UCONN
UNIVERSITY OF CONNECTICUT

Department of Computer Science and Engineering

Data Structures and Object-Oriented Design

(CSE – 2050)

Hasan Baig

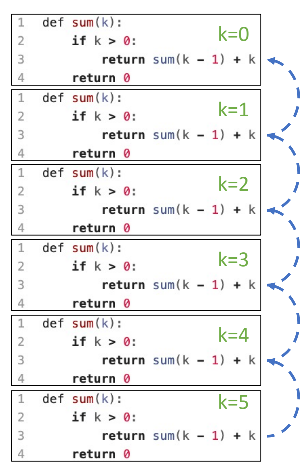
Office: UConn (Stamford), 305C
email: hasan.baig@uconn.edu

1

Module 5 – Recursion and Dynamic Programming

- Recursion
  → Solving a problem using function that recursively call itself.

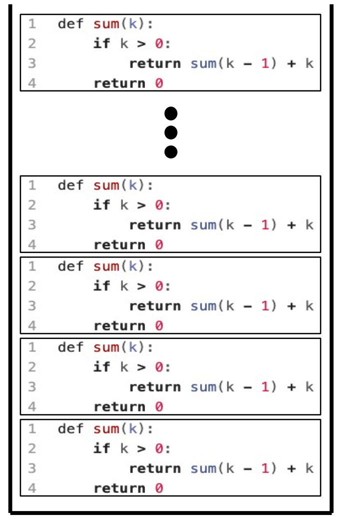
```python
def sum(k):
    if k > 0:
        return sum(k - 1) + k
    return 0
print(sum(5))
```



2

Module 5 – Recursion and Dynamic Programming

- FunctionCall Stack
  - Used Stack (LIFO) structure to hold recursive calls
  - In Python, infinite recursive calls are restricted to 1000 (by default)
    - Can be modified using `sys.setrecursionlimit(N)`



FunctionCall Stack

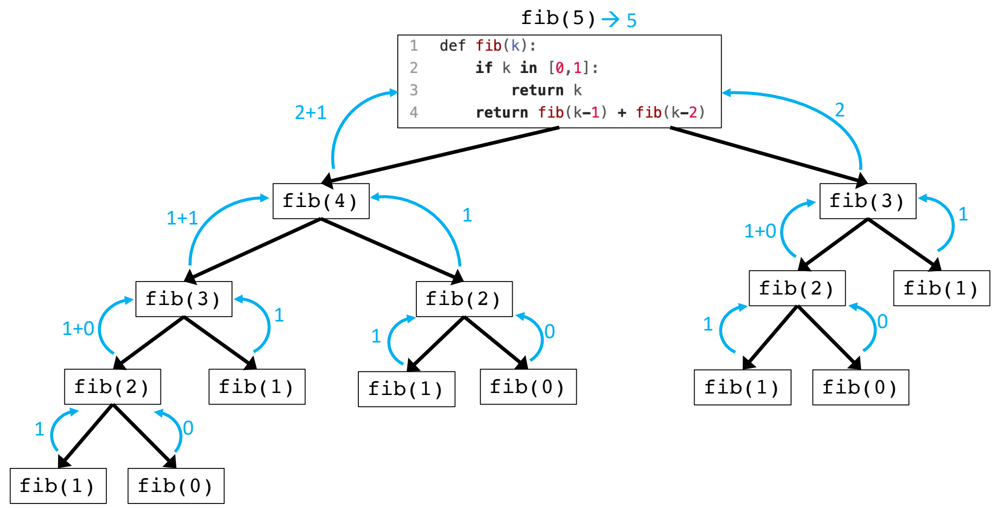
Module 5 – Recursion and Dynamic Programming

1. Have a Base Case:

```
def sum(k):
    if k > 0:
        return sum(k - 1) + k
    return 0
```

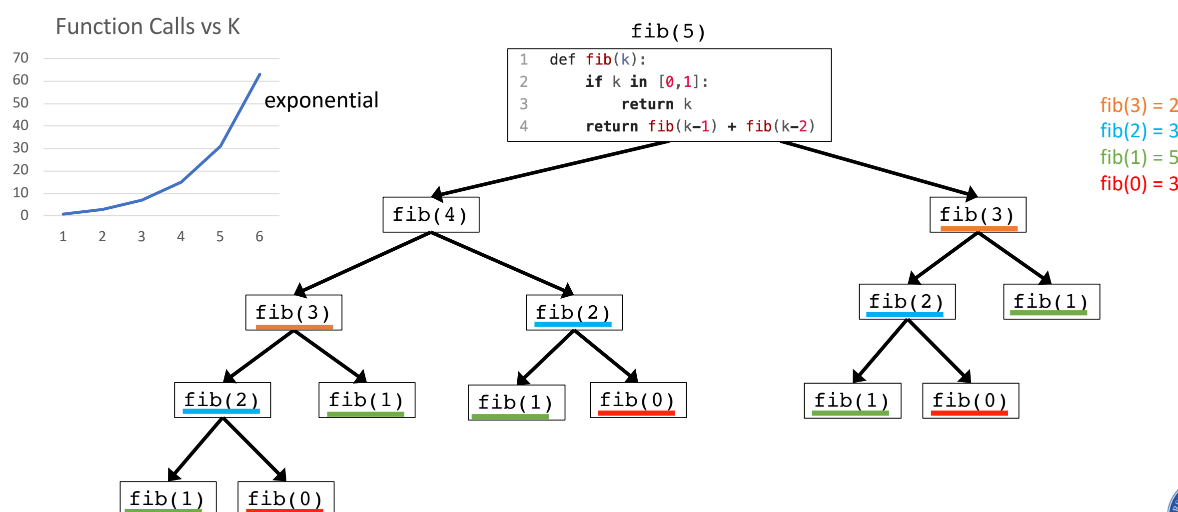
2. Recursion calls should move towards the base case

Module 5 – Recursion and Dynamic Programming



5

Module 5 – Recursion and Dynamic Programming



6

CSE-2050 – Data Structures and Object-Oriented Design

Review

7

Module 5 – Recursion and Dynamic Programming

Memoization

- Avoid making a function call again which has already been executed by:
- storing the intermediate solution of subproblems and use them later wherever needed
- Also called Top-down approach which uses recursion + caching

Hasan Baig

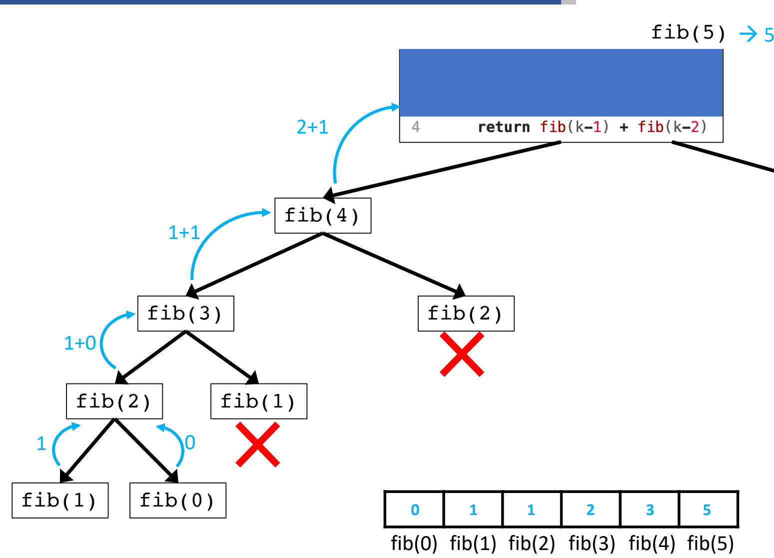
7

CSE-2050 – Data Structures and Object-Oriented Design

Review

8

Module 5 – Recursion and Dynamic Programming



Hasan Baig

8

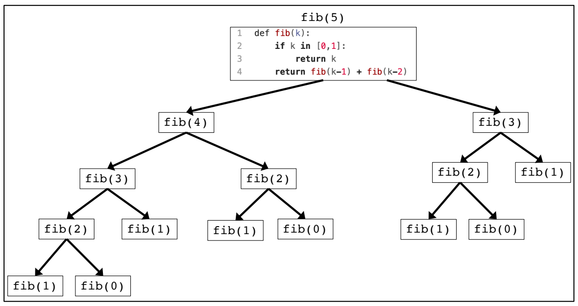
CSE-2050 – Data Structures and Object-Oriented Design

Review

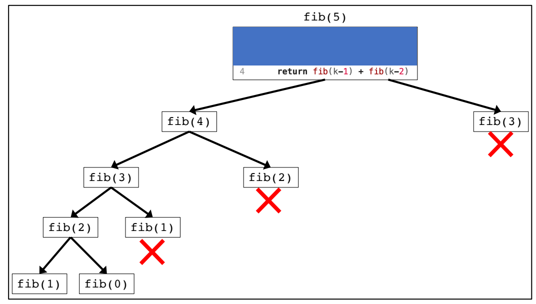
9

Module 5 – Recursion and Dynamic Programming

Total number of function calls



15



9

fib(n) = n + 3

O(n)

The complexity reduces from exponential to polynomial or linear

Hasan Baig

9

CSE-2050 – Data Structures and Object-Oriented Design

Review

10

Module 5 – Recursion and Dynamic Programming

Dynamic Programming –

Bottom up Approach using tabulation (iterative) method

0	1	1	2	3	5

```
def fibo_dyn(k):
    if k <= 1:
        return k

    F = [0, 1]

    for i in range(2, k+1, 1):
        F.append( F[i - 1] + F[i - 2] )
    return F[k]
```

Hasan Baig

10

Module 6 – Searching and Sorting

Binary Search
Search for an element in a sorted array by dividing the search interval in half.

1. Start by examining the middle item → return if it is the required item
2. If the required item is less than the middle item, disregard the upper half of the search space. If it is greater than the middle item, disregard the lower half of the search space.
3. Repeat until the required item is found or until the search space becomes empty

Module 6 – Searching and Sorting

```
student_ids = [1432, 1433, 1434, 1435, 1436, 1437, 1438, 1439, 1440, 1441, 1442, 1443, 1444, 1445, 1446, 1447,
1448, 1449, 1450, 1451, 1452, 1453, 1454, 1455, 1456, 1457, 1458, 1459, 1460, 1461, 1462, 1463,
1464, 1465, 1466, 1467, 1468, 1469, 1470, 1471, 1472, 1473, 1474, 1475, 1476, 1477, 1478, 1479,
1480, 1481, 1482, 1483, 1484, 1485, 1486, 1487, 1488, 1489, 1490, 1491, 1492, 1493, 1494, 1495,
1496, 1497, 1498, 1499, 1500, 1501, 1502, 1503, 1504, 1505, 1506, 1507, 1508, 1509, 1510, 1511,
1512, 1513, 1514, 1515, 1516, 1517, 1518, 1519, 1520, 1521, 1522, 1523, 1524, 1525, 1526, 1527,
1528, 1529, 1530, 1531, 1532, 1533, 1534, 1535, 1536, 1537, 1538, 1539, 1540, 1541, 1542, 1543,
1544, 1545, 1546, 1547, 1548, 1549, 1550, 1551, 1552, 1553, 1554, 1555, 1556, 1557, 1558, 1559,
1560, 1561, 1562, 1563, 1564, 1565, 1566, 1567, 1568, 1569, 1570, 1571, 1572, 1573, 1574, 1575,
1576, 1577, 1578, 1579, 1580, 1581, 1582, 1583, 1584, 1585, 1586, 1587, 1588, 1589, 1590, 1591,
1592, 1593, 1594, 1595, 1596, 1597, 1598, 1599, 1600, 1601, 1602, 1603, 1604, 1605, 1606, 1607,
1608, 1609, 1610, 1611, 1612, 1613, 1614, 1615, 1616, 1617, 1618, 1619, 1620, 1621, 1622, 1623,
1624, 1625, 1626, 1627, 1628, 1629, 1630, 1631, 1632, 1633, 1634, 1635, 1636, 1637, 1638, 1639,
1640, 1641, 1642, 1643, 1644, 1645, 1646, 1647, 1648, 1649, 1650, 1651, 1652, 1653, 1654, 1655,
1656, 1657, 1658, 1659, 1660, 1661, 1662, 1663, 1664, 1665, 1666, 1667, 1668, 1669, 1670, 1671,
1672, 1673, 1674, 1675, 1676, 1677, 1678, 1679, 1680, 1681, 1682, 1683, 1684, 1685, 1686, 1687,
1688, 1689, 1690, 1691, 1692, 1693, 1694, 1695, 1696, 1697, 1698, 1699]
```

CSE-2050 – Data Structures and Object-Oriented Design

Review

13

Module 6 – Searching and Sorting

student_ids = [1634, 1635, 1636, 1637, 1638, 1639, 1640, 1641, 1642, 1643, 1644, 1645, 1646, 1647, 1648, **1649**, 1650, 1651, 1652, 1653, 1654, 1655, 1656, 1657, 1658, 1659, 1660, 1661, 1662, 1663, 1664, 1665]

Hasan Baig

13

CSE-2050 – Data Structures and Object-Oriented Design

Review

14

Module 6 – Searching and Sorting

student_ids = [… **1651** …]

Hasan Baig

14

CSE-2050 – Data Structures and Object-Oriented Design

Review

15

Module 6 – Searching and Sorting

```
def BS(L, item):
    if len(L) == 0:
        return False
    mid_index = len(L) // 2
    if item == L[mid_index]:
        return True
    elif item < L[mid_index]:
        return BS(L[ : mid_index], item)
    else:
        return BS(L[mid_index + 1 : ], item)
```

Using Slicing → O(n)

Hasan Baig

15

CSE-2050 – Data Structures and Object-Oriented Design

Review

16

Module 6 – Searching and Sorting

```
def BS_improved(L, item, lower, upper):
    if lower > upper:
        return False
    else:
        mid_index = (lower + upper) // 2
        if item == L[mid_index]:
            return True
        elif item < L[mid_index]:
            return BS_improved(L, item, lower, mid_index - 1)
        else:
            return BS_improved(L, item, mid_index + 1, upper)
```

Slicing removed

Hasan Baig

16

Module 6 – Searching and Sorting

0^{th} Iteration:
Data size = n

1^{st} Iteration:
Data size = n/2 or $n/2^1$

2^{nd} Iteration:
Data size = (n/2)/2 → n/4 or $n/2^2$

••••

••••

At K^{th} Iteration, the data size becomes 1:

$$n/2^k = 1$$
$$n = 2^k$$
$$\log n = \log 2^k$$
$$\rightarrow O(\log n)$$

Module 6 – Searching and Sorting

Bubble Sort Algorithm

```
def bubble_sort(L):
    for el in range(len(L) - 1):
      for i in range(len(L) - 1 - el):
        if L[i] > L[i+1]:             #If two items are out of order
          L[i], L[i+1] = L[i+1], L[i]   #Switch them
```

i = 0

el = 0

9	8	7	6	5
8	9	7	6	5
8	7	9	6	5
8	7	6	9	5
8	7	6	5	9

el = 1

8	7	6	5	9
7	8	6	5	9
7	6	8	5	9
7	6	5	8	9

el = 2

7	6	5	8	9
6	7	5	8	9
6	5	7	8	9

el = 3

6	5	7	8	9
5	6	7	8	9

$O(n^2)$

Module 6 – Searching and Sorting

Selection Sort Algorithm

- Selection sort algorithm is another approach of sorting data in **O(n²)** quadratic running time
- We can either find the smallest item and place it in the beginning

OR

- Find the biggest item and move it to end

```python
def SS_min(L):
    for i in range(len(L) - 1):
        min = i
        for j in range(i + 1, len(L)):
            if L[j] < L[min]:
                min = j
        #swap
        L[i], L[min] = L[min], L[i]
```

```python
def SS_max(L):
    for i in range(len(L) - 1):
        max = 0
        for j in range(1, len(L)-i):
            if L[j] > L[max]:
                max = j
        #swap
        L[-1 - i], L[max] = L[max], L[-1 - i]
```

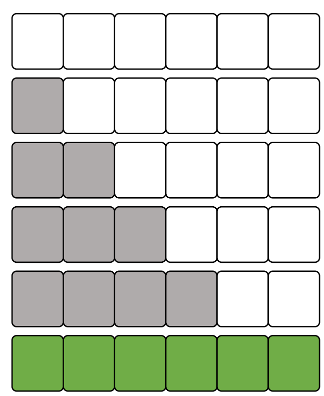
- Selection sort is better for applications where less number of write operations are required

Module 6 – Searching and Sorting

Insertion Sort Algorithm

- Online algorithm – sort array as it receives data (example from web)

```python
def insertion_sort(L):
    for i in range(1 ,len(L)):
        j = i
        while j>0 and L[j] < L[j-1]:
            L[j-1], L[j] = L[j], L[j-1]
            j -= 1
```

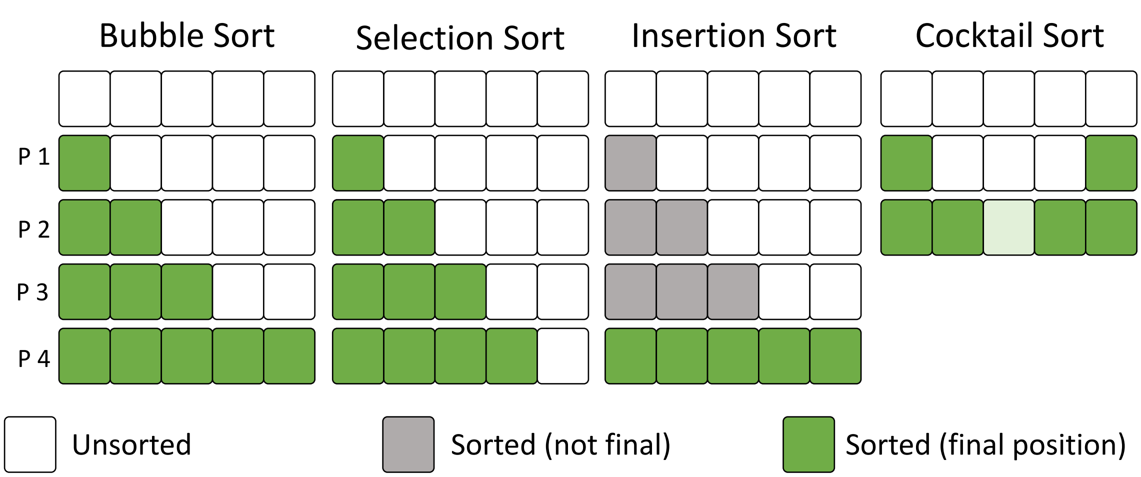


Module 6 – Searching and Sorting

- **Bubble sort**
  - Iterates over every pair in collection, swaps out of order pairs
  - After x iterations, the last x items are in their final (sorted) place
- **Selection sort**
  - Iterates over every unsorted item in collection, selects the next smallest/biggest
  - After x iterations, the last x items are in their final (sorted) place
- **Insertion sort**
  - Iterates over a progressively growing sorted section of the list
  - Bubbles the next un-sorted item into place
  - After x iterations, the first x items are sorted but may not be in their final place.
- **Cocktail sort**
  - Invariant of bubble sort
  - After x iterations, smallest and largest elements are placed at the right place

Module 6 – Searching and Sorting

Summary



Module 7 – Divide and Conquer

- Divide and Conquer is a paradigm for algorithm design and consists of the following three steps:

1. ***Divide:*** Divide the input data D into two or more disjoint subsets, D_1 and D_2.
2. ***Conquer:*** Recursively solve the subproblems associated with the subsets, D_1 and D_2.
3. ***Combine:*** Take the solutions to the subproblems, D_1 and D_2, and merge them into a solution to the original problem D.

Base case: Base case for the recursion are subproblems of size 0 or 1.

Module 7 – Divide and Conquer

Merge-Sort Algorithm

85	24	63	45	17	31	96	50

```
Algorithm mergeSort(D)
  Input sequence D with n
        elements
  Output sequence D sorted

if D.size() > 1
  (D1, D2) ← partition(D, n/2)
  mergeSort(D1)
  mergeSort(D2)
  D ← merge(D1, D2)
```

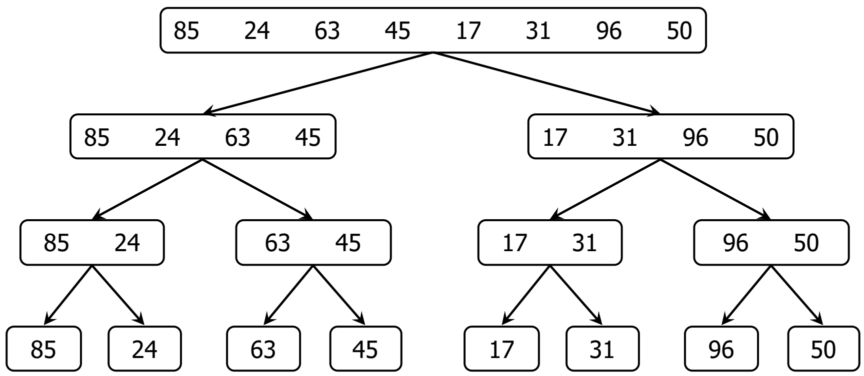
CSE-2050 – Data Structures and Object-Oriented Design

Review

25

Module 7 – Divide and Conquer

Merge-Sort Algorithm




```
Algorithm mergeSort(D)
  Input  sequence D with n
         elements
  Output sequence D sorted

if D.size() > 1
  (D₁, D₂) ← partition(D, n/2)
  mergeSort(D₁)
  mergeSort(D₂)
  D ← merge(D₁, D₂)
```

Hasan Baig

25

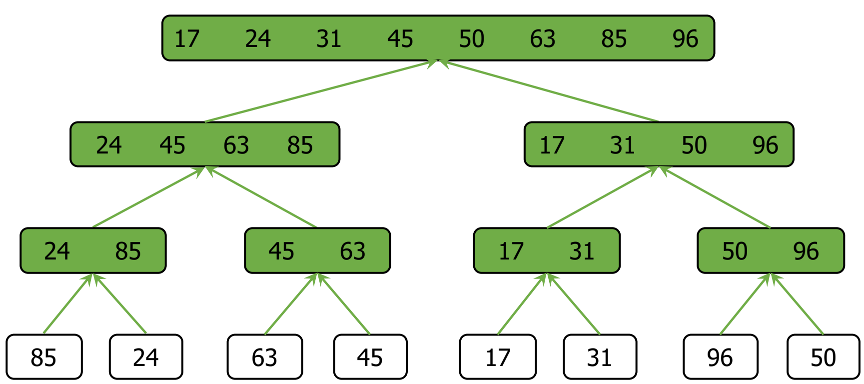
CSE-2050 – Data Structures and Object-Oriented Design

Review

26

Module 7 – Divide and Conquer

Merge-Sort Algorithm




```
Algorithm mergeSort(D)
  Input  sequence D with n
         elements
  Output sequence D sorted

if D.size() > 1
  (D₁, D₂) ← partition(D, n/2)
  mergeSort(D₁)
  mergeSort(D₂)
  D ← merge(D₁, D₂)
```

Hasan Baig

26

Module 7 – Divide and Conquer

Merge-Sort Algorithm

D1

24	**45**	**63**	**85**

i=0

D2

17	**31**	**50**	**96**

j=0

D

i+j=0

```
merge(D1, D2, D):
    i=j=0
    while i < len(D1) and while j < len(D1)
        if D1[i] < D2[j]:
            D[i+j] = D1[i]
            i += 1
        else:
            D[i+j] = D2[j]
            j += 1

    D[i+j:] = D1[i : ] + D2[j : ]
```


Module 7 – Divide and Conquer

Merge-Sort Algorithm

D1

24	**45**	**63**	**85**

i=0

D2

17	**31**	**50**	**96**

j=0

D

i+j=0

If:
D1 contains m1 elements
D2 contains m2 elements

→ O(m1 + m2)

```
merge(D1, D2, D):
    i=j=0
    while i < len(D1) and while j < len(D1)
        if D1[i] < D2[j]:
            D[i+j] = D1[i]
            i += 1
        else:
            D[i+j] = D2[j]
            j += 1

    D[i+j:] = D1[i : ] + D2[j : ]
```

CSE-2050 – Data Structures and Object-Oriented Design

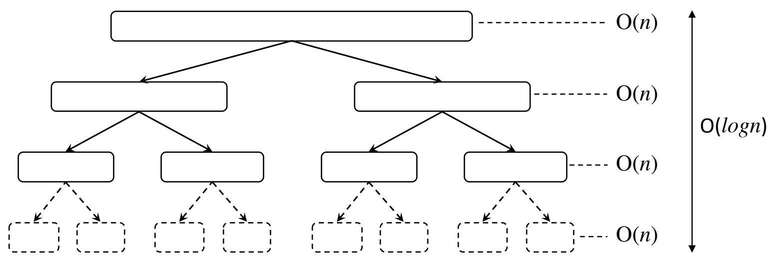
Review

29

Module 7 – Divide and Conquer

Merge-Sort Algorithm

Depth	# nodes/sequences	size
0	1	n
1	2	n/2
2	4	$n/2^2$
...	...	...
i	2^i	$n/2^i$



- The amount of work done at each node is merge + partition
  - Total work done at depth i is: number of nodes x size of nodes = 2^i x $n/2^i$ → O(n)
- What is the stopping condition of recursion? → O($logn$)

➔ Total time complexity = **O($n$log$n$)**

Hasan Baig

29

CSE-2050 – Data Structures and Object-Oriented Design

Review

30

Module 7 – Divide and Conquer

Quick-Sort Algorithm

85	24	63	45	17	31	96	50

```
Algorithm quickSort(D)
 Input sequence D with n
       elements
 Output sequence D sorted

if D.size() > 1
 pivot ← pick x from D
 L ← elements less than x
 E ← element equal to x
 G ← elements greater than x
 L = quickSort(L)
 G = quickSort(G)
 return L + E + G
else:
 return D
```

Hasan Baig

30

Module 7 – Divide and Conquer

Quick-Sort Algorithm

- Picking last element in the list as pivot

85	24	63	45	17	31	96	**50**

```
Algorithm quickSort(D)
 Input sequence D with n
       elements
 Output sequence D sorted

if D.size() > 1
 pivot ← pick x from D
 L ← elements less than x
 E ← element equal to x
 G ← elements greater than x
 L = quickSort(L)
 G = quickSort(G)
 return L + E + G
else:
 return D
```


Module 7 – Divide and Conquer

Quick-Sort Algorithm

- Picking last element in the list as pivot

24	45	17	31	**50**	85	63	96

```
Algorithm quickSort(D)
 Input sequence D with n
       elements
 Output sequence D sorted

if D.size() > 1
 pivot ← pick x from D
 L ← elements less than x
 E ← element equal to x
 G ← elements greater than x
 L = quickSort(L)
 G = quickSort(G)
 return L + E + G
else:
 return D
```

CSE-2050 – Data Structures and Object-Oriented Design

Review

33

Module 7 – Divide and Conquer

Quick-Sort Algorithm

- Picking last element in the list as pivot

24	45	17	31	**50**	85	63	96

24	45	17	31

```
Algorithm quickSort(D)
  Input sequence D with n
    elements
  Output sequence D sorted

if D.size() > 1
  pivot ← pick x from D
  L ← elements less than x
  E ← element equal to x
  G ← elements greater than x
  L = quickSort(L)
  G = quickSort(G)
  return L + E + G
else:
  return D
```

Hasan Baig

33

CSE-2050 – Data Structures and Object-Oriented Design

Review

34

Module 7 – Divide and Conquer

Quick-Sort Algorithm

- Picking last element in the list as pivot

24	45	17	31	**50**	85	63	96

24	45	17	**31**

```
Algorithm quickSort(D)
  Input sequence D with n
    elements
  Output sequence D sorted

if D.size() > 1
  pivot ← pick x from D
  L ← elements less than x
  E ← element equal to x
  G ← elements greater than x
  L = quickSort(L)
  G = quickSort(G)
  return L + E + G
else:
  return D
```

Hasan Baig

34

CSE-2050 – Data Structures and Object-Oriented Design

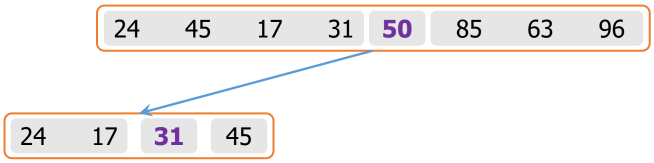
Review

35

Module 7 – Divide and Conquer

Quick-Sort Algorithm

- Picking last element in the list as pivot




```
Algorithm quickSort(D)
  Input sequence D with n
        elements
  Output sequence D sorted

if D.size() > 1
  pivot ← pick x from D
  L ← elements less than x
  E ← element equal to x
  G ← elements greater than x
  L = quickSort(L)
  G = quickSort(G)
  return L + E + G
else:
  return D
```

Hasan Baig

35

CSE-2050 – Data Structures and Object-Oriented Design

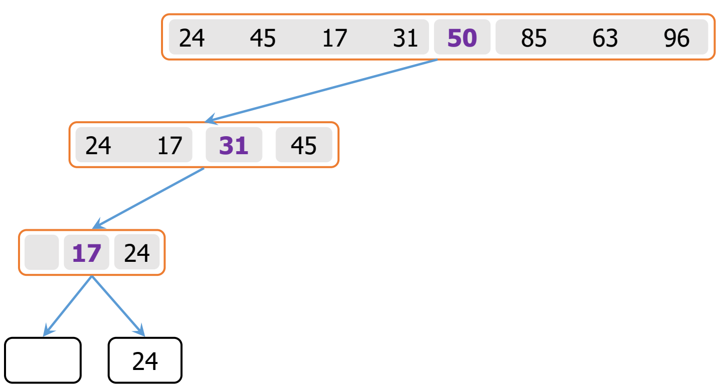
Review

36

Module 7 – Divide and Conquer

Quick-Sort Algorithm

- Picking last element in the list as pivot




```
Algorithm quickSort(D)
  Input sequence D with n
        elements
  Output sequence D sorted

if D.size() > 1
  pivot ← pick x from D
  L ← elements less than x
  E ← element equal to x
  G ← elements greater than x
  L = quickSort(L)
  G = quickSort(G)
  return L + E + G
else:
  return D
```

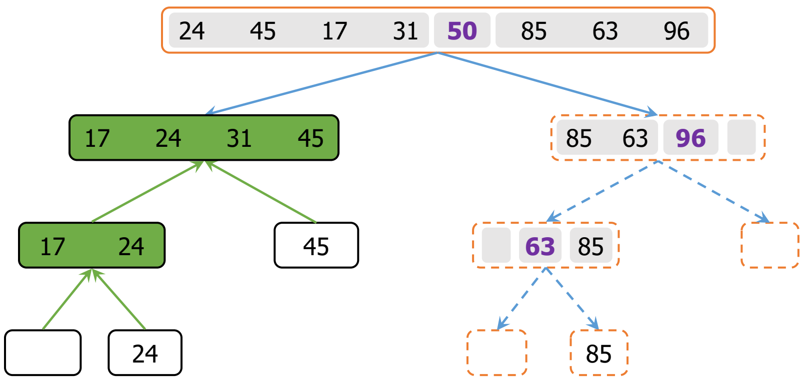
Hasan Baig

36

Module 7 – Divide and Conquer

Quick-Sort Algorithm

- Picking last element in the list as pivot




```
Algorithm quickSort(D)
 Input sequence D with n
       elements
 Output sequence D sorted

if D.size() > 1
 pivot ← pick x from D
 L ← elements less than x
 E ← element equal to x
 G ← elements greater than x
 L = quickSort(L)
 G = quickSort(G)
 return L + E + G
else:
 return D
```


Module 7 – Divide and Conquer

Quick-Sort Algorithm (in-place) implementation

- Data items are manipulated within the same container for sorting → thus saves memory

Module 7 – Divide and Conquer

Quick-Sort Algorithm (in-place) implementation

```
quickSort(D, L_Idx, H_Idx)
    if L_Idx < H_Idx:

        pivot = partition(D, L_Idx, H_Idx)
        quickSort(D, L_Idx, pivot-1)
        quickSort(D, pivot+1, H_Idx)
end
```

D

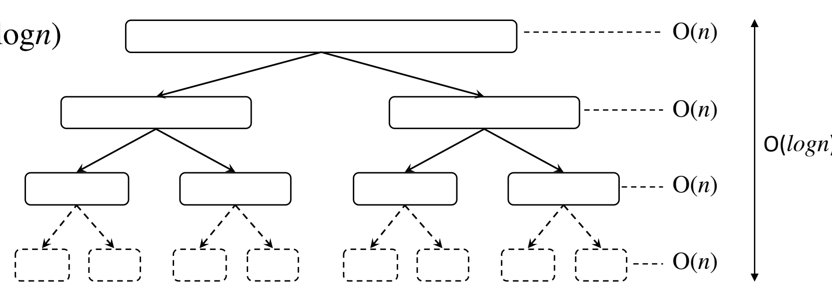
23	**6**	**4**	**-1**	**0**	**12**	**8**	**3**	**1**

L_Idx=0 H_Idx=8

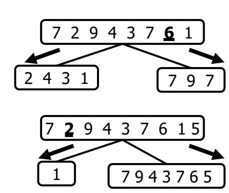
```
partition(D, low, high)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```


Module 7 – Divide and Conquer

- Depends on pivot
- Best case runtime → $O(n\log n)$



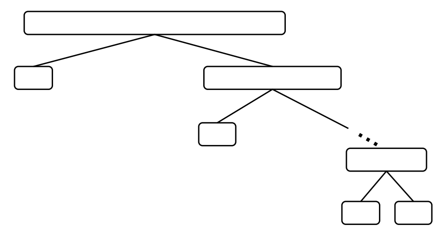
- For a sequence of size D
  - Good Pivot → generates L and G each of size less than ¾D
  - Bad Pivot → generates either L or G of size greater than ¾D



Module 7 – Divide and Conquer

- The worst case for quick-sort occurs when the pivot is the unique minimum or maximum element

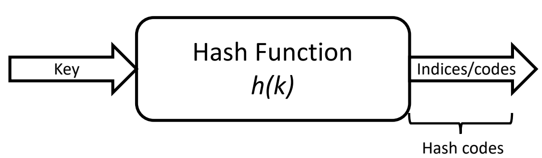
depth	time
0	n
1	$n-1$
…	…
$n-1$	1



- Worst case runtime → $O(n^2)$

Module 8 – Mapping and Hash Tables

- Mapping → association between two objects → key-value pairs
  - Python has dictionary data structure to hold key-value pairs
  - To map keys, hash function is used to generate the index location

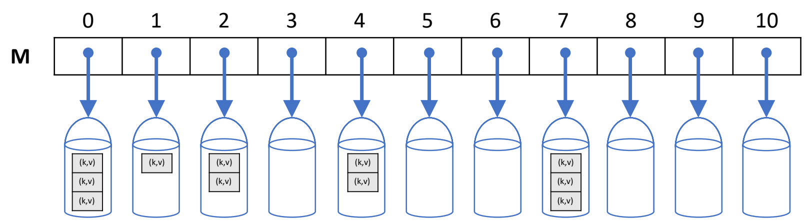


- Hash Collision → hash function generate the same code for different keys to map at the same location

Module 8 – Mapping and Hash Tables

Collision Handling scheme

- Separate Chaining
  - Instead of having a single object at each location in hash table, we conceptualized to have buckets



- In Worst case:
  - Time to search for the bucket is O(1)
  - Time to search a key in the bucket depends on the size of bucket: For size n → O(n)

CSE-2050 – Data Structures and Object-Oriented Design

Module 8 – Mapping and Hash Tables

Collision Handling scheme

- Open addressing – Linear Probing
  - Inserting an element (k,v) at M[j]
    - j is the index generated by hash function
  - If j^{th} place is occupied, we try M[(j+1) % N]
  - If this place is also occupied, we next try M[(j+2) % N], and so on

CSE-2050 – Data Structures and Object-Oriented Design

45

Module 8 – Mapping and Hash Tables

Collision Handling scheme

- Inserting a new element with key k = 15 → k mod N → 15 mod 11 = 4
- This new item should be placed at location 4

M				Ali	Mia	Tim	Sam	15			
	0	1	2	3	4	5	6	7	8	9	10

- This requires additional implementation to search for an existing key
- Accessing cell array is analogous to <u>probing</u> the bucket to find its content

Hasan Baig

45

CSE-2050 – Data Structures and Object-Oriented Design

46

Module 8 – Mapping and Hash Tables

- A good hash function index “n” items of a map in a bucket array of capacity “N”
  → The expected size of a bucket is n/N

- The ration λ = n/N is called “**Load Factor**” of the hash table
  - Bounded by a small constant (preferably below 1)

  - Example:
    - n = 15
    - N = 10

    ➔ λ = 1.5 → collision

	0	1	2	3	4	5	6	7	8	9
M										

Hasan Baig

46

Announcements

- Exam Duration – 50 minutes
  - Total points: 30
  - Total 4 sections → 6 MCQs in each
  - All questions carry 1 point except one that carry 2.5 points
  - Question # 25 is a bonus question which carries 5 points
- No Labs this week

if D has n elements
then looping through D:

for i in range(len(D))
....

will have $O(n)$ time complexity!
This is very simple!

Now, if we say D has nxn elements
i.e n^2 elements, then looping through
D will have $O(n^2)$ time complexity.