CSE221

Lecture 23:
Sorting

Hyungon Moon

Acknowledgment: The content of this file is based on the slides of the textbook as well as the slides provided by Prof. Won-Ki Jeong.

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

Outline

- Motivation
- Internal sort
  - Insertion sort
  - Quick sort
  - Merge sort
  - Radix sort

Outline

- Motivation
- Internal sort
  - Insertion sort
  - Quick sort
  - Merge sort
  - Radix sort

Definitions

- List
  - Collection of records
  - Each record has fields
- Key
  - Field used to distinguish among the records
- Example
  - Telephone dictionary (list)
  - Name, address, phone number (fields)
    - Name is often used as a key

Bundy E H	FraserIsland	724-5944
Burgess Frank W	382ElmS	724-2324
Burns George	Powassan	724-5076
Burns Paul	RR2	724-5170
Bursey William T	RR2	724-5598
Busch Barney	Nipissing	724-5970
Busch Bill	RR3	724-5184
Busch C H	356 ClarkeW	724-3220
Busch Charles	Nipissing	724-5245
Busch Mrs Debra	326 Elm	724-2830
Busch Eddie	RR3	724-3018
Busch George	RR2	724-5580
Busch Joe	Nipissing	724-5242
Busch John	RR3	724-5247
Busch R	RR3	724-3226
Busch R B	288 MainW	724-2836
Busch Stephen	316 EdwardS	724-3024
Busch Wm C	RR3	724-2033
Busuttil A	RR1	724-5035
BUTLER GENERAL STORE	RR4	724-2138
Butler George	MainW	724-2727
Butler Robert	206 ClarkeW	724-2946
Butler William	Powassan	724-2614
Byers Clanton	RR2	724-5614

Search Records using a Key

- Sequential search in a[1:n]
  - Search a[i] : i comparisons
  - Search unsuccessful : n comparisons
  - Average comparison : $(\sum_{1 \le i \le n} i) / n = (n+1)/2$
  - O(n)

- Searching in an <u>ordered</u> list
  - Binary search : O(log n)

we need sorting!

Sorting Problem

- For a given list of records $(R_1, R_2,...,R_n)$
  - Each record R_i has a key value K_i
  - Transitive ordering relation <
    - x<y & y<z then x<z
- Find a permutation σ such that $K_{\sigma(i)} \leq K_{\sigma(i+1)}$ for $1 \leq i \leq n-1$
  - σ is not unique (identical keys may exist)
  - σ_s is *stable* if sorting preserves the order of input list for identical keys

Example of Stable Sorting

- Input records index : (1,2,3,4)
- Corresponding key values : (30, 20, 20, 10)
- Sorted records index can be (4,2,3,1) or (4,3,2,1) because 2 & 3 key values are same
- Stable sort : (4,2,3,1) → 2 & 3 order is not changed

Classification of Sorting Methods

- Internal methods
  - List is small enough to fit in main memory
  - Insertion sort, quick sort, merge sort, heap sort, radix sort
- External methods
  - Larger lists that do not fit in main memory
  - Read / write blocks of records from a disk
  - Out-of-core

Outline

- Motivation
- Internal sort
  - Insertion sort
  - Quick sort
  - Merge sort
  - Radix sort

Insertion Sort

- Assume a[1:i] is ordered
- Insert a new record into the list to get a new ordered list a[1:i+1]

```
void Insert(e, *a, i)
{
    a[0]=e;
    while(e<a[i])
    {
        a[i+1]=a[i];
        i--;
    }
    a[i+1]=e;
}
```

```
void InsertionSort(*a, n)
{
    for(j=2;j<=n;j++)
    {
        temp = a[j];
        Insert(temp, a, j-1);
    }
}
```

insert e to sorted list a[1:i] to make a[1:i+1]

Insertion Sort

- Example
  - Red: sorted

j	[1]	[2]	[3]	[4]	[5]
-	**5**	4	3	2	1
2	**4**	**5**	3	2	1
3	**3**	**4**	**5**	2	1
4	**2**	**3**	**4**	**5**	1
5	**1**	**2**	**3**	**4**	**5**

Insertion Sort

- Analysis
  - Worst case: i+1 comparison for Insert(e,a,i)
    - $O(\sum_{i=1}^{n-1}(i+1)) = O(n^2)$
  - Best case : O(n)
  - For k LOO (left out of order), O(kn)
  - Average case : $O(n^2)$
  - Additional space requirement: O(1)
    - In-place swapping a pair of records
  - Stable

Insertion Sort

- Efficient for small LOO
  - k << n
- Fastest sorting method for small n (n ≤ 30)
- Variation
  - Binary insertion sort
    - Binary search to find where to insert
    - # of records shifting is same, reduce # of comparisons
  - Linked insertion sort
    - Linked list, no record shifting

Outline

- Motivation
- Internal sort
  - Insertion sort
  - Quick sort
  - Merge sort
  - Radix sort

Quick Sort

- Partition the list into three sublists using pivot
  - Left, middle, right
  - Middle = pivot
  - Left <= pivot
  - Right >= pivot
- Recursively sort left and right sublists

Quick Sort

```
void QuickSort(*a, left, right)
{
    if(left<right) {
        int i = left, j=right+1, pivot=a[left];
        do {
            do i++; while(a[i]<pivot); // move i to right
            do j--; while(a[j]>pivot); // move j to left
            if(i<j) swap(a[i],a[j]);
        } while(i<j)
        swap(a[left],a[j]);
        QuickSort(a,left,j-1);
        QuickSort(a,j+1,right);
    }
}
```

Quick Sort

```
do {
      do i++; while(a[i]<pivot);
      do j--; while(a[j]>pivot);
      if(i<j) swap(a[i],a[j]);
} while(i<j)
swap(a[left],a[j]);
QuickSort(a,left,j-1);
QuickSort(a,j+1,right);
```

i j

26 5 37 1 61 11 59 15 48 19

26 5 37 1 61 11 59 15 48 19 — i++, swap

26 5 19 1 61 11 59 15 48 37 — i++, j--, swap

26 5 19 1 15 11 59 61 48 37 — i++, j--, i>j, swap(a[left],a[j])

11 5 19 1 15 | 26 | 59 61 48 37

Quick Sort

R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}	*left*	*right*
[26	5	37	1	61	11	59	15	48	19]	1	10
[11	5	19	1	15]	26	[59	61	48	37]	1	5
[1	5]	11	[19	15]	26	[59	61	48	37]	1	2
1	5	11	[19	15]	26	[59	61	48	37]	4	5
1	5	11	15	19	26	[59	61	48	37]	7	10
1	5	11	15	19	26	[48	37]	59	[61]	7	8
1	5	11	15	19	26	37	48	59	[61]	10	10
1	5	11	15	19	26	37	48	59	61		

Quick Sort

- Analysis
  - O(n) to partition a list with n records
  - Worst : $O(n^2)$
    - Either left or right sublist is empty
  - Best : O(n log n)
    - Left and right sublists are roughly same size
  - Average : O(n log n)
  - Additional space requirement : O(n) for stack
    - O(log n) for optimized implementation
  - Not stable

Quick Sort

- Pivot selection is important to make balanced sublists
- How to select pivot?
  - Left or right
  - Random
  - Median of three (left, middle, right)

Outline

- Motivation
- Internal sort
  - Insertion sort
  - Quick sort
  - Merge sort
  - Radix sort

Merge Ordered Lists

- Merging two ordered lists [l:m] and [m+1:n]
  - O($n - l$ + 1)
  - $n - l$ + 1 : # of records being merged

```
for(neither input list is exhausted)
{
   Copy smaller record among i1 and i2 to iResult
}

Copy remaining records from either i1 or i2 to iResult
```

Merge Ordered Lists

```
Merge() // initList[l:m][m+1:n] are sorted lists, out: mergedList
{
    for(int i1=l, iResult=l,i2=m+1; i1<=m && i2<=n; iResult++)
    {   // neither input is exhausted
        //Copy smaller record among i1 and i2 to iResult
        if(initList[i1] <= initList[i2])
        {
            mergedList[iResult] = initList[i1];
            i1++;
        else
        {
            mergedList[iResult] = initList[i2];
            i2++;
        }
    }
    // copy remaining records from either i1 or i2
    copy(initList + i1, initList + m + 1, mergedList + iResult);
    copy(initList + i2, initList + n + 1, mergedList + iResult);
}
```

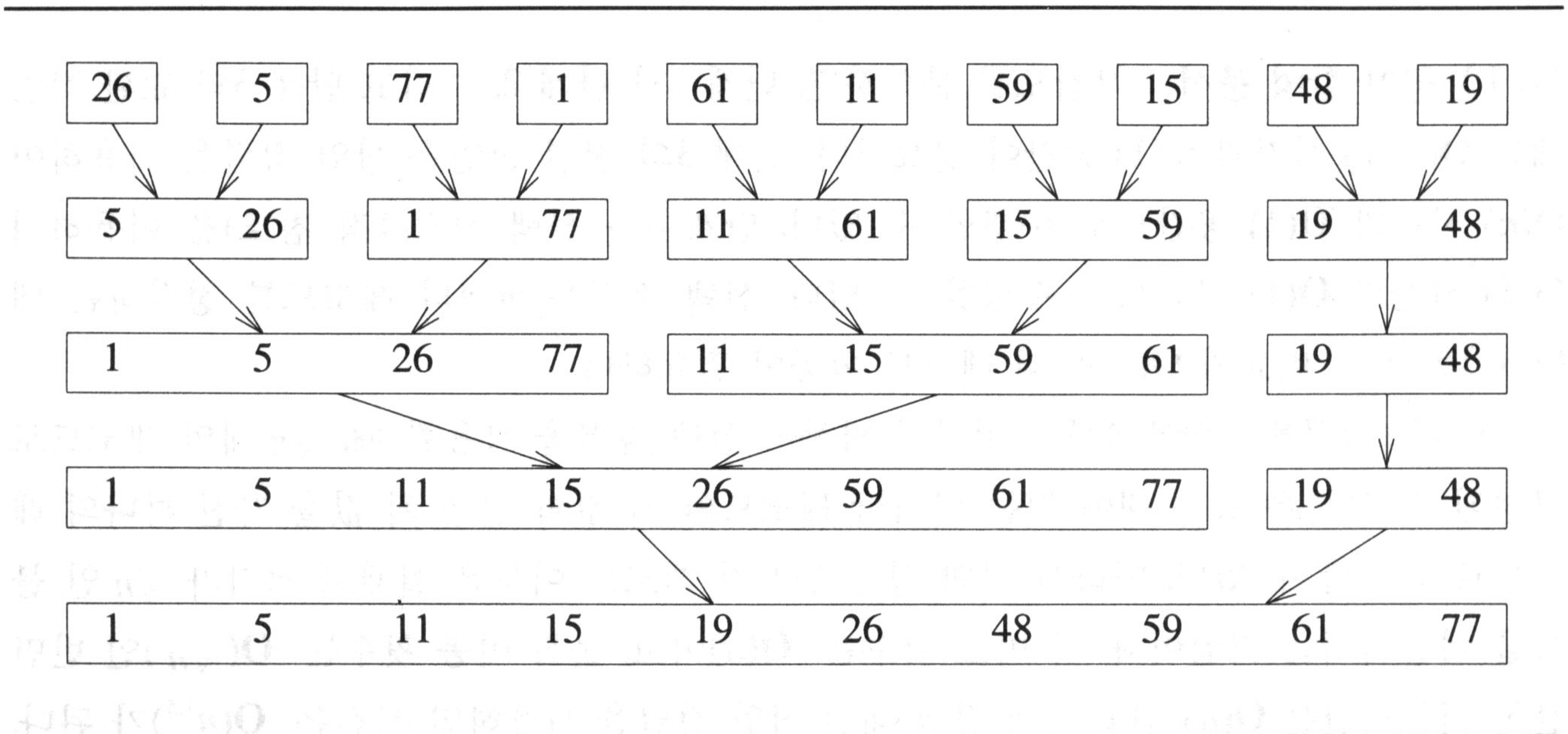
O(?)

Iterative Merge Sort

- Start from sublists of size 1
- Merge every pair
- Double the sublist size and merge
- Repeat until all sublists are merged

Iterative Merge Sort

- Example



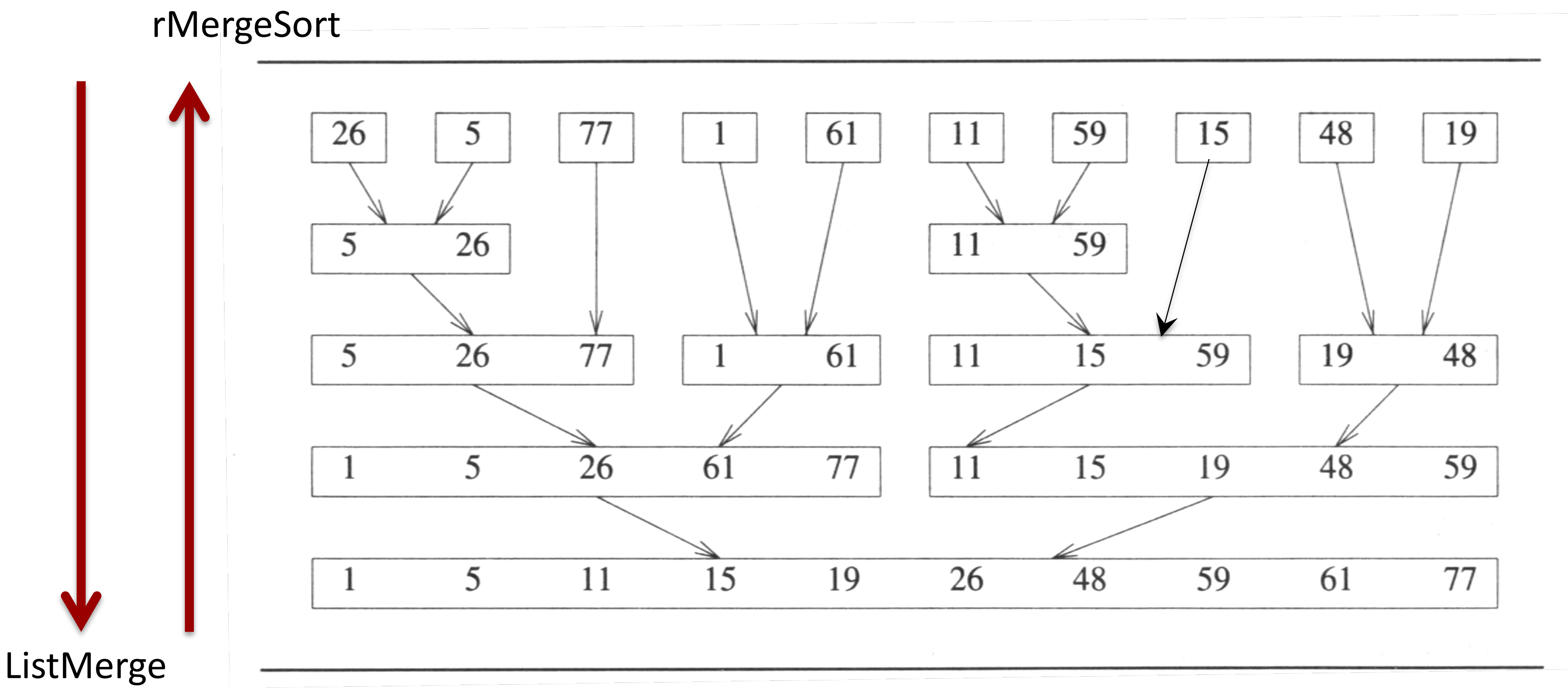
Recursive Merge Sort

- Split list into two equal size lists recursively
- Merge when returns

```
rMergeSort(*a, left, right)
{
   // a[left:right] is to be sorted
    if(left >= right) return left;
    mid = (left+right)/2;

    // actual merge happens when returns
    return ListMerge(a,
                         rMergeSort(a, left, mid),
                         rMergeSort(a, mid+1, right));
}
```

Recursive Merge Sort



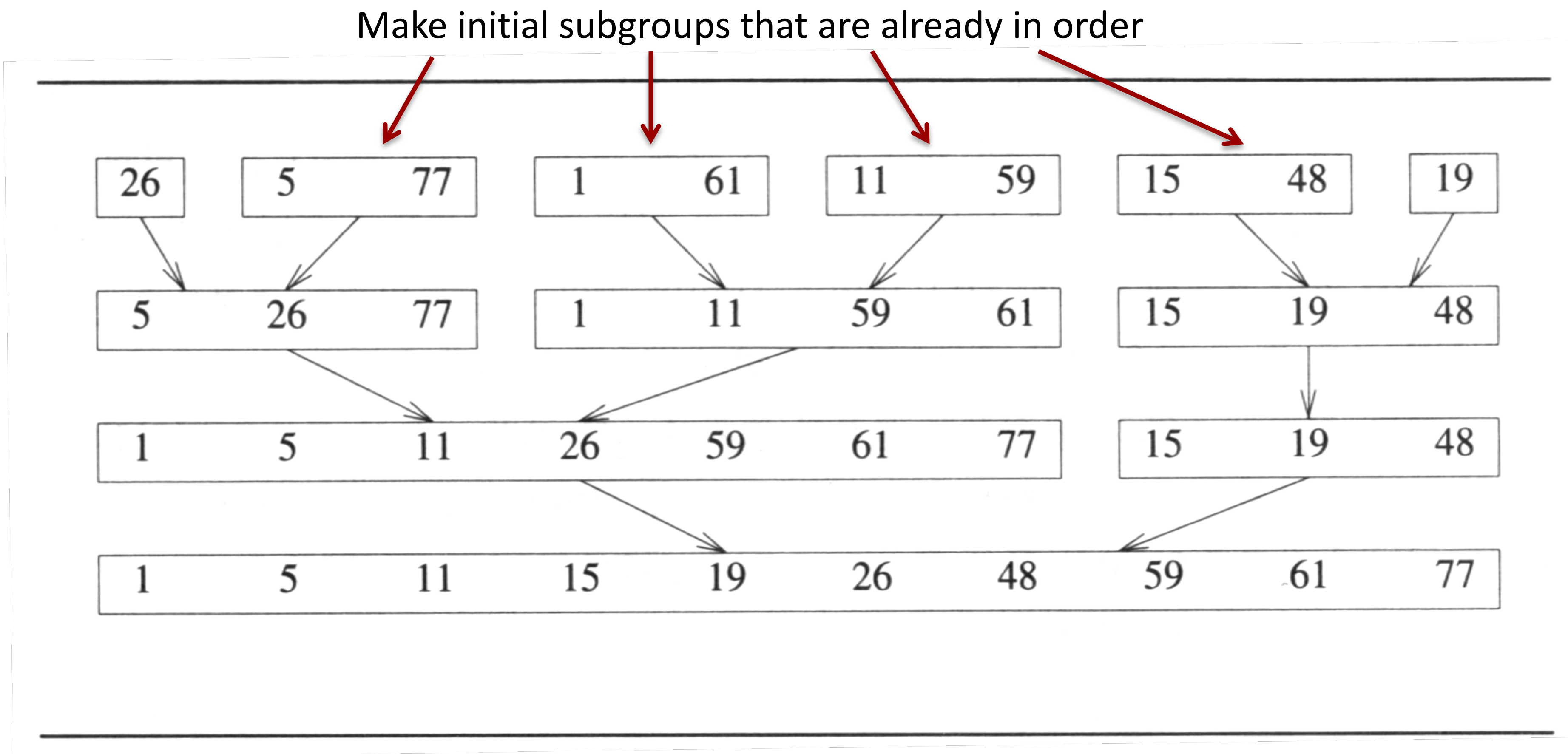
Merge Sort

- Analysis
  - Each step of merge lists : O(n)
  - # of steps : log n
  - Worst, best, average : O(n log n)
  - Additional space requirement : O(n)
    - Need same size buffer to store output
    - In-place merge sort exists, but slow
  - Stable

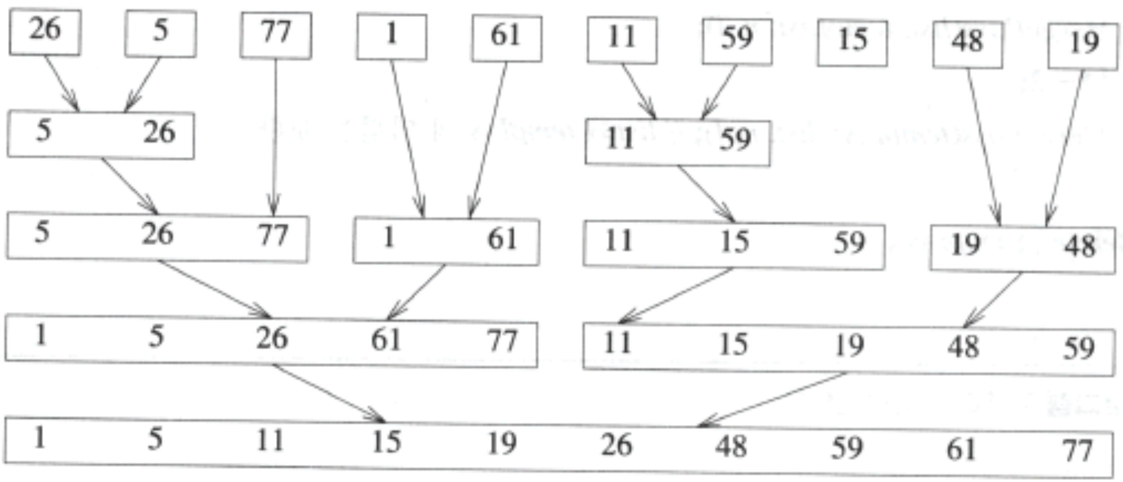
Merge Sort Variants

- Using integer pointer (link)
  - Eliminate record copying time
  - Useful when record size is large
- Natural merge sort
  - Idea
    - Make a sublist if records are already in correct order
    - Reduce merge operations
  - Example: 26, 5, 77, 1, 61, 11, 59
    - Original : [26][5] [77][1] [61][11] [59]
    - Natural : [26][5 77] [1 61][11 59]

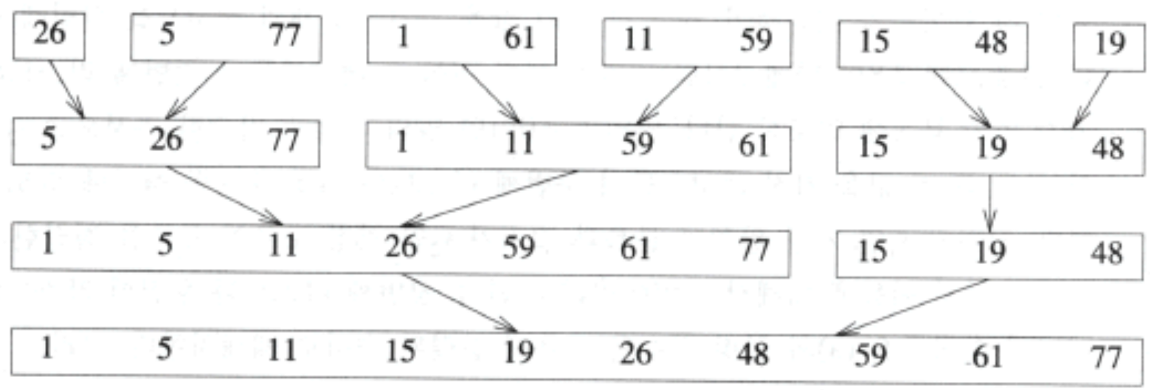
Natural Merge Sort



Iterative vs. Natural Merge Sort



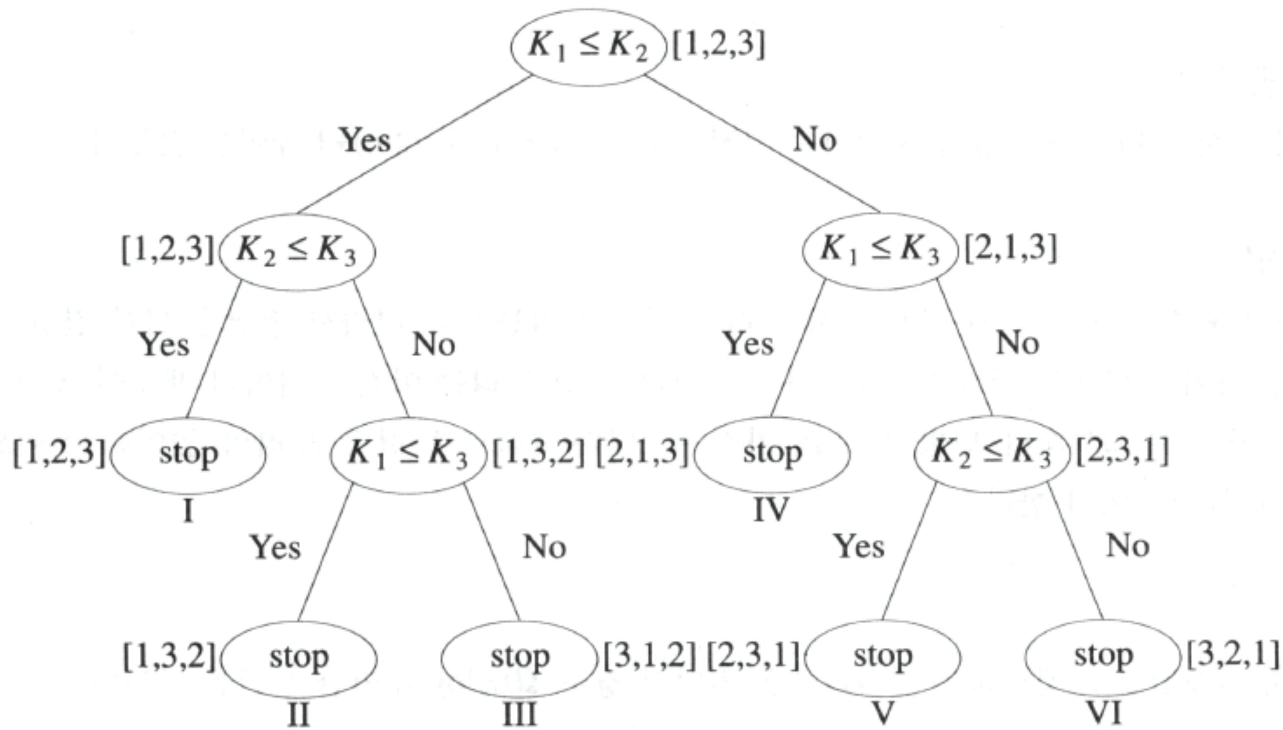
level : 5



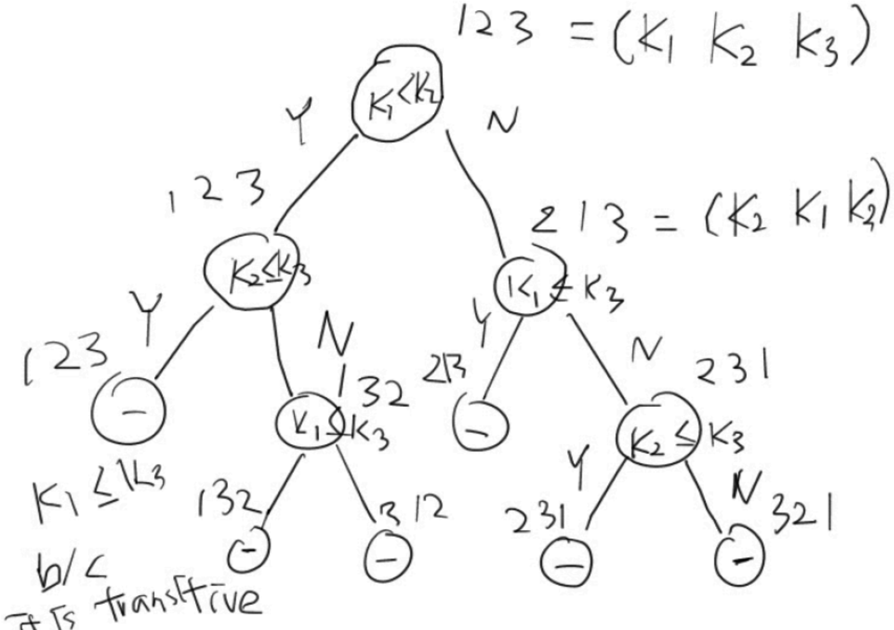
level : 4

How Fast Can We Sort?

- Decision tree
  - Binary tree with all possible comparisons



How fast can we sort?



Total # leaf nodes : $n!$

Minimum binary tree height having $n!$ leaf nodes : $\log_2(n!)+1$

Any comparison-based sorting algorithm should take a path from root to leaf

Then we should show that

b/c min height is $\log_2(n!)+1$,

any path length $\geq \log_2(n!)$

$= \log_2(n\cdot(n-1)\cdot(n-2)\cdots 2\cdot 1)$

$\geq \log_2(n/2)^{\frac{n}{2}}$ b/c $\underbrace{n\cdot(n-1)\cdots\frac{n}{2}}_{\frac{n}{2}}\cdot(\frac{n}{2}-1)\cdots 1$

$\geq\frac{n}{2}$ $\geq\frac{n}{2}$ $\cdots$ $\geq\frac{n}{2}$ $\therefore \left(\frac{n}{2}\right)^{\frac{n}{2}}$

$\Downarrow$

$\Omega(n\log_2 n)$

$\therefore$ Fastest comparison-based sorting algorithm is

$\Omega(n\log n)$

Outline

- Motivation
- Internal sort
  - Insertion sort
  - Quick sort
  - Merge sort
  - Radix sort

Sorting on Several Keys

- Records may have multiple keys
  - $K^1, K^2, \ldots, K^r$
  - K^1 : Most significant key
  - K^r : Least significant key
- A list of records $R^1, R^2, \ldots, R^n$ is sorted with respect to the keys $K^1, K^2, \ldots, K^r$ iff
  - If i < j then $(K_i^1, K_i^2, \ldots, K_i^r) \leq (K_j^1, K_j^2, \ldots, K_j^r)$
  - Compare most significant key first

Sorting on Several Keys

- Example
  - K^1 : A < B < C < D
  - K^2 : 1 < 2 < 3 < 4
  - Ordering
    - A1 < A2 < A3 < A4 <B1 < … < B4 < C1 < … < D4
- How to sort with multiple keys?

Sorting on Several Keys

- Most significant digit first (MSD) sort
  - Sort using most significant key (K^1) first
  - Group having same K^1 values, sort using K^2
  - Repeat this
- Example
  - A1, B3, A4, B2
  - A1, A4, B3, B2 (sort in terms of K^1)
  - A1, A4, B2, B3 (group A & B, sort in terms of K^2)

Sorting on Several Keys

- Least significant digit first (LSD) sort
  - Sort using least significant key (K^r) first
  - Sort using the key K^{r-1} (no grouping as in MSD)
  - Repeat this
- Example
  - A1, B3, A4, B2
  - A1, B2, B3, A4 (sort in terms of K^2)
  - A1, A4, B2, B3 (sort in terms of K^1)

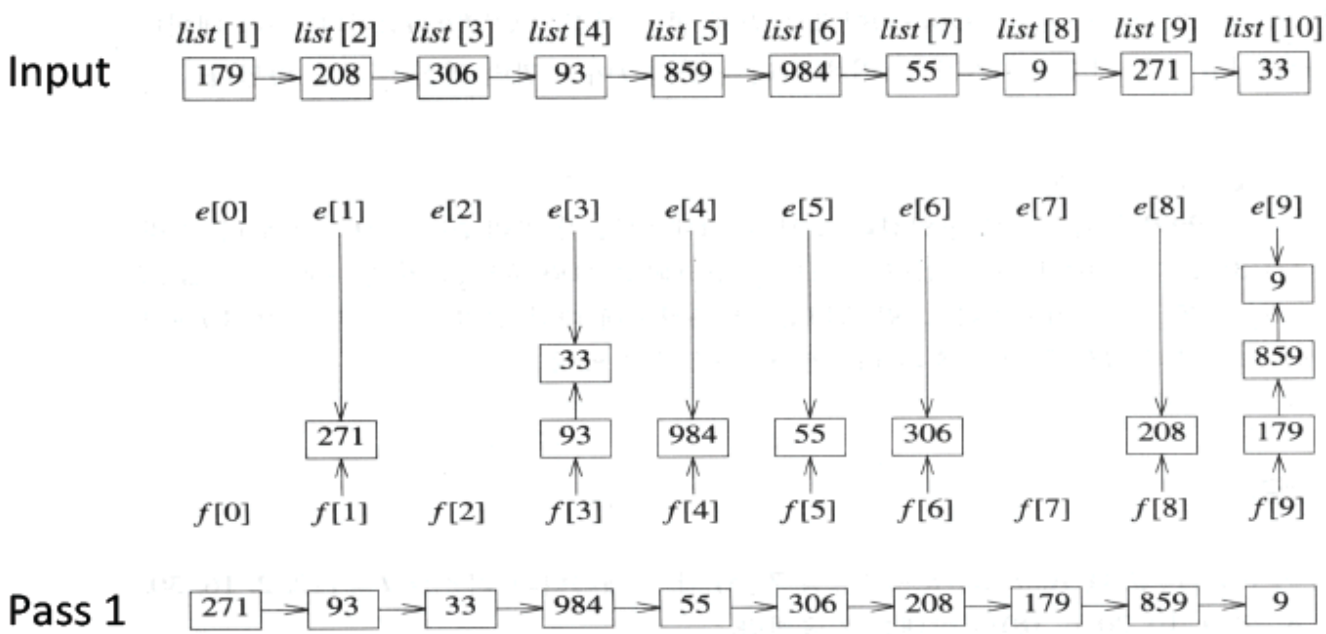
sorting for each K must be stable!

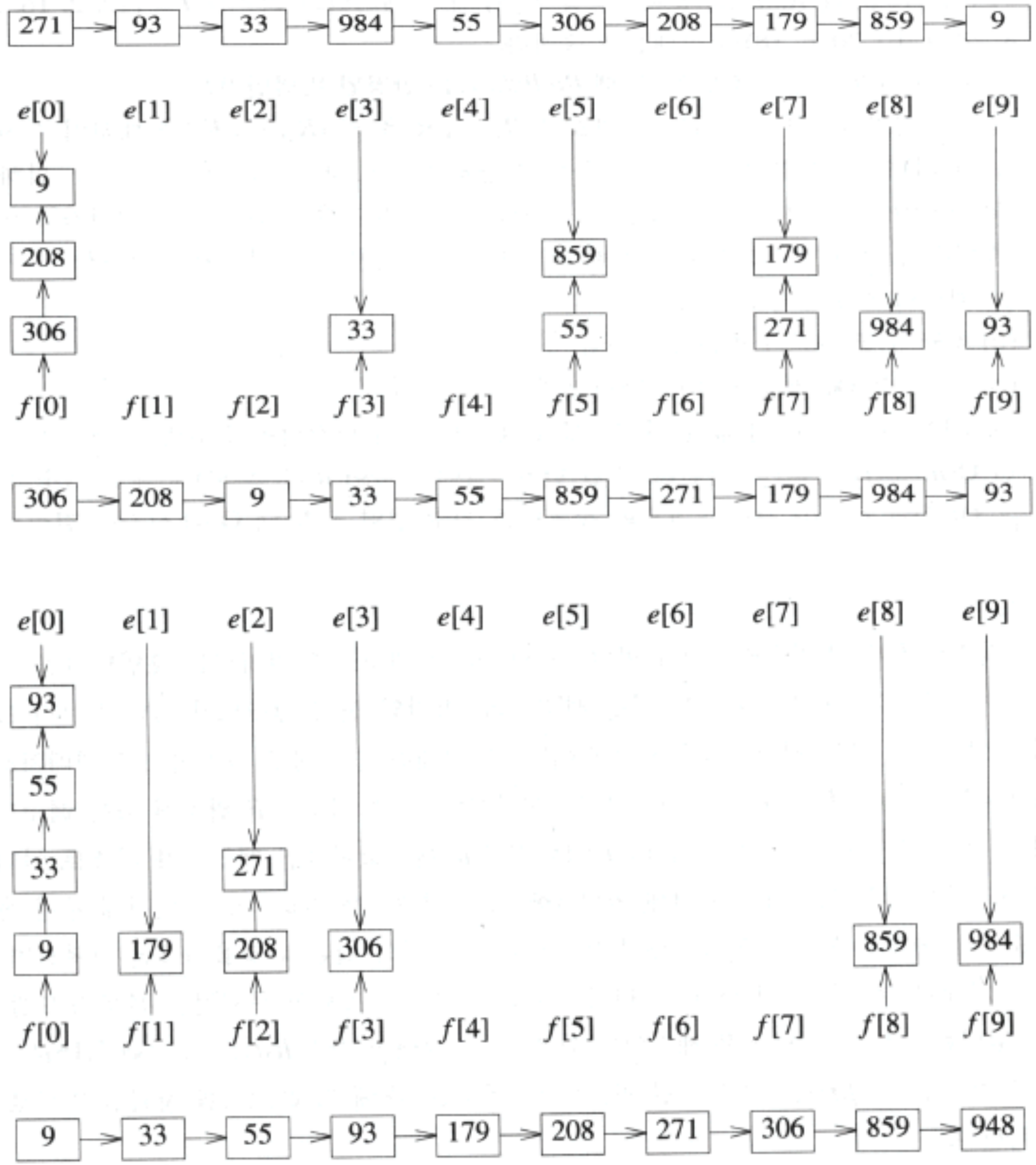
Radix Sort

- Think about integer sort
  - 1423 -> (1, 4, 2, 3)
  - # of keys : digit
  - # of possible different values per key : radix
  - 1423 -> 4 digits, radix-10
- Radix sort
  - Decompose the sort key using radix *r*, *d* digits
  - Make *r* bucket and *d* passes, i-th pass will sort K^i key, LSD order

Radix Sort

- Example
  - Sorting 10 numbers in the range [0, 999]
  - r = 10 (radix), d = 3 (digits)




271 93 33 984 55 306 208 179 859 9
e[0] e[1] e[2] e[3] e[4] e[5] e[6] e[7] e[8] e[9]
9 208 306 33 859 55 179 271 984 93
f[0] f[1] f[2] f[3] f[4] f[5] f[6] f[7] f[8] f[9]
306 208 9 33 55 859 271 179 984 93
e[0] e[1] e[2] e[3] e[4] e[5] e[6] e[7] e[8] e[9]
93 55 33 9 179 271 208 306 859 984
f[0] f[1] f[2] f[3] f[4] f[5] f[6] f[7] f[8] f[9]
9 33 55 93 179 208 271 306 859 948

Pass 2

Pass 3

Algorithm

```
RadixSort() // LSD radix-r sort, max digit of key : d
{
    create r queues;
    for(i=d-1; i>=0; i--) // sort on i-th radix-r digit
    {
        for(j=0; j<n; j++) // n : # of records
        {
            k = i-th radix-r digit of a[j]'s key;
            insert a[j] to queue[k];
        }
        list = {};
        for(k=0; k<r; k++) // collect resulting chain
        {
            list += queue[k];
        }
    }
    return list;
}
```

Discussion

- O(d(n+r))
- *d* depends on the choice of radix and the largest key
  - What if you have 100 numbers, 99 of them are only two-digit numbers (e.g., 12, 34, …) and one of them is 6-digit number (e.g., 123456)?
- If d and r << n, then O(n)
  - Better than comparison-based O(n log n) sorting algorithms

Questions?