Output: alignments of seq1 and seq2

Algorithm:



$$s(i,j) = \max \begin{cases} s(i-1,j-1) + (mis)match \\ s(i-1,j) + gap \\ s(i,j-1) + gap \end{cases}$$

- $s(i-1,j-1) + (\text{mis})\text{match}$: align letter seq1[i] with letter seq2[j] (match: +2, mismatch: -2)
- $s(i-1,j) + \text{gap}$: align a gap "-" from seq2 with seq1[i] (gap: -2)
- $s(i,j-1) + \text{gap}$: align a gap "-" from seq1 with seq2[j] (gap: -2)

Pseudocode

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

```
for each student do
    if student's grade ≥ 60 then
        print 'student has passed'
    else
        print 'student has failed'
    end if
end for
```

Example algorithm: longest hydrophobic patch (L12)

Input:

amino acid sequence

Output:

longest hydrophobic patch

Algorithm:

findLongestHydrophobicPatch(protein)

```
isHydrophobicPatch(sequence)?

EDAYQIALEGAASTE
```

outer for loop:
start position from
start = 0

inner for loop
end position from
end = start + 1

isHydrophobicPatch(sequence)?

```
isHydrophobic('E')          isHydrophobic('L')
# (1) first a.a.            # (2) last a.a.

EDAYQIAL
```

for-loop

```
patchLen += isHydrophobic(s[aa])
# (3) length of hydrophobic amino acids (min 80%)
```

isHydrophobic(aa)?

```
aa in ["G","A","V","L","I","P","F","M","W"]?
```

findLongestHydrophobicPatch Python code

```
# This returns the longest hydrophobic patch found in a sequence
def findLongestHydrophobicPatch(protein):
    longestPatch=""  # the longest patch found so far

    # for every possible starting point
    for start in range(0,len(protein)):

        # and every possible end point
        for end in range(start+1,len(protein)+1):
            # get the sequence
            candidate = protein[start:end]

            # test hydrophobicity
            if isHydrophobicPatch(candidate):

                # if longer than longest seen so far, update
                if len(candidate)>len(longestPatch):
                    longestPatch = candidate

    return longestPatch
```

findLongestHydrophobicPatch pseudocode

Algorithm 2 findLongestHydrophobicPatch

```
while start position < protein length do
    end position ← start position + 1
    while end position < protein length do
        candidate ← protein substring from start to end position
        if candidate is hydrophobic patch then
            if length(candidate) > length(longestHydroPho) then longestHydroPho ← candidate
            end if
        end if
        end position ← end position + 1
    end while
    start position ← start position + 1
end while
```

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

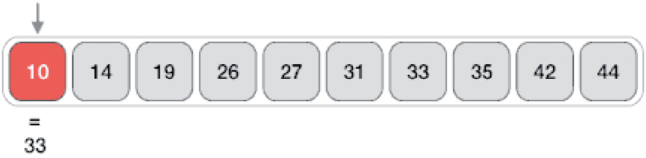
A very simple search algorithm

- 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

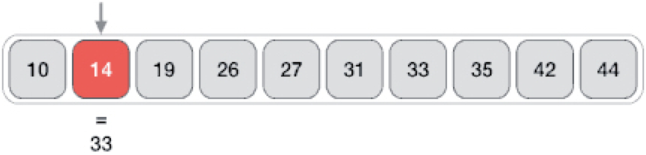
Example: search for the item with value 33

Linear search #2

Starting with the first item in the sequence:

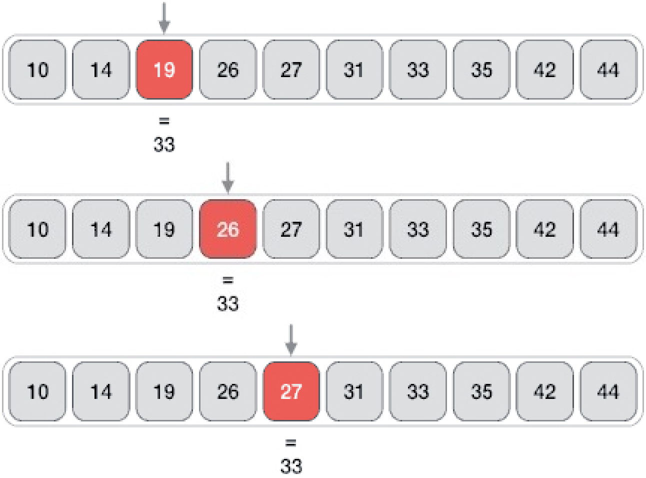


Then the next:



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

Linear search: pseudocode

Algorithm 3 Linear search

```
procedure LINEAR_SEARCH(sequence, key)
    for index = 0 to length(sequence) do
        if sequence[index] == key then
            return index
        end if
    end for
    return None
end procedure
```

Linear search: Python implementation

```python
def linear_search(sequence, key):
    for index in range(0, len(sequence)):
        if sequence[index] == key:
            return index
    return None

# import random
# L = random.sample(range(1,10**9),10**7)
# import time
# time_start = time.time()
# print(f"start: {time.asctime(time.localtime(time_start))
# index = linear_search(L, -1)
# time_finish = time.time()
# print(f"end:   {time.asctime(time.localtime(time_finish))
# print("time taken (seconds):", time_finish-time_start)
```

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

Binary search #2

Example: let's search for the value 31 in the following sorted sequence

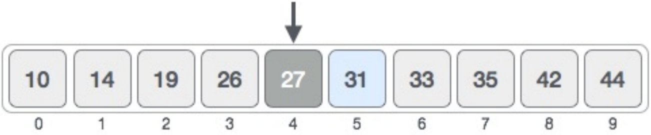
First, we need to determine the middle item:

```
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

low high

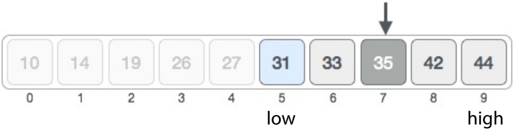
Now, we find the new *mid*

```
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)



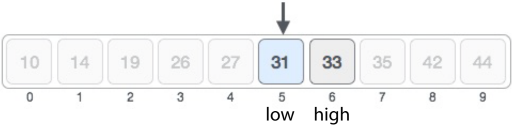
The value stored at location 7 is not a match

- 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

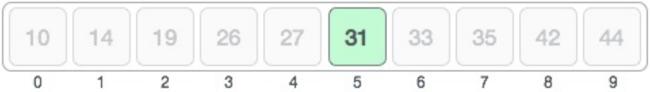
Calculate the mid again

- *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`

Binary search: pseudocode

Algorithm 4 Binary search

```
procedure BINARY_SEARCH(sequence, key)
    low = 0, high = length(sequence) − 1
    while low ≤ high do
        mid = (low + high) / 2
        if sequence[mid] > key then
            high = mid - 1
        else if sequence[mid] < key then
            low = mid + 1
        else
            return mid
        end if
    end while
    return 'Not found'
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,

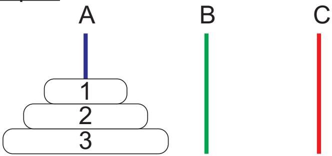
- the running time of linear search is proportional to the length of the list being searched.
- the running time of linear search is proportional to the **logarithm** of the length of the list being searched.

Binary search versus Linear search

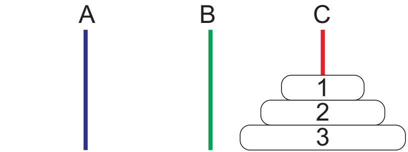
```python
import random
import time
from decimal import Decimal
from linear_search import linear_search
from binary_search import binary_search

# generate list of 100 Million elements,
# where each element is a random number between 0 and 100,000,000
print("Generating list...")
n = 10**7
L = random.sample(range(10**9), n)

L.append(876567) # for testing purpose

print("Sorting list...")
L.sort()

key = int(input("Enter key for linear search: "))

# perform linear search
print("Starting linear search ...")
time_start = time.time()
index = linear_search(L, key)
time_finish = time.time()
linear_search_time = time_finish-time_start
print(f"Found at position: {index}; time taken:", "{:.2e}".format(linear_search_time), "seconds")

print("Starting binary search ...")
time_start = time.time()
index = binary_search(L, key)
time_finish = time.time()
binary_search_time = time_finish-time_start
print(f"Found at position: {index}; time taken:", "{:.2e}".format(binary_search_time), "seconds")
```

Example algorithm: Tower of Hanoi (Advanced)

Input:



Output:



Rules:

- *Only one peg* can be moved at a time
- Take the top disk from one of the stacks and place it on top of another stack or empty rod
- *No larger disk may be placed on top of a smaller disk*

Algorithm (recursive):

- Move n - 1 disks from source peg to spare peg
- Move mth disk from the source to the target peg
- Move the n - 1 disks from spare peg to the target peg

See `tower_of_hanoi.py`

tower_of_hanoi Python code (Advanced)

```python
def move(n, source, target, spare):
    if n > 0:
        # move n - 1 disks from source to spare
        move(n - 1, source, spare, target)

        # move the nth disk from source to target
        target.append(source.pop())

        # Display our progress
        print(A, B, C, '##############', sep = '\n')

        # move the n - 1 disks that we left on spare onto target
        move(n - 1, spare, target, source)

# initiate call from source A to target C with spare B
A = [3, 2, 1]
B = []
C = []

move(3, A, C, B)
```

`tower_of_hanoi` pseudocode (Advanced)

Algorithm 5 Tower of Hanoi Mover

1: Move n - 1 disks from source peg to spare peg
2: Move the nth disk from the source to the target peg
3: Move the n - 1 disks from spare peg to the target peg
4: Do nothing if no disk left on source and spare peg