# Algorithms

# Algorithms

An **algorithm** is a finite set of instructions that accomplish a specific task.

Every algorithm can be constructed using combinations of sequencing, selection, and iteration.

**Sequencing** is the application of each step of an algorithm in the order in which the code statements are given. (for example, following a baking recipe requires that steps are taken in a certain order)

**Iteration** is a repeating portion of an algorithm. Iteration repeats a specified number of times or until a given condition is met.

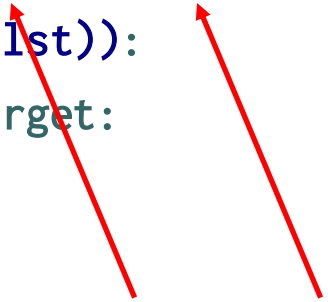
**Selection** determines which parts of an algorithm are executed based on a condition being true or false.

We will discuss some important algorithms in this lecture.

# Sequential search

Linear search or sequential search algorithms check each element of a list, in order, until the desired value is found or all elements in the list have been checked. Implement sequential search using list which returns the index of the target or `-1` if it is not found.

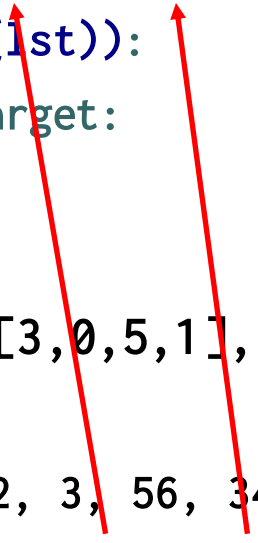
```
def sequential_search(lst, target):
        for i in range(len(lst)):
            if lst[i] == target:
                return i
        return -1
a = sequential_search([3,0,5,1], 0) # a = 1
print(a) # 1
```



# Sequential search

Linear search or sequential search algorithms check each element of a list, in order, until the desired value is found or all elements in the list have been checked. Implement sequential search using list which returns the index of the target or -1 if it is not found.

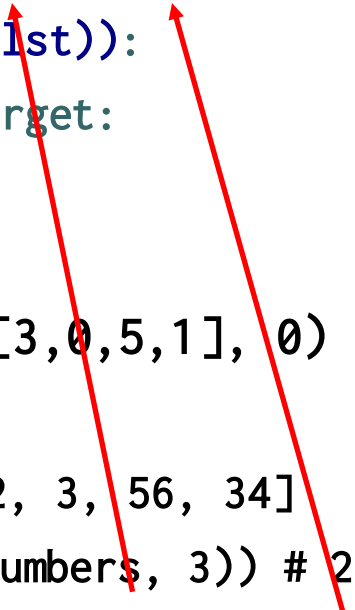
```
def sequential_search(lst, target):
      for i in range(len(lst)):
          if lst[i] == target:
              return i
      return -1
a = sequential_search([3,0,5,1], 0) # a = 1
print(a) # 1
numbers = [4, 2, 3, 7 -12, 3, 56, 34]
print(sequential_search(numbers, 3)) # 2
```



# Sequential search

Linear search or sequential search algorithms check each element of a list, in order, until the desired value is found or all elements in the list have been checked. Implement sequential search using list which returns the index of the target or `-1` if it is not found.

```
def sequential_search(lst, target):
        for i in range(len(lst)):
            if lst[i] == target:
                return i
        return -1
a = sequential_search([3,0,5,1], 0) # a = 1
print(a) # 1
numbers = [4, 2, 3, 7 -12, 3, 56, 34]
print(sequential_search(numbers, 3)) # 2
print(sequential_search(numbers, 100)) # -1
```



# Binary Search

Note that the array below is sorted. How can we take advantage of this?

The binary search algorithm starts at the middle of a sorted data set of numbers and eliminates half of the data; this process repeats until the desired value is found or all elements have been eliminated.

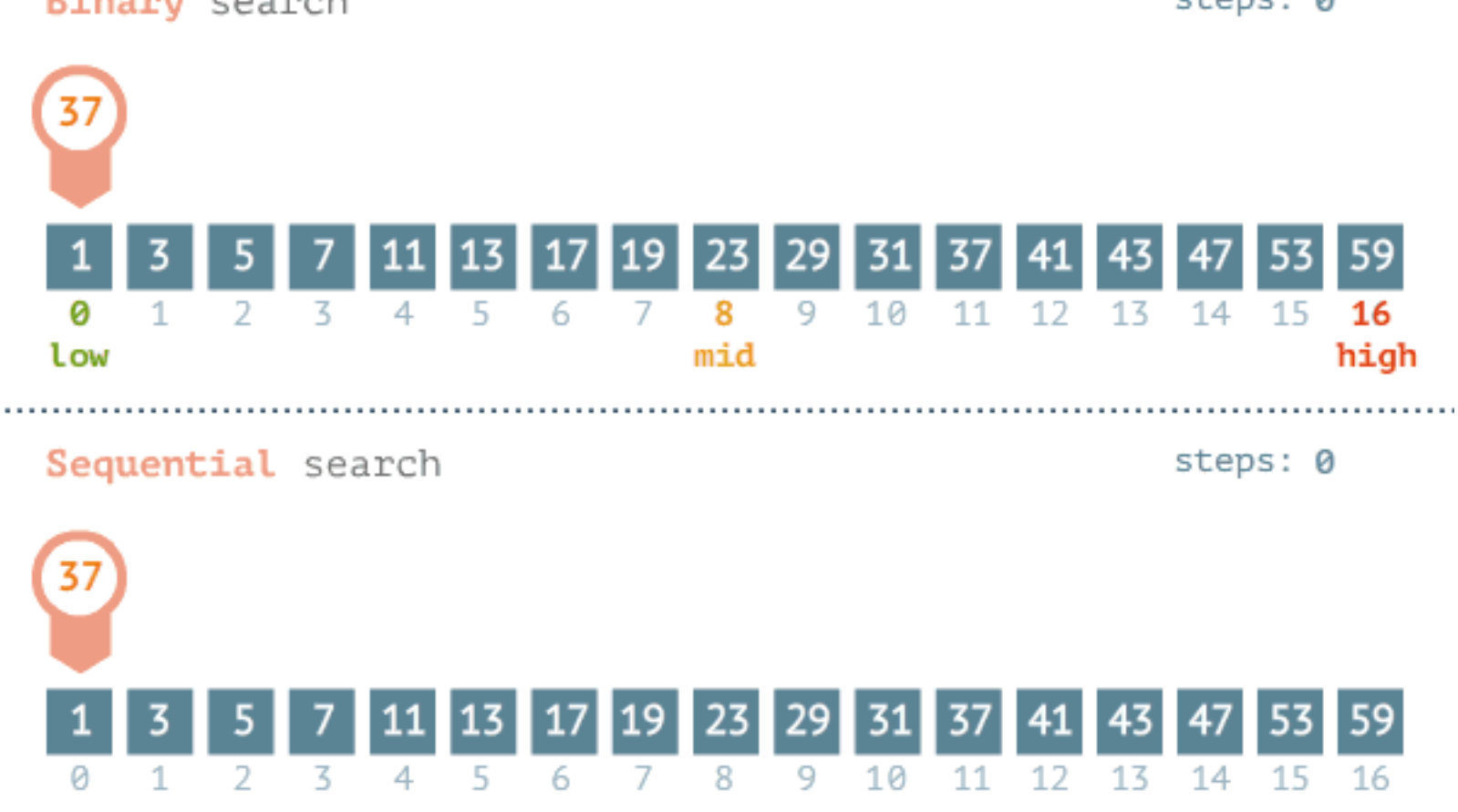
| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| value | -4 | 2 | 7 | 10 | 15 | 20 | 22 | 25 | 30 | 36 | 42 | 50 | 56 | 68 | 85 | 92 | 103 |

1) Look at the middle of the array. If the target is found, we are done. Otherwise, If the target is greater than that value, we can eliminate the left half of the array. And If the target is less than the value, eliminate the right half.
2) Repeat with left or right half of the array accordingly.

The next slide contains two animation explaining binary search vs linear(sequential) search.

Open the powerpoint version to see the animation. The PDF version of slides will NOT animate. See link below.

https://longbaonguyen.github.io/courses/apcsp/lect22.pptx




www.penjee.com

# Binary Search(optional)

Implement binary search. Data must be in sorted order to use the binary search algorithm.

```
def binary_search(sorted_lst, target):
      min, max = 0, len(sorted_lst)-1
      while min <= max:
          mid = (min + max)//2
          if sorted_lst[mid] < target:
              min = mid + 1
          elif sorted_lst[mid] > target:
              max = mid - 1
          elif sorted_lst[mid] == target:
              return mid
      return -1
```

# Algorithmic Efficiency

A **problem** is a general description of a task that can (or cannot) be solved algorithmically.

An **instance** of a problem also includes specific input. For example, sorting is a problem; sorting the list (2,3,1,7) is an instance of the problem.

A **decision problem** is a problem with a yes/no answer (e.g., is there a path from A to B?).

An **optimization problem** is a problem with the goal of finding the "best" solution among many (e.g., what is the shortest path from A to B?).

# Algorithmic Efficiency

**Efficiency** is an estimation of the amount of computational resources used by an algorithm.

Efficiency is typically expressed as a function of the size of the input(e.g the size of the list). Can either be worst-case complexity or average-case complexity.

An algorithm's efficiency can be informally measured by determining the number of times a statement or group of statements executes.

Different correct algorithms for the same problem can have different efficiencies. For example, a sorting algorithm that requires more computations is slower than a different sorting algorithm that requires less.

# Example 1 of Algorithmic Efficiency

```
def sum(lst):
        s = 0
        for x in lst:
            s += x
            s += 1
        return s
```

Let's define efficiency as the number of times a math operation statement is executed. Let the size of lst be n. What is the efficiency of the function sum as a function of n?

Answer: 2n

# Example 2 of Algorithmic Efficiency

```
def sum(lst):
        s = 0
        for x in lst:
            s += x
            s += 1
        for x in lst:
            s -= 1
        return s
```

Let's define efficiency as the number of times a math operation statement is executed. Let the size of lst be n. What is the efficiency of the function sum as a function of n?

Answer: 3n

# Example 3 of Algorithmic Efficiency

```
def sum(lst):
        s = 0
        for x in lst:
                for y in lst:
                        s += y
        return s
```

Let's define efficiency as the number of times a math operation statement is executed. Let the size of lst be n. What is the efficiency of the function sum as a function of n?

Answer: n*n = n^2

# Efficiency for Searching

Suppose we have a list of size n.

1) In the worst-case scenario, what is the number of comparisons needed to find the target using sequential or linear search?

Answer: n

2) In the worst-case scenario, what is the number of comparisons needed to find the target using binary search(when applied to a sorted list)?

Answer: Approximately log_2(n).

# Exponential Complexity Problems

Algorithms with a polynomial efficiency(constant, linear, square, cube, etc.) are said to run in a *reasonable amount of time*. They can be executed quickly on a modern processor.

However, there exists important and practical problems for which there exists no known polynomial time algorithm. Algorithms with exponential or factorial efficiencies are examples of algorithms that run in an *unreasonable amount of time*.

- For example, given a set of integers, find a subset that sums to zero. A brute-force algorithm would try every possible subset. But there are 2^n different subsets. This is an example of an exponential time algorithm. If n is large, even the fastest computers would take too long.

# Heuristic

Some problems cannot be solved in a reasonable amount of time because there is no efficient algorithm for solving them. In these cases, approximate solutions are sought.

A **heuristic** is an approach to a problem that produces a solution that is not guaranteed to be optimal but may be used when techniques that are guaranteed to always find an optimal solution are impractical.

For example, a file-organizing algorithm(sorting a folder based on file types e.g. pdf, docs, jpegs) determines the content of a file based on a certain number of bytes in the beginning of the file. This is an approximate solution since only a few bytes are examined. But it is more practical and faster to run than examining every byte of every file.

# Exponential Complexity Problems

Algorithms with a polynomial efficiency(constant, linear, square, cube, etc.) are said to run in a *reasonable amount of time*. They can be executed quickly on a modern processor.

However, there exists important and practical problems for which there exists no known polynomial time algorithm. Algorithms with exponential or factorial efficiencies are examples of algorithms that run in an *unreasonable amount of time*.

- For example, given a set of integers, find a subset that sums to zero. A brute-force algorithm would try every possible subset. But there are 2^n different subsets. This is an example of an exponential time algorithm. If n is large, even the fastest computers would take too long.

# Decidability

A **decidable problem** is a decision problem for which an algorithm can be written to produce a correct output for all inputs.

- E.g. Is the number even?

An **undecidable problem** is one for which no algorithm can be constructed that is always capable of providing a correct yes-or-no answer.

An undecidable problem may have some instances that have an algorithmic solution, but there is no algorithmic solution that could solve all instances of the problem.

Alan Turing, considered by many to be the father of computer science, proved that there exists undecidable problems. An example he posed is the Halting Problem.

# The Halting Problem(optional)

Can you write a program that takes the source code of another program and some input and returns whether the program will terminate(not go into an infinite loop) with the given input?

```
def halting(function, input):
    # returns whether the function terminates with
    # given input.
    # Is there an implementation of this function?
```

# Example(optional)

IF the halting function can be implemented, it will give the following outputs for the sum function(sum 1 to 10 with step size= increment).

```
def sum(increment):
        x = 1
        while x <= 10:
                x += increment
        return x
print(halting(sum, 1)) # True
print(halting(sum, -1)) # False (infinite loop)
```

Alan Turing proved that such a function(halting) does not exist(cannot be implemented). Alan Turing is portrayed by the incredible Benedict Cumberbatch in the movie "The Imitation Game".

# Algorithms to know for AP Exam

You should know how to implement(write the code for) the following algorithms. The AP exam may give you the code for an algorithm and ask you to find the error or explain what it does.

1) Finding an item in a list(sequential search).
   a) Given a list and an item, return whether(True or False) the item is in the list
   b) Give a list an an item, return the index of the item in the list(-1 if not found).
2) Compute the sum or average of a list of numbers.
3) Find the minimum or maximum value of a list of numbers.

# Simulation

Computer simulation is the process of mathematical modeling, performed on a computer, which is designed to predict the behavior of, or the outcome of, a real-world or physical system.

Simulations often mimic real-world events with the purpose of drawing inferences, allowing investigation of a phenomenon without the constraints of the real world.

Simulations are most useful when real-world events are impractical for experiments (e.g., too big, too small, too fast, too slow, too expensive, or too dangerous).

For example, instead of letting an untrained pilot fly an actual plane, the pilot can learn by using a flight simulator.

# Simulation

The process of developing an abstract simulation involves removing specific details or simplifying functionality.

Simulations can contain bias derived from the choices of real-world elements that were included or excluded.

Simulations facilitate the formulation and refinement of hypotheses related to the objects or phenomena under consideration.

Random number generators can be used to simulate the variability that exists in the real world.