UNIVERSITY OF
LEICESTER

CO1107
Data Structure

Merge Sort

Divide and Conquer Sorting

Divide and Conquer Paradigm

- Divide the problem into smaller sub-problems
- Conquer (solve) each sub-problem and combine the results

Divide and Conquer Sorting Algorithms

- Merge Sort
- Quick Sort

Merge Sort Algorithms

- Two functions are involved:

➢ The mergeSort() function recursively call itself to divide the list till size become one.

➢ The merge() function is used to merge the two halves.

Merge Sort

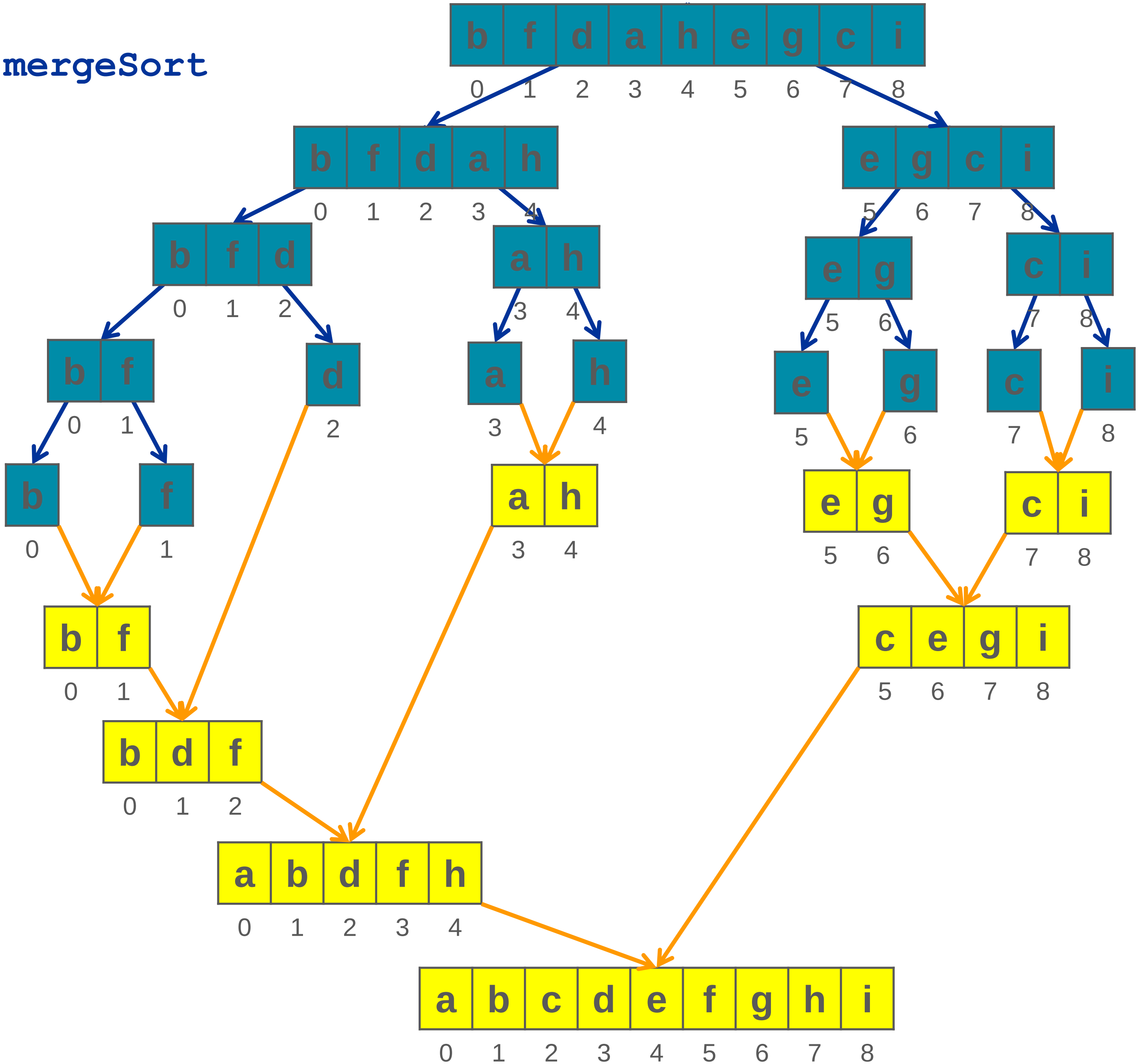
Divide Step

- Divide the list into sub-lists each of length 1

Conquer Step

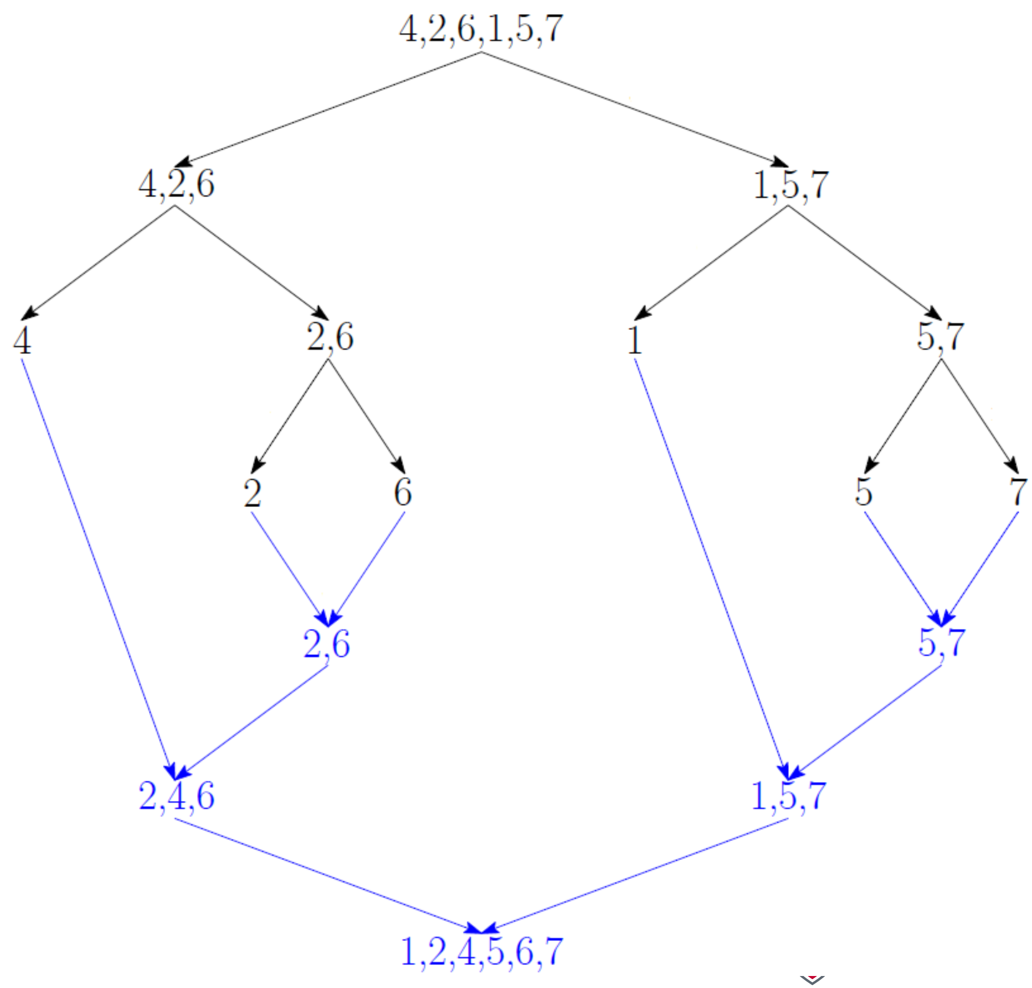
- Repeatedly merge sub-lists to produce new sorted sub-lists until there is only one sorted list

mergeSort


b f d a h e g c i
0 1 2 3 4 5 6 7 8
b f d a h
0 1 2 3 4
e g c i
5 6 7 8
b f d
0 1 2
a h
3 4
e g
5 6
c i
7 8
b f
0 1
d
2
a
3
h
4
e
5
g
6
c
7
i
8
b
0
f
1
a h
3 4
e g
5 6
c i
7 8
b f
0 1
c e g i
5 6 7 8
b d f
0 1 2
a b d f h
0 1 2 3 4
a b c d e f g h i
0 1 2 3 4 5 6 7 8

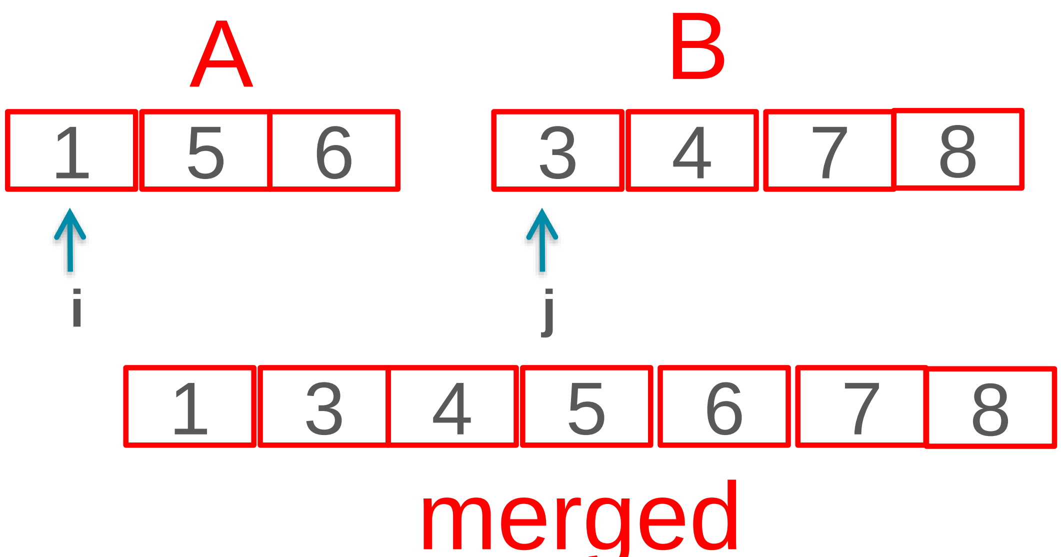
Class Activity

- Sort the following list using Merge Sort.
- L = [4,2,6,1,5,7]



Merging two sorted lists

- Repeat until i or j reach the end of the corresponding list
  - Compare elements at i and j
  - Insert the smaller in merged and increment its pointer
- Append remaining items in the unfinished list to merged



Merging two sorted lists

```
def mergeLists(A, B, merged):
    i,j=0,0 #initialize i and j to 0
    # repeat until reaches the end of at least one list
    while i < len(A) and j < len(B):
        # insert the smaller element in merged and increment pointer
        if A[i] < B[j]:
            merged.append(A[i])
            i=i+1
        else:
            merged.append(B[j])
            j=j+1
    # if A/B is unfinished, add remaining elements to merged
    if i < len(A):
        merged += A[i:]
    if j < len(B):
        merged += B[j:]
A = [1,5,6] # example from previous slide
B = [3,4,7,8]
```

Merge Sort Python Implementation

- Live Demo

Introduction to Time Complexity

Running Time

Depends on a number of factors including:

- The input
- The quality of the code generated by the compiler
- The machine used to execute the program
- The **time complexity** of the algorithm

Running time in RAM model

- Each “simple” operation (e.g., +,-,*,=,+= etc.) take one time step
- Each read, print, and return statement takes one time step.
- Each comparison takes one time step
- The running time of the sequence of statements is the sum of running times of the statements.
- Loops and functions are considered as the composition of many simple operations, and their running time depends upon how many times each of these simple operations are performed.

According to the RAM model, what is the running time for power(2, 5)?

A. 5
B. 18
C. 19
D. None of the above

```
def power(x, N):
    'computes x to the power of N'

    value = 1
    k = 1
    while k <= N:
        value *= x
        k += 1

    return value
```

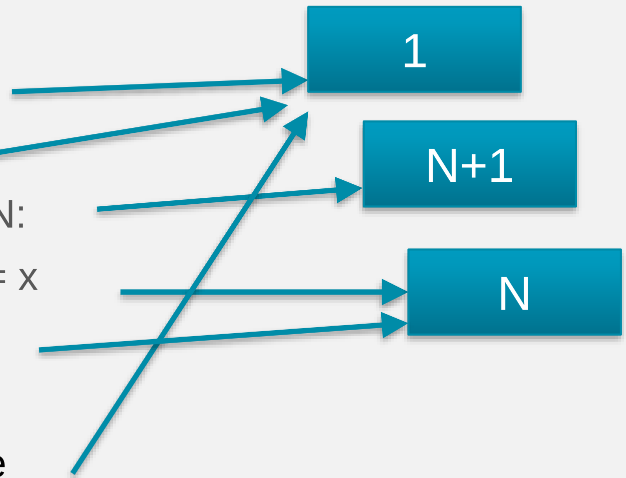
C

Power

```
def power(x, N):
    'computes x to the power of N'

    value = 1
    k = 1
    while k <= N:
        value *= x
        k += 1

    return value
```



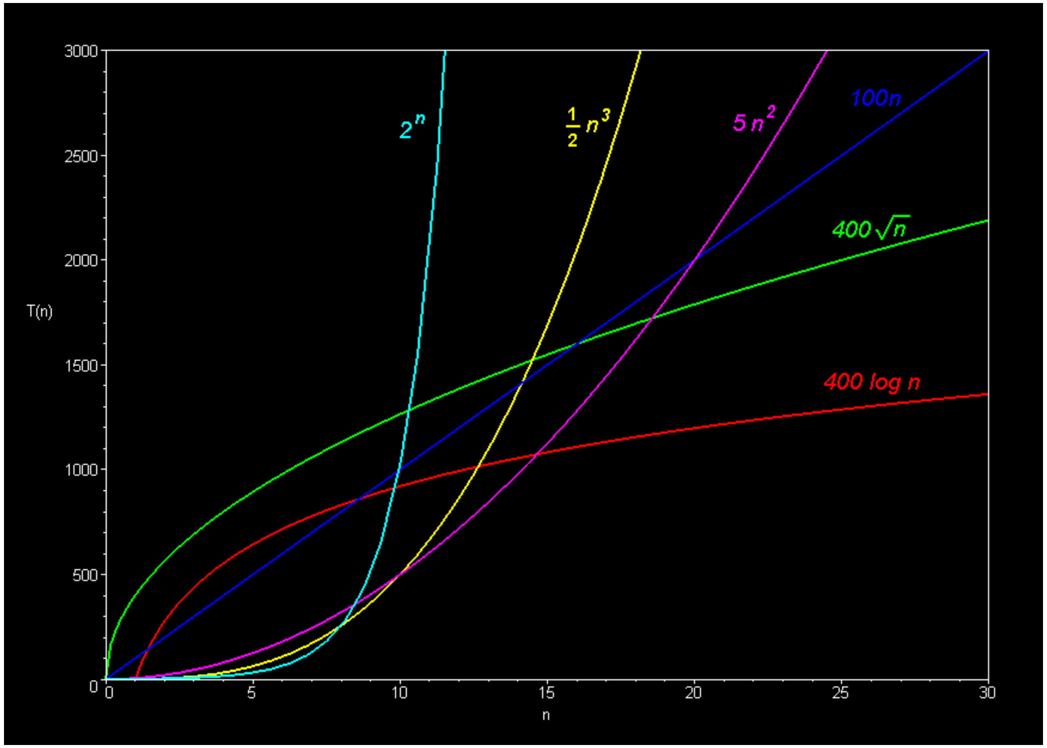
Total running time = 3 + (N + 1) + 2N= 3N + 4

Big O Notation

- The complexity of an algorithm is described using a language called Big O Notation.
- It is how we compare the efficiency of different approaches to a problem.
- With Big O Notation we express the runtime in terms of—how quickly it grows relative to the input, as the input gets larger

Big O notation

- Typically, we use the following simplification rules
  - If f(N) is a product of several terms, any constants that do not depend on N can be ignored
  - If f(N) is a sum of several terms, if there is one with the largest growth rate, it can be kept and others can be omitted
- E.g.,
  - **$12 N^2 + 4 N^3$**
    - **→ $O(N^3)$**
  - **$12 N^2 + 3 N \log(N)$**
    - **→ $O(N^2)$**
  - **$8N^4 + N^2 \log(N) + 12000$**
    - **→ $O(N^4)$**
  - **$1000 + 5000$**
    - **→ $O(N^0)$ → $O(1)$**



What is the complexity of an algorithm in Big-O notation that runs in $8N^3 + 17 N^2 + 150$?

A. $O(8N^3)$
B. $O(N^3 + N^2)$
C. $O(N^3)$
D. $O(8N^3)$
E. $O(8N^3 + 17 N^2 + 150)$
F. None of the above

C

Complexity of power in big-O

```
def power(x, N):
    ’computes x to the power of N’

    value = 1
    k = 1
    while k <= N:
        value *= x
        k += 1

    return value
```

Total running time = 3 + (N + 1) + 2N= 3N + 4
Complexity → O(N)

Order Algorithmic Time Complexity

- The following are in order of increasing time complexity:
    - Constant $O(1)$
    - Logarithmic $O(\log N)$
    - Linear $O(N)$
    - Superlinear $O(N \log N)$
    - Quadratic $O(N^2)$
    - Exponential $O(2^N)$
    - Factorial $O(N!)$

Constant O(1)

- All instructions are performed a fixed amount of times
- Example:

  Print the first number in a list

- The algorithm does not depend on N.
- If N doubles, its running time T remains constant

Logarithmic O(log N)

- Problem is broken up into smaller problems and solved independently. Each step cuts the size by a constant factor
- Example:

Binary search algorithm

- If N doubles, running time T gets slightly slower (T and a bit)

Linear O(N)

- Each element requires a certain (fixed) amount of processing
- Example:

Linear search

- If N doubles, running time T doubles (2*T)

Superlinear O(N log N)

- Problem is broken up into smaller problems and solved independently. Each step cuts the size by a constant factor and the final solution is obtained by combining the sub-solutions.
- Example:

Merge sort

- If N doubles, running time T gets slightly bigger than double (2*T and a bit)

Quadratic $O(N^2)$

- Processes all pairs of data items
  - Often occurs when you have double nested loop

- Example:

Insertion Sort

- If N doubles, running time T increases four times (4*T)

Exponential $O(2^N)$

- Combinatorial explosion
- Example:

  Finding all the subsets of N items

- If N doubles, running time T squares (T*T)

Factorial O(N!)

- Example:

  Finding all the permutations of N items

- Impractical for N > 20

Class Exercises

What is the time complexity:

Print the first number in a list

- All instructions are performed a fixed amount of times
- The algorithm does not depend on N.
- If N doubles, its running time T remains constant
- So : the time complexity is Constant O(1)

What is the time complexity:

```
def function2(aList):
    N = len(aList)
    value = 0
    for i in range(N):
        for j in range(0,2*N,4):
            value += i*j
    return value
```

O(**N**^{2}) as outer loop runs N times and inner loop runs roughly N/2 times.

What is the time complexity:

```
def function1(aList):
    N = len(aList)
    value = 0
    for i in range(N//2):
        for j in range(100):
            value += i*j
    return value
```

O(N) as outer loop runs N//2 times and inner loop runs 100 times.

What is the time complexity:

```python
def fraction_func(n):
    fraction = 1
    for k in range(100):
        for j in range(k):
            fraction = k + j + 1/fraction
    return fraction
```

- O(1) . the outer loop is constant (not dependent on n) and the inner loop is dependent on the outer loop's variable, this makes it 100*k and given k is at most 100 this is at most 100*100 which is still O(1)

What is the time complexity:

```
def test_func(n):
    total = 0
    for k in range(n):
        for j in range(n-k, 0, -1):
            total += k*j
    return total
```

- O(N*N)