UIT2201 Programming and Data Structures

Sorting Algorithms

Chandrabose Aravindan
<AravindanC@ssn.edu.in>

Professor of Information Technology
SSN College of Engineering

August 06, 2022

Introduction

- A sequence of objects can be arranged according to an order imposed by a partial order $\leq$

Introduction

- A sequence of objects can be arranged according to an order imposed by a partial order $\leq$
- A partial order is a relation among the objects which is reflexive, anti-symmetric, and transitive

Introduction

- A sequence of objects can be arranged according to an order imposed by a partial order $\leq$
- A partial order is a relation among the objects which is reflexive, anti-symmetric, and transitive
- A sequence is sorted when every element $x_i \leq x_{i+1}$ (except for the last element!)

A trivial sorting algorithm

```
def isSorted(lst):
    n = len(lst)
    for i in range(n-1):
        if ( lst[i] > lst[i+1] ):
            return False
    return True
```

A trivial sorting algorithm

```
def isSorted(lst):
    n = len(lst)
    for i in range(n-1):
        if ( lst[i] > lst[i+1] ):
            return False
    return True

from itertools import permutations

def permSort(lst):
    for l in permutations(lst):
        res = list(l)
        if isSorted(res):
            return res
```

Inversions

- A sequence of objects can be arranged according to an order imposed by a partial order $\leq$
- A partial order is a relation among the objects which is reflexive, anti-symmetric, and transitive
- Given a sequence of objects, out-of-the-order pairs are known as inversions

Inversions

- A sequence of objects can be arranged according to an order imposed by a partial order $\leq$
- A partial order is a relation among the objects which is reflexive, anti-symmetric, and transitive
- Given a sequence of objects, out-of-the-order pairs are known as inversions
- For any two indices i and j, if $i < j$ and $A[i] > A[j]$ then the pair $(A[i], A[j])$ is an inversion

Inversions

- A sequence of objects can be arranged according to an order imposed by a partial order $\leq$
- A partial order is a relation among the objects which is reflexive, anti-symmetric, and transitive
- Given a sequence of objects, out-of-the-order pairs are known as inversions
- For any two indices i and j, if $i < j$ and $A[i] > A[j]$ then the pair $(A[i], A[j])$ is an inversion
- A sequence is sorted when there are no inversions in it

Inversions

- A sequence of objects can be arranged according to an order imposed by a partial order $\leq$
- A partial order is a relation among the objects which is reflexive, anti-symmetric, and transitive
- Given a sequence of objects, out-of-the-order pairs are known as inversions
- For any two indices i and j, if $i < j$ and $A[i] > A[j]$ then the pair $(A[i], A[j])$ is an inversion
- A sequence is sorted when there are no inversions in it
- Algorithm Idea: Systematically check all the pairs and "correct" the inversions, if any

Basic Sorting Algorithms

- Brute-force approach — exhaustive search (search all the pairs)
  - Selection Sort
  - Bubble Sort
- Divide-Conquer-Merge (Does it help in reducing the number of comparisons?)
  - Insertion Sort
  - Merge Sort
  - Quick Sort
- Hybrid approaches

Selection Sort

- Algorithm idea: In the first pass, select the smallest object and bring it to the beginning of the list. In the next pass, next smallest object should be selected as the second object, and so on.

- Algorithm idea: In the first pass, select the smallest object and bring it to the beginning of the list. In the next pass, next smallest object should be selected as the second object, and so on.

```
def selection_sort(lst):
    n = len(lst)
    for i in range(n-1):
        for j in range(i+1, n):
            if (lst[i] > lst[j]):
                lst[i], lst[j] = lst[j], lst[i]
    return
```

Selection Sort

- Consider the sequence $[32, 108, 14, 72, 12]$

Selection Sort

- Consider the sequence $[32, 108, 14, 72, 12]$
- In the first pass, index $i = 0$ and index j runs from 1 to 4

Selection Sort

- Consider the sequence $[32, 108, 14, 72, 12]$
- In the first pass, index $i = 0$ and index j runs from 1 to 4
- $[32, 108, 14, 72, 12]$

Selection Sort

- Consider the sequence $[32, 108, 14, 72, 12]$
- In the first pass, index $i = 0$ and index j runs from 1 to 4
- $[32, 108, 14, 72, 12]$
- $[32, 108, 14, 72, 12]$

Selection Sort

- Consider the sequence $[32, 108, 14, 72, 12]$
- In the first pass, index $i = 0$ and index j runs from 1 to 4
- $[32, 108, 14, 72, 12]$
- $[32, 108, 14, 72, 12]$
- $[14, 108, 32, 72, 12]$

Selection Sort

- Consider the sequence $[32, 108, 14, 72, 12]$
- In the first pass, index $i = 0$ and index j runs from 1 to 4
- $[32, 108, 14, 72, 12]$
- $[32, 108, 14, 72, 12]$
- $[14, 108, 32, 72, 12]$
- $[14, 108, 32, 72, 12]$

Selection Sort

- Consider the sequence $[32, 108, 14, 72, 12]$
- In the first pass, index $i = 0$ and index j runs from 1 to 4
- $[32, 108, 14, 72, 12]$
- $[32, 108, 14, 72, 12]$
- $[14, 108, 32, 72, 12]$
- $[14, 108, 32, 72, 12]$
- $[12, 108, 32, 72, 14]$

Selection Sort

- Consider the sequence $[32, 108, 14, 72, 12]$
- In the first pass, index $i = 0$ and index j runs from 1 to 4
- $[32, 108, 14, 72, 12]$
- $[32, 108, 14, 72, 12]$
- $[14, 108, 32, 72, 12]$
- $[14, 108, 32, 72, 12]$
- $[12, 108, 32, 72, 14]$
- After the second pass, we will have

Selection Sort

- Consider the sequence $[32, 108, 14, 72, 12]$
- In the first pass, index $i = 0$ and index j runs from 1 to 4
- $[32, 108, 14, 72, 12]$
- $[32, 108, 14, 72, 12]$
- $[14, 108, 32, 72, 12]$
- $[14, 108, 32, 72, 12]$
- $[12, 108, 32, 72, 14]$
- After the second pass, we will have $[12, 14, 108, 72, 32]$

Selection Sort

- Consider the sequence $[32, 108, 14, 72, 12]$
- In the first pass, index $i = 0$ and index j runs from 1 to 4
- $[32, 108, 14, 72, 12]$
- $[32, 108, 14, 72, 12]$
- $[14, 108, 32, 72, 12]$
- $[14, 108, 32, 72, 12]$
- $[12, 108, 32, 72, 14]$
- After the second pass, we will have $[12, 14, 108, 72, 32]$
- After the third pass, we will have

- Consider the sequence $[32, 108, 14, 72, 12]$
- In the first pass, index $i = 0$ and index j runs from 1 to 4
- $[32, 108, 14, 72, 12]$
- $[32, 108, 14, 72, 12]$
- $[14, 108, 32, 72, 12]$
- $[14, 108, 32, 72, 12]$
- $[12, 108, 32, 72, 14]$
- After the second pass, we will have $[12, 14, 108, 72, 32]$
- After the third pass, we will have $[12, 14, 32, 108, 72]$

Selection Sort

- Consider the sequence $[32, 108, 14, 72, 12]$
- In the first pass, index $i = 0$ and index j runs from 1 to 4
- $[32, 108, 14, 72, 12]$
- $[32, 108, 14, 72, 12]$
- $[14, 108, 32, 72, 12]$
- $[14, 108, 32, 72, 12]$
- $[12, 108, 32, 72, 14]$
- After the second pass, we will have $[12, 14, 108, 72, 32]$
- After the third pass, we will have $[12, 14, 32, 108, 72]$
- After the fourth pass, we will have

Selection Sort

- Consider the sequence $[32, 108, 14, 72, 12]$
- In the first pass, index $i = 0$ and index j runs from 1 to 4
- $[32, 108, 14, 72, 12]$
- $[32, 108, 14, 72, 12]$
- $[14, 108, 32, 72, 12]$
- $[14, 108, 32, 72, 12]$
- $[12, 108, 32, 72, 14]$
- After the second pass, we will have $[12, 14, 108, 72, 32]$
- After the third pass, we will have $[12, 14, 32, 108, 72]$
- After the fourth pass, we will have $[12, 14, 32, 72, 108]$

Selection Sort

- It is trivial to reduce the number of swaps by choosing the minimum first and then swap

Selection Sort

- It is trivial to reduce the number of swaps by choosing the minimum first and then swap

```
def selection_sort(lst):
    n = len(lst)
    for i in range(n-1):
        min_index = i
        for j in range(i+1, n):
            if ( lst[min_index] > lst[j] ):
                min_index = j
        if ( i != min_index ):
            lst[i], lst[min_index] = lst[min_index], l
    return
```

Selection Sort

- Consider the same sequence $[32, 108, 14, 72, 12]$

Selection Sort

- Consider the same sequence $[32, 108, 14, 72, 12]$
- In the first pass, index $i = 0$ and index j runs from 1 to 4

Selection Sort

- Consider the same sequence $[32, 108, 14, 72, 12]$
- In the first pass, index $i = 0$ and index j runs from 1 to 4
- $[\underline{32}, 108, 14, 72, 12]$

Selection Sort

- Consider the same sequence $[32, 108, 14, 72, 12]$
- In the first pass, index $i = 0$ and index j runs from 1 to 4
- $[\underline{32}, 108, 14, 72, 12]$
- $[32, 108, \underline{14}, 72, 12]$

Selection Sort

- Consider the same sequence $[32, 108, 14, 72, 12]$
- In the first pass, index $i = 0$ and index j runs from 1 to 4
- $[\underline{32}, 108, 14, 72, 12]$
- $[32, 108, \underline{14}, 72, 12]$
- $[32, 108, \underline{14}, 72, 12]$

Selection Sort

- Consider the same sequence $[32, 108, 14, 72, 12]$
- In the first pass, index $i = 0$ and index j runs from 1 to 4
- $[\underline{32}, 108, 14, 72, 12]$
- $[32, 108, \underline{14}, 72, 12]$
- $[32, 108, \underline{14}, 72, 12]$
- $[32, 108, 14, 72, \underline{12}]$

Selection Sort

- Consider the same sequence $[32, 108, 14, 72, 12]$
- In the first pass, index $i = 0$ and index j runs from 1 to 4
- $[\underline{32}, 108, 14, 72, 12]$
- $[32, 108, \underline{14}, 72, 12]$
- $[32, 108, \underline{14}, 72, 12]$
- $[32, 108, 14, 72, \underline{12}]$
- $[12, 108, 14, 72, 32]$

Selection Sort

- Consider the same sequence $[32, 108, 14, 72, 12]$
- In the first pass, index $i = 0$ and index j runs from 1 to 4
- $[\underline{32}, 108, 14, 72, 12]$
- $[32, 108, \underline{14}, 72, 12]$
- $[32, 108, \underline{14}, 72, 12]$
- $[32, 108, 14, 72, \underline{12}]$
- $[12, 108, 14, 72, 32]$
- After the second pass, we will have

Selection Sort

- Consider the same sequence $[32, 108, 14, 72, 12]$
- In the first pass, index $i = 0$ and index j runs from 1 to 4
- $[\underline{32}, 108, 14, 72, 12]$
- $[32, 108, \underline{14}, 72, 12]$
- $[32, 108, \underline{14}, 72, 12]$
- $[32, 108, 14, 72, \underline{12}]$
- $[12, 108, 14, 72, 32]$
- After the second pass, we will have $[12, 14, 108, 72, 32]$

Selection Sort

- Consider the same sequence $[32, 108, 14, 72, 12]$
- In the first pass, index $i = 0$ and index j runs from 1 to 4
- $[\underline{32}, 108, 14, 72, 12]$
- $[32, 108, \underline{14}, 72, 12]$
- $[32, 108, \underline{14}, 72, 12]$
- $[32, 108, 14, 72, \underline{12}]$
- $[12, 108, 14, 72, 32]$
- After the second pass, we will have $[12, 14, 108, 72, 32]$
- After the third pass, we will have

Selection Sort

- Consider the same sequence $[32, 108, 14, 72, 12]$
- In the first pass, index $i = 0$ and index j runs from 1 to 4
- $[\underline{32}, 108, 14, 72, 12]$
- $[32, 108, \underline{14}, 72, 12]$
- $[32, 108, \underline{14}, 72, 12]$
- $[32, 108, 14, 72, \underline{12}]$
- $[12, 108, 14, 72, 32]$
- After the second pass, we will have $[12, 14, 108, 72, 32]$
- After the third pass, we will have $[12, 14, 32, 72, 108]$

- Consider the same sequence $[32, 108, 14, 72, 12]$
- In the first pass, index $i = 0$ and index j runs from 1 to 4
- $[\underline{32}, 108, 14, 72, 12]$
- $[32, 108, \underline{14}, 72, 12]$
- $[32, 108, \underline{14}, 72, 12]$
- $[32, 108, 14, 72, \underline{12}]$
- $[12, 108, 14, 72, 32]$
- After the second pass, we will have $[12, 14, 108, 72, 32]$
- After the third pass, we will have $[12, 14, 32, 72, 108]$
- After the fourth pass, we will have

- Consider the same sequence $[32, 108, 14, 72, 12]$
- In the first pass, index $i = 0$ and index j runs from 1 to 4
- $[\underline{32}, 108, 14, 72, 12]$
- $[32, 108, \underline{14}, 72, 12]$
- $[32, 108, \underline{14}, 72, 12]$
- $[32, 108, 14, 72, \underline{12}]$
- $[12, 108, 14, 72, 32]$
- After the second pass, we will have $[12, 14, 108, 72, 32]$
- After the third pass, we will have $[12, 14, 32, 72, 108]$
- After the fourth pass, we will have $[12, 14, 32, 72, 108]$

Selection Sort — Analysis

$$T(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$

Selection Sort — Analysis

$$T(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$

$$= \sum_{i=0}^{n-2} (n-1) - (i+1) + 1$$

Selection Sort — Analysis

$$T(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$

$$= \sum_{i=0}^{n-2} (n-1) - (i+1) + 1$$

$$= \sum_{i=0}^{n-2} (n-i-1)$$

Selection Sort — Analysis

$$T(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$

$$= \sum_{i=0}^{n-2} (n-1) - (i+1) + 1$$

$$= \sum_{i=0}^{n-2} (n-i-1)$$

$$= (n-1) + (n-2) + \cdots + 1$$

Selection Sort — Analysis

$$T(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$

$$= \sum_{i=0}^{n-2} (n-1) - (i+1) + 1$$

$$= \sum_{i=0}^{n-2} (n-i-1)$$

$$= (n-1) + (n-2) + \cdots + 1$$

$$= \frac{n(n-1)}{2}$$

$$T(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$

$$= \sum_{i=0}^{n-2} (n-1) - (i+1) + 1$$

$$= \sum_{i=0}^{n-2} (n-i-1)$$

$$= (n-1) + (n-2) + \cdots + 1$$

$$= \frac{n(n-1)}{2}$$

Time complexity is $\Theta(n^2)$

Bubble Sort

- An alternate idea: In the first pass, keep comparing the adjacent objects, and swap if they are out of order

Bubble Sort

- An alternate idea: In the first pass, keep comparing the adjacent objects, and swap if they are out of order
- At the end of the first pass, the largest object bubbles to the last position. In the second pass, second largest object bubbles, and so on.

Bubble Sort

- An alternate idea: In the first pass, keep comparing the adjacent objects, and swap if they are out of order
- At the end of the first pass, the largest object bubbles to the last position. In the second pass, second largest object bubbles, and so on.

```
def bubble_sort_(lst):
    n = len(lst)
    for i in range(n-1):
        for j in range(n-1-i):
            if (lst[j] > lst[j+1]):
                lst[j], lst[j+1] = lst[j+1], lst[j]
    return
```

Bubble Sort

- Consider the same sequence $[32, 108, 14, 72, 12]$

Bubble Sort

- Consider the same sequence $[32, 108, 14, 72, 12]$
- In the first pass, count $i = 0$ and index j runs from 0 to 3

Bubble Sort

- Consider the same sequence $[32, 108, 14, 72, 12]$
- In the first pass, count $i = 0$ and index j runs from 0 to 3
- $[\color{blue}{32}, \color{orange}{108}, 14, 72, 12]$

Bubble Sort

- Consider the same sequence $[32, 108, 14, 72, 12]$
- In the first pass, count $i = 0$ and index j runs from 0 to 3
- $[32, 108, 14, 72, 12]$
- $[32, 108, 14, 72, 12]$

Bubble Sort

- Consider the same sequence $[32, 108, 14, 72, 12]$
- In the first pass, count $i = 0$ and index j runs from 0 to 3
- $[32, 108, 14, 72, 12]$
- $[32, 108, 14, 72, 12]$
- $[32, 14, 108, 72, 12]$

Bubble Sort

- Consider the same sequence $[32, 108, 14, 72, 12]$
- In the first pass, count $i = 0$ and index j runs from 0 to 3
- $[32, 108, 14, 72, 12]$
- $[32, 108, 14, 72, 12]$
- $[32, 14, 108, 72, 12]$
- $[32, 14, 72, 108, 12]$

Bubble Sort

- Consider the same sequence $[32, 108, 14, 72, 12]$
- In the first pass, count $i = 0$ and index j runs from 0 to 3
- $[32, 108, 14, 72, 12]$
- $[32, 108, 14, 72, 12]$
- $[32, 14, 108, 72, 12]$
- $[32, 14, 72, 108, 12]$
- $[32, 14, 72, 12, 108]$

Bubble Sort

- Consider the same sequence $[32, 108, 14, 72, 12]$
- In the first pass, count $i = 0$ and index j runs from 0 to 3
- $[32, 108, 14, 72, 12]$
- $[32, 108, 14, 72, 12]$
- $[32, 14, 108, 72, 12]$
- $[32, 14, 72, 108, 12]$
- $[32, 14, 72, 12, 108]$
- After the second pass, count $i = 1$, index j runs from

Bubble Sort

- Consider the same sequence $[32, 108, 14, 72, 12]$
- In the first pass, count $i = 0$ and index j runs from 0 to 3
- $[32, 108, 14, 72, 12]$
- $[32, 108, 14, 72, 12]$
- $[32, 14, 108, 72, 12]$
- $[32, 14, 72, 108, 12]$
- $[32, 14, 72, 12, 108]$
- After the second pass, count $i = 1$, index j runs from 0 to 2, and we will have

Bubble Sort

- Consider the same sequence $[32, 108, 14, 72, 12]$
- In the first pass, count $i = 0$ and index j runs from 0 to 3
- $[32, 108, 14, 72, 12]$
- $[32, 108, 14, 72, 12]$
- $[32, 14, 108, 72, 12]$
- $[32, 14, 72, 108, 12]$
- $[32, 14, 72, 12, 108]$
- After the second pass, count $i = 1$, index j runs from 0 to 2, and we will have $[14, 32, 12, 72, 108]$

Bubble Sort

- Consider the same sequence $[32, 108, 14, 72, 12]$
- In the first pass, count $i = 0$ and index j runs from 0 to 3
- $[32, 108, 14, 72, 12]$
- $[32, 108, 14, 72, 12]$
- $[32, 14, 108, 72, 12]$
- $[32, 14, 72, 108, 12]$
- $[32, 14, 72, 12, 108]$
- After the second pass, count $i = 1$, index j runs from 0 to 2, and we will have $[14, 32, 12, 72, 108]$
- After the third pass, count $i = 2$, index j runs from 0 to 1, and we will have

Bubble Sort

- Consider the same sequence $[32, 108, 14, 72, 12]$
- In the first pass, count $i = 0$ and index j runs from 0 to 3
- $[32, 108, 14, 72, 12]$
- $[32, 108, 14, 72, 12]$
- $[32, 14, 108, 72, 12]$
- $[32, 14, 72, 108, 12]$
- $[32, 14, 72, 12, 108]$
- After the second pass, count $i = 1$, index j runs from 0 to 2, and we will have $[14, 32, 12, 72, 108]$
- After the third pass, count $i = 2$, index j runs from 0 to 1, and we will have $[14, 12, 32, 72, 108]$

Bubble Sort

- Consider the same sequence $[32, 108, 14, 72, 12]$
- In the first pass, count $i = 0$ and index j runs from 0 to 3
- $[32, 108, 14, 72, 12]$
- $[32, 108, 14, 72, 12]$
- $[32, 14, 108, 72, 12]$
- $[32, 14, 72, 108, 12]$
- $[32, 14, 72, 12, 108]$
- After the second pass, count $i = 1$, index j runs from 0 to 2, and we will have $[14, 32, 12, 72, 108]$
- After the third pass, count $i = 2$, index j runs from 0 to 1, and we will have $[14, 12, 32, 72, 108]$
- After the fourth pass, count $i = 3$, index j runs from 0 to 0, and we will have

Bubble Sort

- Consider the same sequence $[32, 108, 14, 72, 12]$
- In the first pass, count $i = 0$ and index j runs from 0 to 3
- $[32, 108, 14, 72, 12]$
- $[32, 108, 14, 72, 12]$
- $[32, 14, 108, 72, 12]$
- $[32, 14, 72, 108, 12]$
- $[32, 14, 72, 12, 108]$
- After the second pass, count $i = 1$, index j runs from 0 to 2, and we will have $[14, 32, 12, 72, 108]$
- After the third pass, count $i = 2$, index j runs from 0 to 1, and we will have $[14, 12, 32, 72, 108]$
- After the fourth pass, count $i = 3$, index j runs from 0 to 0, and we will have $[12, 14, 32, 72, 108]$

Bubble Sort

- Since adjacent objects are compared, it is possible to detect if the sequence is already sorted!
- So, it may be possible to break the loop and save a lot of comparisons!

Bubble Sort

- Since adjacent objects are compared, it is possible to detect if the sequence is already sorted!
- So, it may be possible to break the loop and save a lot of comparisons!

```
def bubble_sort(lst):
    n = len(lst)
    for i in range(n-1):
        swapped = False
        for j in range(n-1-i):
            if (lst[j] > lst[j+1]):
                lst[j], lst[j+1] = lst[j+1], lst[j]
                swapped = True
        if (not swapped): break
    return
```

Bubble Sort — Analysis

$$T(n) = \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1$$

Bubble Sort — Analysis

$$T(n) = \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1$$

$$= \sum_{i=0}^{n-2} (n-2-i) - 0 + 1$$

Bubble Sort — Analysis

$$T(n) = \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1$$

$$= \sum_{i=0}^{n-2} (n-2-i) - 0 + 1$$

$$= \sum_{i=0}^{n-2} (n-i-1)$$

Bubble Sort — Analysis

$$T(n) = \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1$$

$$= \sum_{i=0}^{n-2} (n-2-i) - 0 + 1$$

$$= \sum_{i=0}^{n-2} (n-i-1)$$

$$= (n-1) + (n-2) + \cdots + 1$$

Bubble Sort — Analysis

$$T(n) = \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1$$

$$= \sum_{i=0}^{n-2} (n-2-i) - 0 + 1$$

$$= \sum_{i=0}^{n-2} (n-i-1)$$

$$= (n-1) + (n-2) + \cdots + 1$$

$$= \frac{n(n-1)}{2}$$

Bubble Sort — Analysis

$$T(n) = \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1$$

$$= \sum_{i=0}^{n-2} (n-2-i) - 0 + 1$$

$$= \sum_{i=0}^{n-2} (n-i-1)$$

$$= (n-1) + (n-2) + \cdots + 1$$

$$= \frac{n(n-1)}{2}$$

Time Complexity is $\Theta(n^2)$

Insertion Sort

- Divide and Conquer!
- Decompose the sequence to head and tail
- Sort the tail
- Insert head in to the sorted tail (at the appropriate position)

Insertion Sort

- Divide and Conquer!
- Decompose the sequence to head and tail
- Sort the tail
- Insert head in to the sorted tail (at the appropriate position)

```
def isort(lst):
    if (len(lst) == 0):
        return []
    else:
        return insert(lst[0], isort(lst[1:]))
```

Insertion Sort

- Consider the same sequence $[32, 108, 14, 72, 12]$

Insertion Sort

- Consider the same sequence $[32, 108, 14, 72, 12]$
- ins(32, isort($[108, 14, 72, 12]$))

Insertion Sort

- Consider the same sequence $[32, 108, 14, 72, 12]$
- ins(32, isort($[108, 14, 72, 12]$))
- ins(32, ins(108, isort($[14, 72, 12]$)))

Insertion Sort

- Consider the same sequence $[32, 108, 14, 72, 12]$
- ins(32, isort([108, 14, 72, 12]))
- ins(32, ins(108, isort([14, 72, 12])))
- ins(32, ins(108, ins(14, isort([72, 12]))))

Insertion Sort

- Consider the same sequence $[32, 108, 14, 72, 12]$
- ins(32, isort($[108, 14, 72, 12]$))
- ins(32, ins(108, isort($[14, 72, 12]$)))
- ins(32, ins(108, ins(14, isort($[72, 12]$))))
- ins(32, ins(108, ins(14, ins(72, isort($[12]$)))))

Insertion Sort

- Consider the same sequence $[32, 108, 14, 72, 12]$
- ins(32, isort([108, 14, 72, 12]))
- ins(32, ins(108, isort([14, 72, 12])))
- ins(32, ins(108, ins(14, isort([72, 12]))))
- ins(32, ins(108, ins(14, ins(72, isort([12])))))
- ins(32, ins(108, ins(14, ins(72, ins(12, isort([]))))))

Insertion Sort

- Consider the same sequence $[32, 108, 14, 72, 12]$
- ins(32, isort([108, 14, 72, 12]))
- ins(32, ins(108, isort([14, 72, 12])))
- ins(32, ins(108, ins(14, isort([72, 12]))))
- ins(32, ins(108, ins(14, ins(72, isort([12])))))
- ins(32, ins(108, ins(14, ins(72, ins(12, isort([]))))))
- ins(32, ins(108, ins(14, ins(72, ins(12, [])))))

Insertion Sort

- Consider the same sequence $[32, 108, 14, 72, 12]$
- ins(32, isort([108, 14, 72, 12]))
- ins(32, ins(108, isort([14, 72, 12])))
- ins(32, ins(108, ins(14, isort([72, 12]))))
- ins(32, ins(108, ins(14, ins(72, isort([12])))))
- ins(32, ins(108, ins(14, ins(72, ins(12, isort([]))))))
- ins(32, ins(108, ins(14, ins(72, ins(12, [])))))
- ins(32, ins(108, ins(14, ins(72, [12]))))

Insertion Sort

- Consider the same sequence $[32, 108, 14, 72, 12]$
- ins(32, isort($[108, 14, 72, 12]$))
- ins(32, ins(108, isort($[14, 72, 12]$)))
- ins(32, ins(108, ins(14, isort($[72, 12]$))))
- ins(32, ins(108, ins(14, ins(72, isort($[12]$)))))
- ins(32, ins(108, ins(14, ins(72, ins(12, isort($[]$))))))
- ins(32, ins(108, ins(14, ins(72, ins(12, $[]$)))))
- ins(32, ins(108, ins(14, ins(72, $[12]$))))
- ins(32, ins(108, ins(14, $[12, 72]$)))

Insertion Sort

- Consider the same sequence $[32, 108, 14, 72, 12]$
- ins(32, isort([108, 14, 72, 12]))
- ins(32, ins(108, isort([14, 72, 12])))
- ins(32, ins(108, ins(14, isort([72, 12]))))
- ins(32, ins(108, ins(14, ins(72, isort([12])))))
- ins(32, ins(108, ins(14, ins(72, ins(12, isort([]))))))
- ins(32, ins(108, ins(14, ins(72, ins(12, [])))))
- ins(32, ins(108, ins(14, ins(72, [12]))))
- ins(32, ins(108, ins(14, [12, 72])))
- ins(32, ins(108, [12, 14, 72]))

Insertion Sort

- Consider the same sequence $[32, 108, 14, 72, 12]$
- ins(32, isort($[108, 14, 72, 12]$))
- ins(32, ins(108, isort($[14, 72, 12]$)))
- ins(32, ins(108, ins(14, isort($[72, 12]$))))
- ins(32, ins(108, ins(14, ins(72, isort($[12]$)))))
- ins(32, ins(108, ins(14, ins(72, ins(12, isort($[]$))))))
- ins(32, ins(108, ins(14, ins(72, ins(12, $[]$)))))
- ins(32, ins(108, ins(14, ins(72, $[12]$))))
- ins(32, ins(108, ins(14, $[12, 72]$)))
- ins(32, ins(108, $[12, 14, 72]$))
- ins(32, $[12, 14, 72, 108]$)

Insertion Sort

- Consider the same sequence $[32, 108, 14, 72, 12]$
- ins(32, isort($[108, 14, 72, 12]$))
- ins(32, ins(108, isort($[14, 72, 12]$)))
- ins(32, ins(108, ins(14, isort($[72, 12]$))))
- ins(32, ins(108, ins(14, ins(72, isort($[12]$)))))
- ins(32, ins(108, ins(14, ins(72, ins(12, isort($[]$))))))
- ins(32, ins(108, ins(14, ins(72, ins(12, $[]$)))))
- ins(32, ins(108, ins(14, ins(72, $[12]$))))
- ins(32, ins(108, ins(14, $[12, 72]$)))
- ins(32, ins(108, $[12, 14, 72]$))
- ins(32, $[12, 14, 72, 108]$)
- $[12, 14, 32, 72, 108]$

Insertion Sort

```
def insert(obj, seq):
    if (len(seq) == 0):
        return [obj]
```

Insertion Sort

```
def insert(obj, seq):
    if (len(seq) == 0):
        return [obj]

    elif (obj <= seq[0]):
        res = [obj]
        res.extend(seq)
        return res
```

Insertion Sort

```python
def insert(obj, seq):
    if (len(seq) == 0):
        return [obj]

    elif (obj <= seq[0]):
        res = [obj]
        res.extend(seq)
        return res

    else:
        res = seq[:1]
        res.extend(insert(obj, seq[1:]))
        return res
```

Insertion Sort

- Consider the case ins(32, $[12, 14, 72, 108]$)

Insertion Sort

- Consider the case ins(32, [12, 14, 72, 108])
- ins(32, [12, 14, 72, 108])

Insertion Sort

- Consider the case ins(32, [12, 14, 72, 108])
- ins(32, [12, 14, 72, 108])
- [12]:::ins(32, [14, 72, 108])

Insertion Sort

- Consider the case ins(32, [12, 14, 72, 108])
- ins(32, [12, 14, 72, 108])
- [12]:::ins(32, [14, 72, 108])
- [12]:::([14]:::ins(32, [72, 108]))

Insertion Sort

- Consider the case ins(32, [12, 14, 72, 108])
- ins(32, [12, 14, 72, 108])
- [12]:::ins(32, [14, 72, 108])
- [12]:::([14]:::ins(32, [72, 108]))
- [12]:::([14]:::([32]:::[72, 108]))

Insertion Sort

- Consider the case ins(32, [12, 14, 72, 108])
- ins(32, [12, 14, 72, 108])
- [12]:::ins(32, [14, 72, 108])
- [12]:::([14]:::ins(32, [72, 108]))
- [12]:::([14]:::([32]:::[72, 108]))
- [12]:::([14]:::[32, 72, 108])

Insertion Sort

- Consider the case ins(32, [12, 14, 72, 108])
- ins(32, [12, 14, 72, 108])
- [12]:::ins(32, [14, 72, 108])
- [12]:::([14]:::ins(32, [72, 108]))
- [12]:::([14]:::([32]:::[72, 108]))
- [12]:::([14]:::[32, 72, 108])
- [12]:::[14, 32, 72, 108]

Insertion Sort

- Consider the case ins(32, [12, 14, 72, 108])
- ins(32, [12, 14, 72, 108])
- [12]:::ins(32, [14, 72, 108])
- [12]:::([14]:::ins(32, [72, 108]))
- [12]:::([14]:::([32]:::[72, 108]))
- [12]:::([14]:::[32, 72, 108])
- [12]:::[14, 32, 72, 108]
- [12, 14, 32, 72, 108]

Analysis of Insertion

$$T(n) = \begin{cases} d & n \leq 1 \\ T(n-1) + c & n > 1 \end{cases}$$

Analysis of Insertion

$$T(n) = \begin{cases} d & n \leq 1 \\ T(n-1) + c & n > 1 \end{cases}$$

Time complexity is $\Theta(n)$

Analysis of Insertion Sort

$$T(n) = \begin{cases} d & n \le 1 \\ T(n-1) + cn & n > 1 \end{cases}$$

Analysis of Insertion Sort

$$T(n) = \begin{cases} d & n \leq 1 \\ T(n-1) + cn & n > 1 \end{cases}$$

Time complexity is $\Theta(n^2)$

Insertion Sort — in-place

- It is possible to implement this algorithm as an “in-place” version

Insertion Sort — in-place

- It is possible to implement this algorithm as an "in-place" version
- When the sub-sequence defined by positions $0 \cdots j$ is already sorted, insert the next object at position $i = j + 1$ by bubbling it down

Insertion Sort — in-place

- It is possible to implement this algorithm as an “in-place” version
- When the sub-sequence defined by positions $0 \cdots j$ is already sorted, insert the next object at position $i = j + 1$ by bubbling it down

```
def insertion_sort(lst):
    n = len(lst)
    for i in range(1,n):
        tmp = lst[i]
        j = i - 1
        while ( (j >= 0) and (lst[j] > tmp) ):
            lst[j+1] = lst[j]
            j -= 1
        lst[j+1] = tmp
    return
```

Insertion Sort — Illustration

- Consider the same sequence $[32, 108, 14, 72, 12]$

Insertion Sort — Illustration

- Consider the same sequence $[32, 108, 14, 72, 12]$
- In the first pass, $i = 1$ and index j runs from 0 to 0

Insertion Sort — Illustration

- Consider the same sequence $[32, 108, 14, 72, 12]$
- In the first pass, $i = 1$ and index j runs from 0 to 0
- That is, up to index 0 is already sorted and insert object at index 1 in to that list

Insertion Sort — Illustration

- Consider the same sequence $[32, 108, 14, 72, 12]$
- In the first pass, $i = 1$ and index j runs from 0 to 0
- That is, up to index 0 is already sorted and insert object at index 1 in to that list
- $[\underline{32}, 108, 14, 72, 12]$

Insertion Sort — Illustration

- Consider the same sequence $[32, 108, 14, 72, 12]$
- In the first pass, $i = 1$ and index j runs from 0 to 0
- That is, up to index 0 is already sorted and insert object at index 1 in to that list
- $[\underline{32}, 108, 14, 72, 12] \rightarrow \left[\underline{32, 108}, 14, 72, 12\right]$

Insertion Sort — Illustration

- Consider the same sequence $[32, 108, 14, 72, 12]$
- In the first pass, $i = 1$ and index j runs from 0 to 0
- That is, up to index 0 is already sorted and insert object at index 1 in to that list
- $[\underline{32}, 108, 14, 72, 12] \rightarrow \left[\underline{32, 108}, 14, 72, 12\right]$
- In the second pass, $i = 2$ and index j runs from 1 to 0

Insertion Sort — Illustration

- Consider the same sequence $[32, 108, 14, 72, 12]$
- In the first pass, $i = 1$ and index j runs from 0 to 0
- That is, up to index 0 is already sorted and insert object at index 1 in to that list
- $[\underline{32}, 108, 14, 72, 12] \rightarrow \left[\underline{32, 108}, 14, 72, 12\right]$
- In the second pass, $i = 2$ and index j runs from 1 to 0
- $\left[\underline{32, 108}, 14, 72, 12\right]$

Insertion Sort — Illustration

- Consider the same sequence $[32, 108, 14, 72, 12]$
- In the first pass, $i = 1$ and index j runs from 0 to 0
- That is, up to index 0 is already sorted and insert object at index 1 in to that list
- $[\underline{32}, 108, 14, 72, 12] \rightarrow \left[\underline{32, 108}, 14, 72, 12\right]$
- In the second pass, $i = 2$ and index j runs from 1 to 0
- $\left[\underline{32, 108}, 14, 72, 12\right] \rightarrow \left[\underline{32, X, 108}, 72, 12\right]$

Insertion Sort — Illustration

- Consider the same sequence $[32, 108, 14, 72, 12]$
- In the first pass, $i = 1$ and index j runs from 0 to 0
- That is, up to index 0 is already sorted and insert object at index 1 in to that list
- $[\underline{32}, 108, 14, 72, 12] \rightarrow \left[\underline{32, 108}, 14, 72, 12\right]$
- In the second pass, $i = 2$ and index j runs from 1 to 0
- $\left[\underline{32, 108}, 14, 72, 12\right] \rightarrow \left[\underline{32, X, 108}, 72, 12\right] \rightarrow [X, 32, 108, 72, 12]$

Insertion Sort — Illustration

- Consider the same sequence $[32, 108, 14, 72, 12]$
- In the first pass, $i = 1$ and index j runs from 0 to 0
- That is, up to index 0 is already sorted and insert object at index 1 in to that list
- $[\underline{32}, 108, 14, 72, 12] \rightarrow \left[\underline{32, 108}, 14, 72, 12\right]$
- In the second pass, $i = 2$ and index j runs from 1 to 0
- $\left[\underline{32, 108}, 14, 72, 12\right] \rightarrow \left[\underline{32, X, 108}, 72, 12\right] \rightarrow [X, 32, 108, 72, 12] \rightarrow \left[\underline{14, 32, 108}, 72, 12\right]$

Insertion Sort — Illustration

- Consider the same sequence $[32, 108, 14, 72, 12]$
- In the first pass, $i = 1$ and index j runs from 0 to 0
- That is, up to index 0 is already sorted and insert object at index 1 in to that list
- $[\underline{32}, 108, 14, 72, 12] \rightarrow \left[\underline{32, 108}, 14, 72, 12\right]$
- In the second pass, $i = 2$ and index j runs from 1 to 0
- $\left[\underline{32, 108}, 14, 72, 12\right] \rightarrow \left[\underline{32, X, 108}, 72, 12\right] \rightarrow [X, 32, 108, 72, 12] \rightarrow \left[\underline{14, 32, 108}, 72, 12\right]$
- In the next pass, $i = 3$ and index j runs from 2 to 0

Insertion Sort — Illustration

- Consider the same sequence $[32, 108, 14, 72, 12]$
- In the first pass, $i = 1$ and index j runs from 0 to 0
- That is, up to index 0 is already sorted and insert object at index 1 in to that list
- $[\underline{32}, 108, 14, 72, 12] \rightarrow \left[\underline{32, 108}, 14, 72, 12\right]$
- In the second pass, $i = 2$ and index j runs from 1 to 0
- $\left[\underline{32, 108}, 14, 72, 12\right] \rightarrow \left[\underline{32, X, 108}, 72, 12\right] \rightarrow [X, 32, 108, 72, 12] \rightarrow \left[\underline{14, 32, 108}, 72, 12\right]$
- In the next pass, $i = 3$ and index j runs from 2 to 0
- We get, $\left[\underline{14, 32, 72, 108}, 12\right]$

Insertion Sort — Illustration

- Consider the same sequence $[32, 108, 14, 72, 12]$
- In the first pass, $i = 1$ and index j runs from 0 to 0
- That is, up to index 0 is already sorted and insert object at index 1 in to that list
- $[\underline{32}, 108, 14, 72, 12] \rightarrow \left[\underline{32, 108}, 14, 72, 12\right]$
- In the second pass, $i = 2$ and index j runs from 1 to 0
- $\left[\underline{32, 108}, 14, 72, 12\right] \rightarrow \left[\underline{32, X, 108}, 72, 12\right] \rightarrow [X, 32, 108, 72, 12] \rightarrow \left[\underline{14, 32, 108}, 72, 12\right]$
- In the next pass, $i = 3$ and index j runs from 2 to 0
- We get, $\left[\underline{14, 32, 72, 108}, 12\right]$
- Finally, $i = 4$ and index j runs from 3 to 0

Insertion Sort — Illustration

- Consider the same sequence $[32, 108, 14, 72, 12]$
- In the first pass, $i = 1$ and index j runs from 0 to 0
- That is, up to index 0 is already sorted and insert object at index 1 in to that list
- $[\underline{32}, 108, 14, 72, 12] \rightarrow \left[\underline{32, 108}, 14, 72, 12\right]$
- In the second pass, $i = 2$ and index j runs from 1 to 0
- $\left[\underline{32, 108}, 14, 72, 12\right] \rightarrow \left[\underline{32, X, 108}, 72, 12\right] \rightarrow [X, 32, 108, 72, 12] \rightarrow \left[\underline{14, 32, 108}, 72, 12\right]$
- In the next pass, $i = 3$ and index j runs from 2 to 0
- We get, $\left[\underline{14, 32, 72, 108}, 12\right]$
- Finally, $i = 4$ and index j runs from 3 to 0
- We get, $\left[\underline{12, 14, 32, 72, 108}\right]$

Insertion Sort — Analysis

$$T(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1$$

Insertion Sort — Analysis

$$T(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1$$

$$= \sum_{i=1}^{n-1} (i-1) - 0 + 1$$

Insertion Sort — Analysis

$$T(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1$$

$$= \sum_{i=1}^{n-1} (i - 1) - 0 + 1$$

$$= \sum_{i=1}^{n-1} i$$

Insertion Sort — Analysis

$$T(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1$$

$$= \sum_{i=1}^{n-1} (i-1) - 0 + 1$$

$$= \sum_{i=1}^{n-1} i$$

$$= (n-1) + (n-2) + \cdots + 1$$

Insertion Sort — Analysis

$$T(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1$$

$$= \sum_{i=1}^{n-1} (i-1) - 0 + 1$$

$$= \sum_{i=1}^{n-1} i$$

$$= (n-1) + (n-2) + \cdots + 1$$

$$= \frac{n(n-1)}{2}$$

Insertion Sort — Analysis

$$T(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1$$

$$= \sum_{i=1}^{n-1} (i-1) - 0 + 1$$

$$= \sum_{i=1}^{n-1} i$$

$$= (n-1) + (n-2) + \cdots + 1$$

$$= \frac{n(n-1)}{2}$$

Time Complexity is $\Theta(n^2)$

Lower bound for brute-force sorting algorithms

- Given a sequence of n objects, there are ${}^{n}C_{2}$ pairs and that many comparisons are made

Lower bound for brute-force sorting algorithms

- Given a sequence of n objects, there are ${}^{n}C_2$ pairs and that many comparisons are made
- Since adjacent objects are compared (either directly or indirectly), each swap corrects one inversion only

Lower bound for brute-force sorting algorithms

- Given a sequence of n objects, there are ${}^nC_2$ pairs and that many comparisons are made
- Since adjacent objects are compared (either directly or indirectly), each swap corrects one inversion only
- Minimum number of inversions: 0

Lower bound for brute-force sorting algorithms

- Given a sequence of n objects, there are ${}^{n}C_2$ pairs and that many comparisons are made
- Since adjacent objects are compared (either directly or indirectly), each swap corrects one inversion only
- Minimum number of inversions: 0
- In such a case, bubble sort and insertion sort perform $\Theta(n)$ comparisons only

Lower bound for brute-force sorting algorithms

- Given a sequence of n objects, there are ${}^{n}C_2$ pairs and that many comparisons are made
- Since adjacent objects are compared (either directly or indirectly), each swap corrects one inversion only
- Minimum number of inversions: 0
- In such a case, bubble sort and insertion sort perform $\Theta(n)$ comparisons only
- Maximum number of inversions: $n(n-1)/2$

Lower bound for brute-force sorting algorithms

- Given a sequence of n objects, there are ${}^nC_2$ pairs and that many comparisons are made
- Since adjacent objects are compared (either directly or indirectly), each swap corrects one inversion only
- Minimum number of inversions: 0
- In such a case, bubble sort and insertion sort perform $\Theta(n)$ comparisons only
- Maximum number of inversions: $n(n-1)/2$
- The number of comparisons and swaps are to that effect and worst-case time complexity is $\Theta(n^2)$

Lower bound for brute-force sorting algorithms

- Given a sequence of n objects, there are ${}^{n}C_2$ pairs and that many comparisons are made
- Since adjacent objects are compared (either directly or indirectly), each swap corrects one inversion only
- Minimum number of inversions: 0
- In such a case, bubble sort and insertion sort perform $\Theta(n)$ comparisons only
- Maximum number of inversions: $n(n-1)/2$
- The number of comparisons and swaps are to that effect and worst-case time complexity is $\Theta(n^2)$
- What is the average case?

Lower bound for brute-force sorting algorithms

- Given a sequence of n objects, there are ${}^{n}C_2$ pairs and that many comparisons are made
- Since adjacent objects are compared (either directly or indirectly), each swap corrects one inversion only
- Minimum number of inversions: 0
- In such a case, bubble sort and insertion sort perform $\Theta(n)$ comparisons only
- Maximum number of inversions: $n(n-1)/2$
- The number of comparisons and swaps are to that effect and worst-case time complexity is $\Theta(n^2)$
- What is the average case?
- Expected number of inversions: $n(n-1)/4$

Lower bound for brute-force sorting algorithms

- Given a sequence of n objects, there are ${}^nC_2$ pairs and that many comparisons are made
- Since adjacent objects are compared (either directly or indirectly), each swap corrects one inversion only
- Minimum number of inversions: 0
- In such a case, bubble sort and insertion sort perform $\Theta(n)$ comparisons only
- Maximum number of inversions: $n(n-1)/2$
- The number of comparisons and swaps are to that effect and worst-case time complexity is $\Theta(n^2)$
- What is the average case?
- Expected number of inversions: $n(n-1)/4$
- Hence, $\Omega(n^2)$ comparisons are required

Lower bound for brute-force sorting algorithms

- Given a sequence of n objects, there are ${}^nC_2$ pairs and that many comparisons are made
- Since adjacent objects are compared (either directly or indirectly), each swap corrects one inversion only
- Minimum number of inversions: 0
- In such a case, bubble sort and insertion sort perform $\Theta(n)$ comparisons only
- Maximum number of inversions: $n(n-1)/2$
- The number of comparisons and swaps are to that effect and worst-case time complexity is $\Theta(n^2)$
- What is the average case?
- Expected number of inversions: $n(n-1)/4$
- Hence, $\Omega(n^2)$ comparisons are required
- Is it possible to eliminate more than one inversion per swap?

Merge Sort

- Divide: Partition the sequence in to two equal halves, positions $begin \cdots mid$ and $mid + 1 \cdots end$

Merge Sort

- Divide: Partition the sequence in to two equal halves, positions $begin \cdots mid$ and $mid + 1 \cdots end$
- Conquer: Sort the sub-sequences separately

Merge Sort

- Divide: Partition the sequence in to two equal halves, positions $begin \cdots mid$ and $mid + 1 \cdots end$
- Conquer: Sort the sub-sequences separately
- Merge: Merge the sorted sub-sequences to single sorted sequence

Merge Sort

- Divide: Partition the sequence in to two equal halves, positions $begin \cdots mid$ and $mid + 1 \cdots end$
- Conquer: Sort the sub-sequences separately
- Merge: Merge the sorted sub-sequences to single sorted sequence

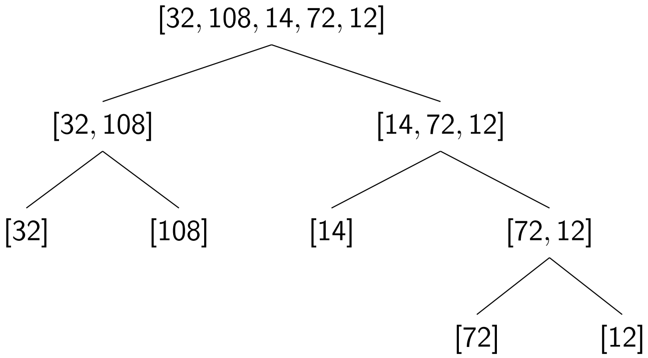
```
def msort(lst):
    length = len(lst)
    if (length < 2):
        return lst[:]
    else:
        mid = length // 2
        return merge(msort(lst[:mid]),
                     msort(lst[mid:])
                )
```

Merge Sort — Illustration

- Consider the same sequence $[32, 108, 14, 72, 12]$

Merge Sort — Illustration

- Consider the same sequence $[32, 108, 14, 72, 12]$

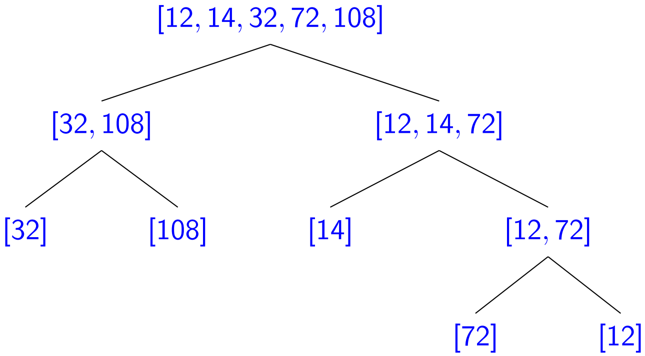


Merge Sort — Illustration

- When the recursions return, at each internal node the two sorted sequences are merged to create a unified sorted sequences

Merge Sort — Illustration

- When the recursions return, at each internal node the two sorted sequences are merged to create a unified sorted sequences



Merging — Illustration

- Merge $[32, 108]$ with $[12, 14, 72]$

Merging — Illustration

- Merge $[32, 108]$ with $[12, 14, 72]$
- $[32, 108]$; $[12, 14, 72]$; $[]$

- Merge $[32, 108]$ with $[12, 14, 72]$
- $[32, 108]$; $[12, 14, 72]$; $[]$
- $[32, 108]$; $[14, 72]$; $[12]$

Merging — Illustration

- Merge [32, 108] with [12, 14, 72]
- [32, 108] ; [12, 14, 72] ; []
- [32, 108] ; [14, 72] ; [12]
- [32, 108] ; [72] ; [12, 14]

Merging — Illustration

- Merge [32, 108] with [12, 14, 72]
- [32, 108] ; [12, 14, 72] ; []
- [32, 108] ; [14, 72] ; [12]
- [32, 108] ; [72] ; [12, 14]
- [108] ; [72] ; [12, 14, 32]

Merging — Illustration

- Merge [32, 108] with [12, 14, 72]
- [32, 108] ; [12, 14, 72] ; []
- [32, 108] ; [14, 72] ; [12]
- [32, 108] ; [72] ; [12, 14]
- [108] ; [72] ; [12, 14, 32]
- [108] ; [] ; [12, 14, 32, 72]

Merging — Illustration

- Merge [32, 108] with [12, 14, 72]
- [32, 108] ; [12, 14, 72] ; []
- [32, 108] ; [14, 72] ; [12]
- [32, 108] ; [72] ; [12, 14]
- [108] ; [72] ; [12, 14, 32]
- [108] ; [] ; [12, 14, 32, 72]
- [] ; [] ; [12, 14, 32, 72, 108]

Merge Sort

```
def merge(lst1, lst2):
    i = j = 0
    res = []
```

Merge Sort

```
def merge(lst1, lst2):
    i = j = 0
    res = []

    while ( (i < len(lst1)) and (j < len(lst2)) ):
        if (lst1[i] < lst2[j]):
            res.append(lst1[i])
            i += 1
        else:
            res.append(lst2[j])
            j += 1
```

Merge Sort

```
def merge(lst1, lst2):
    i = j = 0
    res = []

    while ( (i < len(lst1)) and (j < len(lst2)) ):
        if (lst1[i] < lst2[j]):
            res.append(lst1[i])
            i += 1
        else:
            res.append(lst2[j])
            j += 1

    if (i < len(lst1)):
        res.extend(lst1[i:])
    elif (j < len(lst2)):
        res.extend(lst2[j:])
    return res
```

Merge Sort — Analysis

$$T(n) = \begin{cases} d & n \leq 1 \\ 2T(n/2) + n & n > 1 \end{cases}$$

Merge Sort — Analysis

$$T(n) = \begin{cases} d & n \leq 1 \\ 2T(n/2) + n & n > 1 \end{cases}$$

Time complexity is $O(n \log n)$

Mergesort — Discussion

- Improved worst-case time complexity: $O(n \log n)$
- But, requires extra space for merging
- We have discussed a variant that actually returns a new sorted sequence
- In-place variation is possible, but still requires extra space
- Multi-way merging is also possible
- Suitable for external sorting
- Suitable for sorting dynamic structures such as linked lists

Quicksort

- Invented by C. A. R. Hoare in 1962

Quicksort

- Invented by C. A. R. Hoare in 1962
- Like in the case of merge sort, the input sequence L is partitioned, but not in an uninformed way

Quicksort

- Invented by C. A. R. Hoare in 1962
- Like in the case of merge sort, the input sequence L is partitioned, but not in an uninformed way
- One of the objects is selected as a pivot p. Partitioning is done wrt the pivot: All the objects $x < p$ belong to one partition L_1 and the objects $x > p$ constitute the other partition L_2

Quicksort

- Invented by C. A. R. Hoare in 1962
- Like in the case of merge sort, the input sequence L is partitioned, but not in an uninformed way
- One of the objects is selected as a pivot p. Partitioning is done wrt the pivot: All the objects $x < p$ belong to one partition L_1 and the objects $x > p$ constitute the other partition L_2
- Now L_1 and L_2 are sorted separately

Quicksort

- Invented by C. A. R. Hoare in 1962
- Like in the case of merge sort, the input sequence L is partitioned, but not in an uninformed way
- One of the objects is selected as a pivot p. Partitioning is done wrt the pivot: All the objects $x < p$ belong to one partition L_1 and the objects $x > p$ constitute the other partition L_2
- Now L_1 and L_2 are sorted separately
- Finally sorted L is obtained as: sorted L_1, followed by p, followed by sorted L_2

Quicksort — Illustration

- Consider the same sequence $[32, 108, 14, 72, 12]$

Quicksort — Illustration

- Consider the same sequence $[32, 108, 14, 72, 12]$
- Suppose we select 32 as the pivot

Quicksort — Illustration

- Consider the same sequence $[32, 108, 14, 72, 12]$
- Suppose we select 32 as the pivot
- Then partition may result in $[14, 12]$, $[32]$, $[108, 72]$

Quicksort — Illustration

- Consider the same sequence $[32, 108, 14, 72, 12]$
- Suppose we select 32 as the pivot
- Then partition may result in $[14, 12]$, $[32]$, $[108, 72]$
- Now, sort the two partitions, resulting in $[12, 14]$, $[32]$, $[72, 108]$

Quicksort — Illustration

- Consider the same sequence $[32, 108, 14, 72, 12]$
- Suppose we select 32 as the pivot
- Then partition may result in $[14, 12]$, $[32]$, $[108, 72]$
- Now, sort the two partitions, resulting in $[12, 14]$, $[32]$, $[72, 108]$
- Now, simply append them to get the sorted sequence $[12, 14, 32, 72, 108]$

Quicksort — Illustration

- Consider the same sequence $[32, 108, 14, 72, 12]$
- Suppose we select 32 as the pivot
- Then partition may result in $[14, 12]$, $[32]$, $[108, 72]$
- Now, sort the two partitions, resulting in $[12, 14]$, $[32]$, $[72, 108]$
- Now, simply append them to get the sorted sequence $[12, 14, 32, 72, 108]$
- It is possible to perform these steps in-place, in which case the final append is not necessary

Quicksort — Implementation issues

- How is this different from Mergesort and what may go wrong?

Quicksort — Implementation issues

- How is this different from Mergesort and what may go wrong?
- How to select a pivot?

Quicksort — Implementation issues

- How is this different from Mergesort and what may go wrong?
- How to select a pivot?
  - Obviously median is the best choice, but how to find the median!?

Quicksort — Implementation issues

- How is this different from Mergesort and what may go wrong?
- How to select a pivot?
  - Obviously median is the best choice, but how to find the median!?
  - Pick the object at a pre-determined location — such as first object or the last object

Quicksort — Implementation issues

- How is this different from Mergesort and what may go wrong?
- How to select a pivot?
  - Obviously median is the best choice, but how to find the median!?
  - Pick the object at a pre-determined location — such as first object or the last object
  - Pick a random object

Quicksort — Implementation issues

- How is this different from Mergesort and what may go wrong?
- How to select a pivot?
  - Obviously median is the best choice, but how to find the median!?
  - Pick the object at a pre-determined location — such as first object or the last object
  - Pick a random object
  - Instead of median of the entire sequence, select the median of, say 3 objects

Quicksort — Implementation issues

- How is this different from Mergesort and what may go wrong?
- How to select a pivot?
  - Obviously median is the best choice, but how to find the median!?
  - Pick the object at a pre-determined location — such as first object or the last object
  - Pick a random object
  - Instead of median of the entire sequence, select the median of, say 3 objects
- What may be a better partition strategy?

Quicksort — Implementation issues

- How is this different from Mergesort and what may go wrong?
- How to select a pivot?
  - Obviously median is the best choice, but how to find the median!?
  - Pick the object at a pre-determined location — such as first object or the last object
  - Pick a random object
  - Instead of median of the entire sequence, select the median of, say 3 objects
- What may be a better partition strategy?
  - Lomuto partition algorithm

Quicksort — Implementation issues

- How is this different from Mergesort and what may go wrong?
- How to select a pivot?
  - Obviously median is the best choice, but how to find the median!?
  - Pick the object at a pre-determined location — such as first object or the last object
  - Pick a random object
  - Instead of median of the entire sequence, select the median of, say 3 objects
- What may be a better partition strategy?
  - Lomuto partition algorithm
  - Hoare's partition algorithm

Quicksort — Implementation issues

- How is this different from Mergesort and what may go wrong?
- How to select a pivot?
  - Obviously median is the best choice, but how to find the median!?
  - Pick the object at a pre-determined location — such as first object or the last object
  - Pick a random object
  - Instead of median of the entire sequence, select the median of, say 3 objects
- What may be a better partition strategy?
  - Lomuto partition algorithm
  - Hoare's partition algorithm
- What may be a better base case?

Quicksort — Implementation issues

- How is this different from Mergesort and what may go wrong?
- How to select a pivot?
  - Obviously median is the best choice, but how to find the median!?
  - Pick the object at a pre-determined location — such as first object or the last object
  - Pick a random object
  - Instead of median of the entire sequence, select the median of, say 3 objects
- What may be a better partition strategy?
  - Lomuto partition algorithm
  - Hoare's partition algorithm
- What may be a better base case?
  - When $n < 2$, the sequence is already sorted

Quicksort — Implementation issues

- How is this different from Mergesort and what may go wrong?
- How to select a pivot?
    - Obviously median is the best choice, but how to find the median!?
    - Pick the object at a pre-determined location — such as first object or the last object
    - Pick a random object
    - Instead of median of the entire sequence, select the median of, say 3 objects
- What may be a better partition strategy?
    - Lomuto partition algorithm
    - Hoare's partition algorithm
- What may be a better base case?
    - When $n < 2$, the sequence is already sorted
    - Note that for small sequences, the overhead of divide and conquer may be more

Quicksort — Implementation issues

- How is this different from Mergesort and what may go wrong?
- How to select a pivot?
  - Obviously median is the best choice, but how to find the median!?
  - Pick the object at a pre-determined location — such as first object or the last object
  - Pick a random object
  - Instead of median of the entire sequence, select the median of, say 3 objects
- What may be a better partition strategy?
  - Lomuto partition algorithm
  - Hoare's partition algorithm
- What may be a better base case?
  - When $n < 2$, the sequence is already sorted
  - Note that for small sequences, the overhead of divide and conquer may be more
  - Hence we can think of hybrid sorting — perform insertion sorting when the size of the partition is below a threshold

Quicksort — Implementation

```
def quick_sort(lst):
    QUICK_BASE_CASE = 3

    def quick_sort_rec(lst, begin, end):
        if ((end - begin) < QUICK_BASE_CASE):
            insertion_sort_range(lst, begin, end)
            return

        p = partition(lst, begin, end)
        quick_sort_rec(lst, begin, p-1)
        quick_sort_rec(lst, p+1, end)
        return

    quick_sort_rec(lst, 0, len(lst) - 1)

    return
```

Quicksort — Finding a pivot

```python
def find_pivot_m3(lst, begin, end):
    mid = (begin + end) // 2
    if ( lst[begin] > lst[mid] ):
        lst[begin], lst[mid] = lst[mid], lst[begin]
    if ( lst[begin] > lst[end] ):
        lst[begin], lst[end] = lst[end], lst[begin]
    if ( lst[mid] > lst[end] ):
        lst[mid], lst[end] = lst[end], lst[mid]
    lst[mid], lst[end-1] = lst[end-1], lst[mid]
    return lst[end-1]
```

Quicksort — Finding a pivot

```
def find_pivot_m3(lst, begin, end):
    mid = (begin + end) // 2
    if ( lst[begin] > lst[mid] ):
        lst[begin], lst[mid] = lst[mid], lst[begin]
    if ( lst[begin] > lst[end] ):
        lst[begin], lst[end] = lst[end], lst[begin]
    if ( lst[mid] > lst[end] ):
        lst[mid], lst[end] = lst[end], lst[mid]
    lst[mid], lst[end-1] = lst[end-1], lst[mid]
    return lst[end-1]
```

- Note that this function has side effects!!!

Quicksort — Finding a pivot

```python
def find_pivot_m3(lst, begin, end):
    mid = (begin + end) // 2
    if ( lst[begin] > lst[mid] ):
        lst[begin], lst[mid] = lst[mid], lst[begin]
    if ( lst[begin] > lst[end] ):
        lst[begin], lst[end] = lst[end], lst[begin]
    if ( lst[mid] > lst[end] ):
        lst[mid], lst[end] = lst[end], lst[mid]
    lst[mid], lst[end-1] = lst[end-1], lst[mid]
    return lst[end-1]
```

- Note that this function has side effects!!!
- The pivot is temporarily moved to $(end - 1)$ position

Quicksort — Finding a pivot

```
def find_pivot_m3(lst, begin, end):
    mid = (begin + end) // 2
    if ( lst[begin] > lst[mid] ):
        lst[begin], lst[mid] = lst[mid], lst[begin]
    if ( lst[begin] > lst[end] ):
        lst[begin], lst[end] = lst[end], lst[begin]
    if ( lst[mid] > lst[end] ):
        lst[mid], lst[end] = lst[end], lst[mid]
    lst[mid], lst[end-1] = lst[end-1], lst[mid]
    return lst[end-1]
```

- Note that this function has side effects!!!
- The pivot is temporarily moved to $(end - 1)$ position
- We need to partition the sequence in the range $(begin + 1)$ to $(end - 2)$

Quicksort — Partitioning Algorithm

```python
def partition(lst, begin, end):
    pivot = find_pivot_m3(lst, begin, end)
    i = begin;  j = end - 1
    while (True):
        i += 1
        while (lst[i] < pivot):
            i += 1
        j -= 1
        while (lst[j] > pivot):
            j -= 1
        if (i < j):
            lst[i], lst[j] = lst[j], lst[i]
        else:
            break
    lst[i], lst[end-1] = lst[end-1], lst[i]
    return i
```

Quicksort — Illustration

- Trace the execution of quick sort on the same sequence $[32, 108, 14, 72, 12]$

Quicksort — Analysis

$$T(n) = \begin{cases} d & n \leq 1 \\ T(i) + T(n-i-1) + cn & n > 1 \end{cases}$$

Quicksort — Analysis

$$T(n) = \begin{cases} d & n \leq 1 \\ T(i) + T(n-i-1) + cn & n > 1 \end{cases}$$

Worst case occurs when $i = 0$ or $i = n - 1$

Quicksort — Analysis

$$T(n) = \begin{cases} d & n \leq 1 \\ T(i) + T(n-i-1) + cn & n > 1 \end{cases}$$

Worst case occurs when $i = 0$ or $i = n - 1$

$$T(n) = \begin{cases} d & n \leq 1 \\ T(n-1) + cn & n > 1 \end{cases}$$

Quicksort — Analysis

$$T(n) = \begin{cases} d & n \leq 1 \\ T(i) + T(n-i-1) + cn & n > 1 \end{cases}$$

Worst case occurs when $i = 0$ or $i = n - 1$

$$T(n) = \begin{cases} d & n \leq 1 \\ T(n-1) + cn & n > 1 \end{cases}$$

Worst-case time complexity is $O(n^2)$

Quicksort — Analysis

Best case occurs when $i = n/2$

Quicksort — Analysis

Best case occurs when $i = n/2$

$$T(n) = \begin{cases} d & n \leq 1 \\ 2T(n/2) + cn & n > 1 \end{cases}$$

Quicksort — Analysis

Best case occurs when $i = n/2$

$$T(n) = \begin{cases} d & n \leq 1 \\ 2T(n/2) + cn & n > 1 \end{cases}$$

Best-case time complexity is $O(n \log n)$

To find the expected time on the average, consider $i = 0$ to $i = n - 1$, where each has equal probability $1/n$

Quicksort — Analysis

To find the expected time on the average, consider $i = 0$ to $i = n - 1$, where each has equal probability $1/n$

$$T(n) = \frac{1}{n} \sum_{i=0}^{n-1} T(i) + T(n-i-1) + cn$$

Quicksort — Analysis

To find the expected time on the average, consider $i = 0$ to $i = n - 1$, where each has equal probability $1/n$

$$T(n) = \frac{1}{n}\sum_{i=0}^{n-1} T(i) + T(n-i-1) + cn$$

$$T(n) = \frac{2}{n}\left[\sum_{i=0}^{n-1} T(i)\right] + cn$$

Quicksort — Analysis

To find the expected time on the average, consider $i = 0$ to $i = n - 1$, where each has equal probability $1/n$

$$T(n) = \frac{1}{n} \sum_{i=0}^{n-1} T(i) + T(n - i - 1) + cn$$

$$T(n) = \frac{2}{n} \left[\sum_{i=0}^{n-1} T(i) \right] + cn$$

Average case complexity is $O(n \log n)$

Summary

- Generate-and-Test
    - Check all the permutations of the input sequence
- Brute force (or) Exhaustive search
    - Check all the pairs and swap all the inversions
- Divide-Conquer-Merge
    - Decomposing 1 and $(n-1)$ does not help
    - Decomposing $n/2$ and $n/2$ changes the complexity class
    - Do more work on decomposition where merging becomes trivial
- Hybrid Approaches