CME 2001
Data Structures and Algorithms

Zerrin Işık

zerrin@cs.deu.edu.tr

Course Information

CME 2001

Class hours : Wednesday 8:45 – 10:15 (D4)

Wednesday 10:30 – 12:00 (D4)

Lab hours : Friday 13:00 – 14:30 (Lab 10 -11)

Friday 14:50 – 16:20 (Lab 10 -11)

Office Hours : Tuesday – Wednesday : 13:00 – 15:00

CME 2001

Grading:

2 Homeworks : 25%

1 Midterm : 25%

1 Final : 50%

Plagiarism will get directly ZERO in the homeworks!

Attendence (lectures & labs) is not compulsory !

CME 2001

Text Book:

An Introduction to Algorithms,

Cormen, Leiserson, Rivest and Stein, 3^{rd} Edition,

MIT Press, 2009

Course Page:

GoogleClassroom … CME 2001

Joining code: aau8lit

CME 2001

Google email accounts: name.surname@ceng.deu.edu.tr

Password: email address given in DEU student registration

Please contact with one of account admins in case of problems:

- Onur Çakırgöz
- Mustafa Batar
- İlker Kalaycı

Weekly Schedule

Week	Topic
1.	Introduction, Insertion Sort
2.	Merge Sort, Growth Functions (Asymptotic Notation, Recurrence Solving)
3.	Elementary Data Structures (Stack, Queue, Hash Table)
4.	Open Address Hashing, Resolving Hashing Problems
5.	Heap Sort, Quick Sort
6.	Linear Sorting Algorithms
7.	Binary Search Trees
8.	Midterm 1
9.	Balanced (AVL) Trees
10.	B-Trees
11.	Graph Traversal Algorithms
12.	Minimum Spanning Trees
13.	Midterm 2 – No Lecture
14.	Single Source Shortest Paths

Course Content

- Analysis of Algorithms
- Essential Data Structures (Simple to Complex)

Sorting Algorithms and Their Analysis

Sorting Problem?

- **Input:** A sequence of n numbers $(a_1, a_2, \ldots, a_n)$

- **Output:** A permutation (reordering) $(a_1', a_2', \ldots, a_n')$ of the input sequence such that $a_1' \leq a_2' \leq \ldots \leq a_n'$.

- **Example:**

  Input : 3 7 9 1 2

  Output : 1 2 3 7 9

Insertion Sort

- A good algorithm for sorting a small number of elements.
- Lets assume you will sort a hand of playing cards:
  - Start with an empty left hand and the cards face down on the table.
  - Each time remove one card from the table, and insert it into the correct position in the left hand.
  - To find the correct position for a card, compare it with each of the cards already in the hand, from right to left.
  - At all times, the cards held in the left hand are sorted, and these cards were originally the top cards of the pile on the table.

```
INSERTION-SORT(A, n)
  for j = 2 to n
      key = A[j]
      // Insert A[j] into the sorted sequence A[1 .. j - 1].
      i = j - 1
      while i > 0 and A[i] > key
          A[i + 1] = A[i]
          i = i - 1
      A[i + 1] = key
```

Insertion Sort Example

Sorted **Unsorted**

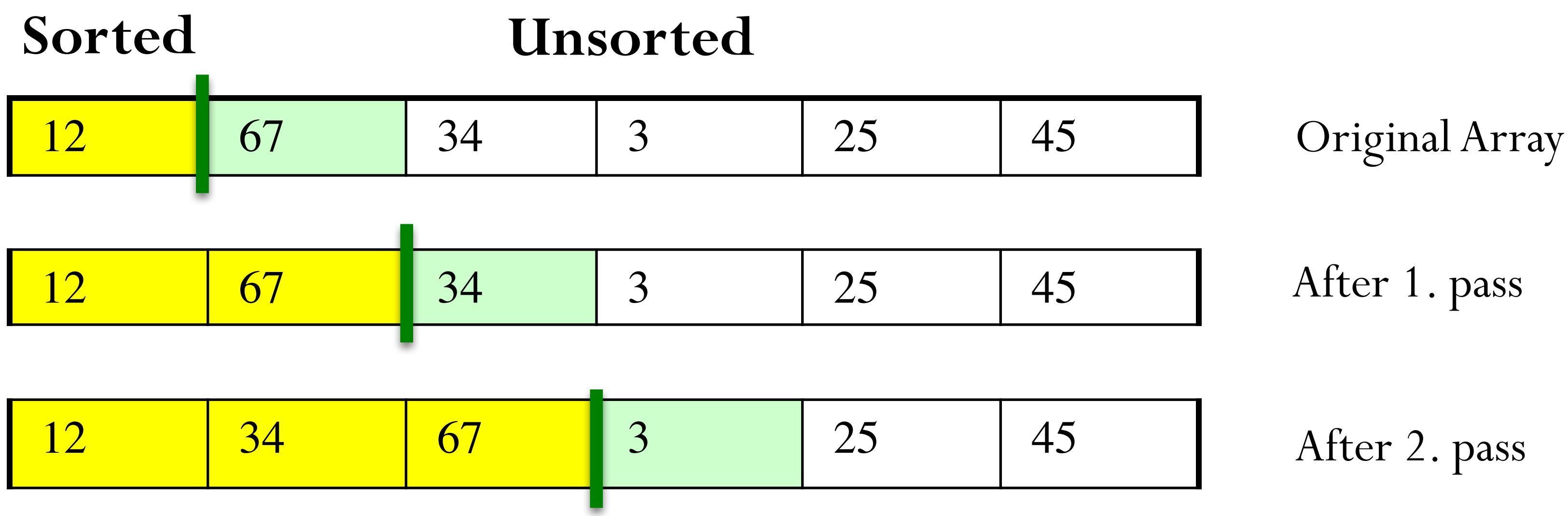
12	67	34	3	25	45	Original Array

Insertion Sort Example

Sorted **Unsorted**

12	67	34	3	25	45	Original Array
12	67	34	3	25	45	After 1. pass

Insertion Sort Example



Insertion Sort Example

Sorted		Unsorted				
12	67	34	3	25	45	Original Array
12	67	34	3	25	45	After 1. pass
12	34	67	3	25	45	After 2. pass
3	12	34	67	25	45	After 3. pass

Insertion Sort Example

Sorted **Unsorted**

12	67	34	3	25	45	Original Array
12	67	34	3	25	45	After 1. pass
12	34	67	3	25	45	After 2. pass
3	12	34	67	25	45	After 3. pass
3	12	25	34	67	45	After 4. pass

Insertion Sort Example

Sorted **Unsorted**

12	67	34	3	25	45	Original Array
12	67	34	3	25	45	After 1. pass
12	34	67	3	25	45	After 2. pass
3	12	34	67	25	45	After 3. pass
3	12	25	34	67	45	After 4. pass
3	12	25	34	45	67	After 5. pass

Analysis of Algorithms

- How is the running time of an algorithm analyzed?
  - Based on the *input itself* and *input size*

- Input:
  - Sorting 100 numbers takes longer than sorting 5 numbers.
  - A sorting algorithm might takes different amounts of time on two inputs of the same size (e.g., assume one input is already sorted).

- Input Size:
  - Usually, the number of items in the input : n
  - For integer multiplication, it is the total number of bits in the two integers.

Types of Analysis

- **Best-Case**
  - Lower bound (i.e., minimum) on the running time for any input
- **Worst-Case** *(often guarantee)*
  - Upper bound (i.e., maximum) on the running time for any input
- **Average-Case**
  - Expected running time for any input, generally as bad as worst-case time

Running time

It is the number of primitive operations (steps) executed.

- Each line of pseudocode takes a constant amount of time.
- Execution of line i always takes the same time c_i.
- Assume that each line consists only of primitive operations.

The running time of an algorithm is:

$$\sum_{\text{all statements}} (\text{cost of statement}) \cdot (\text{number of times statement is executed})$$

Analysis of Insertion Sort

INSERTION-SORT(A, n)	*cost*	*times*
for $j = 2$ **to** n	c_1	n
$\quad key = A[j]$	c_2	$n - 1$
$\quad$ // Insert $A[j]$ into the sorted sequence $A[1\,..\,j-1]$.	0	$n - 1$
$\quad i = j - 1$	c_4	$n - 1$
$\quad$ **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
$\qquad A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$\qquad i = i - 1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$\quad A[i+1] = key$	c_8	$n - 1$

Analysis of Insertion Sort

INSERTION-SORT(A, n)	*cost*	*times*
for $j = 2$ **to** n	c_1	n
$key = A[j]$	c_2	$n - 1$
// Insert $A[j]$ into the sorted sequence $A[1 \mathinner{..} j-1]$.	0	$n - 1$
$i = j - 1$	c_4	$n - 1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
$A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$i = i - 1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$A[i+1] = key$	c_8	$n - 1$

$$
\begin{aligned}
T(n) \quad = \quad & c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) \\
& + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) \; .
\end{aligned}
$$

Best-Case Running Time

Assume the input is already sorted:

- Always find that *A[i]* $\leq$ *key* upon the first time **while** loop is run
- All t_j are 1.
- The running time is:

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)\ . \end{aligned}$$

- Can express *T(n)* as $an + b$ for constants *a* and *b* :

  => *T(n)* is a *linear function* of *n*

Worst-Case Running Time

Assume the input is in reverse sorted order:

- Always find that *A[i]* > *key* in the **while** loop test.
- Compare *key* with all elements to the left of the j^{th} position.
- The **while** loop reaches to 0, one more test after the j-1 test => $t_j = j$.
- The running time is:

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\
&\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\
&= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n \\
&\quad - (c_2 + c_4 + c_5 + c_8)\ .
\end{aligned}
$$

- Can express *T(n)* as $an^2 + bn + c$ for constants *a, b, c* :

  => *T(n)* is a *quadratic function* of *n*

Average-Case Running Time

Assume we randomly choose ***n*** number as the input for insertion sort:

- On average, the key in A[j] is less than half the elements in A[1…j-1] and it's greater than the other half => $t_j \approx (j/2)$.
- The average-case running time is approximately half of the worst-case running time, it's still a *quadratic function* of *n*.

Order of Growth

- Only consider the leading term of the formula for running time.
- Drop lower-order terms
- Ignore constant coefficient in the leading term
- For insertion sort, we already abstracted away the actual statement costs to conclude that the worst-case running time is $an^2 + bn + c$.
  - Drop lower-order terms => an^2.
  - Ignore constant coefficient => n^2.
- We cannot say that the worst-case running time $T(n)=n^2$. It only *grows like* n^2.
- So, the running time is $\Theta(n^2)$ to capture the notion that the *order of growth* is n^2.
- One algorithm is assumed to be more efficient if its worst-case running time has a smaller order of growth.

Next Week Topics

- Merge Sort
- Growth of Functions (Chapter 3-4)