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.

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


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.

Binary 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

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

Sorting

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

- 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, ...

Selection Sort

https://emre.me

Insertion Sort

https://emre.me

Merge Sort

https://emre.me
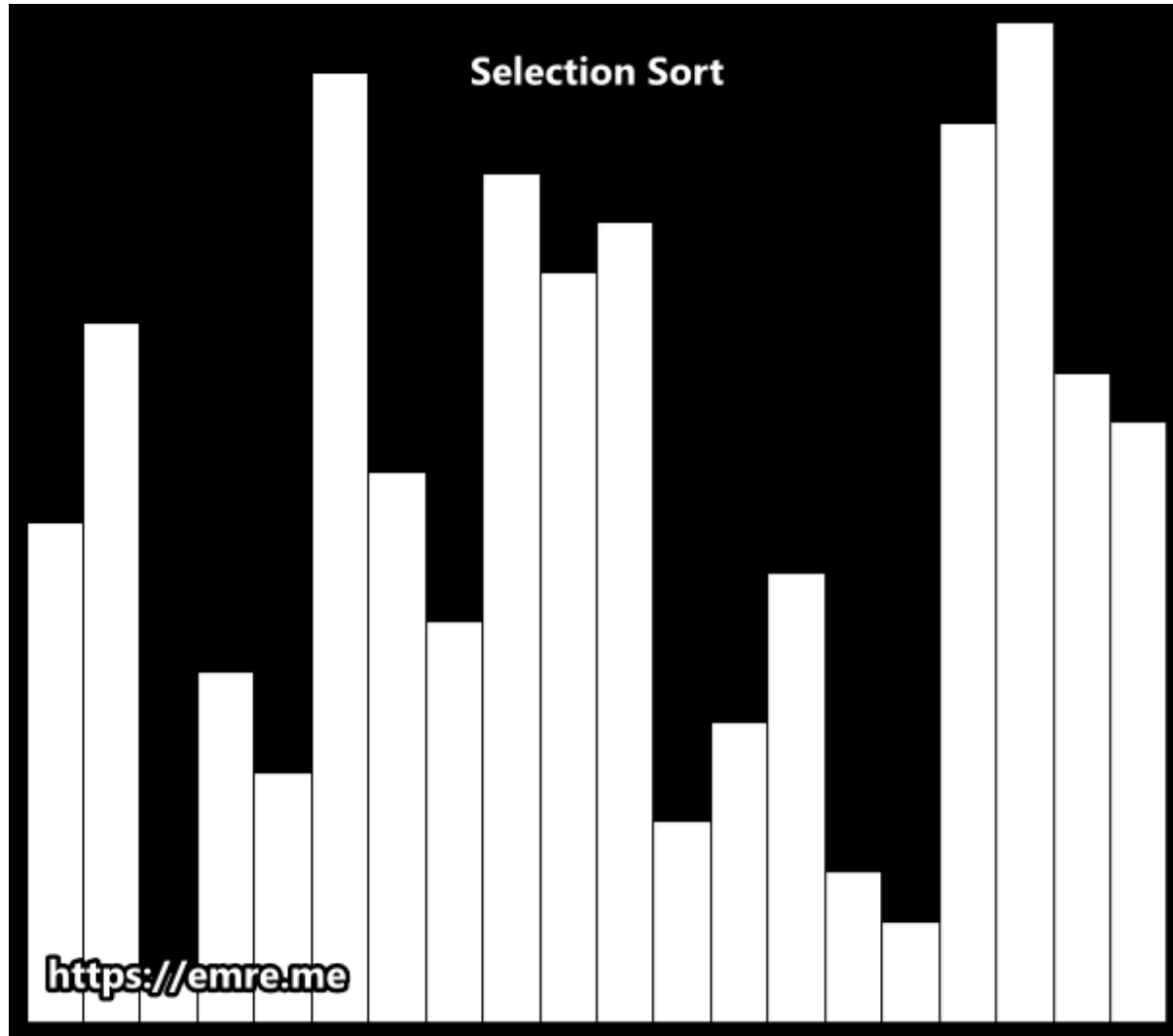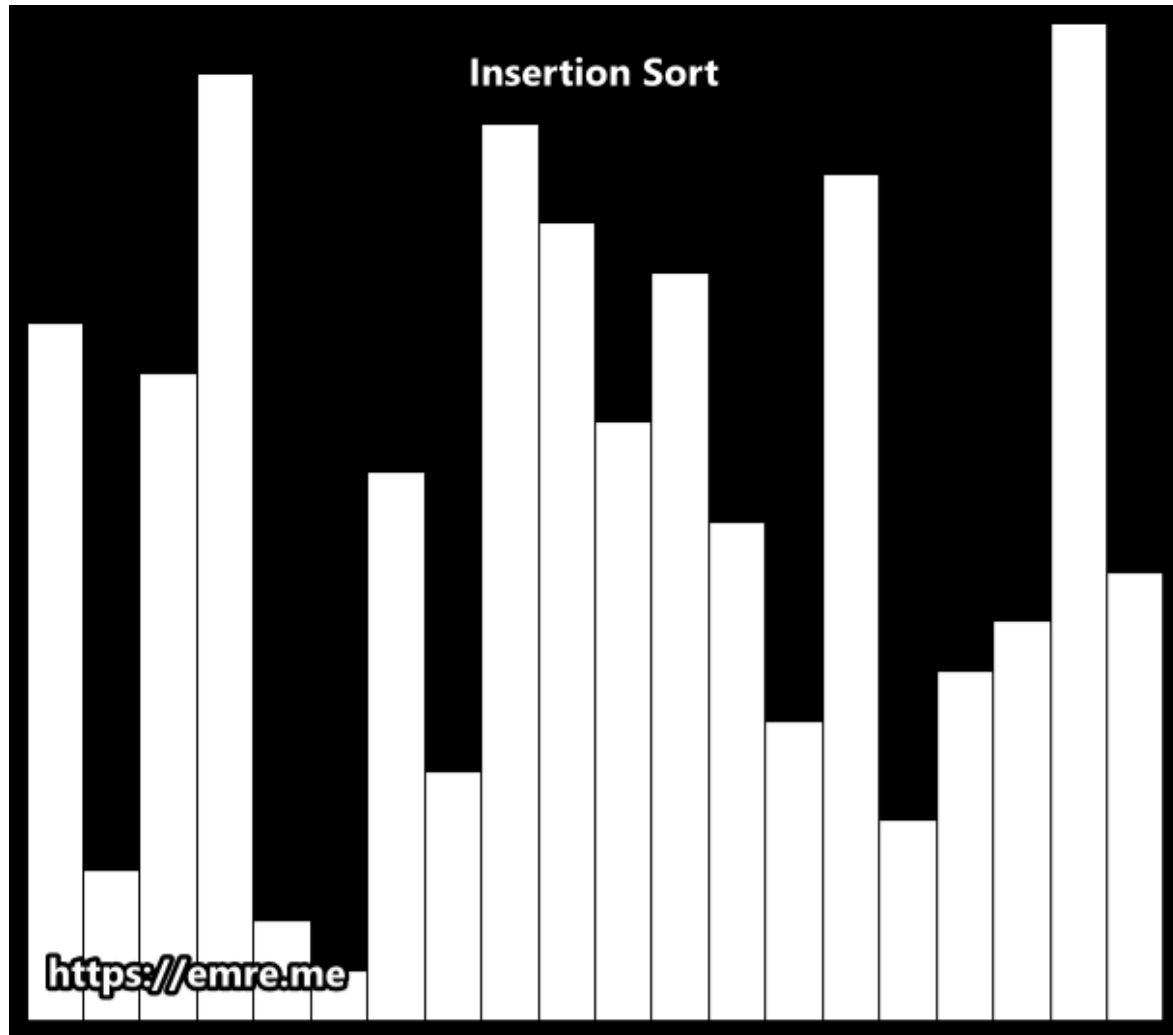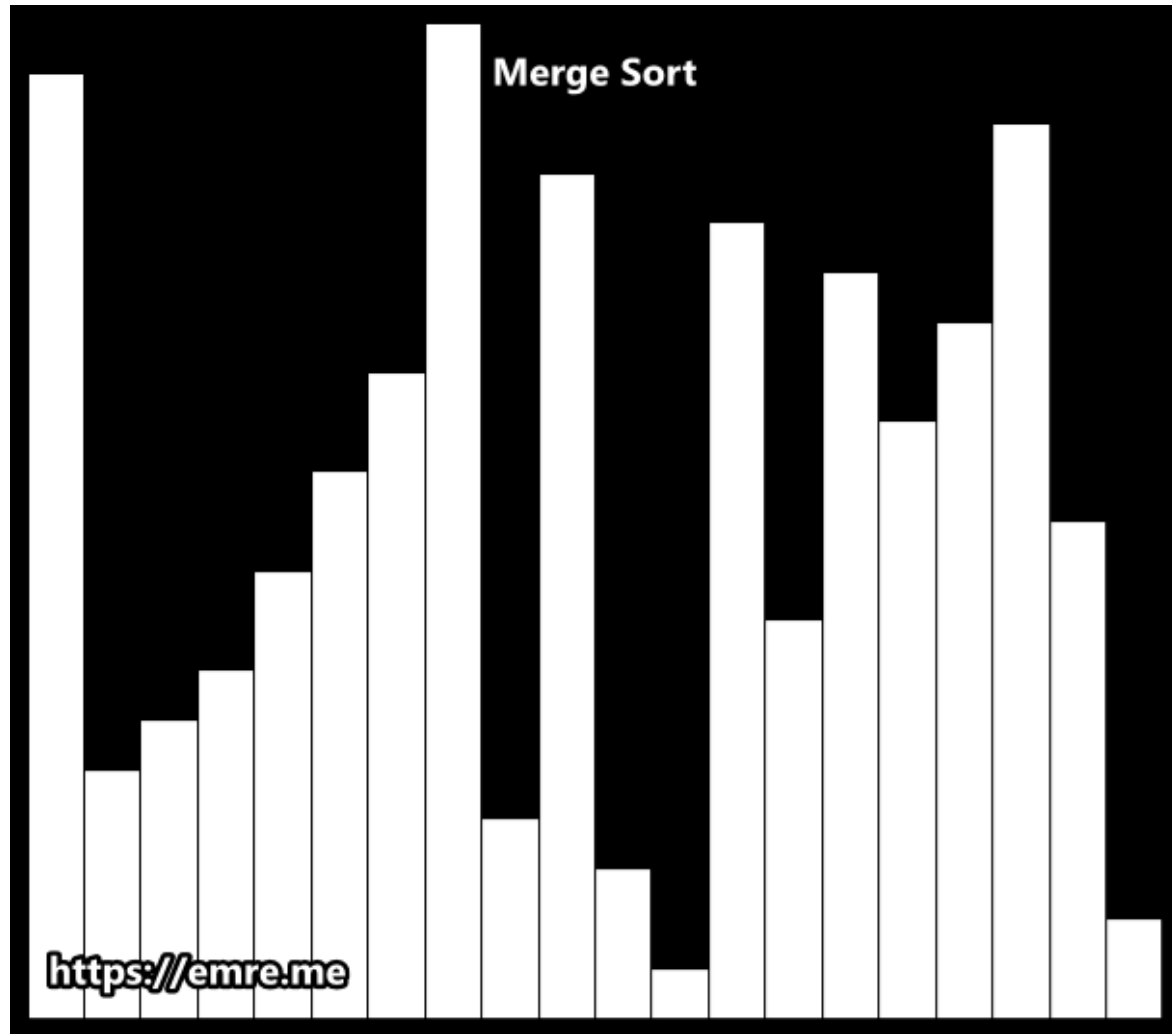
Binary Search

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
```

Top-Down Design

Top-down design is the name given to breaking a problem down into smaller and more manageable parts(also known as **decomposition**).

If any of these subproblems is not easier to solve, we further divide the subproblem into smaller parts. The subdivision of a computer program into separate subprograms is called **modularity.**

We repeat the subdividing process until all small parts are not dividable. Then, we can use a few lines of code to solve every trivial problem. In the end, we put all these little pieces together as a solution to the original problem.

This approach makes it easy to think about the problem, code the solution, read the code, and identify bugs.

Abstraction

Abstraction means displaying only essential information and hiding the details.

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.

When implementing algorithms, procedural abstraction allows a solution to a large problem to be based on the solutions of smaller subproblems.

Index of Two Smallest Values

Given a list of numbers, find the **indices** of the two smallest numbers in the list.

We'll implement two algorithms to solve the above problem. We'll use top-down design by first using pseudocode and break down problems into smaller and smaller parts until the pseudocode is easily translated into Python code.

We'll use both built-in functions(**min, sorted**) and list methods(`index, remove, insert`) to abstract away some of the smaller tasks in the problem(procedure abstraction).

Index of Two Smallest Values

Given a list of numbers, find the **indices** of the two smallest numbers in the list.

Jumping right into coding this problem without planning will be tricky.

Let's think carefully how we would solve this by writing out the pseudocode.

```python
def two_smallest(lst):
    # Find the index of the minimum item in lst
    # Remove that item from the list
    # Find the index of the new minimum item in the list
    # Put the smallest item back in the list
    # If necessary, adjust the second index
    # Return the two indices
```

```python
def two_smallest(lst):
    # Find the index of the minimum item in lst
    smallest = min(lst)
    min1 = lst.index(smallest)
    # Remove that item from the list
    lst.remove(smallest)
    # Find the index of the new minimum item in the list
    next_smallest = min(lst)
    min2 = lst.index(next_smallest)
    # Put the smallest item back in the list
    lst.insert(min1, smallest)
    # If necessary, adjust the second index
    if min1 <= min2:
        min2 += 1
    # Return the two indices
    return (min1, min2)
```

Here's another algorithm to find the indices of the two smallest numbers.

```python
def two_smallest(lst):
        # Sort a copy of lst
        sorted_list = sorted(lst)
        # Get the two smallest numbers
        small1 = sorted_list[0]
        small2 = sorted_list[1]
        # Find their indices in the original list lst
        index1 = lst.index(small1)
        index2 = lst.index(small2)
        # Return the two indices
        return index1, index2
```

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.

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?

Answer: Approximately $\log_2(n)$.

Computational Complexity for Sorting

Suppose we have a list of size n.

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

Answer: Approximately n^2.

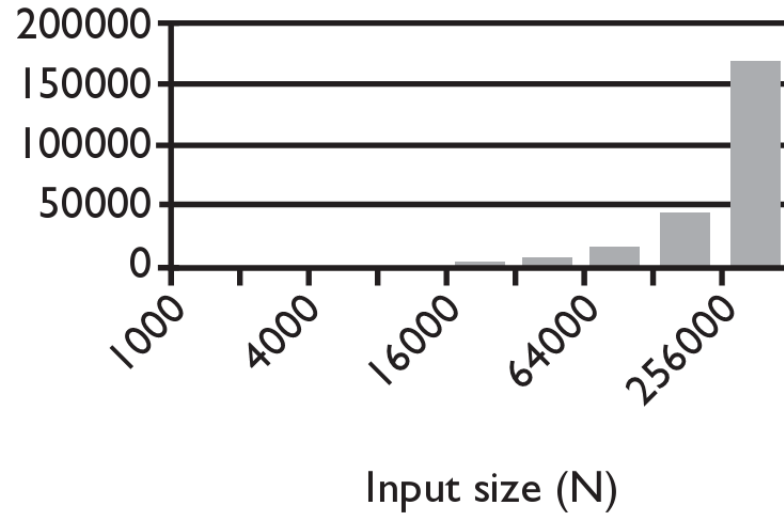2) What is the approximate number of comparisons needed for selection sort?

Answer: Approximately n^2.

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

Answer: Approximately n*log(n).

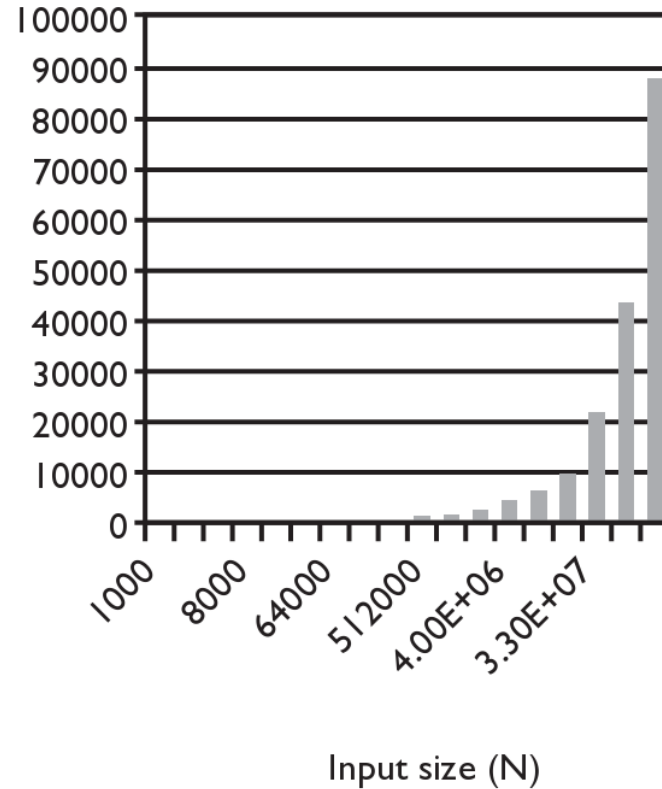# Selection sort runtime

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

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)

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

Here's a problem: In other words, 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?

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

The Halting Problem

Here's a problem: In other words, can you write a function that takes the source code of another function and some input and returns whether the function will terminate(not go into an infinite loop) with the given input?

```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.