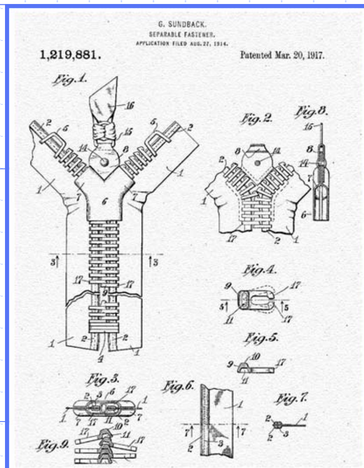
Presentation for use with the textbook, Algorithm Design and Applications, by M. T. Goodrich and R. Tamassia, Wiley, 2015

Merge Sort & Quick Sort



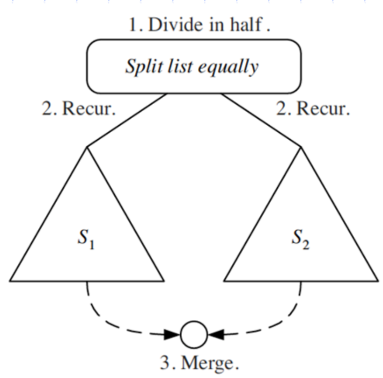
Separable Fastener, U.S. Patent 1,219,881, 1917.
Public domain image.

1

1

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
  - Conquer:
    - Recur: solve the subproblems associated with S_1 and S_2
    - Combine: make 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



2

2

Merge Sort

- Classical example of divide-and-conquer technique
- Problem: Given *n* elements, sort elements into non-decreasing order
- Divide-and-Conquer:
  - If n=1 terminate (every one-element list is already sorted)
  - If n>1, partition elements into two or more sub-collections; sort each; combine into a single sorted list
- How do we partition?

3

Partitioning - Choice 1

- First n-1 elements into set A, last element into set B
- Sort A using this partitioning scheme recursively
  - B already sorted
- Combine A and B using method Insert() (= insertion into sorted array)
- Leads to recursive version of InsertionSort()
  - Number of comparisons: $O(n^2)$
    - Best case = n-1
    - Worst case = $\boxed{c\sum_{i=2}^{n} i = \frac{n(n-1)}{2}}$

4

Partitioning - Choice 2

- Pick the element with largest key in B, remaining elements in A
- Sort A recursively
- To combine sorted A and B, append B to sorted A
  - Use Max() to find largest element → recursive SelectionSort()
  - Use bubbling process to find and move largest element to right-most position → recursive BubbleSort()
- All O(n^2)

5

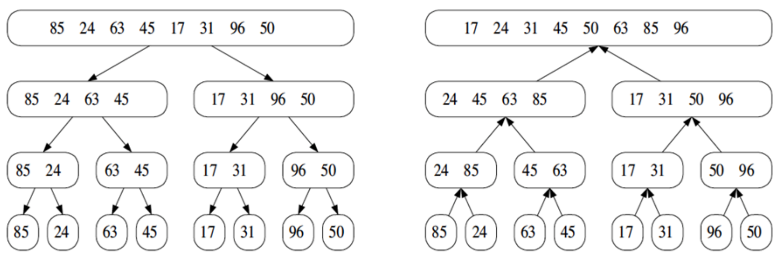
Partitioning - Choice 3

- Let's try to achieve balanced partitioning – that's typical for divide-and-conquer.
- A gets *n/2* elements, B gets the rest half
- Sort A and B recursively
- Combine sorted A and B using a process called *merge*, which combines two sorted lists into one
  - How? We will see soon

6

Merge-Sort

- Merge-sort is a sorting algorithm based on the divide-and-conquer paradigm
- Like heap-sort
  - It has $O(n \log n)$ running time
- Unlike heap-sort
  - It usually needs extra space in the merging process
  - It accesses data in a sequential manner (suitable to sort data on a disk)



7

7

Merging

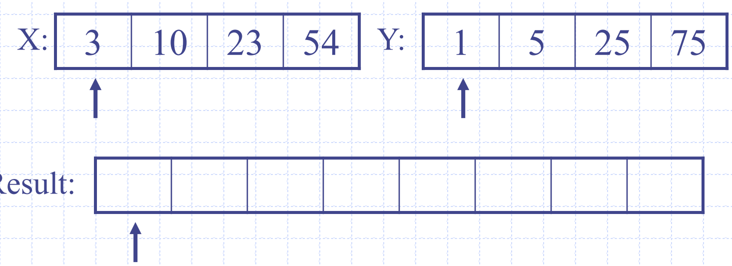
- The key to Merge Sort is merging two sorted lists into one, such that if you have two lists X = $(x_1 \le x_2 \le \cdots \le x_m)$ and Y= $(y_1 \le y_2 \le \cdots \le y_n)$, the resulting list is Z = $(z_1 \le z_2 \le \cdots \le z_{m+n})$
- Example:

  L_1 = { 3 8 9 } L_2 = { 1 5 7 }

  merge(L_1, L_2) = { 1 3 5 7 8 9 }

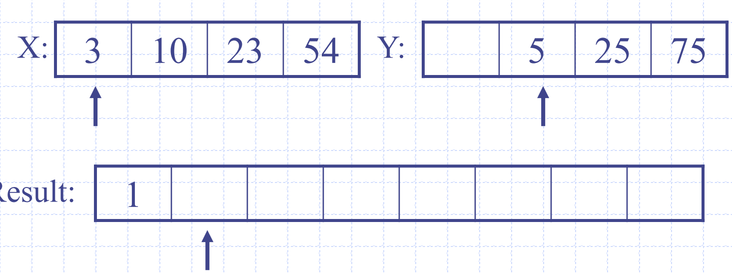
8

Merging



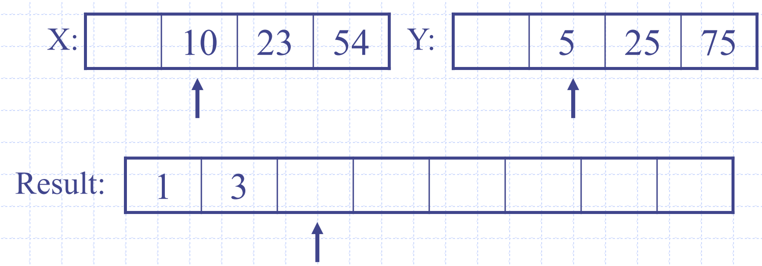
9

Merging (cont.)



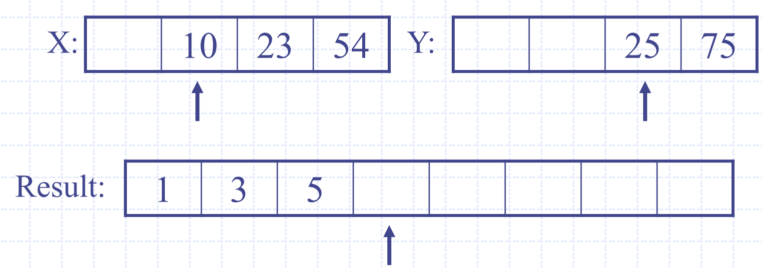
10

Merging (cont.)



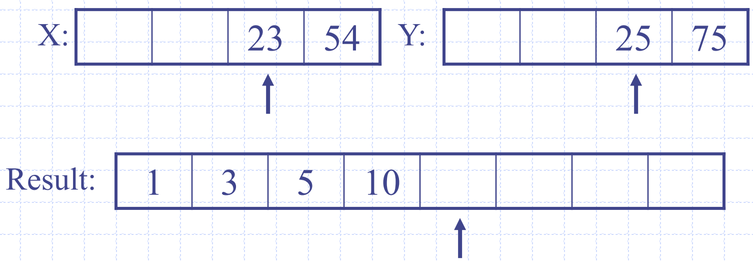
11

Merging (cont.)



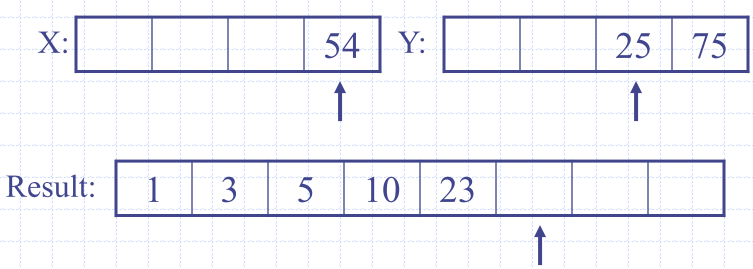
12

Merging (cont.)



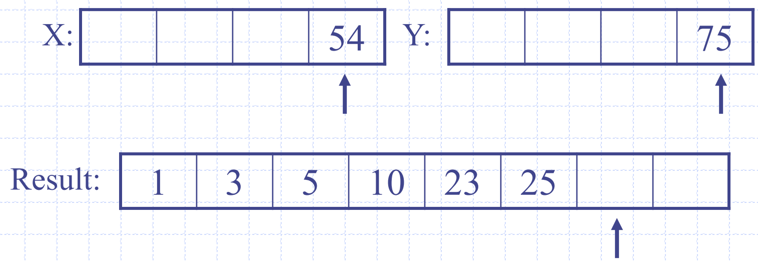
13

Merging (cont.)



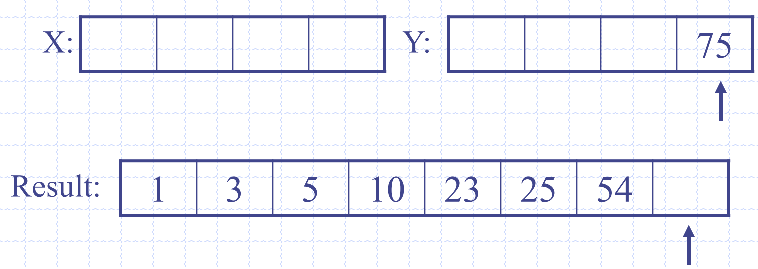
14

Merging (cont.)



15

Merging (cont.)



16

Merging (cont.)

X: | | | | | Y: | | | |

Result:								
	1	3	5	10	23	25	54	75

17

Merging Two Sorted Sequences

- The combine step of merge-sort has to merge 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*** elements, takes $O(2n)$ 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
```

18

18

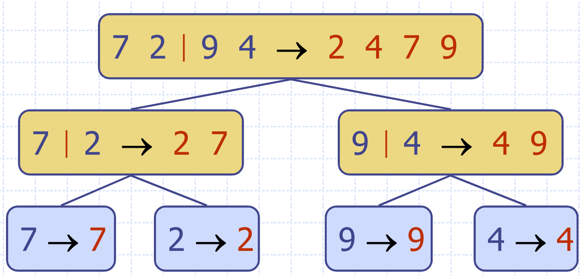
Implementing Merge Sort

- There are two basic ways to implement merge sort:
  - In Place: Merging is done with only the input array
    - Pro: Requires only the space needed to hold the array
    - Con: Takes longer to merge because if the next element is in the right side then all of the elements must be moved down.
  - Double Storage: Merging is done with a temporary array of the same size as the input array.
    - Pro: Faster than In Place since the temp array holds the resulting array until both left and right sides are merged into the temp array, then the temp array is appended over the input array.
    - Con: The memory requirement is doubled.

19

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

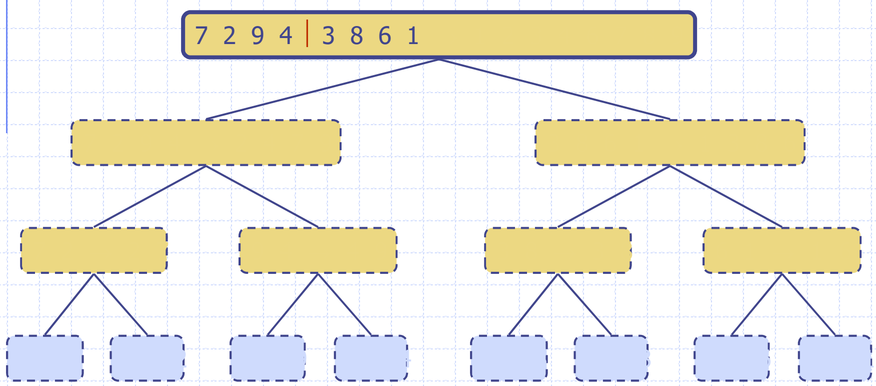


20

20

Execution Example

- Partition

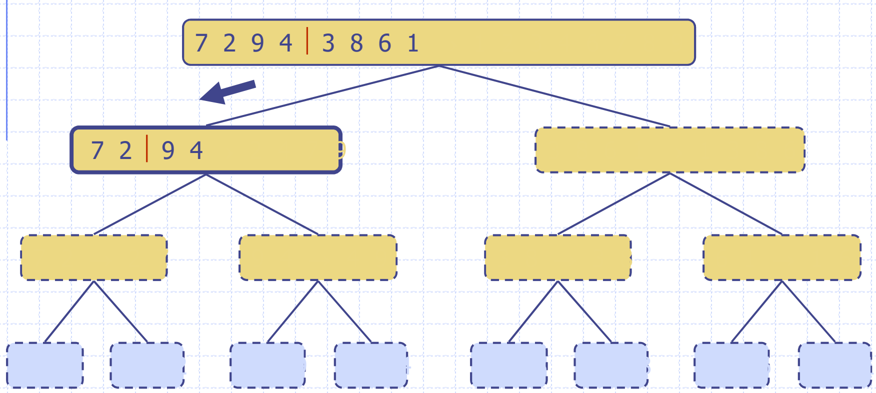


21

21

Execution Example (cont.)

- Recursive call, partition

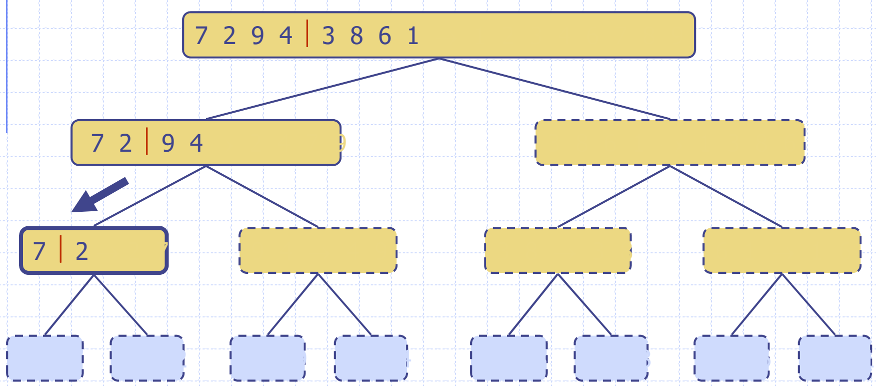


22

22

Execution Example (cont.)

- Recursive call, partition

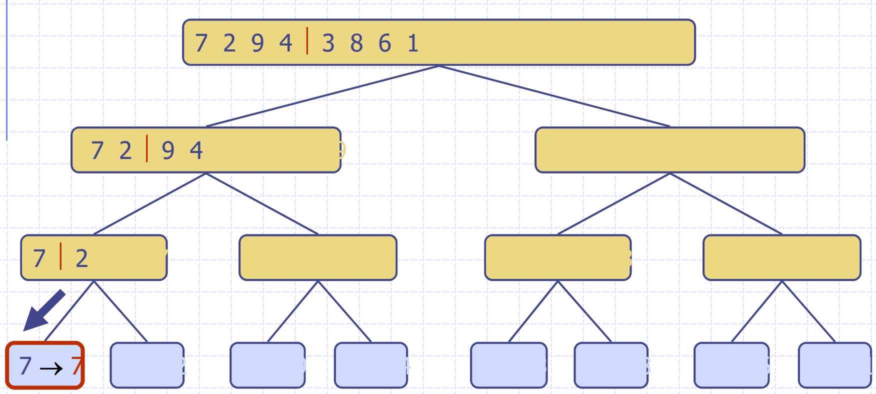


23

23

Execution Example (cont.)

- Recursive call, base case

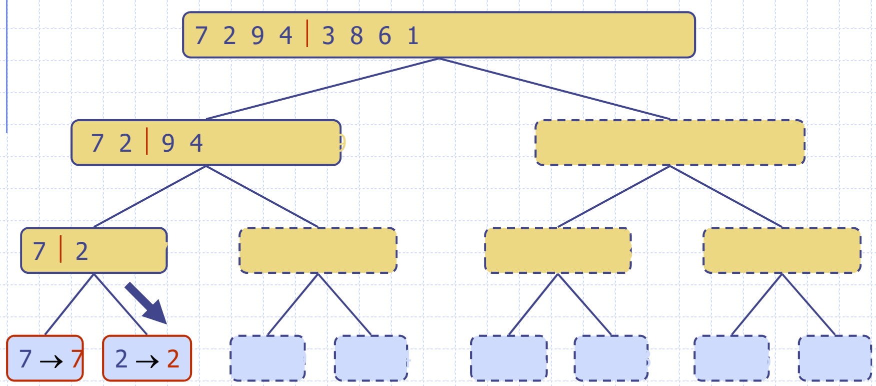


24

24

Execution Example (cont.)

- Recursive call, base case

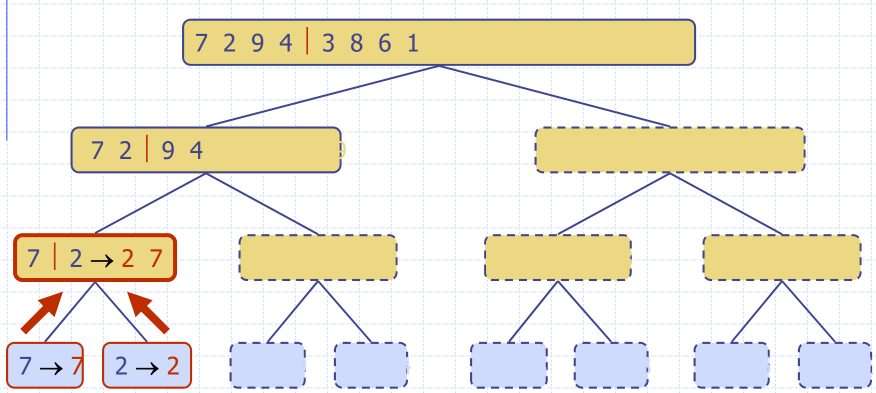


25

25

Execution Example (cont.)

- Merge

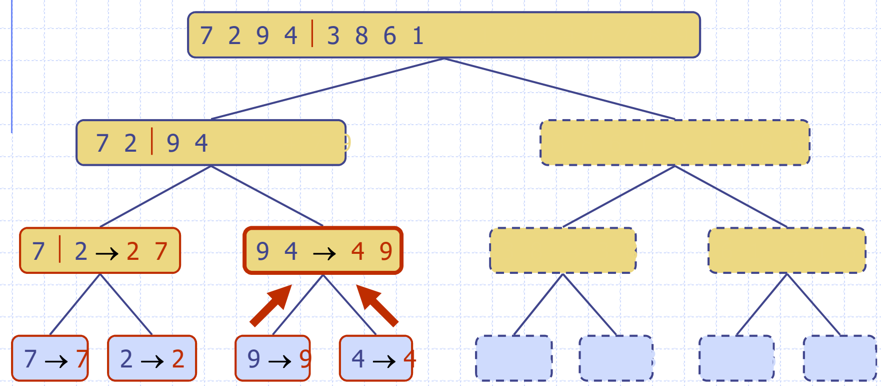


26

26

Execution Example (cont.)

- Recursive call, …, base case, merge

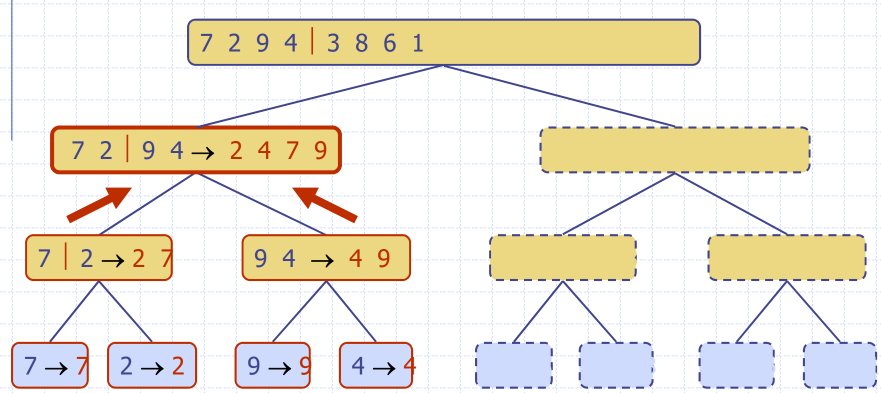


27

27

Execution Example (cont.)

- Merge

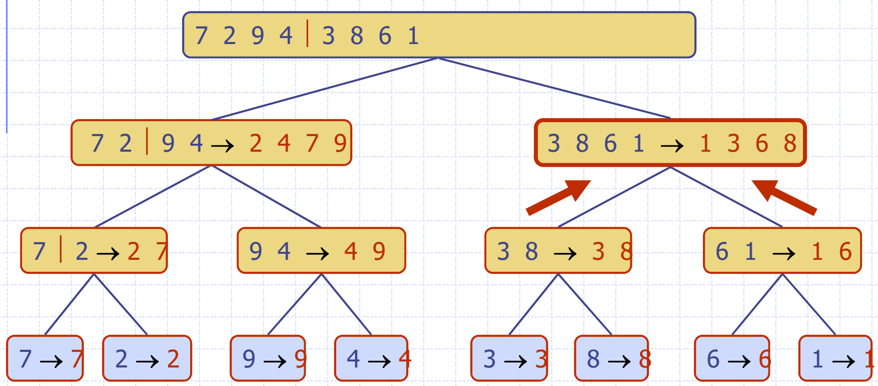


28

28

Execution Example (cont.)

- Recursive call, …, merge, merge

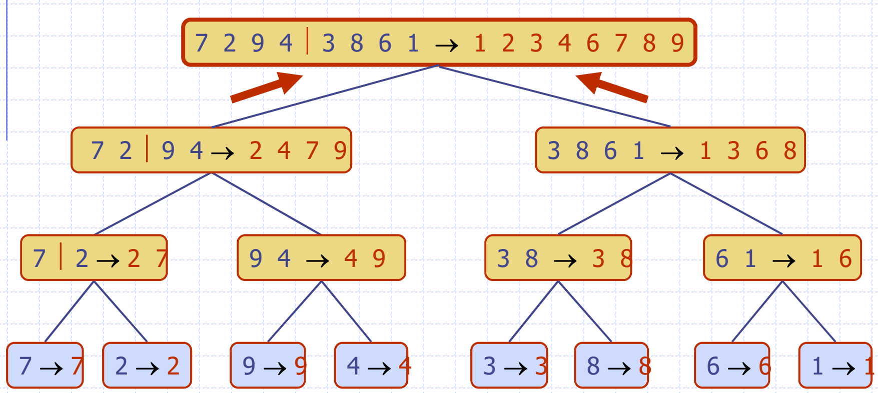


29

29

Execution Example (cont.)

- Merge



30

30

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
  - Combine: merge S_1 and S_2 into a unique sorted sequence

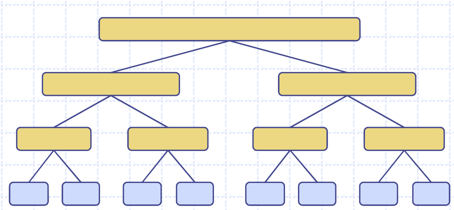
Algorithm ***mergeSort***(S)
  Input sequence S with n elements
  Output sequence S sorted according to C
  if $S.size() > 1$
    $(S_1, S_2) \leftarrow partition(S, n/2)$
    $mergeSort(S_1)$
    $mergeSort(S_2)$
    $S \leftarrow merge(S_1, S_2)$

31

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 total running time of merge-sort is $O(n \log n)$

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



32

Evaluation

- Recurrence equation:
- Assume n is a power of 2

$$T(n) = \begin{cases} c_1 & \text{if } n=1 \\ 2T(n/2) + c_2 n & \text{if } n>1,\ n=2^k \end{cases}$$

33

Solution

By Substitution:

$T(n) = 2T(n/2) + c_2n$
$T(n/2) = 2T(n/4) + c_2n/2$
…
$T(n) = 4T(n/4) + 2\ c_2n$
$T(n) = 8T(n/8) + 3\ c_2n$
…
$T(n) = 2^iT(n/2^i) + ic_2n$

Assuming $n = 2^k$, expansion halts when we get $T(1)$ on right side; this happens when $i=k$ $T(n) = 2^kT(1) + kc_2n$
Since $2^k=n$, we know $k=\log(n)$; since $T(1) = c_1$, we get
$T(n) = c_1n + c_2n\log n$;
thus an upper bound for $T_{mergeSort}(n)$ is $O(n\log n)$

34

Variants and Applications

- There are other variants of Merge Sorts including bottom-up merge sort, k-way merge sorting, natural merge sort,
- Bottom-up merge sort eliminates the divide process and assumes all subarrays have 2^k elements for some k, with exception of the last subarray.
- Natural merge sort is known to be the best for nearly sorted inputs. Sometimes, it takes only O(n) for some inputs.
- Merge sort's double memory demands makes it not very practical when main memory is in short supply.
- Merge sort is the major method for external sorting, parallel algorithms, and sorting circuits.

35

Natural Merge Sort

- Identify sorted (or reversely sorted) sub-lists in the input (each is called a run).
- Merge all runs into one.
- Example:
  - Input A = [10, 6, 2, 3, 5, 7, 3, 8]
  - Three (reversed) runs: [(10, 6, 2), (3, 5, 7), (3, 8)]
  - Reverse the reversed: [(2, 6, 10), (3, 5, 7), (3, 8)]
  - Merge them in one: [2, 3, 4, 5, 6, 7, 8, 10]
  - It takes O(n) to sort [n, n-1, …, 3, 2, 1].
- A good implementation of natural merge sort is called Timsort.

36

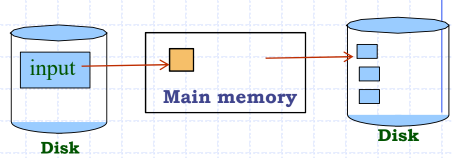
External Sorting

- Using secondary storage effectively
- General Wisdom :
  - I/O costs dominate
  - Design algorithms to reduce I/O

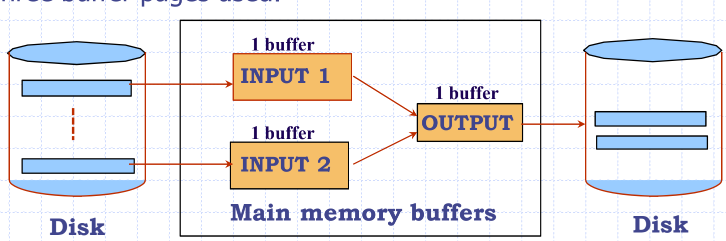
37

2-Way Sort: Requires 3 Buffers

- **Phase 1: PREPARE.**
  - Read a page, sort it, write it.
  - only one buffer page is used



- **Phase 2, 3, …, etc.: MERGE:**
  - Three buffer pages used.



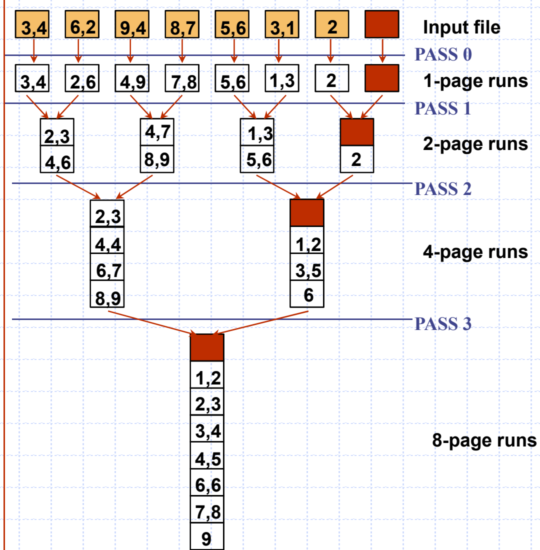
38

Two-Way External Merge Sort

- *Idea:* ***Divide and conquer:*** sort subfiles and merge into larger sorts
- Run: a list of sorted items

Pass 0 ➔ Only one memory block is needed

Pass i > 0 ➔ Only three memory blocks are needed

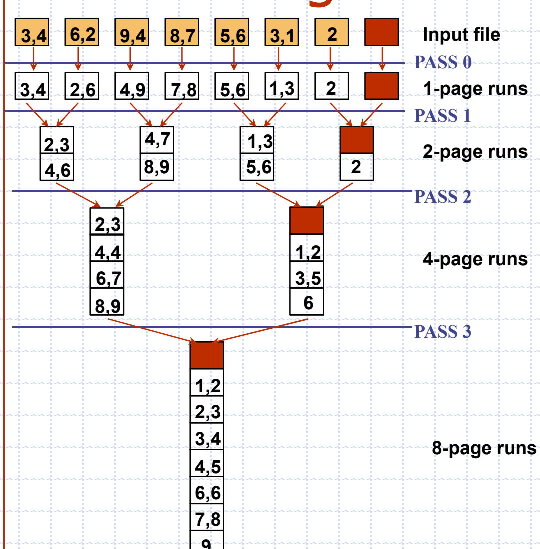


39

Two-Way External Merge Sort

- Costs for one pass: read/write all pages
- # of passes : height of tree
- Total cost : product of above

Notice: We ignored the CPU cost to sort a block in memory or merge two blocks



40

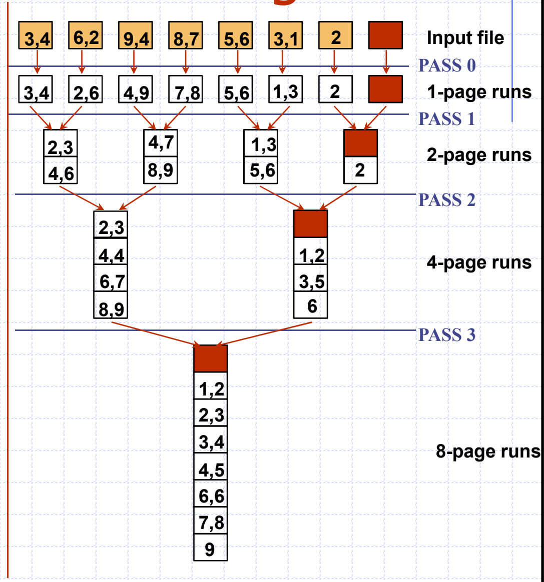
Two-Way External Merge Sort

- Each pass we read/write each page in file.
- N pages in file => 2N
- Number of passes

$$= \lceil \log_2 N \rceil + 1$$

- So total cost is:

$$2N\left(\lceil \log_2 N \rceil + 1\right)$$



41

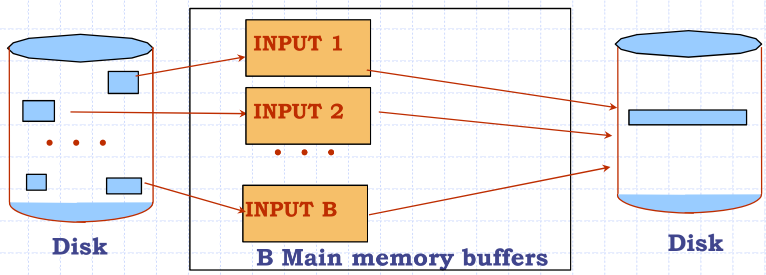
External Merge Sort

- What if we had more buffer pages?
- How do we utilize them wisely ?

-→ Two main ideas !

42

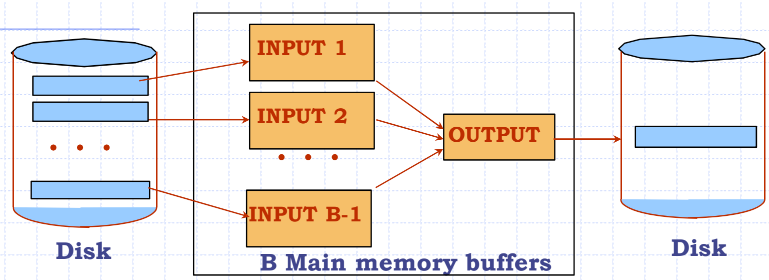
Phase 1 : Prepare



- Construct as large as possible starter lists.
- ➔ Will reduce the number of needed passes

43

Phase 2 : Merge



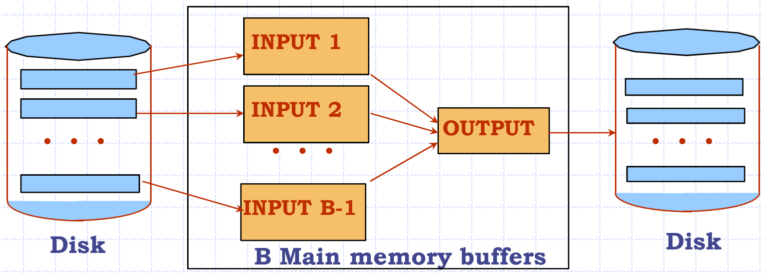
- Merge as many sorted sublists into one long sorted list.
- Keep 1 buffer for the output
- Use B-1 buffers to read from B-1 lists

44

General External Merge Sort

How can we utilize more than 3 buffer pages?

- To sort a file with *N* pages using *B* buffer pages:
  - Pass 0: use *B* buffer pages. Produce $\lceil N / B \rceil$ sorted runs of *B* pages each.
  - Pass 1, 2, …, etc.: merge *B-1* runs.



45

Cost of External Merge Sort

- Number of passes: $1 + \lceil \log_{B-1} \lceil N / B \rceil \rceil$
- Cost = 2N * (# of passes)

46

Example

- Buffer : with 5 buffer pages
- File to sort : 108 pages
  - Pass 0:
    - Size of each run?
    - Number of runs?
  - Pass 1:
    - Size of each run?
    - Number of runs?
  - Pass 2: ???

47

Example

- Buffer : with 5 buffer pages
- File to sort : 108 pages
  - Pass 0: $\lceil 108 / 5 \rceil$ = 22 sorted runs of 5 pages each (last run is only 3 pages)
  - Pass 1: $\lceil 22 / 4 \rceil$ = 6 sorted runs of 20 pages each (last run is only 8 pages)
  - Pass 2: 2 sorted runs, 80 pages and 28 pages
  - Pass 3: Sorted file of 108 pages

- Total I/O costs: ?

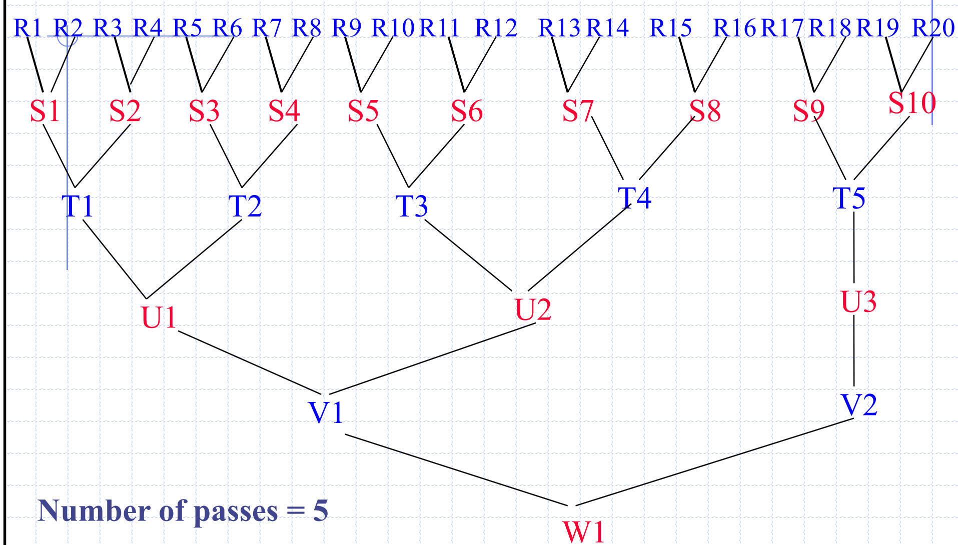
48

Summary of External Sorting

- External sorting is important;
- Choose best internal sorting in Pass 0.
- External merge sort minimizes disk I/O costs:
  - Two-Way External Sorting
    - Only 3 memory buffers are needed
  - Multi-Way External Sorting
    - Use large number of memory buffers available
  - Cost: $1 + \lceil \log_{B-1} \lceil N / B \rceil \rceil$

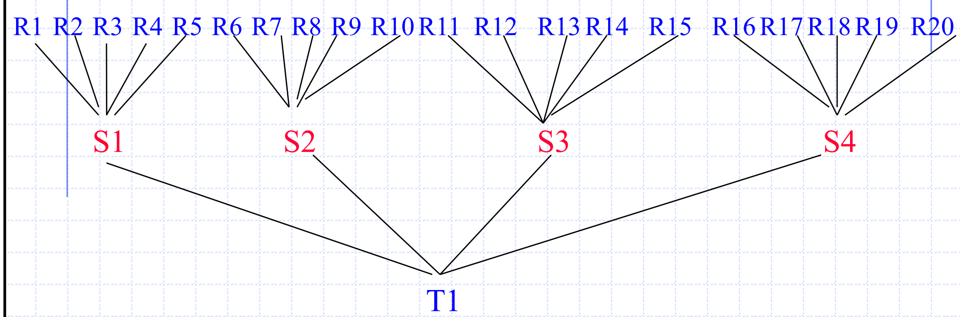
49

Example: 2-Way Merge for 20 Pages



50

Example: 5-Way Merge for 20 Pages



Number of passes = 2

51

Number of Passes of External Sort

- gain of utilizing all available buffers
- importance of a high fan-in during merging

#Buffers available in main-memory

#Pages in File

N	B=3	B=5	B=9	B=17	B=129	B=257
100	7	4	3	2	1	1
1,000	10	5	4	3	2	2
10,000	13	7	5	4	2	2
100,000	17	9	6	5	3	3
1,000,000	20	10	7	5	3	3
10,000,000	23	12	8	6	4	3
100,000,000	26	14	9	7	4	4
1,000,000,000	30	15	10	8	5	4

52

Summary of Sorting Algorithms

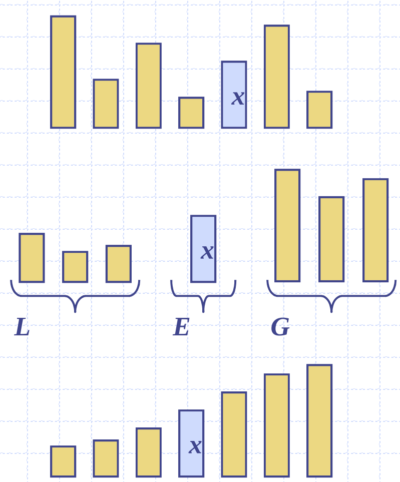
Algorithm	Time	Notes
selection-sort	$O(n^2)$	▪ stable ▪ in-place ▪ for small data sets (< 1K)
insertion-sort	$O(n^2)$	▪ stable ▪ in-place ▪ for small data sets (< 1K)
heap-sort	$O(n \log n)$	▪ non-stable ▪ in-place ▪ for large data sets (1K — 1M)
merge-sort	$O(n \log n)$	▪ stable ▪ sequential data access ▪ for huge data sets (> 1M)

53

53

Quick-Sort

- Quick-sort is also a sorting algorithm based on the divide-and-conquer paradigm:
  - Divide: pick a random element x (called pivot) and partition S into
    - L elements less than x
    - E elements equal x
    - G elements greater than x
  - Recur: sort L and G
  - Combine: join L, E and G



- Alternative: merge E into L or G.

54

54

Quicksort Algorithm

Given an array of *n* elements (e.g., integers):

- If array only contains one element, return
- Else
  - pick one element to use as *pivot.*
  - Partition elements into two sub-arrays:
    - Elements less than or equal to pivot
    - Elements greater than pivot
  - Quicksort two sub-arrays
  - Return results

55

Partitioning Array

Given a pivot, partition the elements of the array such that the resulting array consists of:

1. One sub-array that contains elements >= pivot
2. Another sub-array that contains elements < pivot

The sub-arrays are stored in the original data array.

Partitioning loops through, swapping elements below/above pivot.

56

Partition using Lists

- We partition an input sequence as follows:
  - We remove, in turn, each element y from S and
  - We insert y into L, E or G, depending on the result of the comparison with the pivot x
- Each insertion is at the end of a sequence, and hence takes $O(1)$ time.
- Thus, the partition step of quick-sort takes $O(n)$ time.
- Pro: stable sort
- Cons: not in place

Algorithm *partition*(S, p)
  Input sequence S, position p of pivot
  Output subsequences L, E, G of the elements of S less than, equal to, or greater than the pivot, resp.
  $L, E, G \leftarrow$ empty sequences
  $x \leftarrow S.remove(p)$
  while $\neg S.isEmpty()$
    $y \leftarrow S.remove(S.first())$
    if $y < x$
      $L.addLast(y)$
    else if $y = x$
      $E.addLast(y)$
    else $\{ y > x \}$
      $G.addLast(y)$
  return L, E, G

57

57

In-Place Partitioning Example

We are given array of n integers to sort:

40	20	10	80	60	50	7	30	100

58

Pick Pivot Element

- There are a number of ways to pick the pivot element. In this example, we will use the first element in the array:

40	20	10	80	60	50	7	30	100

59

pivot_index = 0									
	40	20	10	80	60	50	7	30	100
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
		too_big_index							too_small_index

60

1. While data[too_big_index] <= data[pivot]
   ++too_big_index

pivot_index = 0

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
40	20	10	80	60	50	7	30	100

too_big_index → [1]

too_small_index → [8]

61

1. While data[too_big_index] <= data[pivot]
   ++too_big_index

pivot_index = 0

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
40	20	10	80	60	50	7	30	100

too_big_index → [2]

too_small_index → [8]

62

1. While data[too_big_index] <= data[pivot]
   ++too_big_index

pivot_index = 0

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
40	20	10	80	60	50	7	30	100

too_big_index too_small_index

63

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index

pivot_index = 0

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
40	20	10	80	60	50	7	30	100

too_big_index too_small_index

64

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index

pivot_index = 0

40	20	10	80	60	50	7	30	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

too_big_index too_small_index

65

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]

pivot_index = 0

40	20	10	80	60	50	7	30	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

too_big_index too_small_index

66

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]

pivot_index = 0

40	20	10	30	60	50	7	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

too_big_index → [3]; too_small_index → [7]

67

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. If too_small_index > too_big_index, go to 1.

pivot_index = 0

40	20	10	30	60	50	7	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

too_big_index → [3]; too_small_index → [7]

68

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. If too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] and data[pivot_index]

pivot_index = 0

40	20	10	30	7	50	60	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

too_big_index too_small_index

Line 5 is optional

69

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. If too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] and data[pivot_index]

pivot_index = 4

7	20	10	30	40	50	60	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

too_big_index too_small_index

Line 5 is optional

70

Partition Result

7	20	10	30	40	50	60	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

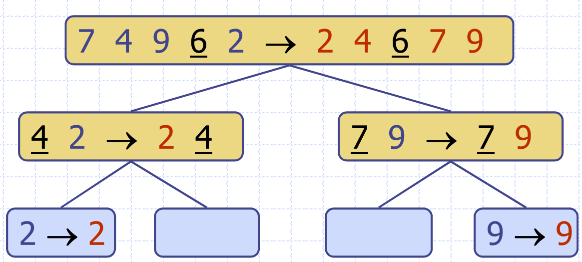
<= data[pivot] > data[pivot]

Recursive calls on two sides to get a sorted array.

71

Quick-Sort Tree

- An execution of quick-sort is depicted by a binary tree
  - Each node represents a recursive call of quick-sort and stores
    - Unsorted sequence before the execution and its pivot
    - 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

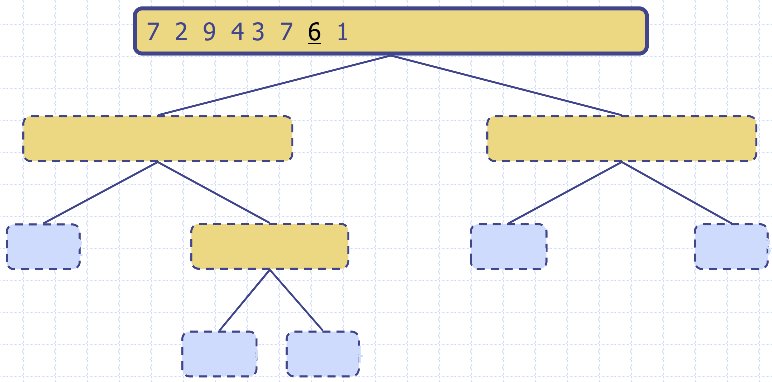


72

72

Execution Example

- Pivot selection

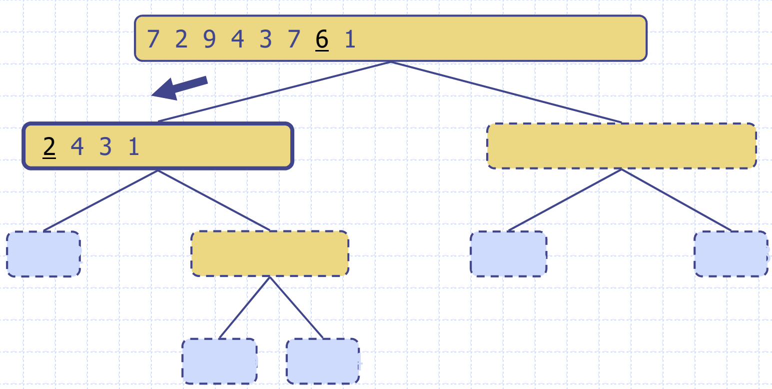


73

73

Execution Example (cont.)

- Partition, recursive call, pivot selection

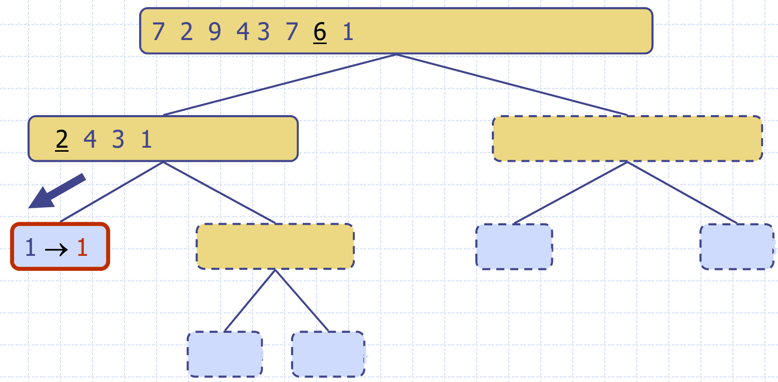


74

74

Execution Example (cont.)

- Partition, recursive call, base case

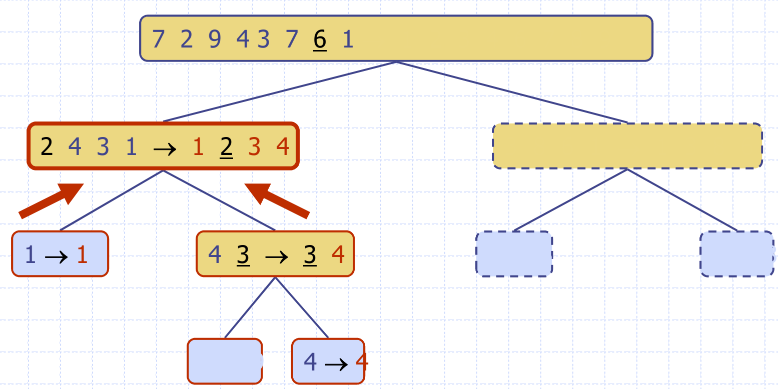


75

75

Execution Example (cont.)

- Recursive call, …, base case, join

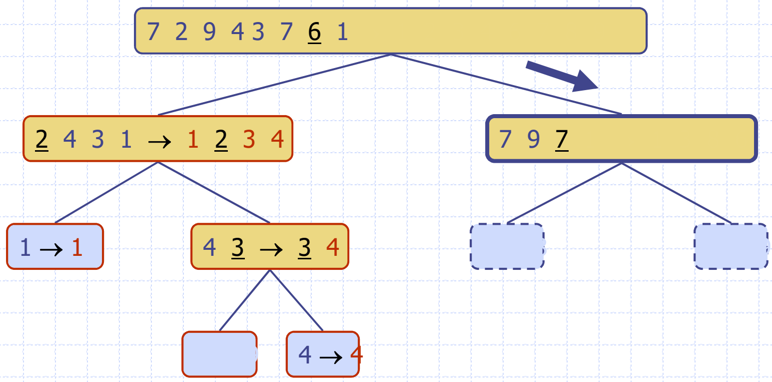


76

76

Execution Example (cont.)

- Recursive call, pivot selection

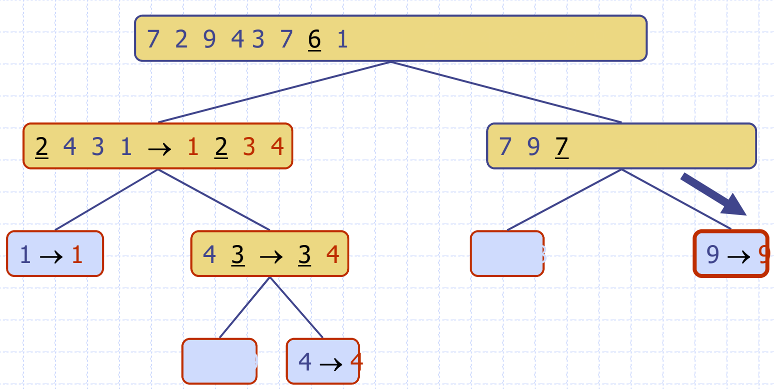


77

77

Execution Example (cont.)

- Partition, …, recursive call, base case

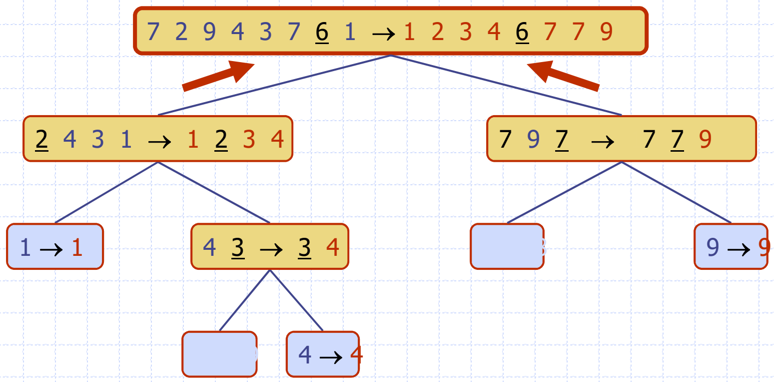


78

78

Execution Example (cont.)

- Join



79

79

Quicksort Analysis

- In-place quick-sort is not stable.
- Stable quick-sort is not in-place.
- Assume that keys are random, uniformly distributed.
- What is best case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays of size n/2
    2. Quicksort each sub-array
  - Depth of recursion tree?

80

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays of size n/2
    2. Quicksort each sub-array
  - Depth of recursion tree? $O(\log_2 n)$
  - Number of accesses in partition? $O(n)$

81

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?

82

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays:
       - one sub-array of size 0
       - the other sub-array of size n-1
    2. Quicksort each sub-array
  - Depth of recursion tree?

83

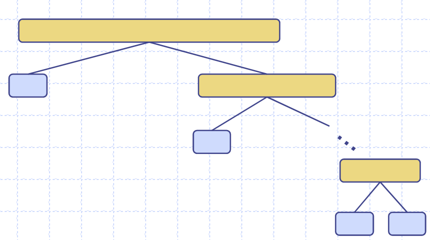
Worst-case Running Time

- The worst case for quick-sort occurs when the pivot is the unique minimum or maximum element
- One of L and G has size $n - 1$ and the other has size 0
- The running time is proportional to the sum

$$n + (n - 1) + \ldots + 2 + 1$$

- Thus, the worst-case running time of quick-sort is $O(n^2)$

depth	time
0	n
1	$n - 1$
…	…
$n - 1$	1



84

84

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays:
       - one sub-array of size 0
       - the other sub-array of size n-1
    2. Quicksort each sub-array
  - Depth of recursion tree? $O(n)$
  - Number of accesses per partition? $O(n)$

85

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time: $O(n^2)$!!!

86

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time: $O(n^2)$!!!
- What can we do to avoid worst case?
  - Randomly pick a pivot

87

Quicksort Analysis

- Bad divide: $T(n) = T(1) + T(n-1)$ -- $O(n^2)$
- Good divide: $T(n) = T(n/2) + T(n/2)$ -- $O(n \log_2 n)$
- Random divide: Suppose on average one bad divide followed by one good divide.
- $T(n) = T(1) + T(n-1) = T(1) + 2T((n-1)/2)$
- $T(n) = c + 2T((n-1)/2)$ is still $O(n \log_2 n)$

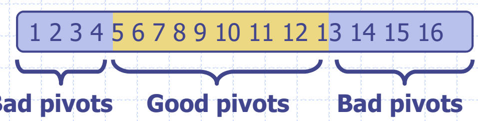
88

Expected Running Time

- Consider a recursive call of quick-sort on a sequence of size s
  - **Good call:** the sizes of L and G are each less than $3s/4$
  - **Bad call:** one of L and G has size greater than $3s/4$

Good call **Bad call**

- A call is good with probability $1/2$
  - 1/2 of the possible pivots cause good calls:

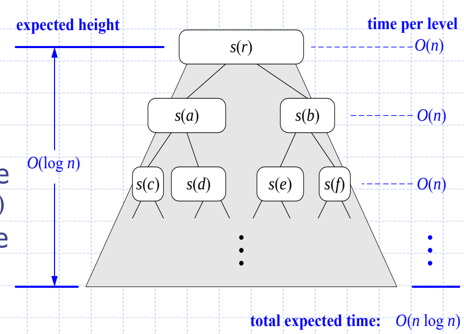


89

89

Expected Running Time, Part 2

- Probabilistic Fact: The expected number of coin tosses required in order to get k heads is $2k$
- For a node of depth i, we expect
  - $i/2$ ancestors are good calls
  - The size of the input sequence for the current call is at most $(3/4)^{i/2}n$
- Therefore, we have
  - For a node of depth $2\log_{4/3}n$, the expected input size is one
  - The expected height of the quick-sort tree is $O(\log n)$
- The amount or work done at the nodes of the same depth is $O(n)$
- Thus, the expected running time of quick-sort is $O(n \log n)$



90

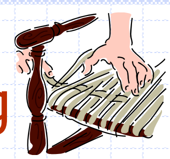
90

Randomized Guarantees

- Randomization is a very important and useful idea. By either picking a random pivot or scrambling the permutation before sorting it, we can say:
  - "With high probability, randomized quicksort runs in O(n log n) time."
- Randomization is a general tool to improve algorithms with bad worst-case but good average-case complexity.
- The worst-case is still there, but we almost certainly won't see it.

91

In-Place 3-Way Partitioning



- Dijkstra's 3-way partititioning (Holland flag problem)

<p | =p | ?.....................? | >p

p=pivot

<p | =p | >p

- Bentley-McIlroy's 3-way partitioning

=p | <p | ?.....................? | >p | =p

p=pivot

<p | =p | >p

92

92

In-Place 3-Way Randomized Quick-Sort

- Quick-sort can be implemented to run in-place
- In the partition step, we use replace operations to rearrange the elements of the input sequence such that
  - the elements less than the pivot have rank less than h
  - the elements equal to the pivot have rank between h and k
  - the elements greater than the pivot have rank greater than k
- The recursive calls consider
  - elements with rank less than h
  - elements with rank greater than k

Algorithm *inPlaceQuickSort*(S, l, r)
 Input sequence S, ranks l and r
 Output sequence S with the elements of rank between l and r rearranged in increasing order
 if $l \geq r$
  return
 $i \leftarrow$ a random integer between l and r
 $p \leftarrow S.elemAtRank(i)$
 $(h, k) \leftarrow inPlace3WayPartition(p)$
 $inPlaceQuickSort(S, l, h - 1)$
 $inPlaceQuickSort(S, k + 1, r)$

93

93

Improved Pivot Selection

Pick median value of three elements from data array:
 data[0], data[n/2], and data[n-1].

Use this median value as pivot.

For large arrays, use the median of three medians from
 {data[0], data[n/8], data[2n/8]}, {data[3n/8], data[4n/8], data[5n/8]}, and {data[6n/8], data[7n/8], data[n-1]}.

94

Improving Performance of Quicksort

- Improved selection of pivot.
- For sub-arrays of size 100 or less, apply brute force search, or insert-sort.
  - Sub-array of size 1: trivial
  - Sub-array of size 2:
    - if(data[first] > data[second]) swap them
  - Sub-array of size 100 or less: call insert-sort.

95

Improving Performance of Quicksort

- Improved selection of pivot.
- For sub-arrays of size 100 or less, apply brute force search, e.g., insert-sort.
- Test if the sub-array is already sorted before recursive calls.

96

Stability of sorting algorithms

- A STABLE sort preserves relative order of records with equal keys

Sorted on first key:

Aaron	4	A	664-480-0023	097 Little
Andrews	3	A	874-088-1212	121 Whitman
Battle	4	C	991-878-4944	308 Blair
Chen	2	A	884-232-5341	11 Dickinson
Fox	1	A	243-456-9091	101 Brown
Furia	3	A	766-093-9873	22 Brown
Gazsi	4	B	665-303-0266	113 Walker
Kanaga	3	B	898-122-9643	343 Forbes
Rohde	3	A	232-343-5555	115 Holder
Quilici	1	C	343-987-5642	32 McCosh

Sort the first file on second key:

Records with key value 3 are not in order on first key!!

Fox	1	A	243-456-9091	101 Brown
Quilici	1	C	343-987-5642	32 McCosh
Chen	2	A	884-232-5341	11 Dickinson
Kanaga	3	B	898-122-9643	343 Forbes
Andrews	3	A	874-088-1212	121 Whitman
Furia	3	A	766-093-9873	22 Brown
Rohde	3	A	232-343-5555	115 Holder
Battle	4	C	991-878-4944	308 Blair
Gazsi	4	B	665-303-0266	113 Walker
Aaron	4	A	664-480-0023	097 Little

97

Summary of Sorting Algorithms

Algorithm	Time	Notes
selection-sort	$O(n^2)$	▪ in-place, stable ▪ slow (not good for any inputs)
insertion-sort	$O(n^2)$	▪ in-place, stable ▪ slow (good for small inputs)
quick-sort	$O(n \log n)$ expected	▪ in-place, not stable ▪ fastest (good for large inputs)
heap-sort	$O(n \log n)$	▪ in-place, not stable ▪ fast (good for large inputs)
merge-sort	$O(n \log n)$	▪ not in-place, stable ▪ fast (good for huge inputs)

98

98

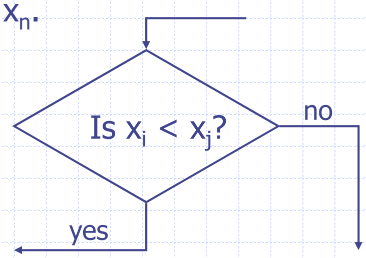
Divide and Conquer

	Simple Divide	Fancy Divide
Uneven Divide 1 vs n-1	Insert Sort	Selection Sort
Even Divide n/2 vs n/2	Merge Sort	Quick Sort

99

Comparison-Based Sorting

- Many sorting algorithms are comparison based.
  - They sort by making comparisons between pairs of objects
  - Examples: bubble-sort, selection-sort, insertion-sort, heap-sort, merge-sort, quick-sort, ...
- Let us therefore derive a lower bound on the running time of any algorithm that uses comparisons to sort n elements, $x_1, x_2, \ldots, x_n$.



100

100

How Fast Can We Sort?

- Selection Sort, Bubble Sort, Insertion Sort: $O(n^2)$
- Heap Sort, Merge sort: $O(nlogn)$
- Quicksort: $O(nlogn)$ - *average*
- What is common to all these algorithms?
  - Make **comparisons** between input elements

$a_i < a_j$, $a_i \le a_j$, $a_i = a_j$, $a_i \ge a_j$, or $a_i > a_j$

101

101

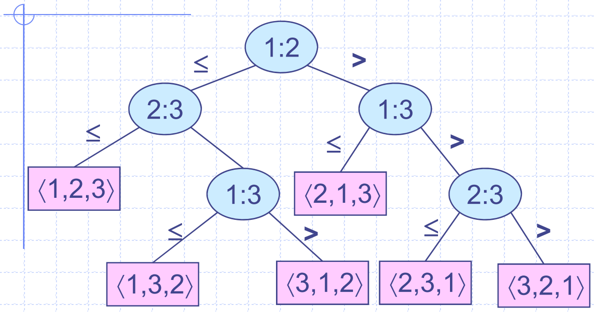
Comparison-based Sorting

- **Comparison sort**
  - Only comparison of pairs of elements may be used to gain order information about a sequence.
  - Hence, a lower bound on the number of comparisons will be a lower bound on the complexity of any comparison-based sorting algorithm.
- All our sorts have been comparison sorts
- The best worst-case complexity so far is $\Theta(n \lg n)$ (e.g., heapsort, merge sort).
- We prove a lower bound of $\Omega(n \lg n)$ for any comparison sort: merge sort and heapsort are optimal.
- The idea is simple: there are $n!$ outcomes, so we need a tree with $n!$ leaves, and therefore $\log(n!)$ = n log n.

102

Decision Tree

For insertion sort operating on three elements.



Simply unroll all loops for all possible inputs.

Node *i*:*j* means compare A[*i*] to A[*j*].

Leaves show outputs;

No two paths go to same leaf!

Contains 3! = 6 leaves.

103

Decision Tree (Cont.)

- Execution of sorting algorithm corresponds to tracing a path from root to leaf.
- The tree models all possible execution traces.
- At each internal node, a comparison $a_i \le a_j$ is made.
  - If $a_i \le a_j$, follow left subtree, else follow right subtree.
  - View the tree as if the algorithm splits in two at each node, based on information it has determined up to that point.
- When we come to a leaf, ordering $a_{\pi(1)} \le a_{\pi(2)} \le \ldots \le a_{\pi(n)}$ is established.
- A correct sorting algorithm must be able to produce any permutation of its input.
  - Hence, each of the $n!$ permutations must appear at one or more of the leaves of the decision tree.

104

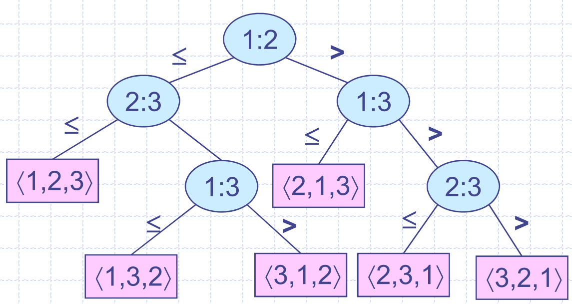
A Lower Bound for Worst Case

- Worst case no. of comparisons for a sorting algorithm is
  - Length of the longest path from root to any of the leaves in the decision tree for the algorithm.
    - Which is the height of its decision tree.
- A lower bound on the running time of any comparison sort is given by
  - A lower bound on the heights of all decision trees in which each permutation appears as a reachable leaf.

105

Optimal sorting for three elements

Any sort of six elements has 5 internal nodes.



There must be a worst-case path of length ≥ 3.

106

A Lower Bound for Worst Case

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

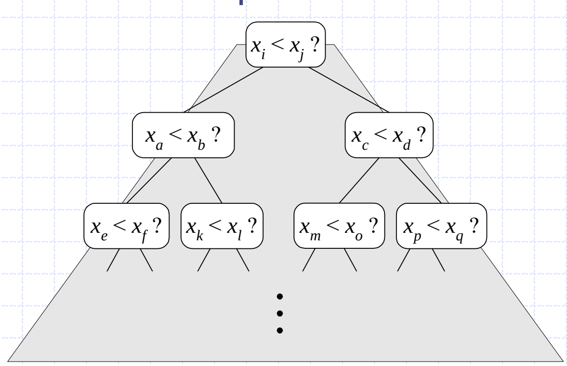
Proof:

- Suffices to determine the height of a decision tree.
- The number of leaves is at least $n!$ (# outputs)
- The number of internal nodes $\geq n! - 1$
- The height is at least $\log(n!-1) = \Omega(n \lg n)$

107

Counting Comparisons

- Let us just count comparisons then.
- Each possible run of the algorithm corresponds to a root-to-leaf path in a decision tree

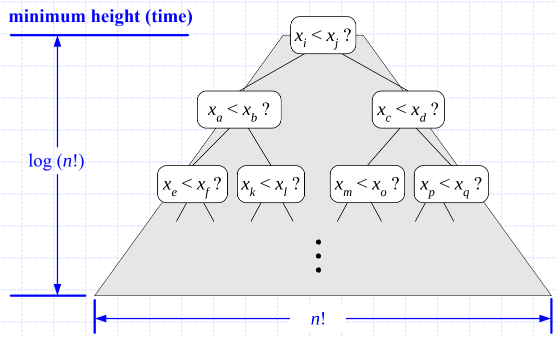


108

108

Decision Tree Height

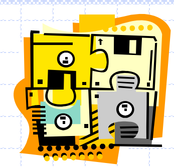
- The height of the decision tree is a lower bound on the running time
- Every input permutation must lead to a separate leaf output
- If not, some input …4…5… would have same output ordering as …5…4…, which would be wrong
- Since there are n!=1·2 · … ·n leaves, the height is at least log (n!)



109

109

The Lower Bound



- Any comparison-based sorting algorithms takes at least log(n!) time
- Therefore, any such algorithm takes worst-case time at least

$$\log (n!) \geq \log \left(\frac{n}{2}\right)^{\frac{n}{2}} = (n/2)\log(n/2).$$

- That is, any comparison-based sorting algorithm must run in Ω(n log n) time in the worst case.

110

110