Design and Analysis of Algorithms

L11: MergeSort

Dr. Ram P Rustagi
Sem IV (2020-Even)
Dept of CSE, KSIT
rprustagi@ksit.edu.in

Resources

- Text book 1: Levitin (Mergesort)
- NPTel - DAA (Prof Madhavan Mukund)
  - https://onlinecourses.nptel.ac.in/noc20_cs27/unit?unit=12&lesson=16
  - https://onlinecourses.nptel.ac.in/noc20_cs27/unit?unit=12&lesson=17

MergeSort

- Problem: Given a set of N elements, sort the elements in ascending (or descending) order
  - Assume that these elements are in an array of size N
- Approaches
  - Divide and Conquer approach

Sort Algorithms

- Bubble sort
- Selection sort
- Insertion sort
- **Mergesort**
- Quicksort
- Shell sort
- Heap sort
- Radix sort

MergeSort

- Basic idea
  - Take two sorted list and merge them into a single sorted list.

- Approach
  - Keep dividing the elements into (almost) equal half size (recursively) till sublist becomes of size 1
  - List of size 1 is sorted by default
  - Merge the sorted lists and keep repeating (recursively back)
  - When all the lists are merged, all elements are sorted.

MergeSort Example
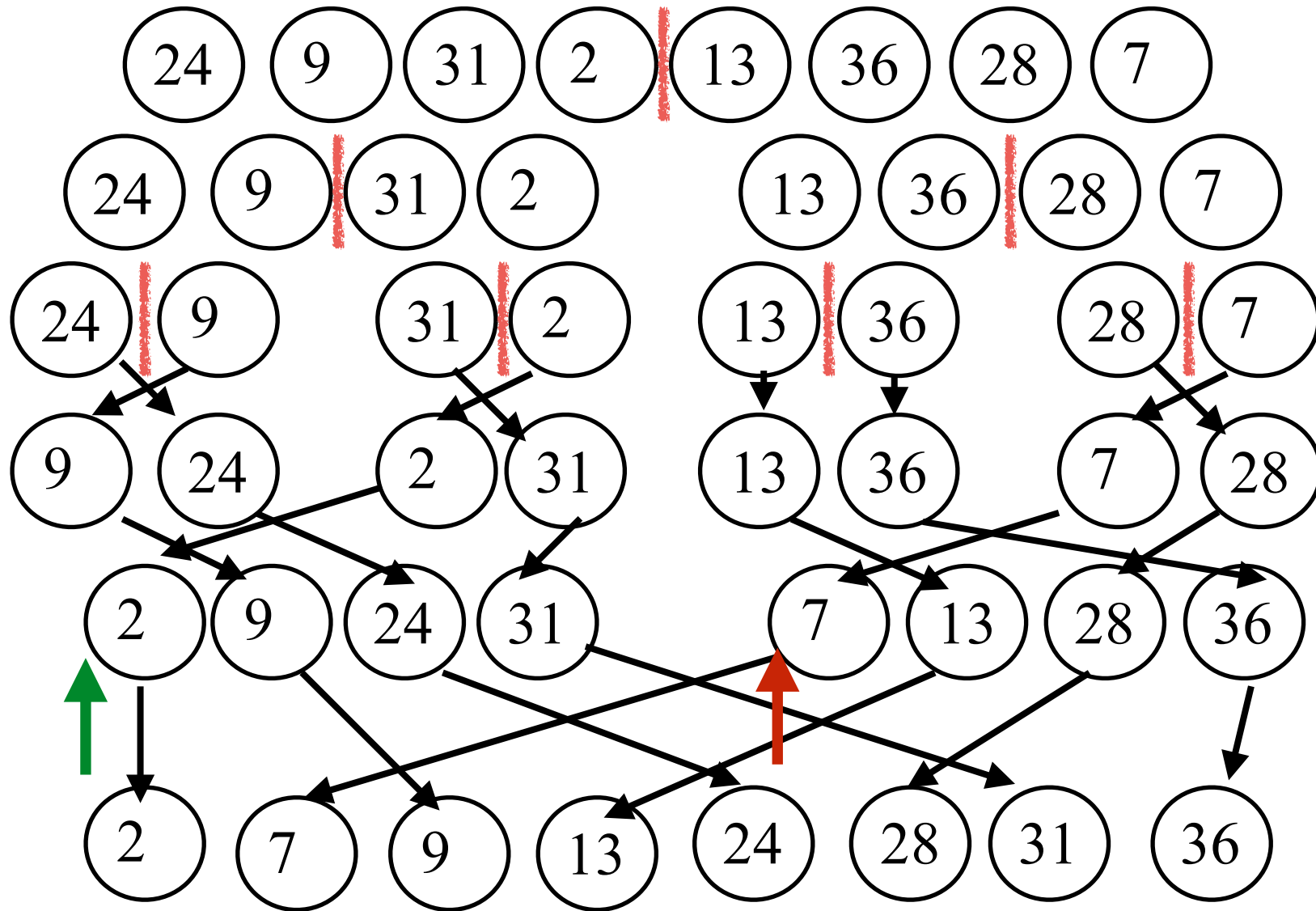
MergeSort

- Split array A[1:n] into about equal halves
  – Make copies of each half in arrays B and C
- Sort arrays B and C recursively
- Merge sorted arrays B and C into A as follows:
  – Repeat until one of the arrays becomes empty
    - Compare the first elements of the remaining unprocessed portions of the arrays
    - Copy the smaller of the two into A,
      – Increment the index of the array (smaller)
  – Once all elements in one of the arrays are copied
    - Copy the remaining unprocessed elements from the other array into A.

Algo: MergeSort

- **Algo** `MergeSort(1,n,A[])`
```
#Sort array A recursive by merging
#i/p: unsorted array A[1:n]
#o/p: sorted array A[1:n]
if n>1, then
   copy A[1:n/2] to B[1:n/2]
   copy A[n/2+1:n] to C[1:n/2]
   Mergesort(1,n/2,B) #recursive
   Mergesort(1,n/2,C) #recursive
   Merge(B,C,A) # merge two arrays
 # else part not required, why?
```

Algo: MergeSort

- **Algo** `Merge(B[1:p],C[1:q],A[1:p+q])`
`#maintain one index for each array`
`i←1; j←1; k←1;`
  while `(i<p+1)` **and** `(j<q+1)` **do**
    if `(B[i]≤C[j])`, **then**
      `A[k] ← B[i]`
      `i ← i+1`
    else
      `A[k] ← C[j]`
      `j ← j+1`
    `k ← k+1`
  if `(i > p)` **then** `#B has been fully copied to A`
    copy `C[j:q]` **to** `A[k:p+q]`
  else
    copy `B[i:p]` **to** `A[k:p+q]`

MergeSort: Analysis

- Each step of Mergesort
  - Two recursive invocations of size $n/2$: $2T(n/2)$
  - Merging of two $n/2$ array into one array of size n
    - Time complexity: n
- Recurrence relation for time complexity becomes

$$T(n) = 2T(n/2) + n$$
$$= 2(2T(n/4) + n/2) + n = 2^2 T(n/2^2) + n + n$$
$$= \ldots$$
$$= 2^k T(n/2^k) + n + \ldots (\log_2 n \text{ times})$$
$$= n*T(1) + n\log_2 n = n + n\log_2 n$$
$$= \Theta(n\log_2 n)$$

  - Better than $\Theta(n)$ for Insertion, Selection sort
- Space complexity $= \Theta(n)$

MergeSort: Master Theorem

T(n)=aT(n/b)+Θ(n^d) for **n=b^k, k=1,2,**
T(1) = c, where, **a≥1, b≥2, c>0**

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d log n) & \text{if } a = b^d \\ \Theta(n^{log_b a}) & \text{if } a > b^d \end{cases}$$

T(n)=2T(n/2) + n
a=2 (a≥1), b=2(b≥2), c=T(1)=1,and
f(n)=n $\in$Θ(n^d) ⇒ f(n)$\in$Θ(n^1) ⇒ d=1
Thus, b^d=b^1=b ⇒a=b^1 **#2nd** case in Master Theorem
T(n)=Θ(n^{d}log$_b$n) = Θ(n^{1}log$_2$n) = Θ(nlog$_2$n)

Mergesort Shortcomings

- Creates a new array i.e. requires additional O(n) space
  - No obvious way to merge in place in linear time.
- It is inherently recursive.
  - Recursive implemenation requires function invocation and return, a costly operation.
- Thus, Generally, not used in pratice.
- Alternative approaches
  - Can we ensure that left part is always less than the rigth part.
    - Thus, no need to merge the two.
    - Approach taken by **QuickSort.**

MergeSort (Inplace)

- If we need to merge in place, what is time and space complexity
  - Space: $O(1)$
  - Time: $O(n^2)$

									Moves
	6	10	15	20	3	4	5	19	Moves
S1	3	10	15	20	4	5	6	19	4
S2	3	4	15	20	5	6	10	19	4
S3	3	4	5	20	6	10	15	19	4
S4	3	4	5	6	10	15	19	20	5

3-way MergeSort

- Divide into 3 parts
- Mergesort each part separately
- Merge the parts.
- Time complexity

```
T(n) = 3T(n) + O(n)

     = O(log₃n)
```

Summary

- Mergesort
  - Not in place sort
  - Stable sort

-

Count Inversions

- Applications of Mergesort
- An inversion is defined as `A[i]>A[j]` if `i<j`.
- Consider the array
  `24, 9, 31, 2, 13, 36, 28, 7`
- Inversions are
  ```
  24-9, 24-2, 24-13, 24-7
  9-2, 9-7
  31-2, 31-13, 31-28, 31-7
  13-7
  36-28, 36-7
  28-7
  ```
- Total inversions: `14`
- Use Merge and count approach (Divide & Conq)

Count Inversions

- Consider the array
  `24, 9, 31, 2, 13, 36, 28, 7`
- Inversions in Left half: `24, 9, 31, 2`
  `(4): 24-31, 24-2, 9-2, 31-2`
  Sorted subarray: `2, 9, 24, 31`
- Inversions in Right half: `13, 36, 28, 7`
  `(4): 13-7, 36-28, 36-7, 28-7`
  Sorted subarray: `7, 13, 28, 36`
- Inversions from left half to right half
  `2,9,24,31 — 7,13,28,36`
  `(6): 9-7,24-7,24-13,31-7,31-13,31-28`
  Total inversions: `4+4+6=14`
- Use Merge and count approach (Divide & Conq)