Faculty of Information Technology,
Monash University

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
  - 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 3 April 2020 at 23:59:00)
  - Start early!
  - Seek help if struggling

Recommended reading

- Basic mathematics used for algorithm analysis: http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Math/
- 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. Auxiliary Space Complexity
C. Recursive Algorithms
D. Output-Sensitive Time complexity
E. Comparison-based Sorting Algorithms
    I. Selection Sort
    II. Lower bound for comparison-based sorting
F. Non-comparison Sorting Algorithms
    I. Counting Sort
    II. Radix Sort

Auxiliary Space Complexity

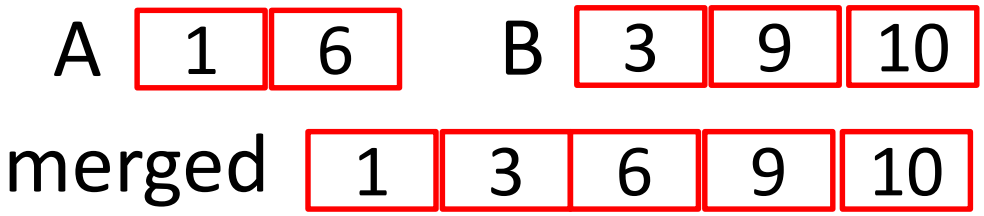
- **Space complexity** is the total amount of space taken by an algorithm as a function of input size
- **Auxiliary space complexity** is the amount of space taken by an algorithm excluding the space taken by the input
  - Many textbooks and online resources do not distinguish between the above two terms 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 between them.

Auxiliary Space Complexity

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?

Quiz time!
https://flux.qa - 92A2WY



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
  - It needs to create the output list which is size N
  - 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.

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

Time/Space Complexity: Binary Search

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

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
  - O(1), 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. Auxiliary Space Complexity
C. Recursive Algorithms
D. Output-Sensitive Time complexity
E. Comparison-based Sorting Algorithms
    I. Selection Sort
    II. Lower bound for comparison-based sorting
F. Non-comparison Sorting Algorithms
    I. Counting Sort
    II. Radix Sort

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)

Solution (as seen last week) is:

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

Hence, the complexity is O(N)

Quiz time!
https://flux.qa - 92A2WY

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

Outline

Complexity Analysis

A. Introduction/Recap (covered last week)
B. Auxiliary Space Complexity
C. Recursive Algorithms
D. Output-Sensitive Time complexity
E. Comparison-based Sorting Algorithms
    I. Selection Sort
    II. Lower bound for comparison-based sorting
F. Non-comparison Sorting Algorithms
    I. Counting Sort
    II. Radix Sort

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

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 2:

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

Time complexity?

Quiz time!
https://flux.qa - 92A2WY

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

Output-Sensitive Time Complexity

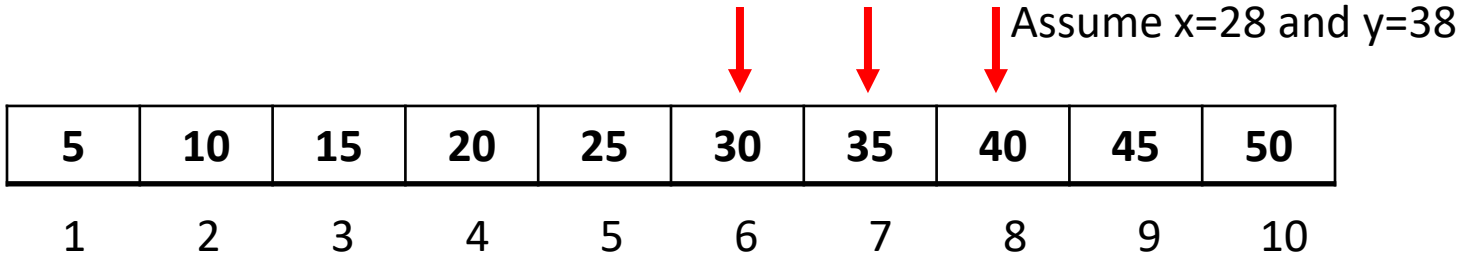
Output-sensitive complexity is the time-complexity that also depends on the size of output.

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.



Outline

Complexity Analysis

A. Introduction/Recap (covered last week)
B. Auxiliary Space Complexity
C. Recursive Algorithms
D. Output-Sensitive Time complexity
E. Comparison-based Sorting Algorithms
   I. Selection Sort
   II. Lower bound for comparison-based sorting
F. Non-comparison Sorting Algorithms
   I. Counting Sort
   II. Radix Sort

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: Generally we assume that numbers are machine-word sized (i.e. fit in a register) and so comparison is O(1). In theory, for very large numbers, comparison would be O(digits)

Comparison Cost

- Typically assume comparison cost is O(1) (small values)
- Sometimes comparison cost is critical
- Genome sequences may have millions of characters – expensive comparison

- 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).
- This ignores comparison cost (which is mostly fine)

In summary:

In this unit we will assume generally assume that string comparison is O(L) and numerical comparison is O(1), unless otherwise specified

Stable sorting algorithms

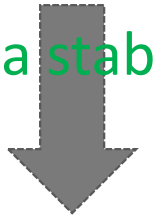
A sorting algorithm is called stable if it maintains the relative ordering of elements that have equal keys.

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

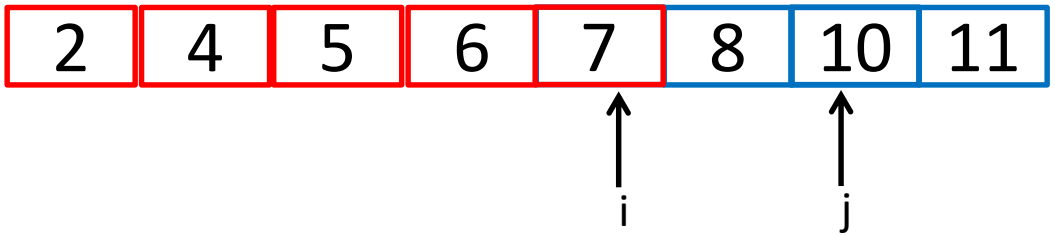
Complexity Analysis

A. Introduction/Recap (covered last week)
B. Auxiliary Space Complexity
C. Recursive Algorithms
D. Output-Sensitive Time complexity
E. Comparison-based Sorting Algorithms
    I. Selection Sort
    II. Lower bound for comparison-based sorting
F. Non-comparison Sorting Algorithms
    I. Counting Sort
    II. Radix Sort

Selection Sort (Correctness)

Sort an array (denoted as arr) in ascending order

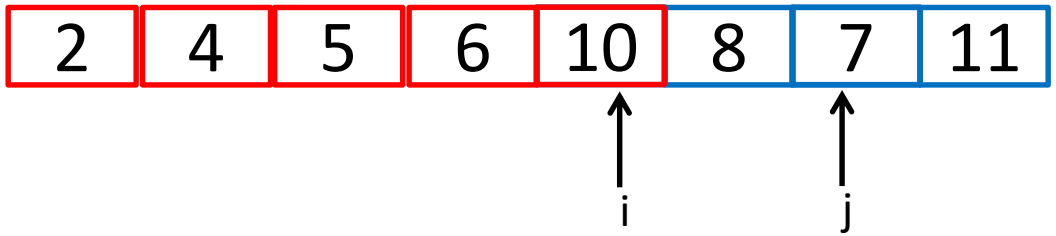
```
for(i = 1; i < N; i++) {
        j = index of minimum element in arr[i … N]
        swap (arr[i],arr[j])
}
```



Selection Sort (Correctness)

Sort an array (denoted as arr) in ascending order

```
// LI: arr[1 … i-1] is sorted AND arr[1 … i-1] <= arr[i … N]
for(i = 1; i < N; i++) {
        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]
```



https://flux.qa
92A2WY

Selection Sort

Sort an array (denoted as arr) in ascending order

```
// LI: arr[1 … i-1] is sorted AND arr[1 … i-1] <= arr[i … N]
for(i = 1; i < N; i++) {
        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]
```

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

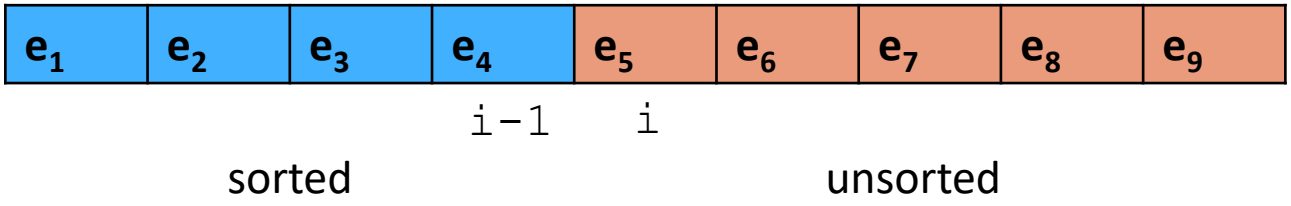
```
// LI: arr[1 … i-1] is sorted (That is Insertion Sort)
```

Selection Sort

```
Could we use a weaker loop invariant, e.g.,
// LI: arr[1 … i-1] is sorted (That is Insertion Sort)
```

While doing the proof, we need to show that if the LI is true at iteration k, it is true at iteration k+1

```
j = index of minimum element in arr[i … N]
      swap (arr[i],arr[j])
```

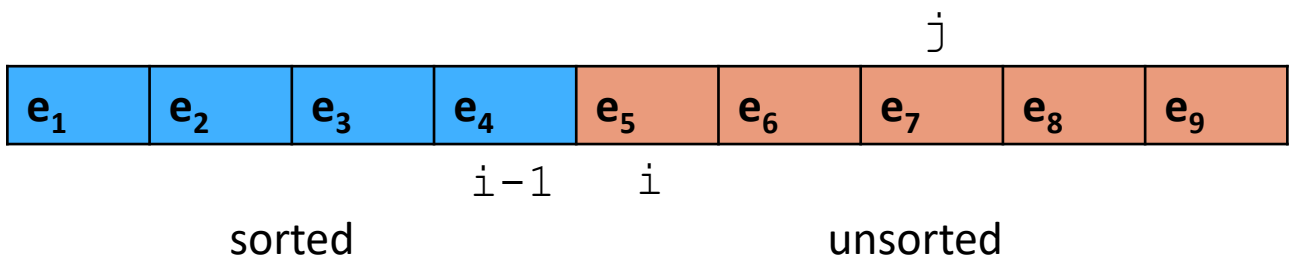


Selection Sort

```
Could we use a weaker loop invariant, e.g.,
// LI: arr[1 … i-1] is sorted (That is Insertion Sort)
```

While doing the proof, we need to show that if the LI is true at iteration k, it is true at iteration k+1

```
j = index of minimum element in arr[i … N]
      swap (arr[i],arr[j])
```

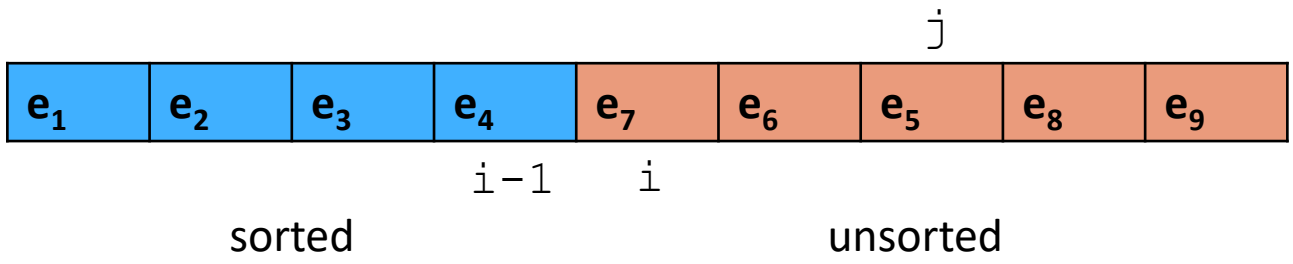


Selection Sort

```
Could we use a weaker loop invariant, e.g.,
// LI: arr[1 … i-1] is sorted (That is Insertion Sort)
```

While doing the proof, we need to show that if the LI is true at iteration k, it is true at iteration k+1

```
j = index of minimum element in arr[i … N]
      swap (arr[i],arr[j])
```

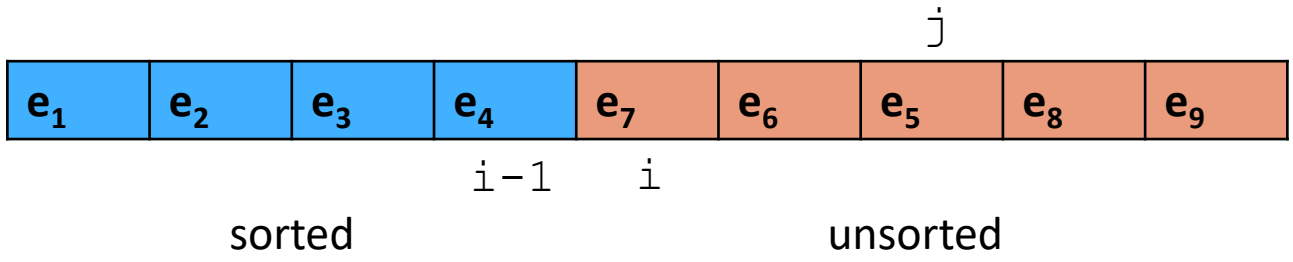


Selection Sort

```
Could we use a weaker loop invariant, e.g.,
// LI: arr[1 … i-1] is sorted (That is Insertion Sort)
```

While doing the proof, we need to show that if the LI is true at iteration k, it is true at iteration k+1

```
j = index of minimum element in arr[i … N]
      swap (arr[i],arr[j])
```



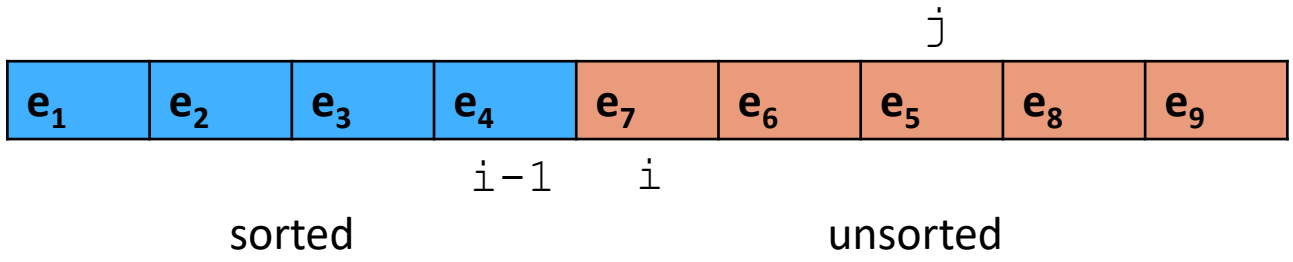
How do we know that $e_7 >= e_4$?

Selection Sort

```
Could we use a weaker loop invariant, e.g.,
// LI: arr[1 … i-1] is sorted (That is Insertion Sort)
```

While doing the proof, we need to show that if the LI is true at iteration k, it is true at iteration k+1

```
j = index of minimum element in arr[i … N]
      swap (arr[i],arr[j])
```



How do we know that $e_7 >= e_4$?

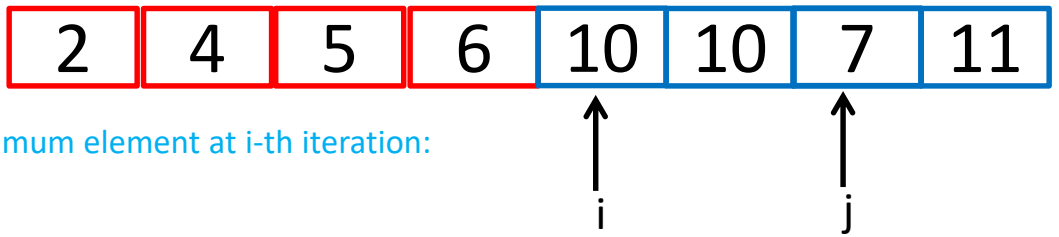
We don't ☹

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



Space Complexity?

Auxiliary Space Complexity?

Selection Sort is an in-place algorithm!

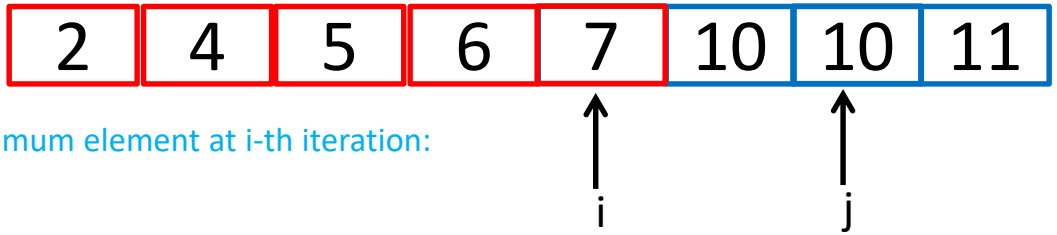
Is selection sort stable?

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



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. Recursive Algorithms
D. Output-Sensitive Time complexity
E. Comparison-based Sorting Algorithms
   I. Selection Sort
   II. Lower bound for comparison-based sorting
F. Non-comparison Sorting Algorithms
   I. Counting Sort
   II. Radix Sort

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 complexity** for a **problem** is the lowest possible complexity **any** algorithm (known or unknown) can achieve to solve the problem
  - It is important because it gives a theoretical bound on what is best possible
  - Unless stated otherwise, lower bound is for the worst-case complexity of the algorithm

- What is the lower bound complexity of finding the minimum element in an array of N elements
  - **Ans:** Ω(N): We (at least) need to look at every element
    - Big-Ω means "at least" (lower bound) whereas big-O means "at most" (upper bound)
  - Since the finding minimum algorithm we saw this week has O(N) worst-case complexity, it is best possible algorithm (i.e., optimal) in terms of time complexity.

Lower Bound Complexity

- **Lower bound complexity** for a **problem** is the lowest possible complexity **any** algorithm (known or unknown) can achieve to solve the problem
  - It is important because it gives a theoretical bound on what is best possible
  - Unless stated otherwise, lower bound is for the worst-case complexity of the algorithm

- What is the lower bound complexity for sorting?
  - For comparison-based algorithm, lower bound complexity is $\Omega(N \log N)$.
  - Read section 3.4 of the notes to see why the lower bound is $\Omega(N \log N)$ -- the proof is not examinable

- Next, we discuss two non-comparison sorting algorithms that do sorting in less than O(N log N)

Break Time

Outline

Complexity Analysis

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

Intuition

- Given an array containing a permutation of the numbers 1…N, sort them.

```
for(i=1, i <= N, i++)
    print(i)
```

- Given an array containing a subset of the numbers 1…N, sort them.

Counting Sort

- Sorting positive integers
- Determine the maximum value in the input
- Create an array “count” of that size

count

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

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Counting Sort

- Iterate through the input and count the number of times each value occurs
- Do this by incrementing the corresponding position in "count"
- If we see a 3, increment count[3]

count

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

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Counting Sort

- Iterate through the input and count the number of times each value occurs
- Do this by incrementing the corresponding position in "count"
- If we see a 3, increment count[3]

count

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

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Counting Sort

- Iterate through the input and count the number of times each value occurs
- Do this by incrementing the corresponding position in "count"
- If we see a 3, increment count[3]

count

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

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Counting Sort

- Iterate through the input and count the number of times each value occurs
- Do this by incrementing the corresponding position in "count"
- If we see a 3, increment count[3]

count

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

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Counting Sort

- Iterate through the input and count the number of times each value occurs
- Do this by incrementing the corresponding position in "count"
- If we see a 3, increment count[3]

count

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

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Counting Sort

- Iterate through the input and count the number of times each value occurs
- Do this by incrementing the corresponding position in "count"
- If we see a 3, increment count[3]

count

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

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Counting Sort

- Iterate through the input and count the number of times each value occurs
- Do this by incrementing the corresponding position in "count"
- If we see a 3, increment count[3]

count

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

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Counting Sort

- Iterate through the input and count the number of times each value occurs
- Do this by incrementing the corresponding position in "count"
- If we see a 3, increment count[3]

count

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

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Counting Sort

- Iterate through the input and count the number of times each value occurs
- Do this by incrementing the corresponding position in "count"
- If we see a 3, increment count[3]

count

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

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Counting Sort

- Create “output”
- For each position in count
- Append count[i] copies of the value i to output
- So if count[7]=2, then append 2 copies of 7

count

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

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Output

Counting Sort

- Create “output”
- For each position in count
- Append count[i] copies of the value i to output
- So if count[7]=2, then append 2 copies of 7

count

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

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Output

1							

Counting Sort

- Create “output”
- For each position in count
- Append count[i] copies of the value i to output
- So if count[7]=2, then append 2 copies of 7

count

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

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Output

1							

Counting Sort

- Create “output”
- For each position in count
- Append count[i] copies of the value i to output
- So if count[7]=2, then append 2 copies of 7

count

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

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Output

1	3	3	3				

Counting Sort

- Create “output”
- For each position in count
- Append count[i] copies of the value i to output
- So if count[7]=2, then append 2 copies of 7

count

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

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Output

1	3	3	3				

Counting Sort

- Create “output”
- For each position in count
- Append count[i] copies of the value i to output
- So if count[7]=2, then append 2 copies of 7

count

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

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Output

1	3	3	3	5			

Counting Sort

- Create “output”
- For each position in count
- Append count[i] copies of the value i to output
- So if count[7]=2, then append 2 copies of 7

count

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

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Output

1	3	3	3	5			

Counting Sort

- Create “output”
- For each position in count
- Append count[i] copies of the value i to output
- So if count[7]=2, then append 2 copies of 7

count

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

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Output

1	3	3	3	5	7	7	

Counting Sort

- Create “output”
- For each position in count
- Append count[i] copies of the value i to output
- So if count[7]=2, then append 2 copies of 7

count

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

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Output

1	3	3	3	5	7	7	8

Analysis of Counting Sort

- Create "count" array of size max, where max is the maximum value in the input

- 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 U be the domain size (e.g., **max)**, i.e., U is the size of **count** array.

Time Complexity:

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

Space Complexity:

- O(N+U)

Is counting sort stable?

No, because it counts the values but does not distinguishes between them. In fact, it loses any data associated with the values, so it is much worse than unstable. Lets fix this!

Stable Counting Sort (Method 1)

- To fix the two problems of stability and losing data, we need a smart idea

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Output

(1,p)	(3,a)	(3,c)	(3,b)	(5,g)	(7,f)	(7,d)	(8,w)

- What information would you need to correctly place the data with key “5” in the output?

Stable Counting Sort (Method 1)

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Construct count:

- For each key in input,
- count[key] += 1

count

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

Output

1	2	3	4	5	6	7	8

Stable Counting Sort (Method 1)

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Construct count:

- For each key in input,
- count[key] += 1

Construct position:

- Initialise first position as a 1

count

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

position

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

Output

1	2	3	4	5	6	7	8

Stable Counting Sort (Method 1)

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Construct count:

- For each key in input,
- count[key] += 1

Construct position:

- Initialise first position as a 1
- position[i] = position[i-1] + count[i-1]

count

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

position

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

Output

1	2	3	4	5	6	7	8

Stable Counting Sort (Method 1)

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Construct count:

- For each key in input,
- count[key] += 1

Construct position:

- Initialise first position as a 1
- position[i] = position[i-1] + count[i-1]

count

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

position

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

Output

1	2	3	4	5	6	7	8

Stable Counting Sort (Method 1)

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Construct count:

- For each key in input,
- count[key] += 1

Construct position:

- Initialise first position as a 1
- position[i] = position[i-1] + count[i-1]

count

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

position

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

Output

1	2	3	4	5	6	7	8

Stable Counting Sort (Method 1)

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Construct count:

- For each key in input,
- count[key] += 1

Construct position:

- Initialise first position as a 1
- position[i] = position[i-1] + count[i-1]

count

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

position

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

Output

1	2	3	4	5	6	7	8

Stable Counting Sort (Method 1)

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Construct count:

- For each key in input,
- count[key] += 1

Construct position:

- Initialise first position as a 1
- position[i] = position[i-1] + count[i-1]

count

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

position

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

Output

1	2	3	4	5	6	7	8

Stable Counting Sort (Method 1)

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Construct count:

- For each key in input,
- count[key] += 1

Construct position:

- Initialise first position as a 1
- position[i] = position[i-1] + count[i-1]

count

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

position

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

Output

1	2	3	4	5	6	7	8

Stable Counting Sort (Method 1)

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Construct count:

- For each key in input,
- count[key] += 1

Construct position:

- Initialise first position as a 1
- position[i] = position[i-1] + count[i-1]

count

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

position

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

Output

1	2	3	4	5	6	7	8

Stable Counting Sort (Method 1)

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Construct count:

- For each key in input,
- count[key] += 1

Construct position:

- Initialise first position as a 1
- position[i] = position[i-1] + count[i-1]

Construct output

- Go through input, looking at each (key, val)
- Set output[position[key]] to the (key, val) pair from input
- Increment position[key]

count

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

position

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

Output

1	2	3	4	5	6	7	8

Stable Counting Sort (Method 1)

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Construct count:

- For each key in input,
- count[key] += 1

Construct position:

- Initialise first position as a 1
- position[i] = position[i-1] + count[i-1]

Construct output

- Go through input, looking at each (key, val)
- Set output[position[key]] to the (key, val) pair from input
- Increment position[key]

count

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

position

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

Output

1	2	3	4	5	6	7	8

Stable Counting Sort (Method 1)

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Construct count:

- For each key in input,
- count[key] += 1

Construct position:

- Initialise first position as a 1
- position[i] = position[i-1] + count[i-1]

Construct output

- Go through input, looking at each (key, val)
- Set output[position[key]] to the (key, val) pair from input
- Increment position[key]

count

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

position

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

Output

	(3,a)						
1	2	3	4	5	6	7	8

Stable Counting Sort (Method 1)

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Construct count:

- For each key in input,
- count[key] += 1

Construct position:

- Initialise first position as a 1
- position[i] = position[i-1] + count[i-1]

Construct output

- Go through input, looking at each (key, val)
- Set output[position[key]] to the (key, val) pair from input
- Increment position[key]

count

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

position

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

Output

(1,p)	(3,a)						
1	2	3	4	5	6	7	8

Stable Counting Sort (Method 1)

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Construct count:

- For each key in input,
- count[key] += 1

Construct position:

- Initialise first position as a 1
- position[i] = position[i-1] + count[i-1]

Construct output

- Go through input, looking at each (key, val)
- Set output[position[key]] to the (key, val) pair from input
- Increment position[key]

count

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

position

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

Output

(1,p)	(3,a)	(3,c)					
1	2	3	4	5	6	7	8

Stable Counting Sort (Method 1)

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Construct count:

- For each key in input,
- count[key] += 1

Construct position:

- Initialise first position as a 1
- position[i] = position[i-1] + count[i-1]

Construct output

- Go through input, looking at each (key, val)
- Set output[position[key]] to the (key, val) pair from input
- Increment position[key]

count

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

position

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

Output

(1,p)	(3,a)	(3,c)			(7,f)		
1	2	3	4	5	6	7	8

Stable Counting Sort (Method 1)

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Construct count:

- For each key in input,
- count[key] += 1

Construct position:

- Initialise first position as a 1
- position[i] = position[i-1] + count[i-1]

Construct output

- Go through input, looking at each (key, val)
- Set output[position[key]] to the (key, val) pair from input
- Increment position[key]

count

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

position

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

Output

(1,p)	(3,a)	(3,c)		(5,g)	(7,f)		
1	2	3	4	5	6	7	8

Stable Counting Sort (Method 1)

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Construct count:

- For each key in input,
- count[key] += 1

Construct position:

- Initialise first position as a 1
- position[i] = position[i-1] + count[i-1]

Construct output

- Go through input, looking at each (key, val)
- Set output[position[key]] to the (key, val) pair from input
- Increment position[key]

count

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

position

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

Output

(1,p)	(3,a)	(3,c)	(3,b)	(5,g)	(7,f)		
1	2	3	4	5	6	7	8

Stable Counting Sort (Method 1)

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Construct count:

- For each key in input,
- count[key] += 1

Construct position:

- Initialise first position as a 1
- position[i] = position[i-1] + count[i-1]

Construct output

- Go through input, looking at each (key, val)
- Set output[position[key]] to the (key, val) pair from input
- Increment position[key]

count

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

position

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

Output

(1,p)	(3,a)	(3,c)	(3,b)	(5,g)	(7,f)	(7,d)	
1	2	3	4	5	6	7	8

Stable Counting Sort (Method 1)

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Construct count:

- For each key in input,
- count[key] += 1

Construct position:

- Initialise first position as a 1
- position[i] = position[i-1] + count[i-1]

Construct output

- Go through input, looking at each (key, val)
- Set output[position[key]] to the (key, val) pair from input
- Increment position[key]

count

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

position

1	2
2	2
3	5
4	5
5	6
6	7
7	7
8	9

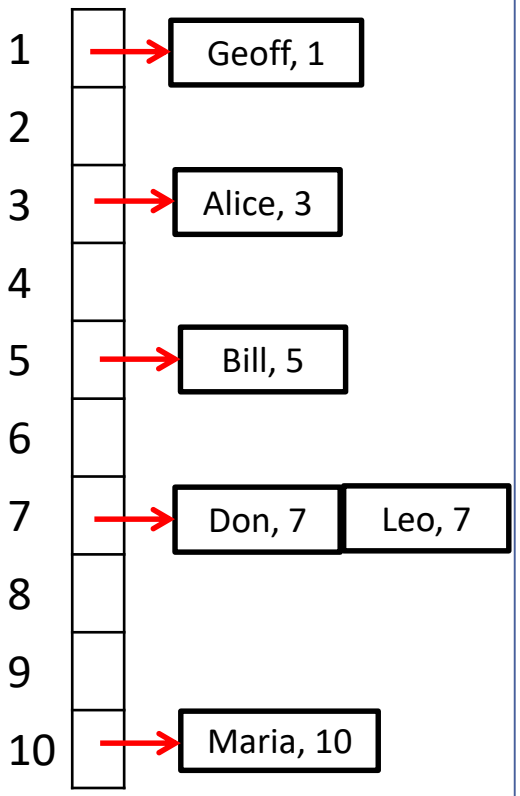
Output

(1,p)	(3,a)	(3,c)	(3,b)	(5,g)	(7,f)	(7,d)	(8,w)
1	2	3	4	5	6	7	8

Stable Counting Sort (Method 2)

- Create an empty array **"count"** of size **max**
- For each item in **"Input"**:
  - **Append item to Count[item.key]**
- 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

Analysis of Counting Sort

- We have made count sort
  - Stable
  - Able to keep associated data

- We have **not** changed the complexity (Though one is faster in practice)

- Count sort is O(N+U). It is O(N) when U is O(N)
- Therefore, counting sort is very inefficient for certain kinds of input.
- Example:
  - N = 2, U = 92233720368547758O7
  - Input: [0, 92233720368547758O7]

- Using counting sort, can we build a fast way to sort numbers?

Outline

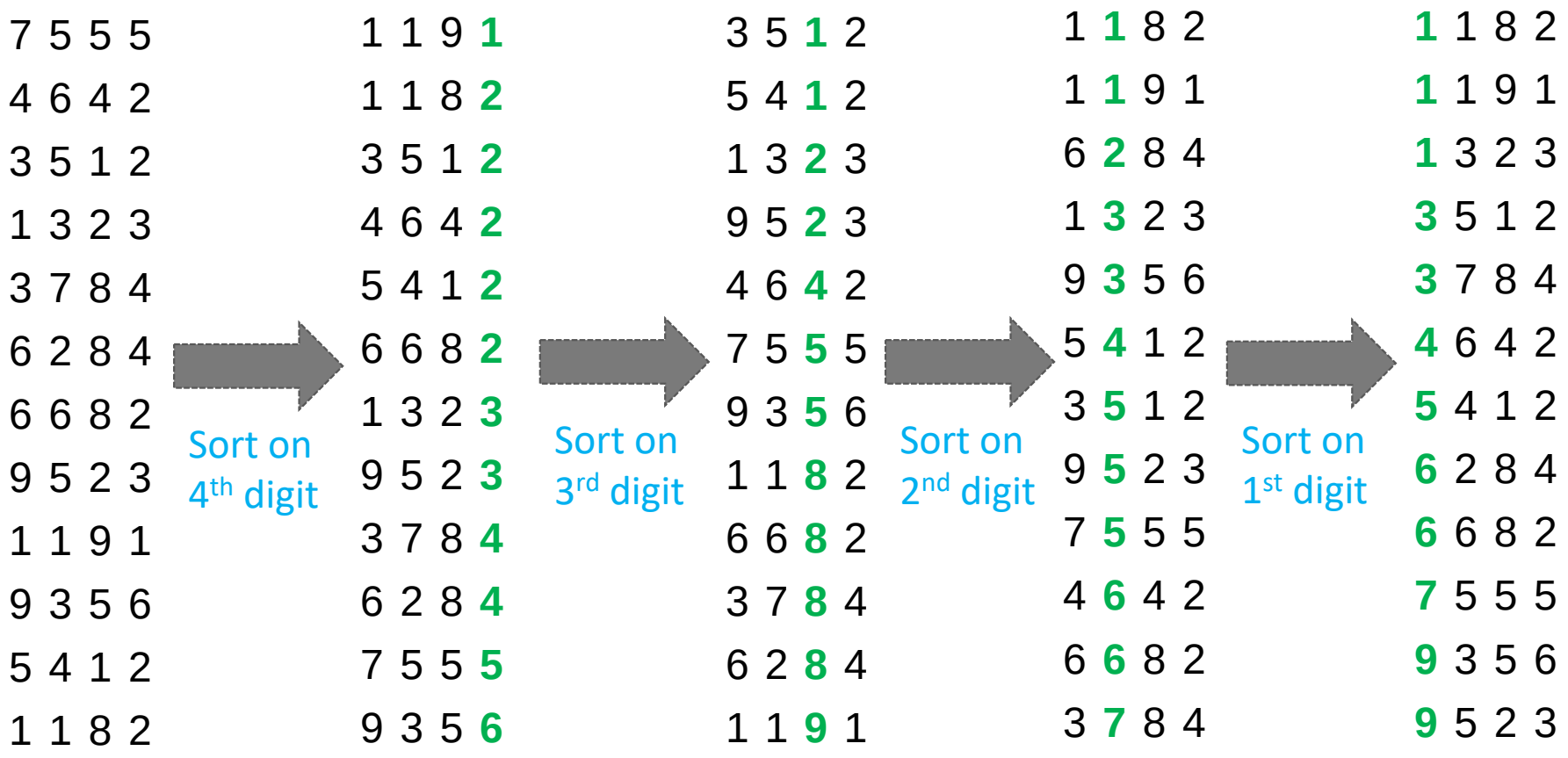
Complexity Analysis

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

Radix Sort

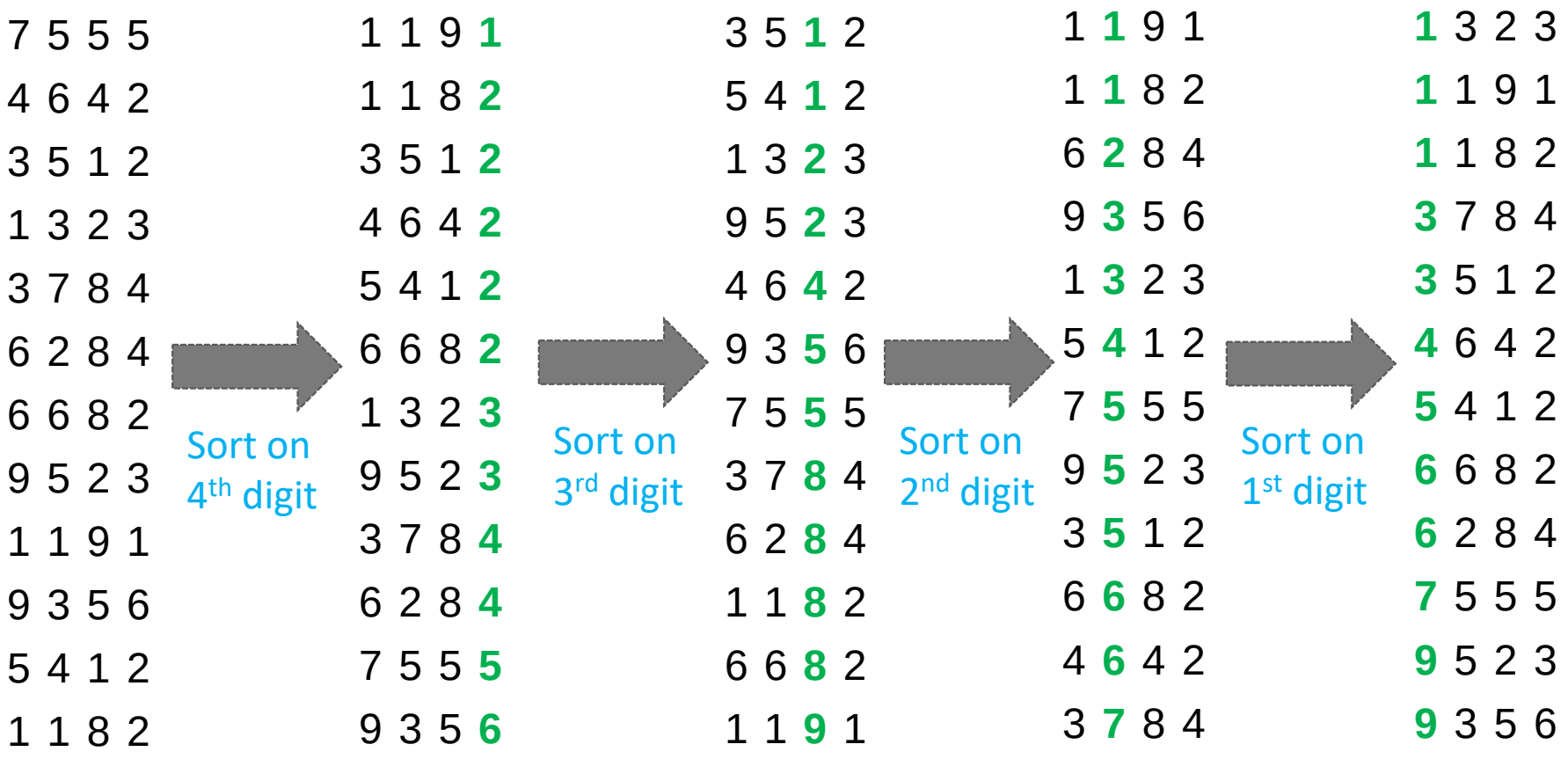
Sort an array of numbers, assuming each number has k digits (why is this often reasonable?)

- Use stable sort to sort them on the k-th digit
- Use stable sort to sort them on the (k-1)-th digit
- …
- Use stable sort to sort them on 1st digit



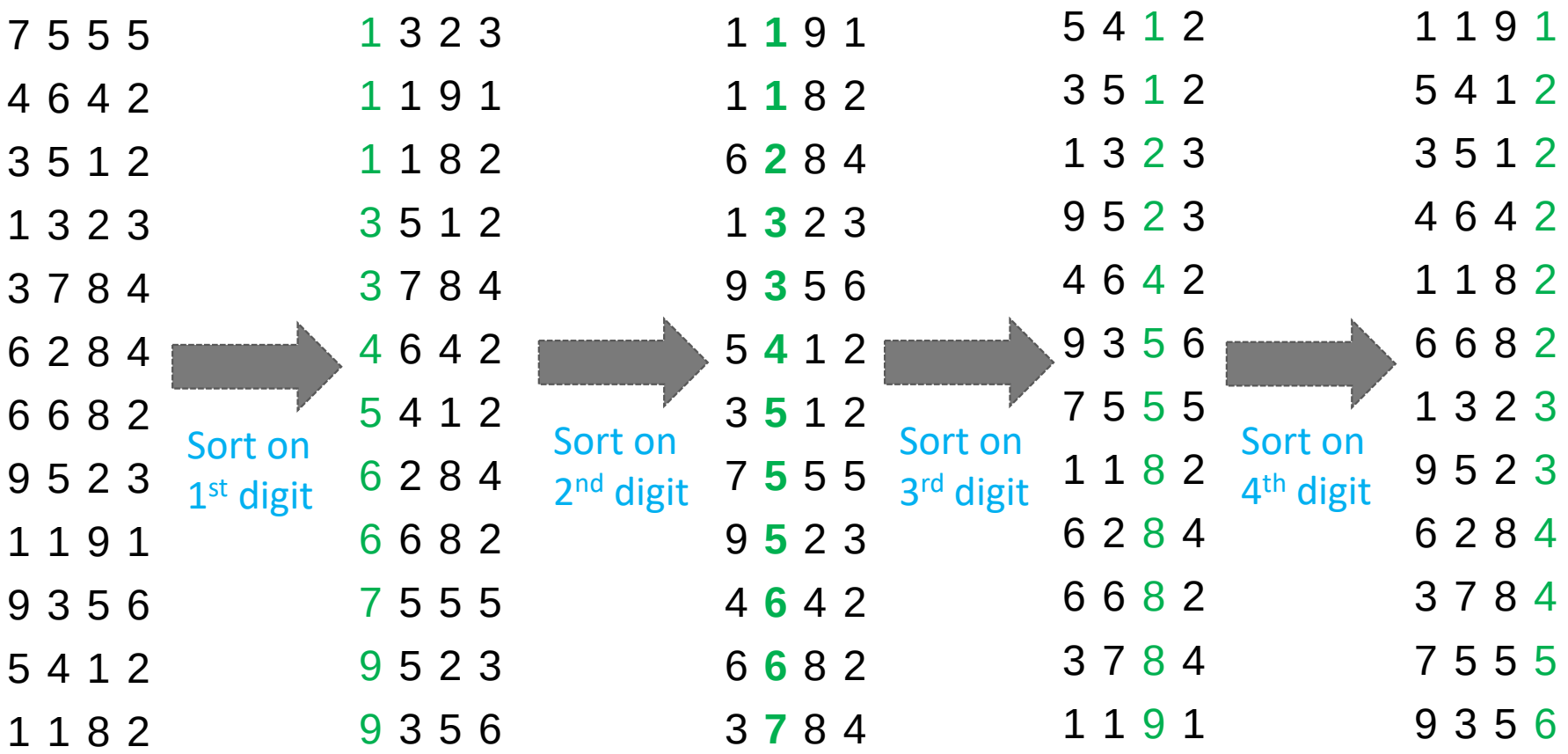
Radix Sort

What happens if we don't use stable sort?



Radix sort

What happens if we process left to right (or most significant digit to least?)



Analysis of Radix Sort

Sort an array of numbers, assuming each number has k digits

- Use stable sort to sort them on the k-th digit
- Use stable sort to sort them on the (k-1)-th digit
- …
- Use stable sort to sort them on 1st digit

Assume that N is the # of numbers to be sorted and each number has k digits

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

Time Complexity of Radix Sort:

- O((N+U)*k) → O(kN) because U is 10

Space Complexity of Radix Sort:

- O((N+U)*k) → O(kN)

Analysis of Radix Sort (Not examinable)

But wait!

- Why base 10?
- Variable base, lets call the base "b"
- How many digits does a number have in base b?
- If the number has **value** M, then it has $\log_b M$
- So we would need $\log_b M$ count sorts to sort numbers of size M

Analysis of Radix Sort (Not examinable)

- So we would need $\log_b M$ count sorts to sort numbers of size M
- Each count sort would take O(N + b)
- Total cost: $O(\log_b M * (N+b))$
- As we increase b…
- Number of count sorts goes **down**
- Cost of each count sort goes **up**

Analysis of Radix Sort (Not examinable)

- Total cost: $O(\log_b M * (N+b))$

- We want to find a good balance…
- Notice that each count sort will be O(N) as long as b is O(N)
- As an example, pick b = N (i.e. the base is the number of elements in the input)

- Total cost: $O(\log_N K * (N))$
- If K is $O(N^c)$…
- Total cost: $O(\log_N N^c * (N)) = O(CN) = O(N)$!!!

- Of course, practical considerations also matter
- Choosing a base which is a power of 2 is probably good
- Cache/localisation considerations…

Sorting Strings

- How can we apply the count sort/radix sort idea to strings?

Quiz time!
https://flux.qa - 92A2WY

- Strings have “digits” which are letters
- The mapping can be done using ASCII
  - e.g., in python
  - ord(“A”) gives 65
  - ord(“B”) gives 66
  - and so on
- The radix is now 26 (or however many characters we are using, e.g. 256)
- This may or may not be useful in assignment 1…

Concluding Remarks

Summary

- Best/worst 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)
- Try to implement radix sort yourself