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.

```python
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.

```python
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.

```python
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.

Binary search

37

1	3	5	7	11	13	17	19	23	29	31	37	41	43	47	53	59
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

low   mid   high

Sequential search

steps: 0

37

1	3	5	7	11	13	17	19	23	29	31	37	41	43	47	53	59
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

www.penjee.com

Binary Search

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

```python
def binary_search(sorted_lst, target):
    min, max = 0, len(sorted_list)-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
```
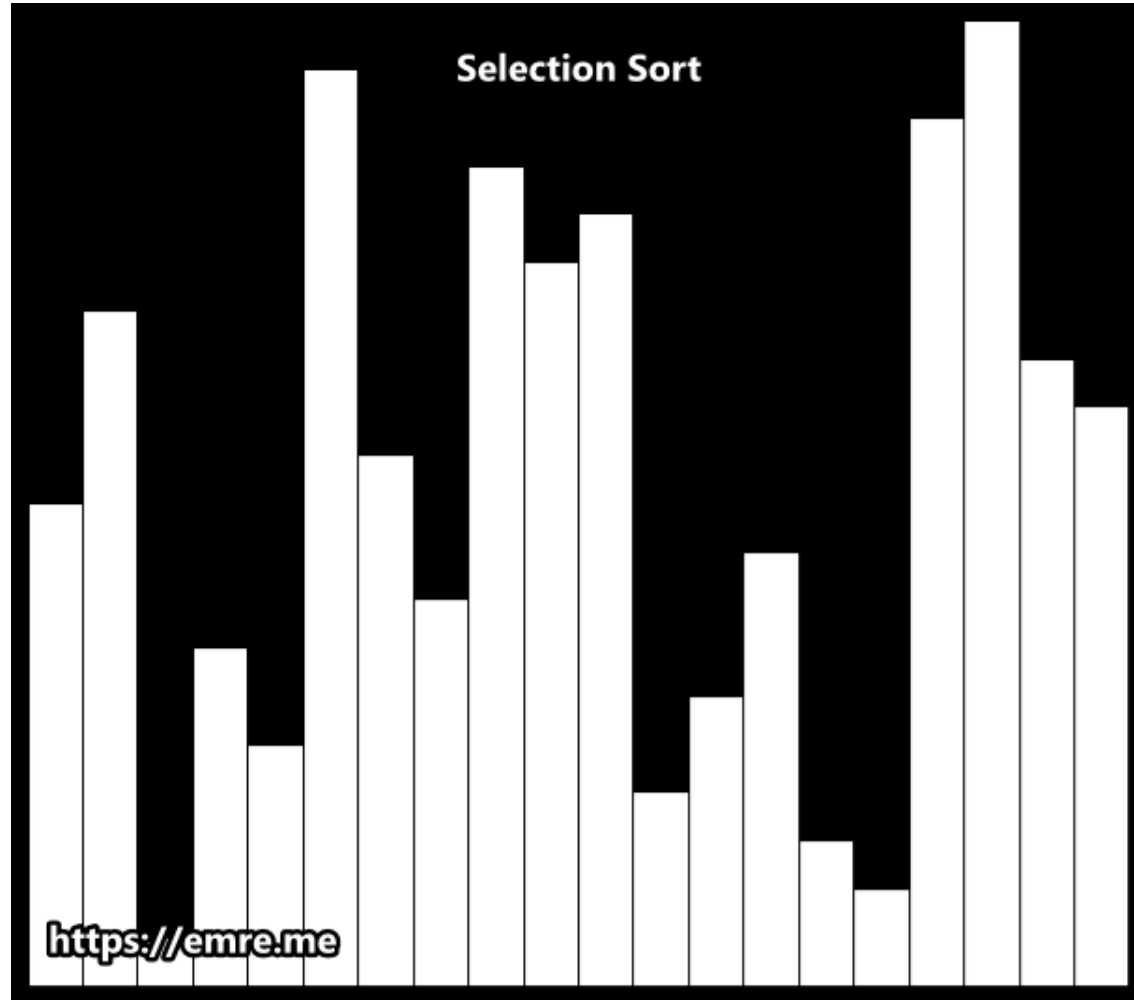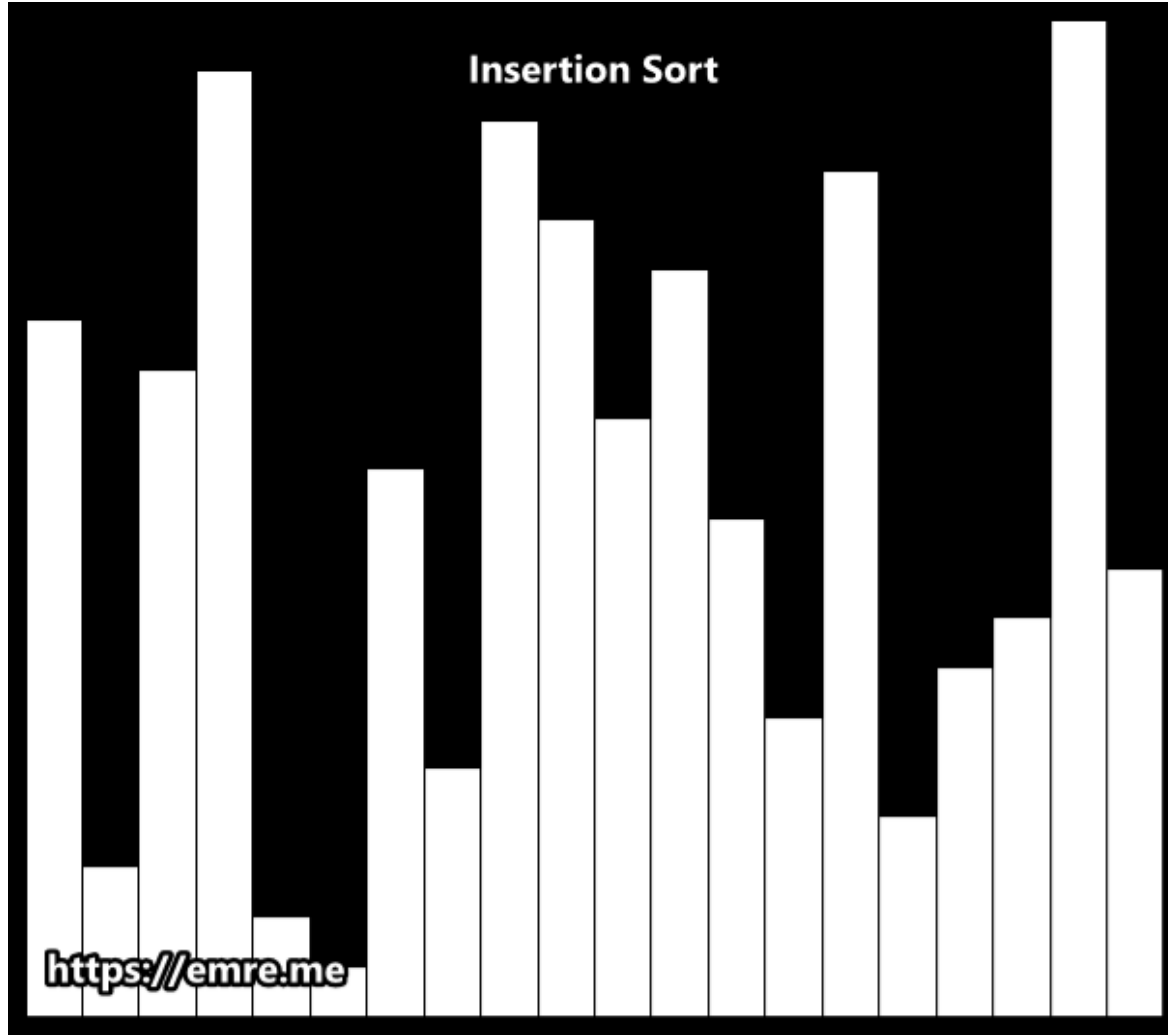
Sorting(not on AP)

sorting: Rearranging the values in an array or collection into a specific order (usually into their "natural ordering").

- one of the fundamental problems in computer science
  - sorts products on Amazon based on ratings or price
  - sorts contacts on phone in alphabetical order

- can be solved in many ways:
  - there are many sorting algorithms
  - some are faster/slower than others
  - some use more/less memory than others
  - some work better with specific kinds of data
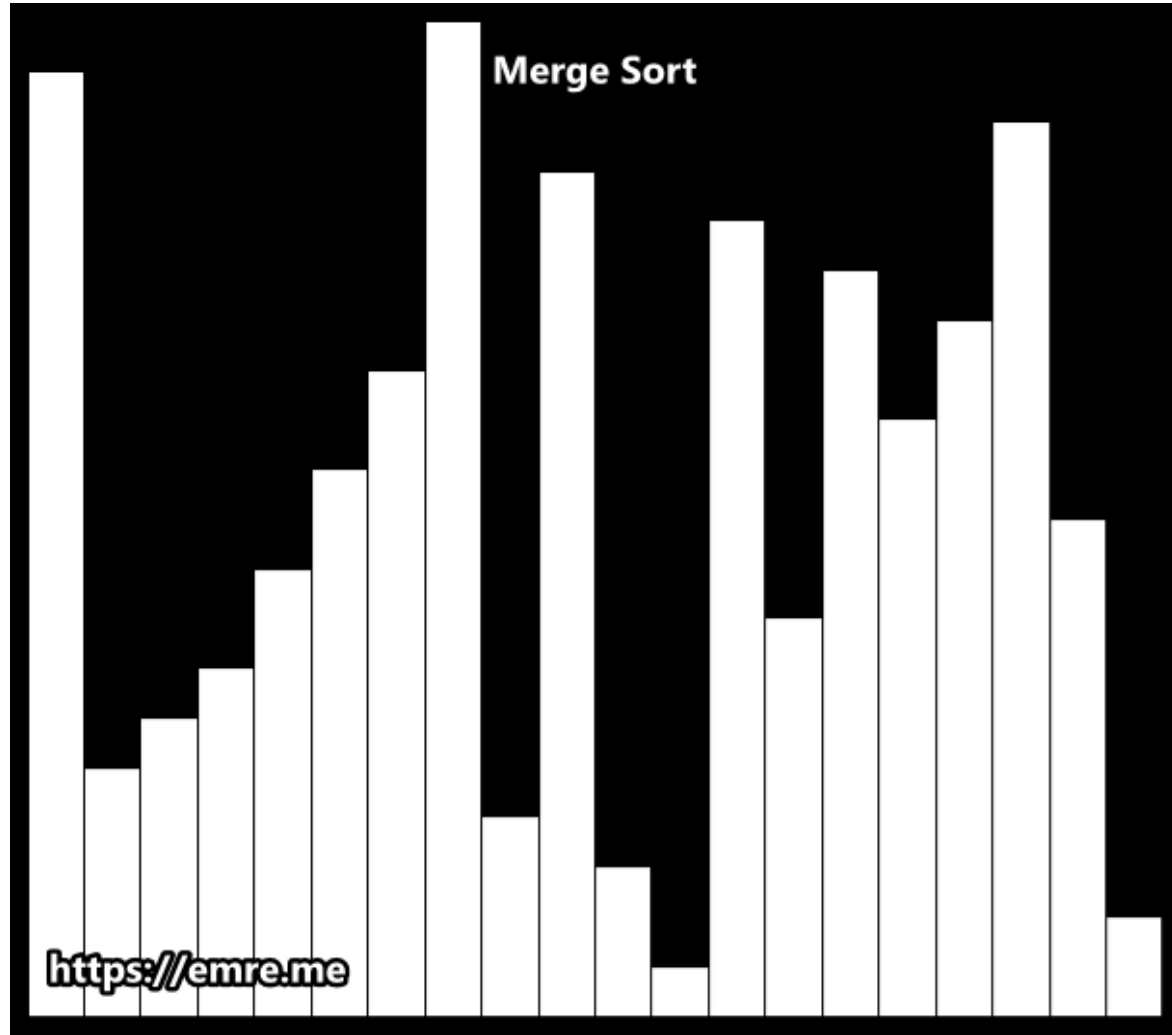 - some can utilize multiple computers / processors, ...

Sorting(not on AP)

Sorting(not on AP)

Insertion Sort

https://emre.me

Sorting(not on AP)

Merge Sort

https://emre.me

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.

Algorithmic Efficiency

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.

Example 1 of Algorithmic Efficiency

```python
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

```python
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

```python
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 $\log_2(n)$.

Note: The AP exam only requires that you know binary search is generally faster than linear search when applied to a sorted list but not how much faster.

Computational Complexity for Sorting(not on AP exam)

Suppose we have a list of size n.

1) What is the approximate number of comparisons needed for selection sort?

Answer: The implementation of this algorithm uses 2 nested loops, with each loops running n times. Thus, approximately $n*n = n^2$.

2) What is the approximate number of comparisons needed for insertion sort?

Answer: Similar to selection sort, the code uses 2 nested loops, with each loops running n times. Thus, approximately $n*n = n^2$.

3) What is the number of comparisons needed for mergesort?

Answer: Approximately $n*\log(n)$. (slower than linear search(n) but faster than both selection and insertion sort(n^2).
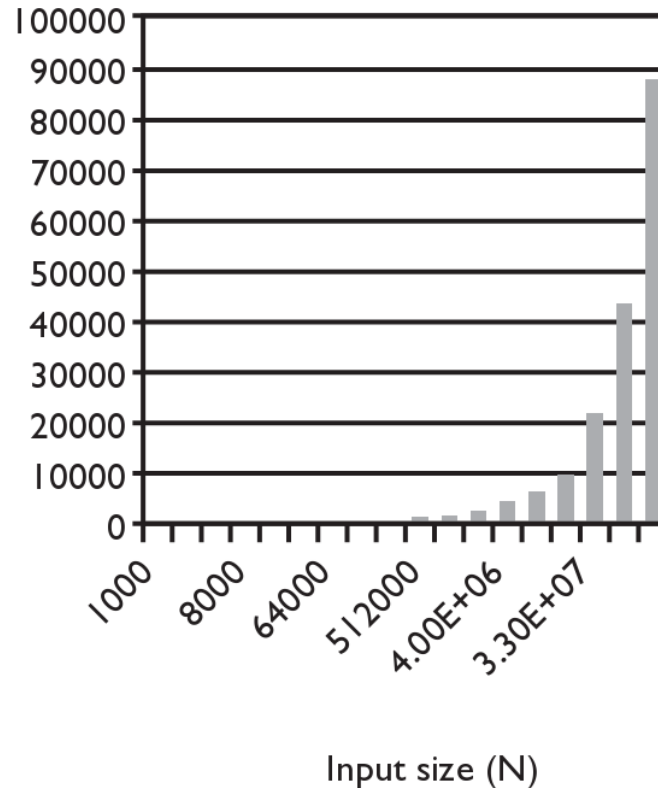
Selection sort runtime(not on AP exam)

N	Runtime (ms)
1000	0
2000	16
4000	47
8000	234
16000	657
32000	2562
64000	10265
128000	41141
256000	164985

Input size (N)
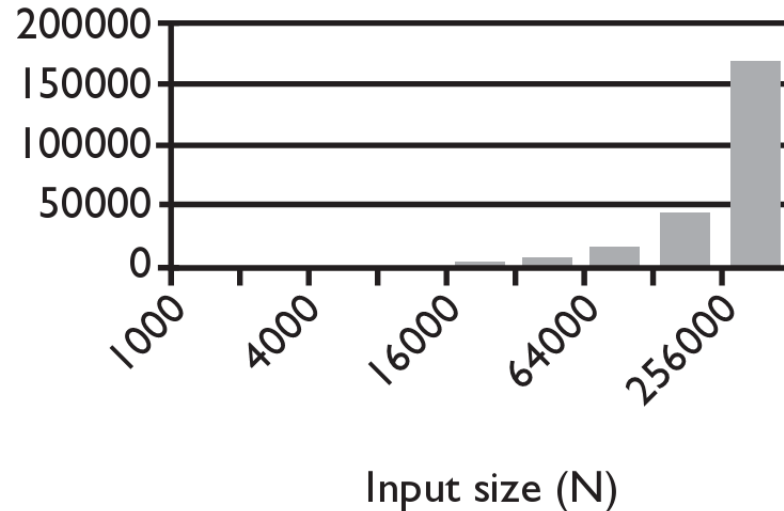
Merge sort runtime(not on AP exam)

N	Runtime (ms)
1000	0
2000	0
4000	0
8000	0
16000	0
32000	15
64000	16
128000	47
256000	125
512000	250
1e6	532
2e6	1078
4e6	2265
8e6	4781
1.6e7	9828
3.3e7	20422
6.5e7	42406
1.3e8	88344

Input size (N)

For a list of length 256000 items, selection sort takes about 2.75 minutes but mergesort takes 1/10 of a second.

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.

Abstraction

Abstraction is the process of reducing complexity by focusing on the main idea.

By hiding details irrelevant to the question at hand and bringing together related and useful details, abstraction reduces complexity and allows one to focus on the idea.

One common type of abstraction is **procedural abstraction**, which provides a name for a procedure(function) and allows it to be used only knowing what it does, not how it does it.

```
import random
print(random.randrange(10)) # random number from 0 - 9
```

We don't need to know how the randrange() function is implemented to be able use it.

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

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

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).

```python
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.