Advanced Analysis of Algorithm

Department of Computer Science
Swat College of Science & Technology

CS Course : Advanced Analysis of Algorithm
Course Instructor : Muzammil Khan

Chapter 6

Sorting Algorithms
(Advanced Sorting Algorithms)

Sorting Algorithms - Classification

- The sorting algorithms are classified into two categories
  - On the basis of *underlying procedure* used
- Sorting by Comparison
  - These are based on comparison of keys
    - The method has general applicability
    - Examples are selection sort, quick sort
- Sorting by Counting
  - These depend on the characteristics of individual data items
    - The sort method has limited application
    - Examples include radix sort, bucket sort,

Sorting by Comparison

- The *Sorting by Comparison* method sorts input
  - By comparing pairs of keys in the input.
- Elementary Algorithms
  - Are *inefficient* but *easy to implement*
  - Their running times are $\theta(n^2)$
  - Common algorithms are
    - *Insertion sort, Exchange sort, Selection sort*
- Advanced Algorithms
  - Advanced algorithms are *efficient*
  - Implementations are usually based on *recursive function calls*
  - Their running times are $\theta(n \lg n)$
  - The typical advanced algorithms include
    - *Merge sort, Quick sort, Heap sort*

Sorting Algorithms

Algorithm	Design Approach	Sort in Place	Complexity
Insertion Sort	Incremental	Yes	$\Theta(n^2)$
Selection Sort	Incremental	Yes	$\Theta(n^2)$
Bubble Sort	Incremental	Yes	$\Theta(n^2)$
Merge Sort	Divide & Conquer	No	Let's see !!

Divide and Conquer Approach

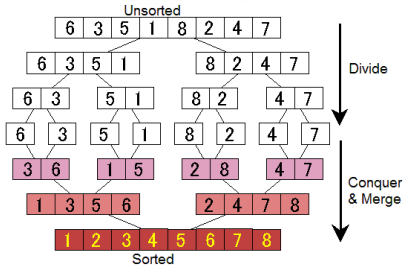
- Divide the problem into a number of sub-problems
  - $D(n)$ - Time to divide a problem into sub-problems
- Conquer the sub-problems by solving them recursively
  - $aT(n/b)$ - Time to conquer each sub-problem
    - If sub-problem sizes are small,
      - They can be solved in a straightforward manner
- Combine the solution of sub-problems into the solution for the original problem
  - $C(n)$ – Time to combine sub-problems
- Total time
  - $T(n) = aT(n/b) + D(n) + C(n)$

Merge Sort

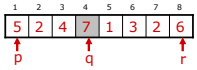
Merge Sort Idea

- To sort an array A[p ... r]
- Divide
  - Divide the array of n-element sequence to be sorted into two subsequences of n/2 elements each
    - Until the subsequence size become 1
- Conquer
  - Sort the subsequences recursively using merge sort
- Combine
  - Merge the two sorted subsequences
    - Until the whole array is sorted

Merge Sort - Illustration



Merge Sort



MERGE-SORT(A, p, r)

if p < r ►Check for base case

then q ← ⌊(p + r)/2⌋ ►Divide

MERGE-SORT(A, p, q) ►Conquer 1st half (Recursively sort)

MERGE-SORT(A, q + 1, r) ►Conquer 2nd half (Recursively sort)

MERGE(A, p, q, r) ►Combine the 2 halves

- Initial call: MERGE-SORT(A, 1, n)

Merge Sort Calls

Merge-Sort(A, 1, 8) Here p = 1, r = 8 & q = 4 as p < r

MS(A, 1, 4)

MS(A, 1, 2)

MS(A, 1, 1) ► 5

MS(A, 2, 2) ► 7

M(A, 1, 1, 2) ► 5, 7

MS(A, 3, 4)

MS(A, 3, 3) ► 3

MS(A, 4, 4) ► 8

M(A, 3, 3, 4) ► 3, 8

M(A, 1, 2, 4) ► 3, 5, 7, 8

MS(A, 5, 8) ► Next slide

Merge Sort Calls (Cont...)

Merge-Sort(A, 1, 8)

MS(A, 5, 8)

MS(A, 5, 6)

MS(A, 5, 5) ► 2

MS(A, 6, 6) ► 6

M(A, 5, 5, 6) ► 2, 6

MS(A, 7, 8)

MS(A, 7, 7) ► 1

MS(A, 8, 8) ► 4

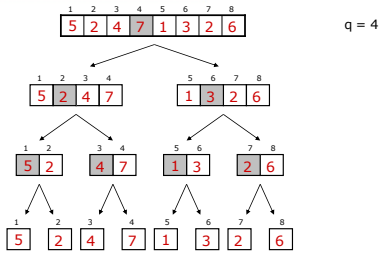
M(A, 7, 7, 8) ► 1, 4

M(A, 5, 6, 8) ► 1, 2, 4, 6

M(A, 1, 4, 8) ► 1, 2, 3, 4, 5, 6, 7, 8 ► Sorted

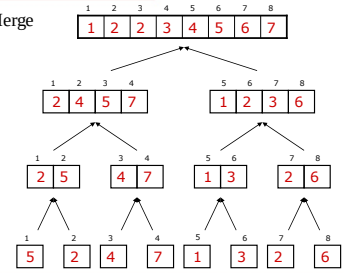
Merge Sort (Cont...)

- Divide

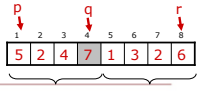


Merge Sort (Cont...)

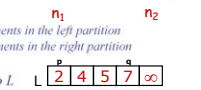
- Conquer & Merge



Merge Sort - Pseudocode




```
MERGE(A, p, q, r)
1   n1 ← q − p + 1                    ▶ set n1 to number of elements in the left partition
2   n2 ← r − q                        ▶ set n2 to number of elements in the right partition
3   for i ← 1 to n1
4     do L[i] ← A[p + i - 1]          ▶ Copy left partition into L
5   for j ← 1 to n2
6     do R[j] ← A[q + j]              ▶ Copy right partition into R
7   L[n1 + 1] ← ∞                     ▶ Insert a large value into the last cell to mark the end of array L
8   R[n2 + 1] ← ∞                     ▶ Insert a large value into the last cell to mark the end of array R
9   i ← 1                             ▶ Counter to scan array L , initially set to 1
10  j ← 1                             ▶ Counter to scan array R, initially set to 1
11  for k ← p to r                    ▶ Process elements from the beginning of left partition to end of right partition
12    do if L[i] ≤ R[j]               ▶ Identify smaller of the two elements in L and R
13      then A[k] ← L[i]              ▶ If smaller element in L, copy it into A
14      i ← i + 1                     ▶ Increase counter for L
15      else A[k] ← R[j]              ▶ If smaller element in R, copy it into A
16      j ← j + 1                     ▶ Increase counter for R
```



Merge Sort – Recurrence Relation

- Recall for Divide and Conquer algorithms
  $T(n) = aT(n/b) + D(n) + C(n)$
- Here ***a=2***, and if we assume ***n*** is a power of ***2***, then each divide step leads to sub-arrays of size ***n/2***
- $D(n)=\theta(1)$ => *division take constant time*
- $C(n)=\theta(n)$ => *combining n elements*
- $T(n)=2T(n/2)+\theta(n)$

Merge Sort – Worst-Case Scenario

In ***worst case*** the array partitioning and merging is done right up to single element arrays. The running time $T(n)$ for the worst case is given by the recurrence:

$$T(1) = c \quad (c \text{ cost of merging single element})$$

$$T(n) = \underbrace{2\,T(n/2)}_{\text{Time to sort two sub-arrays each of size } n/2} + \underbrace{c.n}_{\text{Cost of merging two sub-arrays each of size } n/2}$$

Time to sort array of size n

- The recurrence can be solved by ***substitution method.***
- After k*th* iteration
  $T(n) = ck.n + c.2^k T(n/2^k)$
- Setting $n/2^k = 1$, gives $k = \lg n$
- Substituting for k gives
  $T(n) = c\,n \lg n + c2^{\lg n} = c.n \lg n + c.\,n$

Merge Sort – Worst-Case Scenario

Solving by Substitution Method

As $T(n) = 2T(n/2)+c.n$ ..... (1)

Solve for T(n/2), T(n/4), T(n/8) . . .

$T(n/2) = 2\,T(n/4)+c.n/2$

Eq (1) => $T(n) = 2(2T(n/4)+c.n/2)+c.n$

$= 4T(n/4)+ 2.c.n/2 + c.n$

$= 2^2 T(n/2^2)+2c.n$ ..... (2)

Now T(n/4)

$T(n/4) = 2T(n/8)+c.n/4$

Eq (2) => $T(n) = 2^2(2T(n/8)+c.n/4)+2c.n$

$= 2^3 T(n/2^3)+4.c.n/4+2cn$

$= 2^3 T(n/2^3)+3c.n$

Merge Sort – Worst-Case Scenario

Solving by Substitution Method (Cont...)

Similarly

$T(n) = 2^4 T(n/2^4) + 4c.n$

If $n=2^i$ Then

$T(2^i) = 2^i T(2^i/2^i) + i\ c.n$

$= 2^i T(1) + i\ c.n$

As $T(1) = \Theta(1) = c$

So $T(2^i) = 2^i c + ic.n$

As assumed

$n = 2^i$

$i = \lg_2 n$

Merge Sort – Worst-Case Scenario

Solving by Substitution Method (Cont...)

So $T(n) = nc + \lg_2 n.c.n$

$= cn(1+\lg_2 n)$

$= cn\lg n + cn$

- Ignoring low order Terms and constants
  - ***T(n) = Θ(nlgn)***

Advanced Analysis of Algorithm

Merge Sort
Worst, Average and Best-Case Scenario

- Worst case running time of merge sort is
  - *O(nlgn)*
- Average case running time of merge sort is also
  - *θ(nlgn)*
- Best case ?

Advanced Analysis of Algorithm

Merge Sort (Cont...)

- Running time of merge sort is
  - ***Insensitive of the input***
- Advantage
  - Guaranteed to run in *Θ(nlgn)*
- Disadvantage
  - Requires extra space ≈N

Advanced Analysis of Algorithm

Sort Challenge 1

- Problem
  - Sort a huge **randomly-ordered** file of small records
- Application: Process transaction record for a phone company
- Which sorting method to use?
  - A. Bubble sort
  - B. Selection sort
  - C. Merge sort guaranteed to run in time ~ ***nlgn***
  - D. Insertion sort

Advanced Analysis of Algorithm

Solution

- Selection sort?
  - NO, always takes quadratic time
- Bubble sort?
  - NO, quadratic time for randomly-ordered keys
- Insertion d?
  - NO, quadratic time for randomly-ordered keys
- Merge sort?
  - YES, it is designed for this problem

Advanced Analysis of Algorithm

Sort Challenge 2

- Problem
  - Sort a file that is already almost in order
- Applications
  - Re-sort a huge database after a few changes
  - Double check that someone else sorted a file
- Which sorting method to use?
  - A. Merge sort, guaranteed to run in time ~ ***nlgn***
  - B. Selection sort
  - C. Bubble sort
  - D. A custom algorithm for almost in-order files
  - E. Insertion sort

Solution

- Selection sort?
  - NO, always takes quadratic time
- Bubble sort?
  - NO, bad for some definitions of "almost in order"
  - Example:
    - B C D E F G H I J K L M N O P Q R S T U V W X Y Z A
- Merge sort or custom method?
  - Probably not: insertion sort simpler and faster
- Insertion sort?
  - YES, takes linear time for most definitions of "almost in order"

Sort Challenge 3 (Homework)

- Problem
  - Sort a file of huge records with tiny keys
- Example application: Reorganize your MP-3 files
- Which method to use?
  - A. Merge sort, guaranteed to run in time ~ ***nlgn***
  - B. Selection sort
  - C. Bubble sort
  - D. Insertion sort
  - E. A custom algorithm for huge records/tiny keys

Solution

- Insertion sort or bubble sort?
  - NO, too many exchanges
- Merge sort or custom method?
  - Probably not
- Selection sort?
  - YES, it takes linear time for exchanges
  - Selection sort simpler, does less swaps

Quick Sort

Quick Sort

- Design Approach
  - Divide and Conquer
- Procedure take place in two phases
- Given array

51	67	11	78	90	13	66	80	92	30	40	59	68	71	65	23

- Phase #1 (Divide)
  - An array element is chosen as ***Pivot***

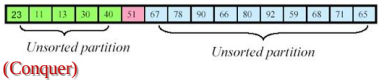
51	67	11	78	90	13	66	80	92	30	40	59	68	71	65	23

  - The array is partitioned into two sub-arrays (by pivot value)
    - All elements on the left are less then pivot value and
    - All elements on the right are greater or equal to pivot value

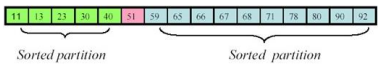
23	11	13	30	40	51	67	78	90	66	80	92	59	68	71	65

Unsorted partition *Unsorted partition*

Quick Sort (Cont...)



- Phase #2 (Conquer)
  - The left and right sub-arrays are sorted recursively



Quick Sort Pseudo code

QUICKSORT(A, p, r)

1. if p < r then ►Check for base case
2. q ← PARTITION(A, p, r) ►Create Partitions
3. QUICKSORT(A, p, q-1) ►Sort left side of the pivot
4. QUICKSORT(A, q+1, r) ►Sort right side of the pivot

Quick Sort Pseudo code (Cont...)

PARTITION (A, p, r)

1. x ← A[r]
2. i ← p-1
3. for j ← p to r-1
4. do if A[j] <= x
5. then i ← i+1
6. exchange A[i] ↔ A[j]
7. exchange A[i+1] ↔ A[r]
8. return i+1

Description

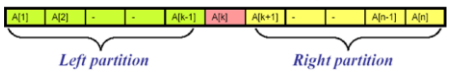
► Need to addddd description

Analysis of Quick Sort

- Running time of Quick sort depends on
  - Whether partition is balanced or unbalanced
    - This in turn depends on elements (pivot) used for partitioning

Quick Sort – Best-Case Scenario

- In best case
  - The array is partitioned into *two nearly equal sub-array*



- Recurrence relation for best case running time is

$$T(n) = 2\,T(n/2) + n - 1$$

Time to sort array of size n = *Time to sort two subarrays each of size n/2* + *Cost of partitioning subarray of size n-1*

Quick Sort – Best-Case Scenario (Cont...)

- The best case running time recurrence

$$T(n) = 2\,T(n/2) + n - 1$$

can be solved by ***iteration-substitution*** method

- After k^{th} iteration:

$$T(n) = 2^k T(n/2^k) + k.n - (2^0 + 2^1 + 2^2 + \ldots\ldots + 2^{k-1})$$

- Summing the series

$$T(n) = 2^k T(n/2^k) + k.n - (2^k - 1)$$

- Setting $n/2^k = 1$, gives $k = \lg n$

$$T(n) = 2^{\lg n}.T(1) + n.\lg n - (2^{\lg n} - 1)$$

$$= n + n \lg n - (n - 1)$$

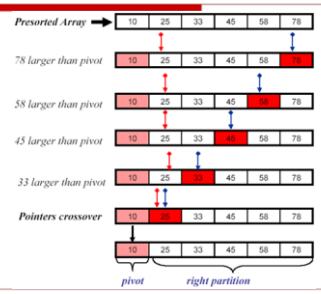
$$= n \lg n + 1 = \theta(n \lg n)$$

- The best case running time of quick sort is $\theta(n \lg n)$

Quick Sort – Worst-Case Scenario

- The *worst case arises*
  - When the given array is in *presorted*
  - The partitioned result in a *single right partition*
    - The *left partition is empty*
  - In this case
    - The array is scanned from left to right until
      - The pointer cross over
  - As shown
    - Next slide

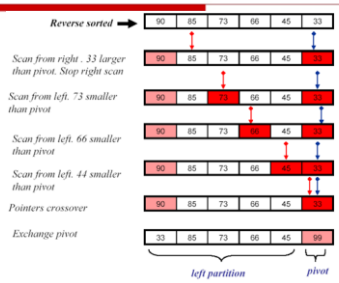
Quick Sort – Worst-Case Scenario (Cont...)



Quick Sort – Worst-Case Scenario (Cont...)

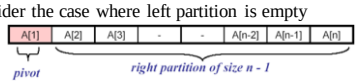
- The *worst case arises*
  - When the given array is in *reverse sorted*
  - The partitioned result in a *single left partition*
    - The *right partition is empty*
  - In this case
    - The array is scanned from left to right until
      - The pointer cross over
  - As shown
    - Next slide

Quick Sort – Worst-Case Scenario (Cont...)



Quick Sort – Worst-Case Scenario (Cont...)

- Consider the case where left partition is empty



- The recurrence relation for worst case

$T(n) = T(0) + T(n-1) + n-1$

Time to sort array of size n; Time to sort left partition $T(0)=0$; Time to sort right partition of size n-1; Cost of partitioning subarray of size n-1

- Since left partition is empty, recurrence simplifies to

$T(n) = T(n-1) + n-1, \quad n>1$

  - Same recurrence applies when right partition is empty

Quick Sort – Worst-Case Scenario (Cont...)

- The recurrence

$T(n) = T(n-1) + n-1, \quad n > 0$
$T(0) = 0$

  - Can be solved by *iteration method*
- Iteration yields the solution as

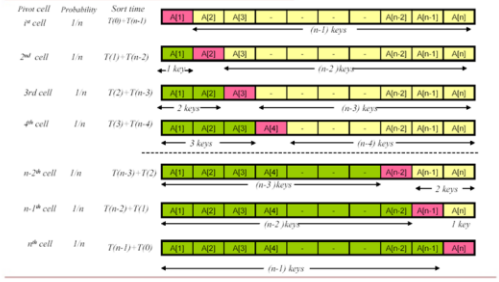
$T(n) = n-1 + (n-2) + (n-3) + \ldots + 2 + 1$
$= n(n-1)/2 = O(n^2)$

- The worst case running time of worst case is
  - $\Theta(n^2)$

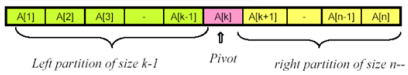
Quick Sort – Average-Case Scenario

- In *average case*, we assume that
  - The *pivot* can be any array element of size ***n***
  - By *probabilistic analysis*
    - ***1/n*** is the probability of any array element to be pivot
  - As shown
    - Next slide

Quick Sort – Average-Case Scenario (Cont...)



Quick Sort – Average-Case Scenario (Cont...)



- By summing over running time for all probable locations of pivot and associated cost
- The Recurrence

$$T(n) = n-1 + \sum_{k=1}^{n} \frac{1}{n} [\, T(k-1) + T(n-k) \,]$$

Average time to sort array of size n | *Cost of partitioning subarrays of size n-1* | *Probability of pivot being in cell k* | *Time to sort subarray of size k-1* | *Time to sort subarray of size n - k*

Quick Sort – Average-Case Scenario (Cont...)

$$T(n) = n-1 + \sum_{k=1}^{n} \frac{1}{n} [\, T(k-1) + T(n-k) \,]$$

- Expanding the summation

$$\sum_{k=1}^{n} T(n-k) = T(n-1)+T(n-2)+\ldots+T(2)+T(1)+T(0)$$
$$= T(0)+T(1)+T(2)+\ldots+T(n-2)+T(n-1) \quad (\text{Rearranging in reverse order})$$
$$= \sum_{k=1}^{n} T(k-1)$$

- The recurrence can be written as:

$$T(n) = n-1 + \frac{2}{n}\sum_{k=1}^{n} T(k-1)$$

- Multiplying both sides with n

$$n.T(n) = n(n-1) + 2\sum_{k=1}^{n} T(k-1)$$

- Substituting *n-1* for *n* in the last recurrence

$$(n-1).T(n-1) = (n-1)(n-2) + 2\sum_{k=1}^{n-1} T(k-1)$$

Quick Sort – Average-Case Scenario (Cont...)

$$n.T(n) = n(n-1) + 2\sum_{k=1}^{n} T(k-1)$$

$$(n-1).T(n-1) = (n-1)(n-2) + 2\sum_{k=1}^{n-1} T(k-1)$$

- Subtracting second equality from the first, and rearranging

$$nT(n) - (n-1)T(n-1) = 2(n-1) + 2T(n-1)$$

- Simplifying and rearranging

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

- Setting $S(n)$

$$S(n) = \frac{T(n)}{n+1}$$

gives the new recurrence

$$S(n) = S(n-1) + \frac{2(n-1)}{n(n+1)}, \quad S(0)=0$$

Quick Sort – Average-Case Scenario (Cont...)

- The following recurrence is solved by ***iteration method.***

$$S(n) = S(n-1) + \frac{2(n-1)}{n(n+1)}, \quad S(0)=0$$

- The solution is as follows

$$S(n) = \sum_{i=1}^{n} \frac{2(i-1)}{i(i+1)} = \sum_{i=1}^{n} \frac{2}{i+1} - \sum_{i=1}^{n} \frac{2}{i(i+1)}$$
$$= \sum_{i=1}^{n} \frac{2}{i+1} - \sum_{i=1}^{n} [\frac{2}{i} - \frac{2}{(i+1)}]$$
$$= \sum_{i=1}^{n} \frac{4}{i+1} - \sum_{i=1}^{n} \frac{2}{i} = \sum_{i=1}^{n} \frac{2}{i} - \frac{4n}{(n+1)}$$

Since $\sum_{i=1}^{n} \frac{1}{i} \approx \lg n + 0.386$

- $S(n) = (\ln n + 0.386) - 4n/(n+1)$
- $T(n) = 1.386\, n \lg n - 2.846\, n$ (changing ln n to lg n)

Average running time of quick sort is ***θ(n lg n)***

Pivot Selection

- *Pivot influence performance* of quick sort algorithm
  - There are four choices
- Left-most element
  - The *1st* element is chosen as pivot
    - There is chance of bad partitioning

1	2	3	4	5	6	7	8	9	10
56	90	82	88	12	50	89	22	77	25

pivot=56

- Right-most element
  - The *last* element is chosen as pivot
    - There is chance of bad partitioning

1	2	3	4	5	6	7	8	9	10
56	90	82	88	12	50	89	22	77	25

pivot=25

Pivot Selection

- Medium element
  - The middle element is chosen as pivot

1	2	3	4	5	6	7	8	9	10
56	90	82	88	12	50	89	22	77	25

pivot=12

- Random element
  - An element is selected randomly
    - The random selection reduces/ minimize the chances of bad partitioning

1	2	3	4	5	6	7	8	9	10
56	90	82	88	12	50	89	22	77	25

pivot=89

Randomized Quick Sort

In ***randomized quick sort***, the pivot is chosen ***randomly*** from the elements in subarray. A ***random number generator*** is used to generate an index which lies in the range of upper and lower indexes of the array. The procedures for partitioning and recursive sorting are given below.

```
RANDOMIZED-PARTITION( A, p, r)
1  i← RANDOM(p,r)          ►RANDOM generates a random number in range p to r
2  exchange A[l]↔A[i]      ►Exchange left-most element with the randomly selected element
3  k ← PARTITION(A, p, r)  ►Invoke PARTITION method to divide the array
4  return k                ►Return index of pivot in the partitioned array
```

```
RANDOMIZED-QUIKSORT( A, p, r)
1  if  p < r
2  then q← RANDOMIZED-PARTITION(A, p, r)
3       RANDOMIZED-QUIKSORT( A, p, q-1)
4       RANDOMIZED-QUIKSORT( A, q+1, r)
```


Homework

- Count Sort
- Radix Sort
- Bucket Sort
- Binary Sort