Faculty of Information Technology, Monash University

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

This material has been reproduced and communicated to you by or on behalf of Monash University pursuant to Part VB of the Copyright Act 1968 (the Act). The material in this communication may be subject to copyright under the Act. Any further reproduction or communication of this material by you may be the subject of copyright protection under the Act. Do not remove this notice

FIT2004: Algorithms and Data Structures

Week 2: Analysis of Algorithms

These slides are prepared by M. A. Cheema and are based on the material developed by Arun Konagurthu and Lloyd Allison.

Things to Note

- Consultation times
  - My consultation moved from Thursday 4:15pm-5pm
  - Two more consultations:
    - Wednesday 2pm-3pm
    - Friday 1pm-2pm
  - Details on Moodle
- Tutorial week 2 have been uploaded
  - You need to attempt the questions under "Assessed preparation" before your lab this week to meet the hurdle for participation marks
- Assignment 1 has been released (due 24 March 2019 at 23:55:00)
  - Start early! Don't live dangerously
  - Seek help if struggling
  - Don't submit version

Recommended reading

- Basic mathematics used for algorithm analysis: http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Math/
- Program verification: http://www.csse.monash.edu.au/courseware/cse2304/2006/03logic.shtml
- Section 1,2 and 3.5 of Unit Notes
- For more about Loop invariants: Also read Cormen et al. Introduction to Algorithms, Pages 17-19, Section 2.1: Insertion sort.).

Outline

Complexity Analysis

A. Introduction/Recap (covered last week)
B. Finding minimum
C. Binary Search
D. Comparison-based Sorting Algorithms
   I. Selection Sort
   II. Insertion Sort
   III. Lower bound for comparison-based sorting
E. Non-comparison Sorting Algorithms
   I. Counting Sort
   II. Radix Sort
F. Recursive Algorithms

Time Complexity: Finding minimum value

```
//Find minimum value in an unsorted array of N>0 elements
min = array[1]
index = 2

while index <= N

      if array[index] < min
            min = array[index]
      index = index + 1

return min
```

Time Complexity?

- Worst-case
  - O(N)
- Best-case
  - O(N)
  - We cannot say best-case is when N=1. Complexity must be defined in terms of input size N.
- Average
  - O(N)

Auxiliary Space Complexity

- **Space complexity** is the total amount of space taken by an algorithm as a function of input size
- **Auxiliary space complexity** is the amount of space taken by an algorithm in addition to the space taken by the input
  - Many textbooks and online resources do not distinguish between the above two terms and use the term “space complexity” when they are in fact referring to auxiliary space complexity. In this unit, we use these two terms to differentiate b/w them.

Example:

- Merge() in merge sort merges two sorted lists A and B. Assume total # of elements in A and B is N.
- What is the space complexity?
- What is the auxiliary space complexity?
- **In-place algorithm:** An algorithm that has O(1) auxiliary space complexity
  - i.e., it only requires constant space in addition to the space taken by input
  - Merging is not an in-place algorithm
    - Be mindful that some books use a different definition (e.g., space taken by recursion may be ignored). For the sake of this unit, we will use the above definition.

A

1	6

B

3	9	10

merged

1	3	6	9	10

Space Complexity: Finding minimum

```
//Find minimum value in an unsorted array of N>0 elements
min = array[1]
index = 2

while index <= N

      if array[index] < min
            min = array[index]
      index = index + 1

return min
```

- Space complexity?
  - O(N)
- Auxiliary space complexity?
  - O(1)
- This is an in-place algorithm

Outline

Complexity Analysis

A. Introduction/Recap (covered last week)
B. Finding minimum
C. Binary Search
D. Comparison-based Sorting Algorithms
   I. Selection Sort
   II. Insertion Sort
   III. Lower bound for comparison-based sorting
E. Non-comparison Sorting Algorithms
   I. Counting Sort
   II. Radix Sort
F. Recursive Algorithms

Time/Space Complexity: Binary Search

```
lo = 1
hi = N + 1

while ( lo < hi - 1 )
    mid = floor( (lo+hi)/2 )
    if key >= array[mid]
        lo=mid
    else
        hi=mid

if N > 0 and array[lo] == key
    print(key found at index lo)
else
    print(key not found)
```

Time Complexity?

- Worst-case
  - Search space at start: N
  - Search space after 1st iteration: N/2
  - Search space after 2nd iteration: N/4
  - …
  - Search space after x-th iteration: 1

  What is x? i.e., how many iterations in total?
  O(log N)
- Best-case
  - Can be improved to O(1) by returning key when key == array[mid]

Space Complexity?

- O(N)

Auxiliary Space Complexity?

- O(1)

Binary search is an in-place algorithm!

Outline

Complexity Analysis

A. Introduction/Recap (covered last week)
B. Finding minimum
C. Binary Search
D. Comparison-based Sorting Algorithms
   I. Selection Sort
   II. Insertion Sort
   III. Lower bound for comparison-based sorting
E. Non-comparison Sorting Algorithms
   I. Counting Sort
   II. Radix Sort
F. Recursive Algorithms

Comparison-based Sorting

- Comparison-based sorting algorithms sort the input array by comparing the items with each other. E.g.,
  - Selection Sort
  - Insertion Sort
  - Quick Sort (to be analysed next week)
  - Merge Sort
  - Heap Sort
  - …

- The algorithms that do not require comparing items with each other are called non-comparison sorting algorithms. E.g., Counting sort, radix sort, bucket sort etc.

Comparison Cost

- Typically, we assume that comparing two elements takes O(1), e.g., array[i] <= array[j]. This is not necessarily true.
- String Comparison: The worst-case cost of comparing two strings is O(L) where L is the number of characters in the smaller string. E.g.,
  - "Welcome to Faculty of IT" <= "Welcome to FIT2004" ??
  - We compare strings character by character (from left to right) until the two characters are different – all green letters are compared in above example
- Number Comparison: Similarly, the worst-case cost to compare two numbers is O(L) where L is the number of digits in the smaller number.
  - Note that for a number N, the number of digits is O(log N).

Comparison Cost

- Typically, we assume the comparison cost to be O(1) because we usually don't deal with numbers having a lot of digits and strings having a lot of characters.
- However, in many cases, comparison cost is a critical factor. E.g., genome sequences may have millions of characters which makes comparing two sequences very expensive.
- The cost of comparison-based sorting is often taken as in terms of # of comparisons, e.g., # of comparisons in merge sort is O(N log N)

In this unit, **unless specified otherwise**, we will assume that comparison cost is O(1).

- E.g., For Assignment 1, the comparison cost is O(L) because the assignment specifies that comparison cost for two strings is **not** O(1).

Stable sorting algorithms

A sorting algorithm is called stable if it maintains the relative ordering of elements that have equal keys.

Input is sorted by names

Input

Marks	80	75	70	90	85	75
Name	Alice	Bill	Don	Geoff	Leo	Maria

Sort on Marks using a stable algorithm

Output

Marks	70	75	75	80	85	90
Name	Don	Bill	Maria	Alice	Leo	Geoff

Note: Output is sorted on marks then names.
Unstable sorting cannot guarantee this (e.g., Maria may appear before Bill)

Outline

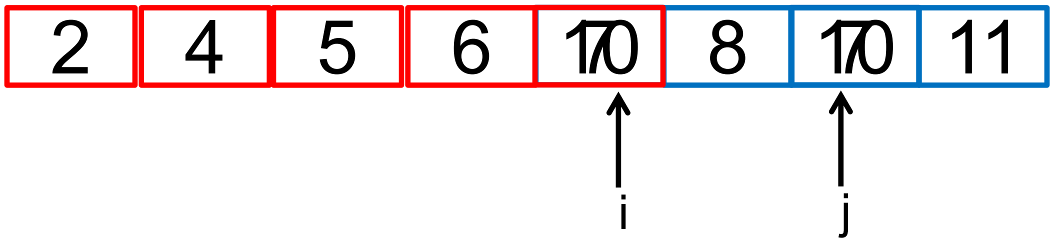
Complexity Analysis

A. Introduction/Recap (covered last week)
B. Finding minimum
C. Binary Search
D. Comparison-based Sorting Algorithms
   I. Selection Sort
   II. Insertion Sort
   III. Lower bound for comparison-based sorting
E. Non-comparison Sorting Algorithms
   I. Counting Sort
   II. Radix Sort
F. Recursive Algorithms

Selection Sort (Correctness)

Sort an array (denoted as arr) in ascending order

```
for(i = 1; i < N; i++) {
// LI: arr[1 … i-1] is sorted AND arr[1 … i-1] <= arr[i … N]
      j = index of minimum element in arr[i … N]
      swap (arr[i],arr[j])
}
// i=N when the loop terminates
// LI: arr[1 … N-1] is sorted AND arr[1 … N-1] <= arr[N]
```



Selection Sort

Sort an array (denoted as arr) in ascending order

```
for(i = 1; i < N; i++) {
// LI: arr[1 … i-1] is sorted AND arr[1 … i-1] <= arr[i+1 … N]
        j = index of minimum element in arr[i … N]
        swap (arr[i],arr[j])
}
// LI: arr[1 … N-1] is sorted AND arr[1 … N-1] <= arr[N]
```

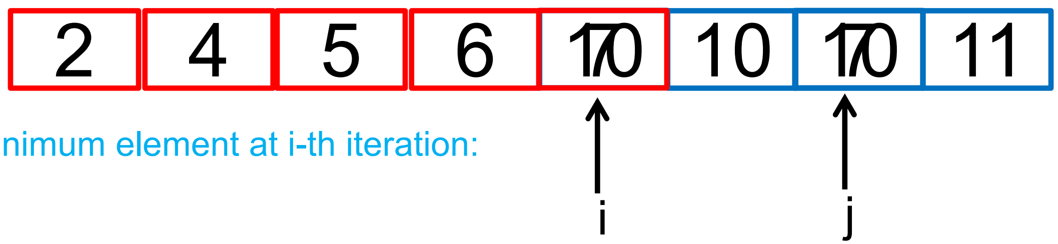
Could we use a weaker loop invariant, e.g.,

// LI: arr[1 … i-1] is sorted (That is Insertion Sort)

Selection Sort Analysis

```
for(i = 1; i < N; i++) {
        j = index of minimum element in arr[i … N]
        swap (arr[i],arr[j])
}
```

Time Complexity?



- Worst-case
  - Complexity of finding minimum element at i-th iteration:
  - Total complexity:
- Best-case
- Average

Space Complexity?

Auxiliary Space Complexity?

Selection Sort is an in-place algorithm!

Is selection sort stable?

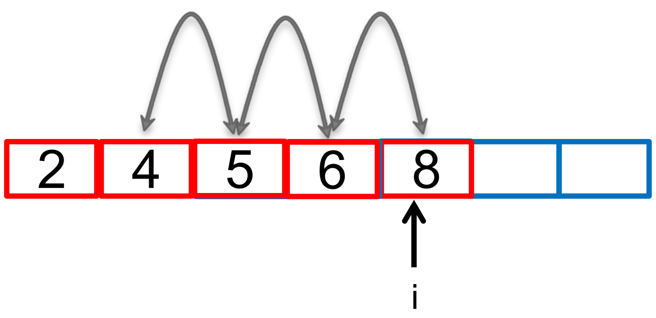
Outline

Complexity Analysis

A. Introduction/Recap (covered last week)
B. Finding minimum
C. Binary Search
D. Comparison-based Sorting Algorithms
   I. Selection Sort
   II. Insertion Sort
   III. Lower bound for comparison-based sorting
E. Non-comparison Sorting Algorithms
   I. Counting Sort
   II. Radix Sort
F. Recursive Algorithms

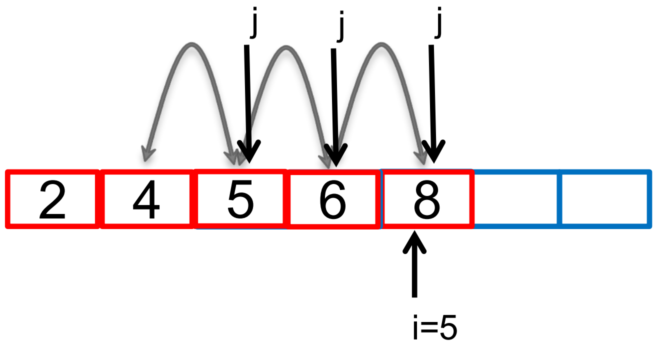
Insertion Sort (Correctness)

```
for(i = 1; i <= N; i++) {
        #LI: arr[1…i-1] is sorted
        #insert arr[i] in arr[1…i] in sorted order
        #idea: continue swapping the item with the item on left as
 long as the item on the left is bigger
}
#i=N+1 when the loop terminates
#LI: arr[1…N] is sorted
```



Insertion Sort

```
for(i = 1; i <= N; i++) {
   j = i
   while arr[j-1] > arr[j] and j>1:
      #swap elements at arr[j] and arr[j-1]
      swapElements(j-1,j)
      j = j-1
}
```

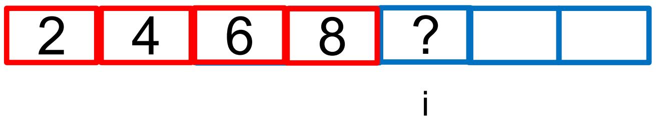


Insertion Sort Analysis

```
for(i = 1; i <= N; i++) {
    j = i
    while arr[j-1] > arr[j] and j>1:
        #swap elements at arr[j] and arr[j-1]
        swapElements(j-1,j)
        j = j-1
}
```

Time Complexity?

- Worst-case
    - Complexity of while loop at i-th iteration ;
    - Total complexity:



- Best-case
    - Complexity of while loop at i-th iteration:
    - Total complexity:
- Average
    - On average, arr[i] will be bigger than 50% of the elements on its left
    - Total cost on average is half the cost of worst-case: still $O(N^2)$

Space Complexity?

Auxiliary Space Complexity?

Is Insertion Sort stable?

- Yes, because swapping stops when the element on left is smaller **or equal**

Outline

Complexity Analysis

A. Introduction/Recap (covered last week)
B. Finding minimum
C. Binary Search
D. Comparison-based Sorting Algorithms
    I. Selection Sort
    II. Insertion Sort
    III. Lower bound for comparison-based sorting
E. Non-comparison Sorting Algorithms
    I. Counting Sort
    II. Radix Sort
F. Recursive Algorithms

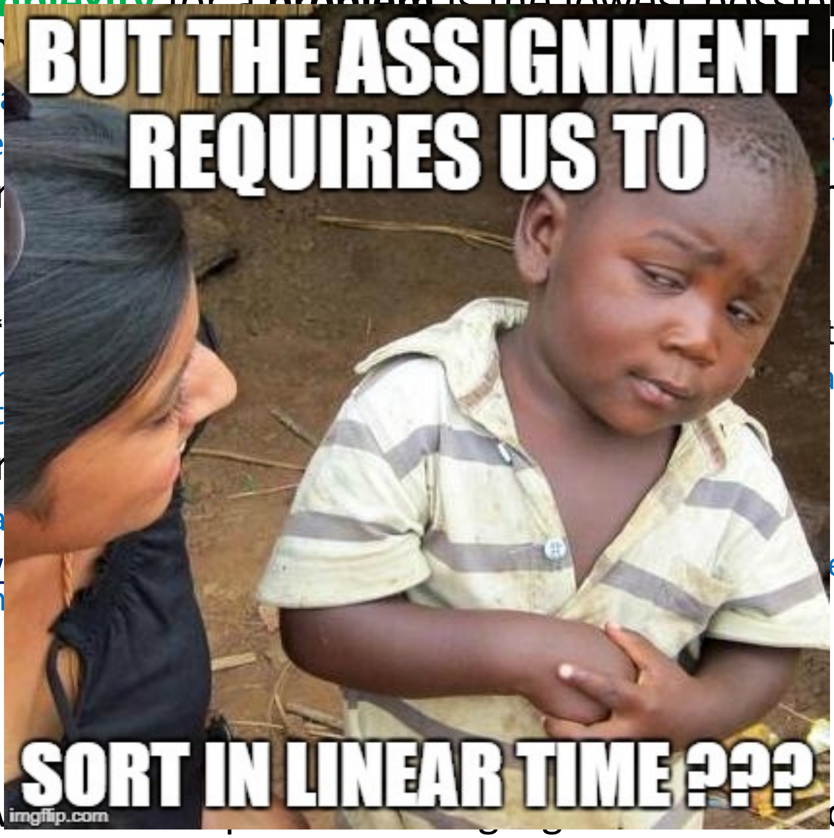
Summary of comparison-based sorting algorithms

	Best	Worst	Average	Stable?	In-place?
Selection Sort	$O(N^2)$	$O(N^2)$	$O(N^2)$	No	Yes
Insertion Sort	$O(N)$	$O(N^2)$	$O(N^2)$	Yes	Yes
Heap Sort	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	No	Yes
Merge Sort	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	Yes	No
Quick Sort	$O(N \log N)$	$O(N^2)$ – can be made $O(N \log N)$	$O(N \log N)$	Depends	No

Is it possible to develop a sorting algorithm with worst-case time complexity better than $O(N \log N)$?

Lower Bound Complexity

- **Lower bound com**plexity for a **problem** is the lowest possible complexity **any** algorithm (known ... blem
  - It is important beca ... le?
  - Unless stated othe ... the algorithm.
- What is the lower ... element in an array of N elements
  - **Ans:** Ω(N)
    - Big-Ω means " ... " (upper bound)
  - Since the finding m ... se complexity, it is best possible algorithm (i.e., opt
- What is the lower
  - For comparison-ba
  - Read https://www ... ee why the lower bound is Ω(N log N) -- the proof is n



- Next, we discuss t ... o sorting in less than O(N log N)

Outline

Complexity Analysis

A. Introduction/Recap (covered last week)
B. Finding minimum
C. Binary Search
D. Comparison-based Sorting Algorithms
   I. Selection Sort
   II. Insertion Sort
   III. Lower bound for comparison-based sorting
E. Non-comparison Sorting Algorithms
   I. Counting Sort
   II. Radix Sort
F. Recursive Algorithms

Game

- Enter an integer between 1 to 200 (inclusive) on MARS
- The person entering the smallest unique integer wins
  - i.e., if two people enter the same integer, both are disqualified
  - So the winner is the person who entered the smallest positive unique integer

- Algorithm to determine the winner?
- What is the lower bound complexity for this problem?

Counting Sort

Assume we have to sort the input containing positive integers.

- Find the maximum integer in the array and call it **max**.
- Create an empty array **"count"** of size **max** each value initialized to 0

// count # of occurrences for each value in input array

- For each value in **"Input"**:
  - **count[value]+= 1**
- Output = empty
- For x=1 to len(**count**):
  - NumOfOccurrences = count[x]
  - Append x to Output NumOfOccurrences times

count

1	1
2	0
3	3
4	0
5	1
6	0
7	2
8	1

Input

3	**1**	**3**	**7**	**5**	**3**	**7**	**8**

Output

1	3	3	3	5	7	7	8

Analysis of Counting Sort

- Find the maximum integer in the array and call it **max**.
- Create an empty array **"count"** of size **max** each value initialized to 0

// count # of occurrences for each value

- For each value in **"Input":**
  - **Count[value]+= 1**
- Output = empty
- For x=1 to len(**count**):
  - NumOfOccurrences = Count[x]
  - Append x to Output NumOfOccurrences times

Let N be the size of Input array and D be the domain size (e.g., **max)**, i.e., D is the size of **count** array.

Time Complexity:

- O(N+D) – worst-case, best-case, average-case all are the same

Space Complexity:

- O(N+D)

Is counting sort stable?

No, because it counts the values but does not distinguishes between them. However, it can be made stable (shown later).

Counting Sort for alphabets

- Counting sort can also be applied to sort an array of alphabets
- e.g., Array = [B,C,D,B,C,A]
- Count # occurrences for each letter
  - A refers to index 1 in count array
  - B refers to index 2 in count array
  - and so on
- The mapping can be done using ASCII
  - e.g., in python
  - ord("A") gives 65
  - ord("B") gives 66
  - and so on
- For any letter char, we can get its index
  - ord(char) – 64
- After counting, print # occurrences as in counting sort
- Conversion from integer to character can be done easily
  - chr(65) → a
  - chr(66) →b
  - chr(index+64)

A	1	1
B	2	2
C	3	2
D	4	1
E	5	0
…	…	…
Y	25	0
Z	26	0

Analysis of Counting Sort for English Alphabets

- Create an empty array **“count”** of size 26 each value initialized to 0

// count # of occurrences for each alphabet

- For each char in **“Input”:**
  - **count [ord(char) - 64]+= 1**
- Output = empty
- For x=1 to len(**count**):
  - NumOfOccurrences = count[x]
  - Append(chr(x+64) to Output NumOfOccurrences times

The domain size D in the case for English alphabets is 26 which can be considered a constant.

Time Complexity:

- O(N+D) → O(N)

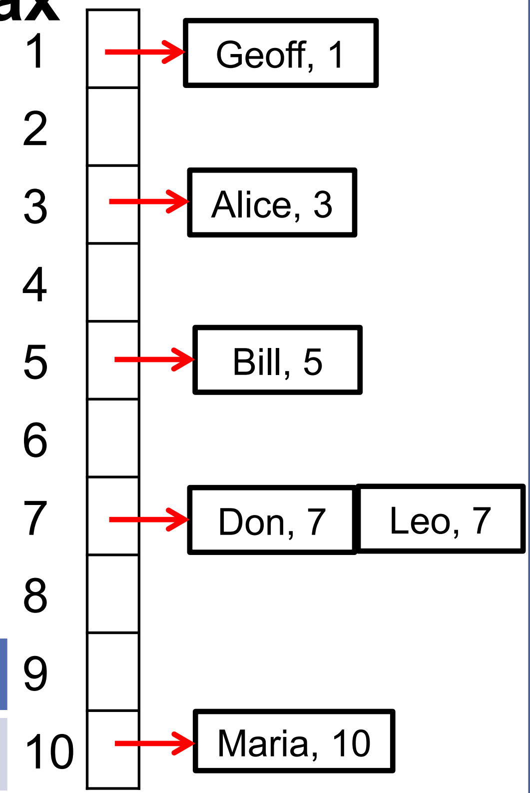
Space Complexity:

- O(N+D) → O(N)

Stable Counting Sort

- Find maximum mark in the array called **max**
- Create an empty array **“count”** of size **max**
- For each item in **“Input”:**
  - **Append item to Count[item.marks]**
- Output = empty
- For x=1 to len(**count**):
  - Append elements in count[x] to Output

count



Input

Marks	**3**	**5**	**7**	**1**	**7**	**10**
Name	Alice	Bill	Don	Geoff	Leo	Maria

Output

Geoff, 1	Alice, 3	Bill, 5	Don, 7	Leo, 7	Maria, 10

Analysis of Stable Counting Sort

- Find maximum mark in the array called **max**
- Create an empty array **“count”** of size **max**
- For each item in **“Input”:**
  - **Append item to Count[item.marks]**
- Output = empty
- For x=1 to len(**count**):
  - Append elements in count[x] to Output

Let D be the domain size and N be the number of values in Input.

Time Complexity:

- O(N+D)
- Note: For our example, since domain is marks (0 to 100), D can be considered a constant!

Space Complexity:

- O(N+D)

Stable sorting can also be used for sorting English alphabets using the same idea!

Outline

Complexity Analysis

A. Introduction/Recap (covered last week)
B. Finding minimum
C. Binary Search
D. Comparison-based Sorting Algorithms
   I. Selection Sort
   II. Insertion Sort
   III. Lower bound for comparison-based sorting
E. Non-comparison Sorting Algorithms
   I. Counting Sort
   II. Radix Sort
F. Recursive Algorithms

Radix Sort

Sort an array of words in alphabetical order assuming each word consists of M letters each

- Use stable sort to sort them on the M-th column
- Use stable sort to sort them on the (M-1)-th column
- …
- Use stable sort to sort them on 1st column

	Sort on 4^{th} column	Sort on 3^{rd} column	Sort on 2^{nd} column	Sort on 1^{st} column
GOAL	HER**B**	GO**A**L	T**A**LL	**A**IMS
TALL	BIR**D**	RI**D**E	H**E**RB	**A**NTS
AIMS	LIK**E**	LI**K**E	R**I**DE	**B**IKE
ANTS	BIK**E**	BI**K**E	L**I**KE	**B**IRD
BIRD	RID**E**	MI**K**E	B**I**KE	**F**ISH
FISH	MIK**E**	TA**L**L	M**I**KE	**G**OAL
LIKE	KIN**G**	KI**N**G	K**I**NG	**H**ERV
BIKE	FIS**H**	AI**M**S	A**I**MS	**K**ING
KING	GOA**L**	HE**R**B	B**I**RD	**L**IKE
RIDE	TAL**L**	BI**R**D	F**I**SH	**M**IKE
HERB	AIM**S**	FI**S**H	A**N**TS	**R**IDE
MIKE	ANT**S**	AN**T**S	G**O**AL	**T**ALL

What happens if we don’t use stable sorting?

Sort an array of words in alphabetical order assuming each word consists of M letters each

- Use unstable sort to sort them on last column
- Use unstable sort to sort them on 2nd last column
- …
- Use unstable sort to sort them on first column

B A T
B I D
T E A

Sort on 1st column

T E A
B I D
B A T

Sort on 2nd column

B A T
T E A
B I D

Sort on 3rd column

B A T
B I D
T E A

OR

B I D
B A T
T E A

What will be the order of elements?

A. BAT, BID, TEA
B. BID, BAT, TEA
C. TEA, BID, BAT
D. All of the above
E. Either A or B
F. None of the above

What happens if we process columns from left to right?

Sort an array of words in alphabetical order assuming each word consists of M letters each

- Use stable sort to sort them on first (left most) column
- Use stable sort to sort them on the 2nd column
- …
- Use stable sort to sort them on last (right most) column

B A T		B A T		B A T		T E A
B A D	→	B A D	→	B A D	→	B A D
T E A	Sort on 3rd column	T E A	Sort on 2nd column	T E A	Sort on 1st column	B A T

What will be the order of elements?

A. BAT, BAD, TEA
B. BAD, BAT, TEA
C. TEA, BAD, BAT
D. TEA, BAT, BAD
E. None of the above

Analysis of Radix Sort

Sort an array of words in alphabetical order assuming each word consists of M letters each

- Use stable sort to sort them on the M-th Column column
- Use stable sort to sort them on the (M-1)-th column
- …
- Use the stable sort to sort them on 1st column

Assume that N is the number of words and each word has M characters each.

Assuming we are using stable counting sort which has time and space complexity O(N+D).

Time Complexity of Radix Sort:

- O((N+D)*M) → O(MN) because D is constant for English alphabets

Space Complexity of Radix Sort:

- O((N+D)*M) → O(MN)

What is the cost of Merge Sort assuming comparing two strings of length M takes O(M)?

- O(MN log N)
- Radix sort can also be used to sort integers using the similar idea!

Outline

- sis
  - (covered last week)
  - Sorting A
  - compariso
  - rting Algo
    - Counting Sort
    - Radix Sort
- Recursive Algorithms

If this is you, great!
But don't live dangerously!!!

If this is you, stay calm (but stop laughing) and talk to me!

Outline

Complexity Analysis

A. Introduction/Recap (covered last week)
B. Finding minimum
C. Binary Search
D. Comparison-based Sorting Algorithms
   I. Selection Sort
   II. Insertion Sort
   III. Lower bound for comparison-based sorting
E. Non-comparison Sorting Algorithms
   I. Counting Sort
   II. Radix Sort
F. Recursive Algorithms

Complexity of recursive algorithms

```
// Compute Nth power of x
power(x,N)
{
    if (N==0)
        return 1
    if (N==1)
        return x
    else
        return x * power(x, N–1)
}
```

Time Complexity

Cost when N = 1: T(1) = b (b&c are constant)

Cost for general case: T(N) = T(N-1) + c (A)

It solution (as seen last week) is:

T(N) = b + (N-1)*c = c*N + b - c

Hence, the complexity is O(N)

Complexity of recursive algorithms

```
// Recursive version
power(x,N)
{
    if (N==0)
        return 1
    if (N==1)
        return x
    else
        return x * power(x, N–1)
}
// Iterative version
result = 1
for i=1; i<= N; i++{
        result = result * x
}
return result
```

Space Complexity?

Total space usage = Local space used by the function * maximum depth of recursion

= c * maximum depth of recursion = c*N

= O(N)

Note: We will not discuss tail-recursion in this unit because it is language specific, e.g., Python doesn't utilize tail-recursion

Auxiliary Space Complexity?

- Recursive power() is not an in-place algorithm

Note that an iterative version of power uses O(1) space and is an in-place algorithm

Complexity of recursive algorithms

```
// Compute Nth  power x
power2(x,N)
{
  if (N==0)
    return 1
  if (N==1)
    return x
  if (N is even)
   return power2( x * x, N/2)
 else
   return power2( x * x, N/2 ) * x
}
```

Time Complexity

Cost when N = 1: T(1) = b (b&c are constant)

Cost for general case: T(N) = T(N/2) + c   (A)

It solution (as seen last week) is:

$T(N) = b + c^* \log_2 N$

Hence, the complexity is O(log N)

Complexity of recursive algorithms

```
// Compute Nth  power x
power2(x,N)
{
   if (N==0)
     return 1
   if (N==1)
     return x
   if (N is even)
    return power2( x * x, N/2)
 else
   return power2( x * x, N/2 ) * x
}
```

Space Complexity

Space usage = Local space used by the function * maximum depth of recursion

= c * maximum depth of recursion

= c*log N

= O(log N)

Is this algorithm in-place?

Output-Sensitive Time Complexity

Problem: Given a sorted array of **unique** numbers and two values x and y, find all numbers greater than x and smaller than y.

Algorithm 1:

- For each number n in array:
  - if n > x and n < y:
    - print(n)

Time complexity:

$O(N)$

Output-Sensitive Time Complexity

Algorithm 2:

- Binary search to find the smallest number greater than x
- Continue linear search from x until next number is >= y

Time complexity?

- O(N) in the worst-case because in the worst-case all numbers may be within the range x to y

Output-sensitive complexity is the time-complexity that also depends on the size of output.

Let W be the number of values in the range (i.e., in output).

Output-sensitive complexity of Algorithm 2? O(W + log N) – note W may be N in the worst-case.

Output-sensitive complexity of Algorithm 1? O(N)

Output-sensitive complexity is only relevant when output-size may vary, e.g., it is not relevant for sorting, finding minimum value etc.

Assume x=28 and y=38

5	**10**	**15**	**20**	**25**	**30**	**35**	**40**	**45**	**50**
1	2	3	4	5	6	7	8	9	10

Concluding Remarks

Summary

- Best/worst/average space/time complexities
- Stable sorting, in-place algorithms
- Non-comparison sorting
- Complexity analysis of recursive algorithms

Coming Up Next

- Quick Sort and its best/worst/average case analysis
- How to improve worst-case complexity of Quick Sort to O(N log N)

Things to do before next lecture

- Make sure you understand this lecture completely (especially the complexity analysis)
- Solve all the recurrence relations yourself (including the ones we solved in lectures)
- Using induction, prove your solutions for the previous task are correct