Analysis of Algorithms 1
(Fall 2013)
Istanbul Technical University
Computer Eng. Dept.

Chapter 8: Sorting in Linear Time

Course slides from
Susan Bridges @MS State
have been used in
preparation of these slides.

Last updated: October 25, 2011

Purpose

- Introduce decision-tree model to study performance limitations of comparison sorts
- Prove lower bound on worst-case running time of any comparison sort
- Learn about counting sort, radix sort, and bucket sort

Outline

- Lower Bounds for Sorting
  - Decision Tree Model
- Counting Sort
- Radix Sort
- Bucket Sort

Sorting

- Discussed several algorithms that can sort *n* numbers in O(n lg n) time
  - Mergesort and heapsort achieve this upper bound in worst case
  - Quicksort achieves it on average
- For each of these algorithms, we can produce a sequence of *n* input numbers that causes the algorithm to run in Ω(n lg n)

Comparison Sorts

- These algorithms share an interesting property:

    The sorted order they determine is based only on comparisons between the input elements

- We call such sorting algorithms comparison sorts
- All the sorting algorithms we have discussed so far are comparison sorts

Detailed Outline

- Prove that any comparison sort must make Θ(n lg n) comparisons in the worst case to sort *n* elements
  - Thus, merge sort and heapsort are asymptotically optimal, and no comparison sort exists that is faster by more than a constant factor
- Three sorting algorithms that run in linear time
  - counting sort, radix sort, and bucket sort

Lower Bounds for Comparison Sorts

Assume:

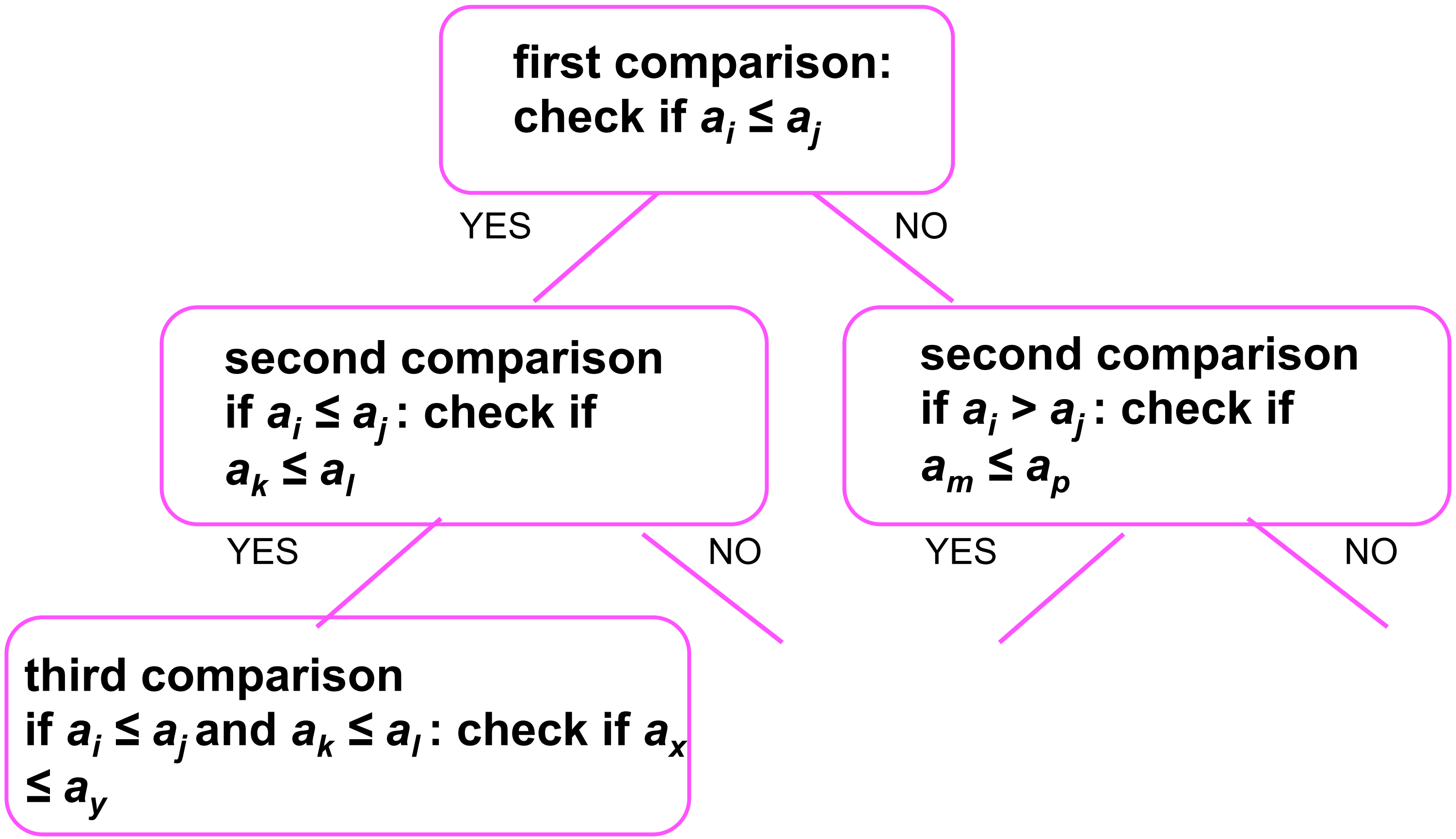
- All elements are distinct
- All comparisons are of form $a_i \leq a_j$

Can view any comparison sort in terms of a decision tree

Decision Tree

- Consider *any* comparison based sorting algorithm
- Represent its behavior on all inputs of a fixed size with a *decision tree*
- Each tree node corresponds to the execution of a comparison
- Each tree node has two children, depending on whether the parent comparison was true or false
- Each leaf represents correct sorted order for that path

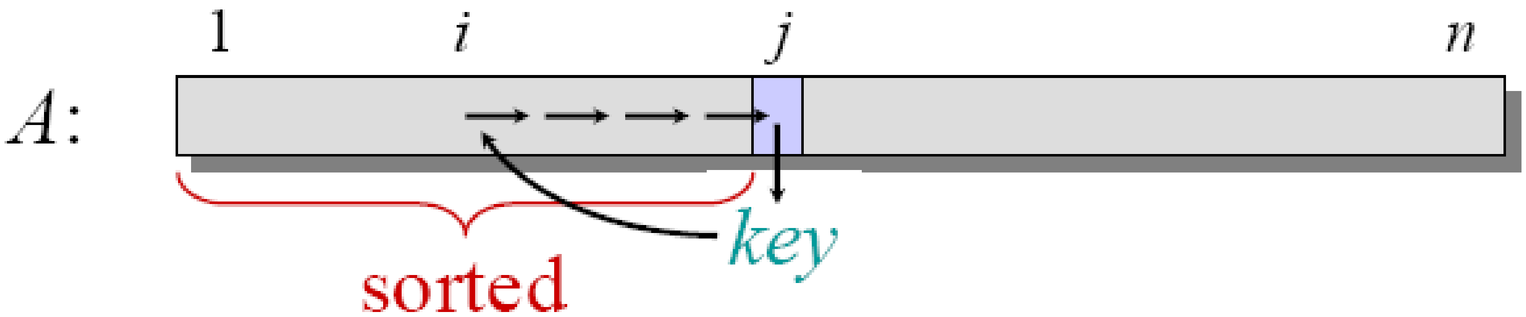
Decision Tree Diagram



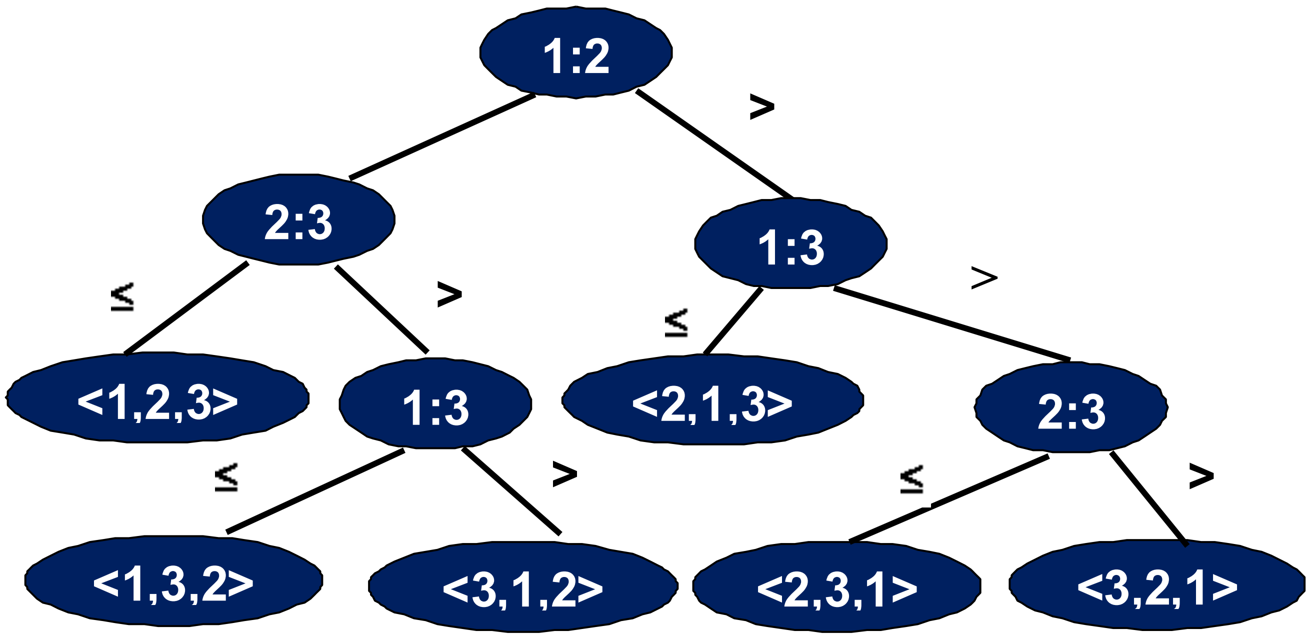
Insertion Sort Algorithm

"pseudocode"

```
INSERTION-SORT (A, n)        ▷ A[1 . . n]
    for j ← 2 to n
        do key ← A[j]
           i ← j – 1                        comparison
           while i > 0 and A[i] > key
               do A[i+1] ← A[i]
                  i ← i – 1
           A[i+1] = key
```



Decision Tree for Insertion Sort (n=3)



How Many Leaves?

- Must be at least one leaf for each permutation of the input
  - Otherwise, there would be a situation that was not correctly sorted
- Number of permutations of *n* keys is *n!*
- Idea: since there must be a lot of leaves, but each decision tree node only has two children, tree cannot be too shallow
  - depth of tree is a lower bound on running time

Lower Bounds for Worst Case

Theorem:

Any comparison sort algorithm requires $\Omega(n \lg n)$ comparisons in the worst case.

Lower Bounds for Worst Case (cont.)

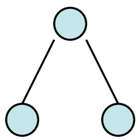
Proof:

- Let h be height of tree and l be number of leaves
- Tree must have at least n! leaves since each permutation of input must be a leaf

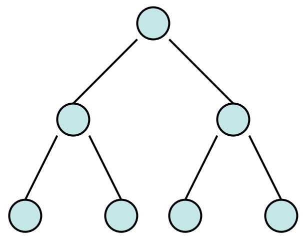
Lower Bounds for Worst Case (cont.)

Proof (cont.):

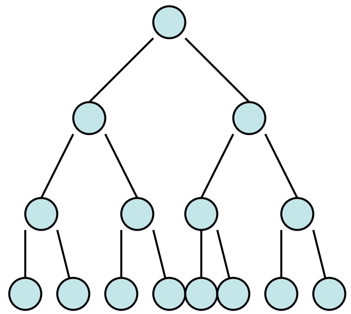
- Binary tree of height h has at most 2^h leaves



h = 1,
***2^1* leaves**



***h = 2, 2^2* leaves**



***h = 3, 2^3* leaves**

- Thus: $n! \le l \le 2^h$
- By taking logarithms:

$$h \ge \lg(n!) = \lg(n)+\lg(n-1)+\ldots+\lg(1) \le n\lg n$$

$$= \Omega(n \lg n)$$

Lower Bounds for Worst Case (cont.)

Corollary:

Heapsort and merge sort are asymptotically optimal comparison sorts.

Proof:

$O(n \lg n)$ upper bounds on running times for heapsort and merge sort match $\Omega(n \lg n)$ worst-case lower bound from the theorem

Counting Sort

- Assumes each of the *n* input elements is an integer in range 0 to *k*, for some integer *k*
- For each element x, determine number of values $\leq$ x
  - This information can be used to place element x into its position
  - Example: 17 elements less than x, x belongs in position 18
- Requires three arrays
  - Input array A[1..*n*]
  - Array B[1..*n*] for sorted output
  - Array C[0..*k*] for counting number of times each element occurs (temporary working storage)

Counting Sort (cont.)

```
COUNTING-SORT (A, B, k)
for i ← 0 to k
    do C[i] ← 0
for j ← 1 to length[A]
    do C[A[j]] ← C[A[j]] + 1
▷C[i] now contains the number of elements equal to i
for i ← 1 to k
    do C[i] ← C[i] + C[i-1]
▷C[i] now contains the number of elements less than or
equal to i
for j ← length[A] downto 1
    do B[C[A[j]]] ← A[j]
        C[A[j]] ← C[A[j]] - 1
```

Counting Sort

A	4	5	1	0	1	3	2	4	1

Max{A[i]}=5 $\rightarrow$ k = 5

	0	1	2	3	4	5
C	0	0	0	0	0	0
C	1	3	1	1	2	1
C	1	4	5	6	8	9

```
for i ← 0 to k
    do C[i] ← 0

for j ← 1 to length[A]
    do C[A[j]] ← C[A[j]] + 1

for i ← 1 to k
do C[i] ← C[i] + C[i-1]
```

	1	2	3	4	5	6	7	8	9
A	4	5	1	0	1	3	2	4	1
B	0	1	1	1	2	3	4	4	5

```
for j ← length[A] downto 1
    do B[C[A[j]]] ← A[j]
       C[A[j]] ← C[A[j]] - 1
```

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

Complexity of Counting Sort

```
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 ← length[A] downto 1
10       do B[C[A[j]]] ← A[j]
11           C[A[j]] ← C[A[j]] - 1
```

Lines 1-2	$\Theta(k)$
Lines 3-4	$\Theta(n)$
Lines 6-7	$\Theta(k)$
Lines 9-11	$\Theta(n)$
Total	$\Theta(n+k)$

Counting Sort

- Beats the lower bound of $\Omega(n \lg n)$ because it is not a comparison sort
- Makes assumptions about the input data
- Is a stable sort
  - Numbers with the same value appear in the output array in the same order as they do in the input array
  - Important if a satellite data are attached to the element being sorted
  - Counting sort often used as a subroutine in radix sort

Radix Sort

- Origin
  - Goes back to late 19th century: Herman Hollerith's card-sorting machine for 1890 US census
- Approach
  - Sort numbers digit-by-digit
  - Start with the least significant digit

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

Radix Sort

3 2 9	7 2 0	7 2 0	3 2 9
4 5 7	3 5 5	3 2 9	3 5 5
6 5 7	4 3 6	4 3 6	4 3 6
8 3 9	4 5 7	8 3 9	4 5 7
4 3 6	6 5 7	3 5 5	6 5 7
7 2 0	3 2 9	4 5 7	7 2 0
3 5 5	8 3 9	6 5 7	8 3 9

Inductive Analysis of Radix Sort

- Say it is sorted for the first n digits (columns), consider sorting for the n+1th digit (column)

Case 1

Two numbers differ in digit n+1

Obvious

7 2 0	3 2 9
3 2 9	3 5 5
4 3 6	4 3 6
8 3 9	4 5 7
3 5 5	6 5 7
4 5 7	7 2 0
6 5 7	8 3 9

Case 2

Two numbers same in digit n+1

Sorting must be stable!

Running Time of Radix Sort

- Given:
  - n d-digit number
  - each digit can take k possible values
- Radix sort sorts these numbers in $\Theta(d(n+k))$ time
- Why?

Running Time of Radix Sort

- The running time depends upon the intermediate stable sorting algorithm run for each digit
- Assume that counting sort is used for intermediate sorting
- Each pass over n d-digit number takes $\Theta(n+k)$ time.
- Since there are d passes the total running time for radix sort is $\Theta(d(n+k))$

Further Analysis of Radix Sort

- Given
  - n b-bit number to sort
  - $r \leq b$
- Radix sort sorts these numbers in $\Theta((b/r)(n+2^r))$ time

Further Analysis of Radix Sort

- Words can viewed as having $d = \lceil b/r \rceil$ digits of r bits each
- Each digit can be considered as an integer between 0 to 2^r-1, i.e., $k=2^r - 1$
- Each pass of counting sort $\Theta(n+k) = \Theta(n+2^r)$
- There are d passes
- Total running time

$$\Theta(d(n+2^r)) = \Theta((b/r)(n+2^r))$$

Further Analysis of Radix Sort

- Given b and n, minimize the running time, $\Theta((b/r)(n+2^r))$, where $r \leq b$
- If $b < \lfloor \log_2 n \rfloor$ any $r \leq b$ $r < \lfloor \log_2 n \rfloor$
  $\rightarrow (n+2^r) = \Theta(n)$
- If r=b, $\Theta((b/r)(n+2^r)) = \Theta(n+2^b) = \Theta(n)$

Further Analysis of Radix Sort

- Given b and n, minimize the running time, $\Theta((b/r)(n+2^r))$, where $r \le b$
- If $b >= \lfloor \log_2 n \rfloor$, $r < \lfloor \log_2 n \rfloor$ gives the best time
- if $r = \lfloor \log_2 n \rfloor$ running time $\Theta(bn/\log n)$
- if $r > \lfloor \log_2 n \rfloor$ running time $\Omega(bn/\log n)$
- if $r < \lfloor \log_2 n \rfloor$ running time $\Theta(n)$

Bucket Sort

- Runs in linear time when the input is drawn from a uniform distribution
- Assumption: Input is uniformly distributed
- Approach:
  - Divide distribution interval into n equal-sized intervals, i.e. buckets
  - Sort the numbers in each bucket
  - Go through the buckets in order

Bucket Sort

BUCKET-SORT(A)

n ← length[A]

for i ← 1 **to** n

    do insert A[i] into list B[$\lfloor nA[i] \rfloor$]

for i ← 0 to n-1

    do sort list B[i] with insertion sort

concatenate lists B[0], B[1], …, B[n-1] together in order

Bucket Sort

<table>
<tr><th>List</th><th>Ptr. to Buckets</th><th>Buckets</th></tr>
<tr><td>.78</td><td>null</td><td></td></tr>
<tr><td>.17</td><td>→</td><td>.12|→ .17|null</td></tr>
<tr><td>.39</td><td>→</td><td>.21|→ .23|→ .26|null</td></tr>
<tr><td>.26</td><td>→</td><td>.39|null</td></tr>
<tr><td>.72</td><td>null</td><td></td></tr>
<tr><td>.94</td><td>null</td><td></td></tr>
<tr><td>.21</td><td>→</td><td>.68|null</td></tr>
<tr><td>.12</td><td>→</td><td>.72|→ .78|null</td></tr>
<tr><td>.23</td><td>null</td><td></td></tr>
<tr><td>.68</td><td>→</td><td>.94|null</td></tr>
</table>

Running Time of Bucket Sort

for i ← 1 **to** n $\Theta(n)$

  do insert A[i] into list B[$\lfloor nA[i] \rfloor$]

for i ← 0 to n-1 $\sum_{i=0}^{n-1} O(n_i^2)$

  do sort list B[i] with insertion sort

concatenate lists B[0], B[1], …, B[n-1] together in order $\Theta(n)$

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)$$

Running Time of Bucket Sort

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^{\,2})$$

Taking expected values of both sides

$$E[T(n)] = E\left[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^{\,2})\right]$$

$$E[T(n)] = \Theta(n) + \sum_{i=0}^{n-1} E\left[O(n_i^{\,2})\right]$$

$$E[T(n)] = \Theta(n) + \sum_{i=0}^{n-1} O(E[n_i^{\,2}])$$

$$E[n_i^{\,2}] = 2 - \frac{1}{n} \quad \rightarrow \quad \Theta(\mathrm{n}) + \mathrm{n}\ O(2 - 1/\mathrm{n}) = \Theta(\mathrm{n})$$

See page 175-176 of the book

Summary

- Lower Bounds for Sorting
  - Decision Tree Model
- Counting Sort
- Radix Sort
- Bucket Sort