Selected Sorting Algorithms

CS 165: Project in Algorithms and Data Structures

Michael T. Goodrich

UNIVERSITY of CALIFORNIA IRVINE

Some slides are from J. Miller, CSE 373, U. Washington

Why Sorting?

- Practical application
  - People by last name
  - Countries by population
  - Search engine results by relevance
- Fundamental to other algorithms
- Different algorithms have different asymptotic and constant-factor trade-offs
  - No single ‘best’ sort for all scenarios
  - Knowing one way to sort just isn’t enough
- Many to approaches to sorting which can be used for other problems

Problem statement

There are *n* comparable elements in an array and we want to rearrange them to be in increasing order

Pre:

- An array **`A`** of data records
- A value in each data record
- A comparison function
  - <, =, >, compareTo

Post:

- For each distinct position **`i`** and **`j`** of **`A`**, if **`i < j`** then **`A[i]`** $\leq$ **`A[j]`**
- **`A`** has all the same data it started with

Insertion sort

- **insertion sort**: orders a list of values by repetitively inserting a particular value into a sorted subset of the list

- more specifically:
  - consider the first item to be a sorted sublist of length 1
  - insert the second item into the sorted sublist, shifting the first item if needed
  - insert the third item into the sorted sublist, shifting the other items as needed
  - repeat until all values have been inserted into their proper positions

Insertion sort

- Simple sorting algorithm.
  - n-1 passes over the array
  - At the end of pass *i*, the elements that occupied A[0]…A[*i*] originally are still in those spots and in sorted order.

	2	15	8	1	17	10	12	5
	0	1	2	3	4	5	6	7
after pass 2	2	8	15	1	17	10	12	5
	0	1	2	3	4	5	6	7
after pass 3	1	2	8	15	17	10	12	5
	0	1	2	3	4	5	6	7

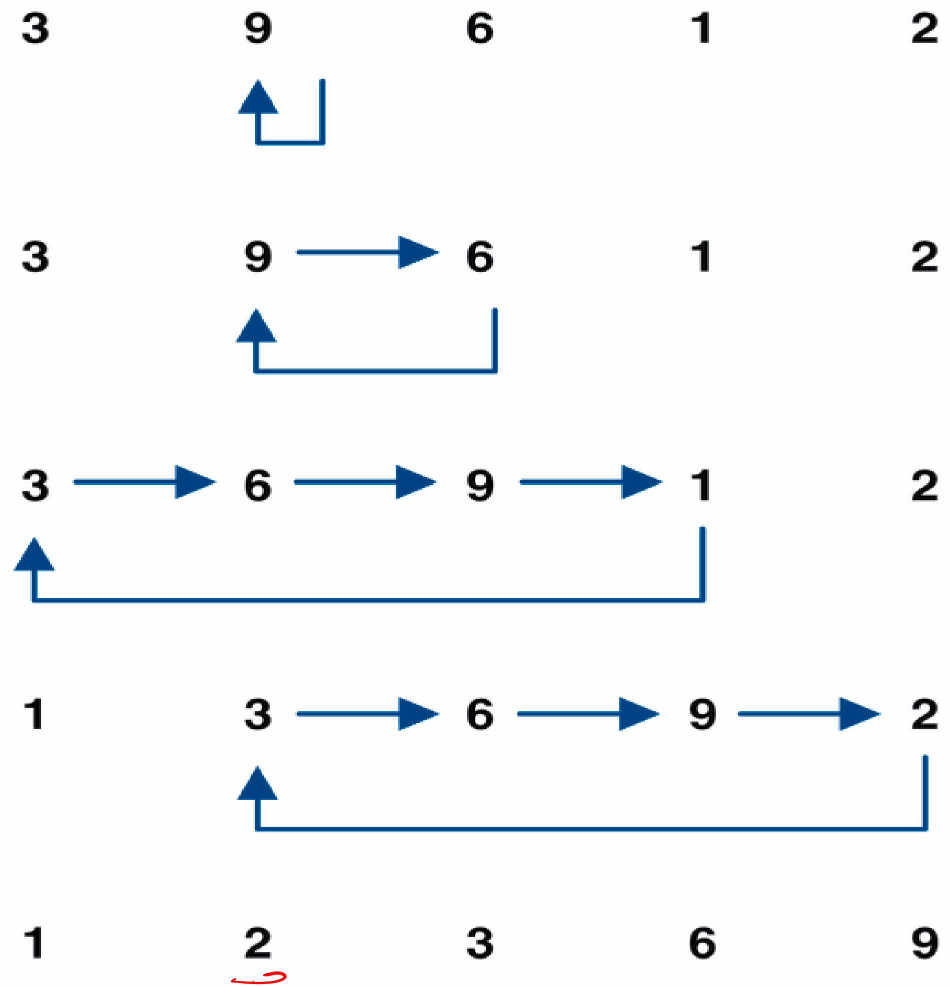
Insertion sort example

3 is sorted.
Shift nothing. Insert 9.

3 and 9 are sorted.
Shift 9 to the right. Insert 6.

3, 6, and 9 are sorted.
Shift 9, 6, and 3 to the right. Insert 1.

1, 3, 6, and 9 are sorted.
Shift 9, 6, and 3 to the right. Insert 2.



Insertion sort code

```
public static void insertionSort(int[] a) {
    for (int i = 1; i < a.length; i++) {
        int temp = a[i];

        // slide elements down to make room for a[i]
        int j = i;
        while (j > 0 && a[j - 1] > temp) {
            a[j] = a[j - 1];
            j--;
        }

        a[j] = temp;
    }
}
```

Insertion-sort Analysis

- An **inversion** in a permutation is the number of pairs that are out of order, that is, the number of pairs, (i,j), such that $i<j$ but $x_i>x_j$.
- Each step of insertion-sort fixes an inversion or stops the while-loop.
- Thus, the running time of insertion-sort is $O(n + k)$, where k is the number of inversions.

Insertion-sort Analysis

- The worst case for the number of inversions, k, is

$$1 + 2 + 3 + \cdots + n-1$$

$$= \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = O(n^2)$$

- This occurs for a list in reverse-sorted order.

Insertion-sort Analysis

- The average case for k is $A(n)$

$\cancel{x}$ = smallest element

remove it $\longrightarrow A(n-1)$

reinserting x:

$$E(I_x) = 0 \cdot \frac{1}{n} + 1 \cdot \frac{1}{n} + 2 \cdot \frac{1}{n} + \dots + (n-1)\frac{1}{n}$$

$$= \frac{1}{n} \sum_{i=1}^{n-1} i = \frac{1}{n}\left(\frac{n \cdot (n-1)}{2}\right)$$

$$= \frac{n-1}{2}$$

$$A(n) = A(n-1) + \frac{n-1}{2}$$

$$= \frac{1}{2} \sum_{i=1}^{n-1} i$$

$$= \frac{1}{2} \cdot \frac{n(n-1)}{2} = \frac{n(n-1)}{4}$$

$$= O(n^2)$$

Shell sort description

- **shell sort**: orders a list of values by comparing elements that are separated by a gap of >1 indexes
  - a generalization of insertion sort
  - invented by computer scientist Donald Shell in 1959

- based on some observations about insertion sort:
  - insertion sort runs fast if the input is almost sorted
  - insertion sort's weakness is that it swaps each element just one step at a time, taking many swaps to get the element into its correct position

Shell sort example

- Idea: Sort all elements that are 5 indexes apart, then sort all elements that are 3 indexes apart, ...

Original	32	95	16	82	24	66	35	19	75	54	40	43	93	68	
After 5-sort	32	35	16	68	24	40	43	19	75	54	66	95	93	82	6 swaps
After 3-sort	32	19	16	43	24	40	54	35	75	68	66	95	93	82	5 swaps
After 1-sort	16	19	24	32	35	40	43	54	66	68	72	82	93	95	15 swaps

Shell sort code

original

```
public static void shellSort(int[] a) {
    for (int gap = a.length / 2; gap > 0; gap /= 2) {
        for (int i = gap; i < a.length; i++) {
            // slide element i back by gap indexes
            // until it's "in order"
            int temp = a[i];
            int j = i;
            while (j >= gap && temp < a[j - gap]) {
                a[j] = a[j - gap];
                j -= gap;
            }
            a[j] = temp;
        }
    }
}
```

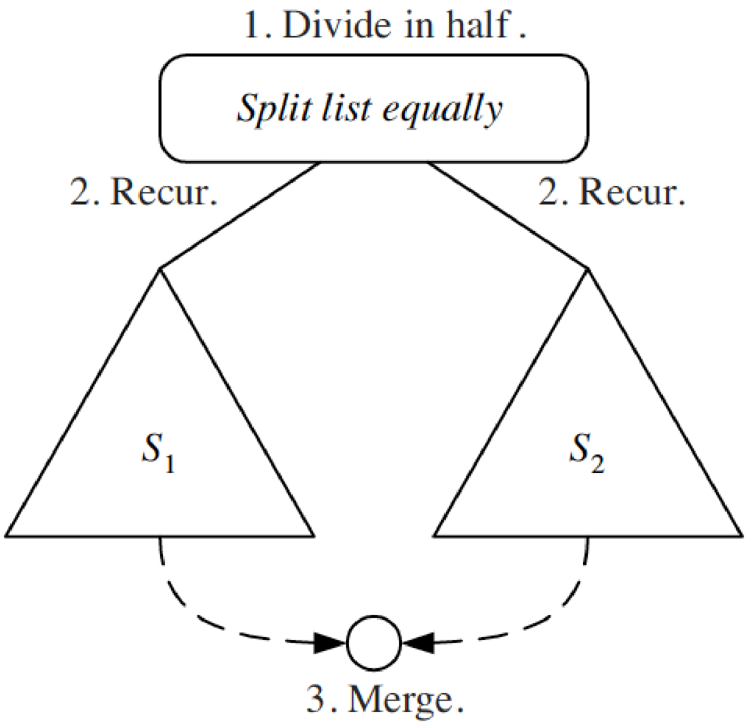
Shell sort Analysis

- Harder than insertion sort
- But certainly no worse than insertion sort
- Worst-case: $O(n^2)$
- Average-case: ????

Experiments

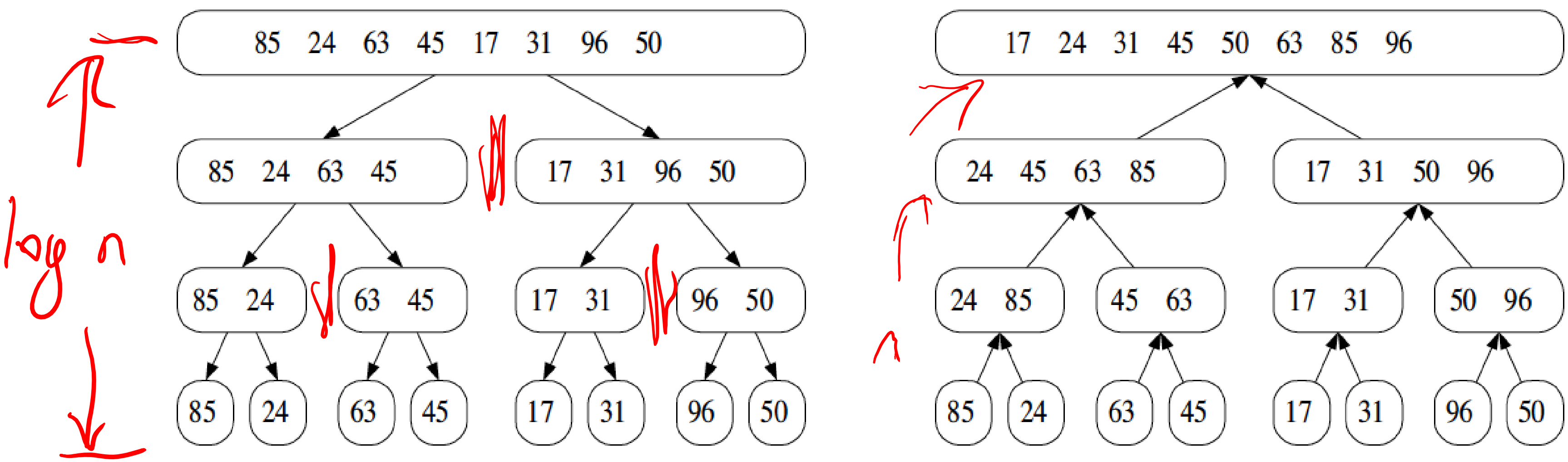
Divide-and-Conquer

- Divide-and conquer is a general algorithm design paradigm:
  - Divide: divide the input data S in two disjoint subsets S_1 and S_2
  - Recur: solve the subproblems associated with S_1 and S_2
  - Conquer: combine the solutions for S_1 and S_2 into a solution for S
- The base case for the recursion are subproblems of size 0 or 1



Merge-Sort

- Merge-sort is a sorting algorithm based on the divide-and-conquer paradigm
  - It has $O(n \log n)$ running time



The Merge-Sort Algorithm

- Merge-sort on an input sequence S with n elements consists of three steps:
  - Divide: partition S into two sequences S_1 and S_2 of about $n/2$ elements each
  - Recur: recursively sort S_1 and S_2
  - Conquer: merge S_1 and S_2 into a unique sorted sequence

Algorithm **mergeSort**(S):
  Input array S of n elements
  Output array S sorted

  if $n > 1$ **then**
    $(S_1, S_2) \leftarrow$ **partition**$(S, n/2)$
    mergeSort(S_1)
    mergeSort(S_2)
    $S \leftarrow$ **merge**(S_1, S_2)

Merging Two Sorted Sequences

- The conquer step of merge-sort consists of merging two sorted sequences ***A*** and ***B*** into a sorted sequence ***S*** containing the union of the elements of ***A*** and ***B***
- Merging two sorted sequences, each with $n/2$ elements and implemented by means of a doubly linked list, takes $O(n)$ time

Algorithm merge(S_1, S_2, S):

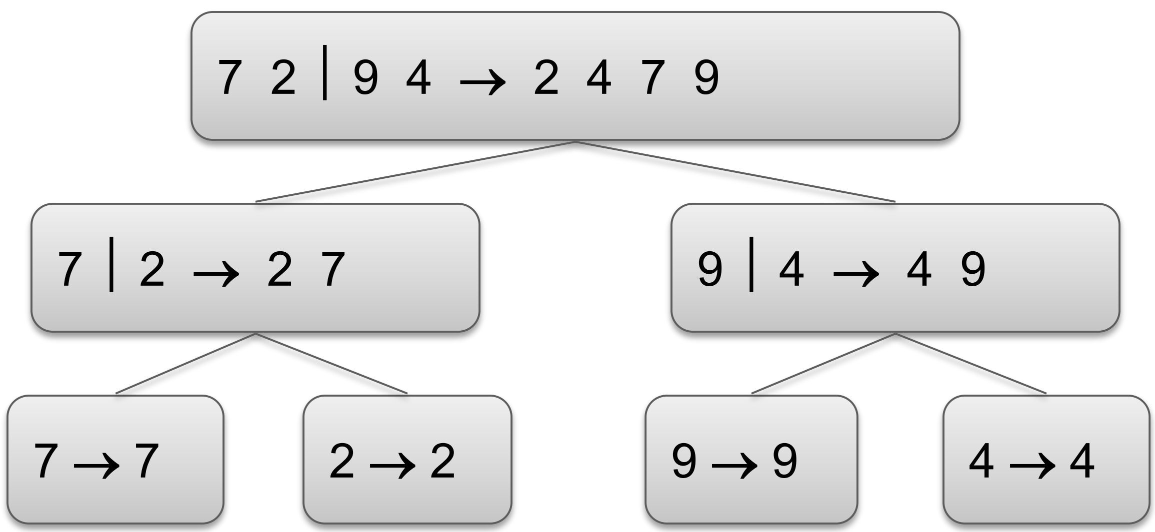
  Input: Two arrays, S_1 and S_2, of size n_1 and n_2, respectively, sorted in non-decreasing order, and an empty array, S, of size at least $n_1 + n_2$

  Output: S, containing the elements from S_1 and S_2 in sorted order

```
i ← 1
j ← 1
while i ≤ n and j ≤ n do
    if S1[i] ≤ S2[j] then
        S[i + j − 1] ← S1[i]
        i ← i + 1
    else
        S[i + j − 1] ← S2[j]
        j ← j + 1
while i ≤ n do
    S[i + j − 1] ← S1[i]
    i ← i + 1
while j ≤ n do
    S[i + j − 1] ← S2[j]
    j ← j + 1
```

Merge-Sort Tree

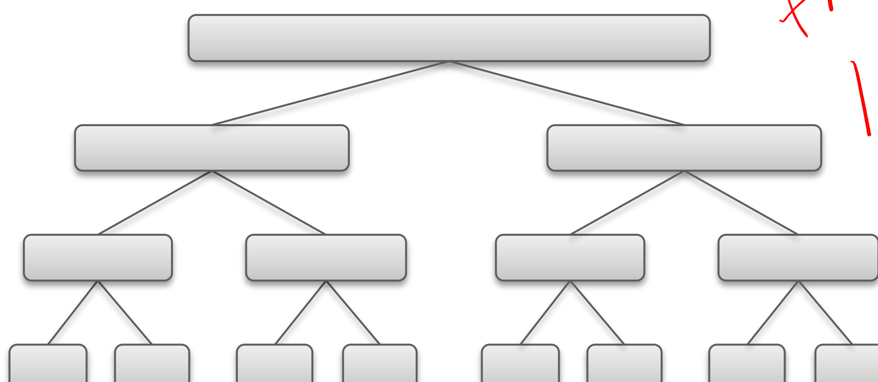
- An execution of merge-sort is depicted by a binary tree
  - each node represents a recursive call of merge-sort and stores
    - unsorted sequence before the execution and its partition
    - sorted sequence at the end of the execution
  - the root is the initial call
  - the leaves are calls on subsequences of size 0 or 1



Analysis of Merge-Sort

- The height h of the merge-sort tree is $O(\log n)$
  - at each recursive call we divide in half the sequence,
- The overall amount or work done at the nodes of depth i is $O(n)$
  - we partition and merge 2^i sequences of size $n/2^i$
  - we make 2^{i+1} recursive calls
- Thus, the best/worst/average running time of merge-sort is $O(n \log n)$

depth	#seqs	size
0	1	n
1	2	$n/2$
i	2^i	$n/2^i$
…	…	…



times log n

n
n
n
n

A Hybrid Sorting Algorithm

- A **hybrid** sorting algorithm is a blending of two different sorting algorithms, typically, a divide-and-conquer algorithm, like merge-sort, combined with an incremental algorithm, like insertion-sort.
- The algorithm is parameterized with hybridization value, ***H***, and an example with merge-sort and insertion-sort would work as follow:
  - Start out performing merge-sort, but switch to insertion sort when the problem size goes below H.

A Hybrid Sorting Algorithm

- Pseudo-code:

- Running time:
  - Depends on ***H***
  - Interesting experiments:
    1. $H = n^{1/2}$
    2. $H = n^{1/3}$
    3. $H = n^{1/4}$

Algorithm HybridMergeSort(S, H):
  Input array S of n elements
  Output array S sorted

  if $n > H$ **then**
    $(S_1, S_2) \leftarrow$ **partition**(S, $n/2$)
    HybridMergeSort(S_1, H)
    HybridMergeSort(S_2, H)
    $S \leftarrow$ **merge**(S_1, S_2)
  else
    InsertionSort(S)

Hybrid Merge-sort Analysis

- Hint: combine the tree-based merge-sort analysis and the insertion-sort analysis…