Algorithm and Data Structure Analysis (ADSA)

Lecture 5: Linear-Time Sorting Algorithms (Counting sort)

Sorting So Far

- Insertion sort:
  - Easy to code
  - Fast on small inputs (less than ~50 elements)
  - Fast on nearly-sorted inputs
  - At each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain.
  - $O(n^2)$ worst case
  - $O(n^2)$ average case
  - $O(n)$ best case

Sorting So Far

- Merge sort:
  - Divide-and-conquer:
    - Split array in half
    - Recursively sort subarrays
    - Linear-time merge step
  - O(n lg n) worst case
  - O(n lg n) best case
  - Doesn't sort in place

Sorting So Far

- Heap sort:
  - Uses the very useful heap data structure
    - Complete binary tree
    - Heap property: parent key > children's keys
  - O(n lg n) worst case
  - Sorts in place
  - Fair amount of shuffling memory around

Sorting So Far

- Quick sort:
  - Divide-and-conquer:
    - Partition array into two subarrays, recursively sort
    - All of first subarray < all of second subarray
    - No merge step needed!
  - Sorts in place
  - $O(n^2)$ worst case
    - Naïve implementation: worst case on sorted input
    - Address this with randomized quicksort
  - O(n lg n) best case

How Fast Can We Sort?

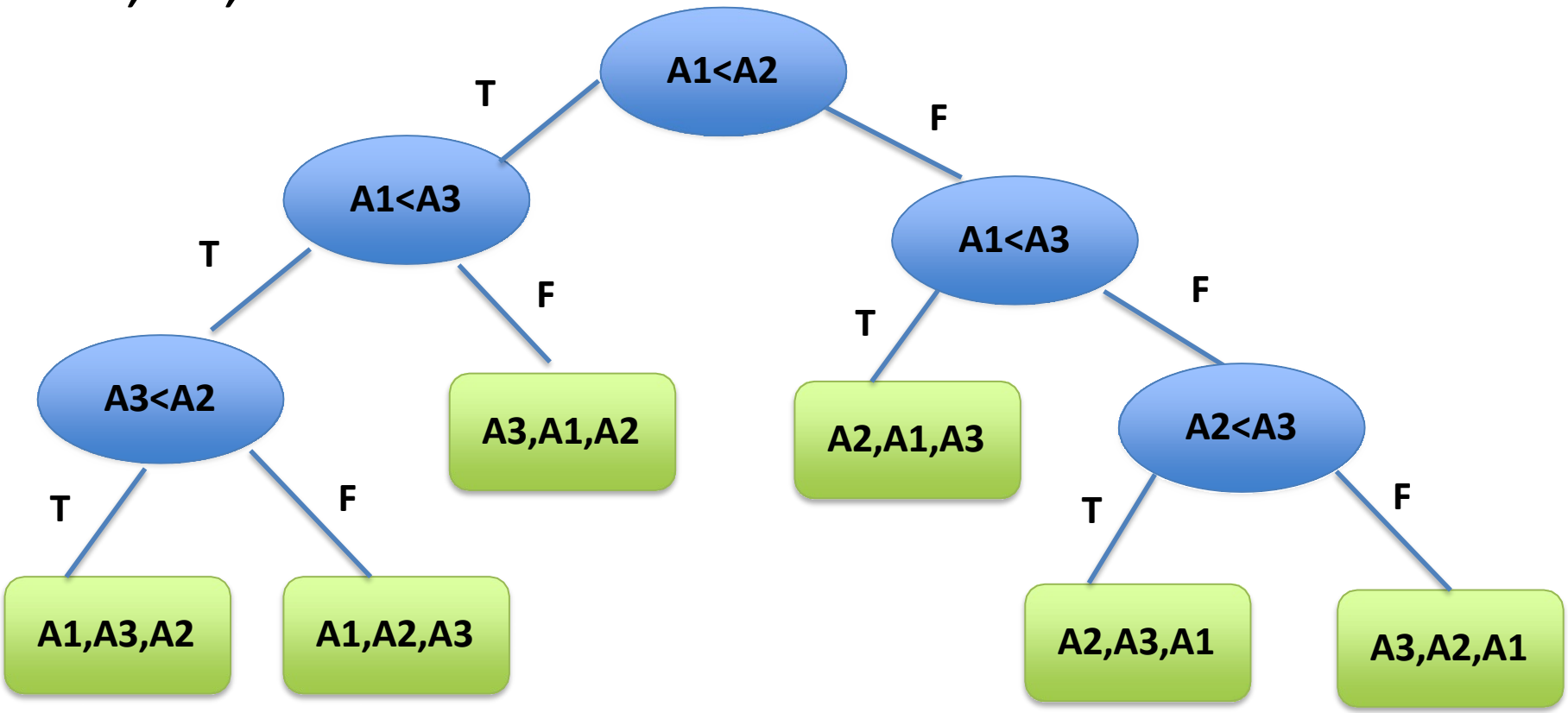
- We will provide a lower bound, then beat it
  - *How do you suppose we'll beat it?*
- First, an observation: all of the sorting algorithms so far are *comparison sorts*
  - The only operation used to gain ordering information about a sequence is the pairwise comparison of two elements
  - Theorem: all comparison sorts are $\Omega(n \lg n)$
    - A comparison sort must do O(n) comparisons
      - *why?* Need to examine every element at least once
    - What about the gap between O(n) and O(n lg n)

Decision Trees

- *Decision trees* provide an abstraction of comparison sorts
  - Any comparison algorithm can be viewed as a tree of all possible comparisons, their outcomes and resulting answer
  - **A decision tree represents the comparisons made by a comparison sort. Every thing else ignored.**
  - The tree models **all possible execution traces for that algorithm on the input size**
  - **A path from the root to a leaf is one computation.**
- *What do the leaves represent?*
- *How many leaves must there be?*

Decision Trees

A1,A2,A3



Decision Trees - example

Decision tree	Algorithm
Internal node	Binary decision (comparison)
Leaf	Found answer
Going from root-to-leaf	Algorithm execution
Path length	Running time
Height of the tree	Worse case running time

Lower Bound For Comparison Sorting

Theorem: Any decision tree that sorts *n* elements has height $\Omega(n \log n)$

- *What's the minimum # of leaves?*
- *What's the maximum # of leaves of a binary tree of height h?*
- Clearly the minimum # of leaves is less than or equal to the maximum # of leaves

Lower Bound For Comparison Sorting

So we have…

$n! \leq 2^h$

Taking logarithms:

$\log (n!) \leq h$

Lower Bound For Comparison Sorting …

$$\log(n!) = \log(1) + \ldots + \log(n/2) + \ldots + \log(n)$$
$$\geq \log(n/2) + \ldots + \log(n)$$
$$= \log(n/2) + \log(n/2+1) + \ldots + \log(n-1) + \log(n)$$
$$\geq \log(n/2) + \ldots + \log(n/2)$$
$$= n/2 * \log(n/2)$$
$$= n/2 \log(n) - n/2$$

$$n/2 \log(n) - n/2 \leq \log(n!) \leq h$$

Thus the minimum height of a decision tree is $\Omega(n \lg n)$

Lower Bound For Comparison Sorts

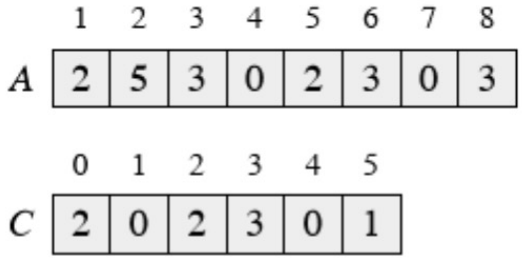
- All comparison sorts are $\Omega(n \lg n)$
- Corollary: Heapsort and Merge sort are asymptotically optimal comparison sorts
- But the name of this lecture is "Sorting in linear time"!
  - *How can we do better than $\Omega(n \lg n)$?*

Sorting In Linear Time

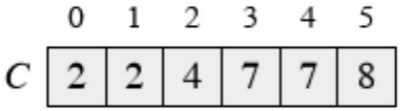
- Counting sort
  - No comparisons between elements!
  - ***But***…depends on assumption about the numbers being sorted
    - We assume numbers are in the range *1.. k*
  - The algorithm:
    - Input: A[1..*n*], where A[j] $\in$ {1, 2, 3, …, *k*}
    - Output: B[1..*n*], sorted (notice: not sorting in place)
    - Also: Array C[1..*k*] for auxiliary storage

Counting Sort

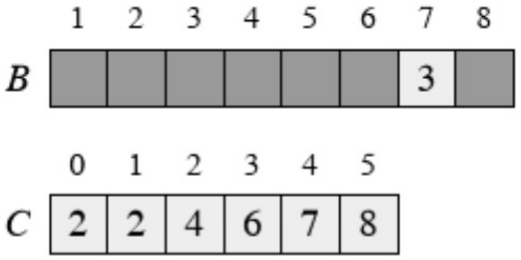
```
1  CountingSort(A, B, k)
2for i=1 to k 3
      C[i]= 0;
4     for j=1 to n
5        C[A[j]] += 1;
6     for i=2 to k
7        C[i] = C[i] + C[i-1];
8     for j=n downto 1
9        B[C[A[j]]] = A[j];
10       C[A[j]] -= 1;
```



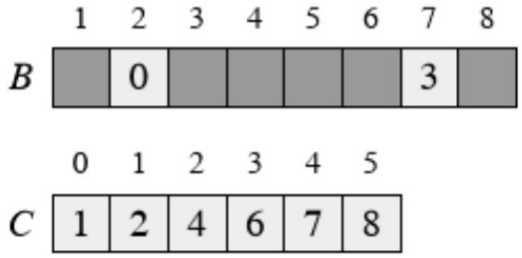
(a)



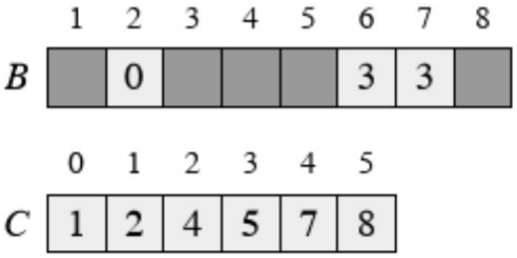
(b)



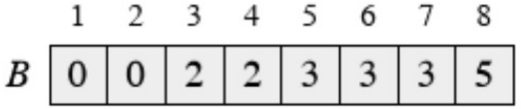
(c)



(d)



(e)



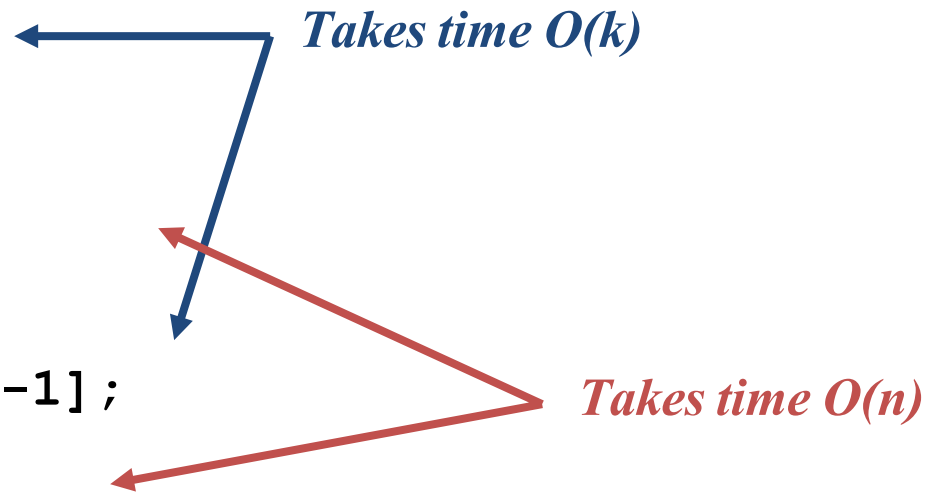
(f)

```
COUNTING-SORT(A, B, k)
 1  let C[0..k] be a new array
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to A.length
 5      C[A[j]] = C[A[j]] + 1
 6  // C[i] now contains the number of elements equal to i.
 7  for i = 1 to k
 8      C[i] = C[i] + C[i − 1]
 9  // C[i] now contains the number of elements less than or equal to i.
10  for j = A.length downto 1
11      B[C[A[j]]] = A[j]
12      C[A[j]] = C[A[j]] − 1
```

The operation of COUNTING-SORT on an input array A[1…8], where each element of A is a nonnegative integer no larger than k=5. (a) The array A and the auxiliary array C after line 5. (b) The array C after line 8. (c)–(e) The output array B and the auxiliary array C after one, two, and three iterations of the loop in lines 10–12, respectively. Only the lightly shaded elements of array B have been filled in. (f) The final sorted output array B.

Counting Sort

```
1  CountingSort(A, B, k)
2for i=1 to k 3
       C[i]= 0;
4      for j=1 to n
5          C[A[j]] += 1;
6      for i=2 to k
7          C[i] = C[i] + C[i-1];
8      for j=n downto 1
9          B[C[A[j]]] = A[j];
10         C[A[j]] -= 1;
```



What will be the running time?

Counting Sort

- Total time: $O(n + k)$
  - Usually, $k = O(n)$
  - Thus counting sort runs in $O(n)$ time
- But sorting is $\Omega(n \lg n)$!
  - No contradiction--this is not a comparison sort (in fact, there are *no* comparisons at all!)
  - Notice that this algorithm is *stable*