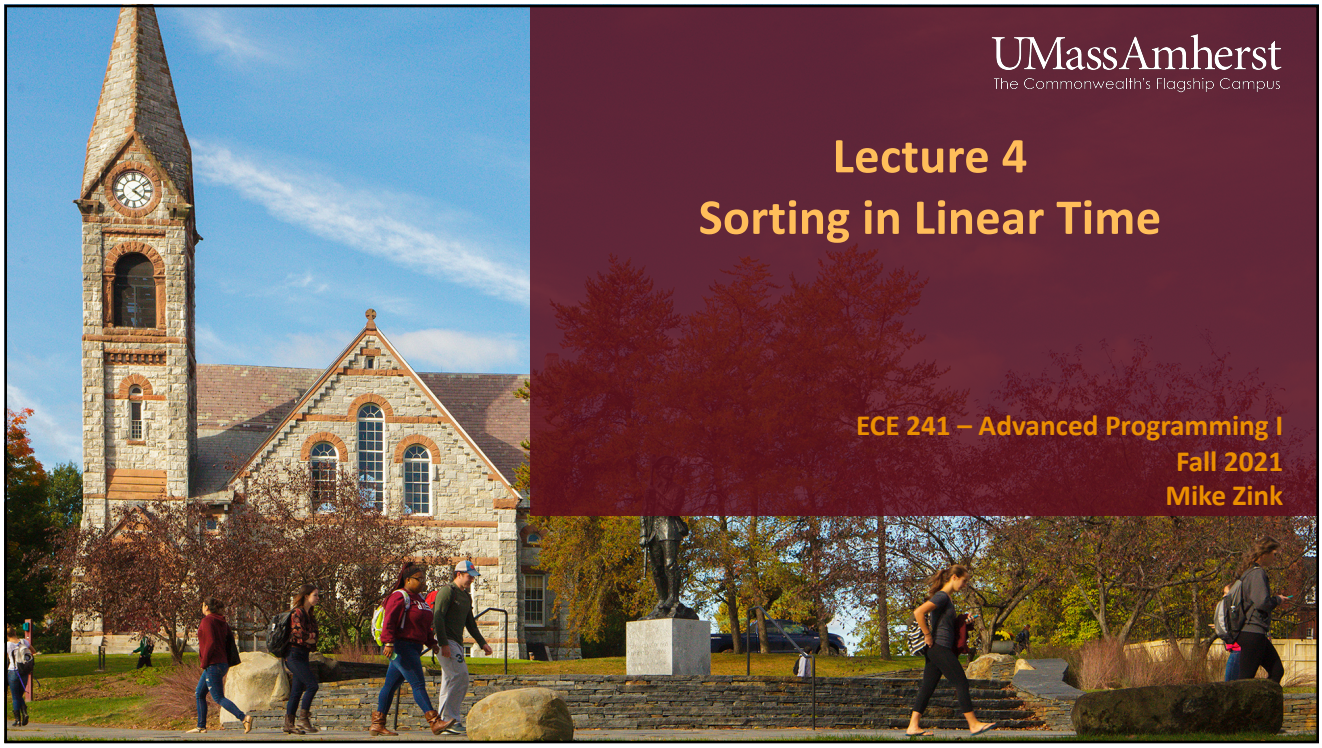


0

UMassAmherst

Overview

- Counting Sort
- Radix Sort
- Bucket Sort

1

UMassAmherst

Objective

- Understand that comparison is not the only method to sort lists
- Learn about sorting algorithms that run in linear time

2

UMassAmherst

Introduction

- Now know several algorithms that can sort in $O(n \log n)$ time
  - *Merge Sort* and *Heapsort* achieve upper bound in worst case
  - *Quicksort* achieves this on average
- Property: *the sorted order they determine is based only on comparison between the input elements* -> **comparison sorts**

3

UMassAmherst

Lower Bounds for Sorting

- Comparison sort:
  - Use only comparison between input sequence $<a_1, a_2, \ldots, a_n>$ to gain order information
  - Given a_i and a_j, perform tests $a_i < a_j$, $a_i <= a_j$, $a_i = a_j$, $a_i >= a_j$, or $a_i > a_j$
- No value inspection, no gaining of order information in any other way

4

UMassAmherst

Lower Bounds for Sorting

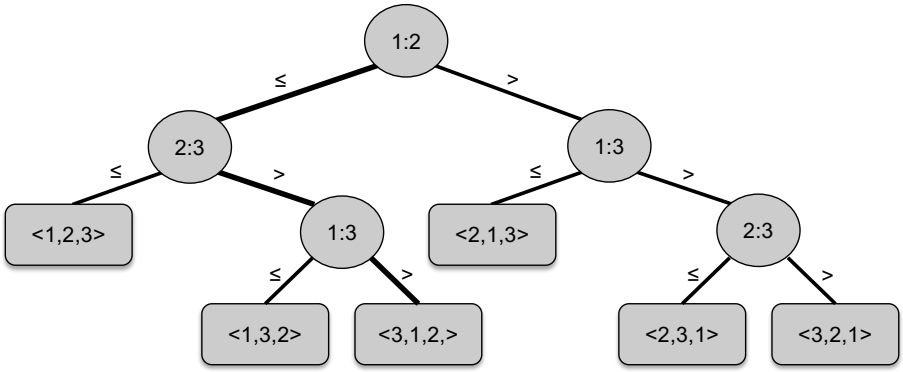
- Assumption: all input elements are distinct
  - Comparison $a_i = a_j$ is useless
  - Comparisons, $a_i < a_j$, $a_i <= a_j$, $a_i >= a_j$, and $a_i > a_j$ all equivalent since they yield identical information about relative order of a_i and a_j
  - Therefore all comparisons have form $a_i <= a_j$

5

UMassAmherst

Decision Tree Model

- $a_1 = 6, a_2 = 8, a_3 = 5$



UMassAmherst

Lower Bound for Worst Case

- Worst case number of comparisons:
  - Longest path from root to any reachable leave
  - Height of decision tree is lower bound

UMassAmherst

Counting Sort

- Assumes *n* input elements (all integers) in range 0 to *k*
- Basic idea:
  - Determine for each input element *x*, number of elements less than *x*
  - Information can be used to place element *x* directly into its position in output array
  - E.g., if there are 17 elements less than *x,* it belongs in output position 18.

8

UMassAmherst

Counting Sort - Algorithm

- Input Array *A*[1 .. *n*]
- *B*[1 .. *n*] holds stored output
- *C*[0 .. *k*] provides temporary working storage

9

UMassAmherst

```
COUNTING-SORT(A, B, K)
1  for i <- 0 to k
2          do C[i] <- 0
3   for j <- 1 to length[A]
4          do C[A[j]] <- C[A[j]] + 1
5  # C[i] now contains the number of elements equal to i.
6   for i <- 1 to k
7          do C[i] <- C[i] + C[i - 1]
8   # C[i] now contains the number of elements less than or equal to i.
9  for j <- 1 to length[A] downto 1
10         do B[C[A[j]]] <-  A[j]
11         C[A[j]] <- C[A[j]] - 1
```


UMassAmherst

Counting Sort - Example

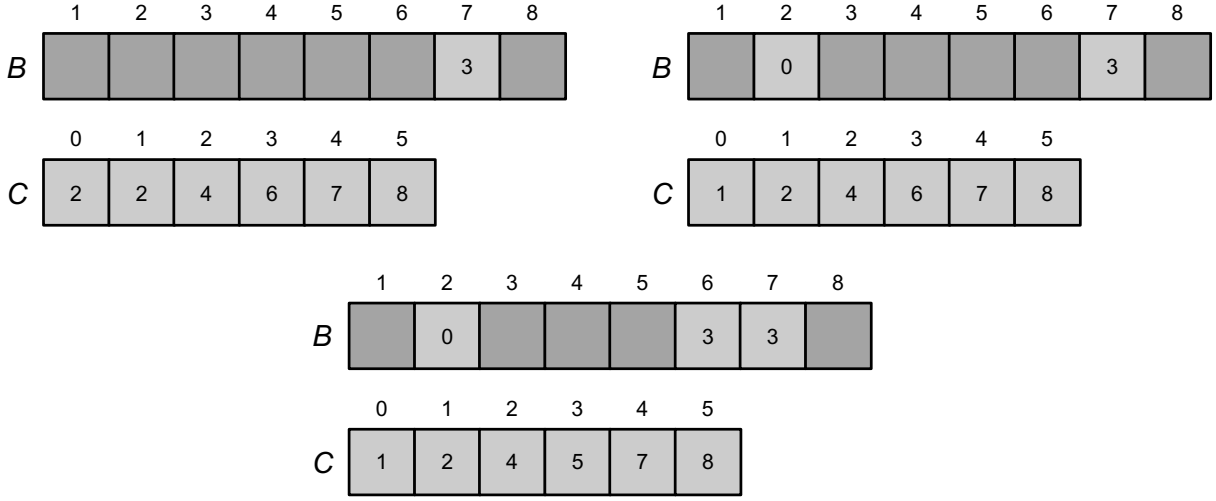
	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
C	2	0	2	3	0	1

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

UMassAmherst

Counting Sort - Example



UMassAmherst

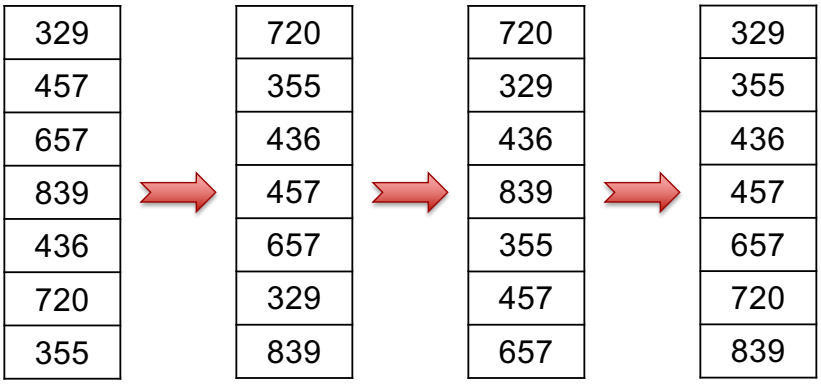
Counting Sort - Analysis

- for loop lines 1-2: O(k)
- For loop lines 3-4: O(n)
- For loop lines 6-7: O(k)
- For loop lines 9-11: O(n)
- Total: O(k+n)
- Use when k = O(n) -> O(n)

UMassAmherst

Radix Sort

- Sort numbers in a column digit-by-digit
- Starting with the least significant bit



UMassAmherst

Radix Sort

- In computer, used to sort records of information keyed by multiple fields
- E.g., sort date by three keys: year, month, day
- Compare years, if tie compare months, if tie compare days

UMassAmherst

Radix Sort

- Sort dates

2017-12-01	2011-03-30	2011-03-30
2016-04-16	2013-01-10	2013-01-10
2017-05-05	2014-11-14	2014-11-14
2013-01-10	2016-04-16	2016-04-16
2016-06-01	2016-06-01	2016-06-01
2014-11-14	2017-12-01	2017-05-05
2011-03-30	2017-05-05	2017-12-01

UMassAmherst

Radix Sort

```
RADIX-SORT(A, d)
1  for i <- 1 to d
2          do  use a stable sort to sort array A on digit i
```

UMassAmherst

Radix Sort - Analysis

Lemma

Given n d-digit numbers in which each digit can take on up to k possible values, Radix-Sort correctly sorts these numbers in $O(d(n + k))$ time if the stable sort it uses takes $O(n + k)$ time.

Proof

The correctness of Radix-Sort follows by induction on the column being sorted. The analysis of the running time depends on the stable sort used as the intermediate sorting algorithm. When each digit is in the range 0 to $k-1$ (so that it can take on k positive values), and k is not too large, counting sort is the obvious choice. Each pass over n d-digit numbers then takes time $O(n + k)$. There are d passes, so the total time for radix sort is $O(d(n + k))$.

UMassAmherst

Bucket Sort

- Assumption about input
  - Input is drawn from uniform distribution
  - Input is generated by random process that distributes elements uniformly and independently over interval [0, 1)
  - (In Counting Sort: input consists of integers in a small range)

UMassAmherst

Bucket Sort

- Idea: divide interval [0,1) into *n* equal-sized sub-intervals or **buckets**
- Distribute *n* input numbers into buckets
- Since input uniformly and independently distributed over [0, 1) => expect not too many numbers fall into one bucket
- **Output:** simply sort numbers in each bucket and go through buckets in order

UMassAmherst

Bucket Sort

```
BUCKET-SORT(A)
1  n <- length[A]
2   for i <- 1 to n
3              do insert A[i] into list B[[n A[i]]]
4  for i <- 0 to n - 1
5              do sort list B[i] with insertion sort
6   concatenate the lists B[0], B[1], …, B[n - 1] together in order
```

UMassAmherst

Bucket Sort - Example

	A
1	.78
2	.17
3	.39
4	.26
5	.72
6	.94
7	.21
8	.12
9	.23
10	.68

	B
0	/
1	→ .12 → .17 /
2	→ .21 → .23 → .26 /
3	→ .39
4	/
5	/
6	→ .68 /
7	→ .72 → .78 /
8	/
9	→ .94

UMassAmherst

Bucket Sort – Analysis

- All lines except line 5 take 0(n) time in worst case
- Analysis of cost of insertion sort reveals that expected time for Bucket Sort is:
  $O(n) + n * O(2 - 1/n) = O(n)$
- Runs in linear expected time

UMassAmherst

Next Steps

- Next lecture on Thursday
- Discussion on Thursday
- HW2 will be posted on Thursday

UMassAmherst
The Commonwealth's Flagship Campus