2CS503 – Design and Analysis of Algorithms

Introduction

References

1. Charles E. Leiserson, Thomas H. Cormen, Ronald L. Rivest, Clifford Stein - Introduction to Algorithms, PHI

2. Ellis Horowitz, SartajSahni, Sanguthevar Rajasekharan, Fundamentals of Computer Algorithms, Galgotia.

3. Jean-Paul Tremblay and Paul G. Sorenson, An Introduction to Data Structures with Applications, Tata McGraw Hill

4. Karumanchi, Narasimha, Data Structures and Algorithms Made Easy, CareerMonk Publications.

Syllabus

Syllabus:	**Teaching Hours**
Unit I	**2**
Elementary Algorithmic: Efficiency of Algorithms, Average & worst-case analysis, Elementary Operation	
Unit II	**4**
Analysis Techniques: Empirical, mathematical, Asymptotic analysis and related unconditional and conditional notations	
Analysis of Algorithms: Analysing control structures: sequencing, “For” loops, Recursive calls, “While” and “repeat” loops, Amortized analysis	
Unit III	**4**
Solving Recurrences: Intelligent guesswork, Homogeneous recurrences, Inhomogeneous Recurrences, Change of variable, Range transformations, Master Theorem, Recurrence Tree.	
Unit IV	**7**
Data Structures: Heaps, Binomial heaps, Disjoint set structures.	
Greedy Algorithms: Graphs: Minimum spanning trees-Kruskal’s algorithm, Prim’s algorithm, Graphs: Shortest path algorithms.	
Unit V	**8**
Divide-and-Conquer: Multiplying large integers, Binary search, sorting: sorting by merging, quick sort, finding the median, Matrix multiplication, Exponentiation, approaches using recursion, Memory functions.	
Dynamic Programming: Principles of optimality, Various applications using dynamic programming.	
Unit VI	**5**
Branch and Bound, Backtracking: Design of some classical problems using branch and bound and Backtracking approaches.	
Randomized and Approximation Algorithms: Design of some classical problems using Randomized and Approximation Algorithms.	

Resources

All resources will be available on Moodle.

Teaching & Evaluation Scheme

Teaching Scheme:

Theory	Tutorial	Practical	Credits
2	1	2	4

	LPW	SEE	CE
Exam Duration	Continuous Evaluation + 2 Hrs. End Semester Exam	3.0 Hrs.	Continuous Evaluation
Component Weightage	0.2 (0.75+0.25)	0.4	0.4 (0.35+0.35+0.3)

Assignments in CE:

MOOC

Introduction

- What is an algorithm?
  - An algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output. An algorithm is thus a sequence of computational steps that transform the input into the output.

Introduction

- What is an algorithm?
  - We can also view an algorithm as a tool for solving a well-specified computational problem. The statement of the problem specifies in general terms the desired input/output relationship. The algorithm describes a specific computational procedure for achieving that input/output relationship.

Introduction

- ## What is an algorithm?
  - We can also view an algorithm as a tool for solving a well-specified computational problem. The statement of the problem specifies in general terms the desired input/output relationship. The algorithm describes a specific computational procedure for achieving that input/output relationship.

  - An algorithmic problem is specified by describing the complete set of instances it must work on and what desired properties the output must have.

Introduction

- What is an algorithm?
  - We can also view an algorithm as a tool for solving a well-specified computational problem. The statement of the problem specifies in general terms the desired input/output relationship. The algorithm describes a specific computational procedure for achieving that input/output relationship.
  - An algorithmic problem is specified by describing the complete set of instances it must work on and what desired properties the output must have.
  - For example, we might need to sort a sequence of numbers into nondecreasing order.

Introduction

- What is an algorithm?
  - Here is how we formally define the sorting problem:

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.

Output: A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \le a'_2 \le \cdots \le a'_n$.

Introduction

- What is an algorithm?
  - Here is how we formally define the sorting problem:

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.

Output: A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \le a'_2 \le \cdots \le a'_n$.

- Instance of a Problem
  - For example, given the input sequence (31, 41, 59, 26, 41, 58), a sorting algorithm returns as output the sequence (26, 31, 41, 41, 58, 59). Such an input sequence is called an instance of the sorting problem.

Introduction

- What is an algorithm?
  - Here is how we formally define the sorting problem:

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.

Output: A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \le a'_2 \le \cdots \le a'_n$.

- Instance of a Problem
  - For example, given the input sequence (31, 41, 59, 26, 41, 58), a sorting algorithm returns as output the sequence (26, 31, 41, 41, 58, 59). Such an input sequence is called an instance of the sorting problem.
  - In general, an instance of a problem consists of the input (satisfying whatever constraints are imposed in the problem statement) needed to compute a solution to the problem.

Introduction

- When can we say that the algorithm is correct?

Introduction

- When can we say that the algorithm is correct?
    - An algorithm is said to be correct if, for every input instance, it halts with the correct output.

Introduction

- When can we say that the algorithm is correct?
  - An algorithm is said to be correct if, for every input instance, it halts with the correct output.
  - We say that a correct algorithm solves the given computational problem.

Introduction

- When can we say that the algorithm is correct?
  - An algorithm is said to be correct if, for every input instance, it halts with the correct output.
  - We say that a correct algorithm solves the given computational problem.
  - An incorrect algorithm might not halt at all on some input instances, or it might halt with an incorrect answer.

Introduction

- When can we say that the algorithm is correct?
  - An algorithm is said to be correct if, for every input instance, it halts with the correct output.
  - We say that a correct algorithm solves the given computational problem.
  - An incorrect algorithm might not halt at all on some input instances, or it might halt with an incorrect answer.
  - Contrary to what you might expect, incorrect algorithms can sometimes be useful, if we can control their error rate.

Introduction

- When can we say that the algorithm is correct?
  - An algorithm is said to be correct if, for every input instance, it halts with the correct output.
  - We say that a correct algorithm solves the given computational problem.
  - An incorrect algorithm might not halt at all on some input instances, or it might halt with an incorrect answer.
  - Contrary to what you might expect, incorrect algorithms can sometimes be useful, if we can control their error rate.
  - Ordinarily, however, we shall be concerned only with correct algorithms.

Introduction

- How can algorithm be specified?
  - An algorithm can be specified in English, as a computer program, or even as a hardware design.

Introduction

- How can algorithm be specified?
  - An algorithm can be specified in English, as a computer program, or even as a hardware design.
  - The only requirement is that the specification must provide a precise description of the computational procedure to be followed.

Introduction

- Types of problem which can be solved by algorithms
    - Managing, manipulating & retrieving the voluminous data

Introduction

- Types of problem which can be solved by algorithms
  - Managing, manipulating & retrieving the voluminous data
  - Shortest Route finding

Introduction

- Types of problem which can be solved by algorithms
  - Managing, manipulating & retrieving the voluminous data
  - Shortest Route finding
  - Public-key cryptography and digital signatures

Introduction

➢ Types of problem which can be solved by algorithms

- Managing, manipulating & retrieving the voluminous data
- Shortest Route finding
- Public-key cryptography and digital signatures
- Resource scheduling, etc.
  - A political candidate may want to determine where to spend money buying campaign advertising in order to maximize the chances of winning an election.

Introduction

- Types of problem which can be solved by algorithms
  - Managing, manipulating & retrieving the voluminous data
  - Shortest Route finding
  - Public-key cryptography and digital signatures
  - Resource scheduling, etc.
    - A political candidate may want to determine where to spend money buying campaign advertising in order to maximize the chances of winning an election.
    - An airline may wish to assign crews to flights in the least expensive way possible, making sure that each flight is covered and that government regulations regarding crew scheduling are met.

Introduction

- Types of problem which can be solved by algorithms
  - Managing, manipulating & retrieving the voluminous data
  - Shortest Route finding
  - Public-key cryptography and digital signatures
  - Resource scheduling, etc.
    - A political candidate may want to determine where to spend money buying campaign advertising in order to maximize the chances of winning an election.
    - An airline may wish to assign crews to flights in the least expensive way possible, making sure that each flight is covered and that government regulations regarding crew scheduling are met.
    - An Internet service provider may wish to determine where to place additional resources in order to serve its customers more effectively.

Introduction

- Types of problem which can be solved by algorithms
  - Managing, manipulating & retrieving the voluminous data
  - Shortest Route finding
  - Public-key cryptography and digital signatures
  - Resource scheduling, etc.
    - A political candidate may want to determine where to spend money buying campaign advertising in order to maximize the chances of winning an election.
    - An airline may wish to assign crews to flights in the least expensive way possible, making sure that each flight is covered and that government regulations regarding crew scheduling are met.
    - An Internet service provider may wish to determine where to place additional resources in order to serve its customers more effectively.
    - All of these are examples of problems that can be solved using linear programming

Efficiency of an Algorithm – Does it Matter?

Efficiency of an Algorithm – Does it Matter?

- Different algorithms devised to solve the same problem often differ dramatically in their efficiency.

Efficiency of an Algorithm – Does it Matter?

- Different algorithms devised to solve the same problem often differ dramatically in their efficiency.

- These differences can be much more significant than differences due to hardware and software.

Efficiency of an Algorithm – Does it Matter?

- Different algorithms devised to solve the same problem often differ dramatically in their efficiency.

- These differences can be much more significant than differences due to hardware and software.

- As an example, we will see two algorithms for sorting.

Efficiency of an Algorithm – Does it Matter?

- Different algorithms devised to solve the same problem often differ dramatically in their efficiency.

- These differences can be much more significant than differences due to hardware and software.

- As an example, we will see two algorithms for sorting.

- The first, known as insertion sort, takes time roughly equal to c_1n^2 to sort n items, where c_1 is a constant that does not depend on n. That is, it takes time roughly proportional to n^2.

Efficiency of an Algorithm – Does it Matter?

- Different algorithms devised to solve the same problem often differ dramatically in their efficiency.

- These differences can be much more significant than differences due to hardware and software.

- As an example, we will see two algorithms for sorting.

- The first, known as insertion sort, takes time roughly equal to c_1n^2 to sort n items, where c_1 is a constant that does not depend on n. That is, it takes time roughly proportional to n^2.

- The second, merge sort, takes time roughly equal to c_2 n lg n, where lg n stands for $\log_2$ n and c_2 is another constant that also does not depend on n.

Efficiency of an Algorithm – Does it Matter?

- Insertion sort typically has a smaller constant factor than merge sort, so that $c1 < c2$.

Efficiency of an Algorithm – Does it Matter?

- Insertion sort typically has a smaller constant factor than merge sort, so that $c1 < c2$.

- We shall see that the constant factors can have far less of an impact on the running time than the dependence on the input size n.

Efficiency of an Algorithm – Does it Matter?

- Insertion sort typically has a smaller constant factor than merge sort, so that c1 < c2.

- We shall see that the constant factors can have far less of an impact on the running time than the dependence on the input size n.

- Let's write insertion sort's running time as c_1 n n and merge sort's running time as c_2 n lg n.

Efficiency of an Algorithm – Does it Matter?

- Insertion sort typically has a smaller constant factor than merge sort, so that $c1 < c2$.

- We shall see that the constant factors can have far less of an impact on the running time than the dependence on the input size n.

- Let's write insertion sort's running time as c_1 n n and merge sort's running time as c_2 n lg n.

- Then we see that where insertion sort has a factor of n in its running time, merge sort has a factor of lg n, which is much smaller. (For example, when n = 1000, lg n is approximately 10, and when n equals one million, lg n is approximately only 20.)

Efficiency of an Algorithm – Does it Matter?

- Although insertion sort usually runs faster than merge sort for small input sizes, once the input size n becomes large enough, merge sort's advantage of lg n vs. n will more than compensate for the difference in constant factors.

Efficiency of an Algorithm – Does it Matter?

- Although insertion sort usually runs faster than merge sort for small input sizes, once the input size n becomes large enough, merge sort's advantage of lg n vs. n will more than compensate for the difference in constant factors.

- No matter how much smaller c_1 is than c_2, there will always be a crossover point beyond which merge sort is faster.

Efficiency of an Algorithm – Does it Matter?

- Different algorithms, constants & hardware proficiency
  - Problem: Sort 10 million numbers
  - Computer A: insertion sort, 10bi/s, c_1=2
  - Computer B: merge sort, 10mi/s, c_2=50

  - Computer A: $$\frac{2 \cdot (10^7)^2 \text{ instructions}}{10^{10} \text{ instructions/second}} = 20{,}000 \text{ seconds (more than 5.5 hours)}$$

  - Computer B: $$\frac{50 \cdot 10^7 \lg 10^7 \text{ instructions}}{10^7 \text{ instructions/second}} \approx 1163 \text{ seconds (less than 20 minutes)}$$

Analyzing Algorithms

- What is to be analyzed?

Analyzing Algorithms

- What is to be analyzed?
  - memory, communication bandwidth, hardware, computational time

Analyzing Algorithms

- What is to be analyzed?
  - memory, communication bandwidth, hardware, computational time
- One processor Random Access Machine (RAM) model
  - Implementation Technology

Analyzing Algorithms

- What is to be analyzed?
  - memory, communication bandwidth, hardware, computational time
- One processor Random Access Machine (RAM) model
  - Implementation Technology
  - Instructions are executed one after another, with no concurrent operations

Analyzing Algorithms

- What is to be analyzed?
  - memory, communication bandwidth, hardware, computational time
- One processor Random Access Machine (RAM) model
  - Implementation Technology
  - Instructions are executed one after another, with no concurrent operations
  - Instructions of RAM model
    - Arithmetic (such as add, subtract, multiply, divide, remainder, floor, ceiling)
    - Data movement (load, store, copy) and control (conditional and unconditional branch, subroutine call and return)

Analyzing Algorithms

- What is to be analyzed?
  - memory, communication bandwidth, hardware, computational time
- One processor Random Access Machine (RAM) model
  - Implementation Technology
  - Instructions are executed one after another, with no concurrent operations
  - Instructions of RAM model
    - Arithmetic (such as add, subtract, multiply, divide, remainder, floor, ceiling)
    - Data movement (load, store, copy) and control (conditional and unconditional branch, subroutine call and return)
  - Each such instruction takes constant amount of time – Primitive / Elementary Operations

Analyzing Algorithms

- Input Size
  - The best notion for input size depends on the problem being studied.

Analyzing Algorithms

- Input Size
  - The best notion for input size depends on the problem being studied.

  - For many problems, such as **sorting** or computing discrete Fourier transforms, the most natural measure is the number of items in the input—for example, the array size n for sorting.

Analyzing Algorithms

- Input Size
  - The best notion for input size depends on the problem being studied.
  - For many problems, such as **sorting** or computing discrete Fourier transforms, the most natural measure is the number of items in the input—for example, the array size n for sorting.
  - For many other problems, such as **multiplying two integers**, the best measure of input size is the total number of bits needed to represent the input in ordinary binary notation.

Analyzing Algorithms

- Input Size
  - The best notion for input size depends on the problem being studied.
  - For many problems, such as **sorting** or computing discrete Fourier transforms, the most natural measure is the number of items in the input—for example, the array size n for sorting.
  - For many other problems, such as **multiplying two integers**, the best measure of input size is the total number of bits needed to represent the input in ordinary binary notation.
  - Sometimes, it is more appropriate to describe the size of the input with two numbers rather than one. For instance, if the input to an algorithm is a **graph**, the input size can be described by the numbers of vertices and edges in the graph.

Analyzing Algorithms

- Input Size
  - The best notion for input size depends on the problem being studied.

  - For many problems, such as **sorting** or computing discrete Fourier transforms, the most natural measure is the number of items in the input—for example, the array size n for sorting.

  - For many other problems, such as **multiplying two integers**, the best measure of input size is the total number of bits needed to represent the input in ordinary binary notation.

  - Sometimes, it is more appropriate to describe the size of the input with two numbers rather than one. For instance, if the input to an algorithm is a **graph**, the input size can be described by the numbers of vertices and edges in the graph.

  - We shall indicate which input size measure is being used with each problem we study.

Analyzing Algorithms

- Running Time
  - The running time of an algorithm on a particular input is the number of primitive operations or “steps” executed.

Analyzing Algorithms

- Running Time
  - The running time of an algorithm on a particular input is the number of primitive operations or "steps" executed.
  - It is convenient to define the notion of step so that it is as machine-independent as possible.

Analyzing Algorithms

- Running Time
  - The running time of an algorithm on a particular input is the number of primitive operations or "steps" executed.
  - It is convenient to define the notion of step so that it is as machine-independent as possible.
  - For the moment, let us adopt the following view.

Analyzing Algorithms

- Running Time
  - The running time of an algorithm on a particular input is the number of primitive operations or “steps” executed.
  - It is convenient to define the notion of step so that it is as machine-independent as possible.
  - For the moment, let us adopt the following view.
  - A constant amount of time is required to execute each line of our pseudocode.

Analyzing Algorithms

- Running Time
  - The running time of an algorithm on a particular input is the number of primitive operations or “steps” executed.
  - It is convenient to define the notion of step so that it is as machine-independent as possible.
  - For the moment, let us adopt the following view.
  - A constant amount of time is required to execute each line of our pseudocode.
  - One line may take a different amount of time than another line, but we shall assume that each execution of the i^{th} line takes time c_i, where c_i is a constant.

Analyzing Algorithms

- Running Time
  - The running time of an algorithm on a particular input is the number of primitive operations or "steps" executed.
  - It is convenient to define the notion of step so that it is as machine-independent as possible.
  - For the moment, let us adopt the following view.
  - A constant amount of time is required to execute each line of our pseudocode.
  - One line may take a different amount of time than another line, but we shall assume that each execution of the i^{th} line takes time c_i, where c_i is a constant.
  - This viewpoint is in keeping with the RAM model, and it also reflects how the pseudocode would be implemented on most actual computers.

Analyzing Algorithms

- Comparison of Running Time

	lg n	n	n lg n	n^2	n^3	2^n
1	0.0	1.0	0.0	1.0	1.0	2.0
2	1.0	2.0	2.0	4.0	8.0	4.0
5	2.3	5.0	11.6	25.0	125.0	32.0
10	3.3	10.0	33.2	100.0	1000.0	1024.0
15	3.9	15.0	58.6	225.0	3375.0	32768.0
20	4.3	20.0	86.4	400.0	8000.0	1048576.0
30	4.9	30.0	147.2	900.0	27000.0	1073741824.0
40	5.3	40.0	212.9	1600.0	64000.0	1099511627776.0
50	5.6	50.0	282.2	2500.0	125000.0	1125899906842620.0
60	5.9	60.0	354.4	3600.0	216000.0	1152921504606850000.0
70	6.1	70.0	429.0	4900.0	343000.0	1180591620717410000000.0
80	6.3	80.0	505.8	6400.0	512000.0	1208925819614630000000000.0
90	6.5	90.0	584.3	8100.0	729000.0	1237940039285380000000000000.0
100	6.6	100.0	664.4	10000.0	1000000.0	1267650600228230000000000000000.0

Analyzing Algorithms

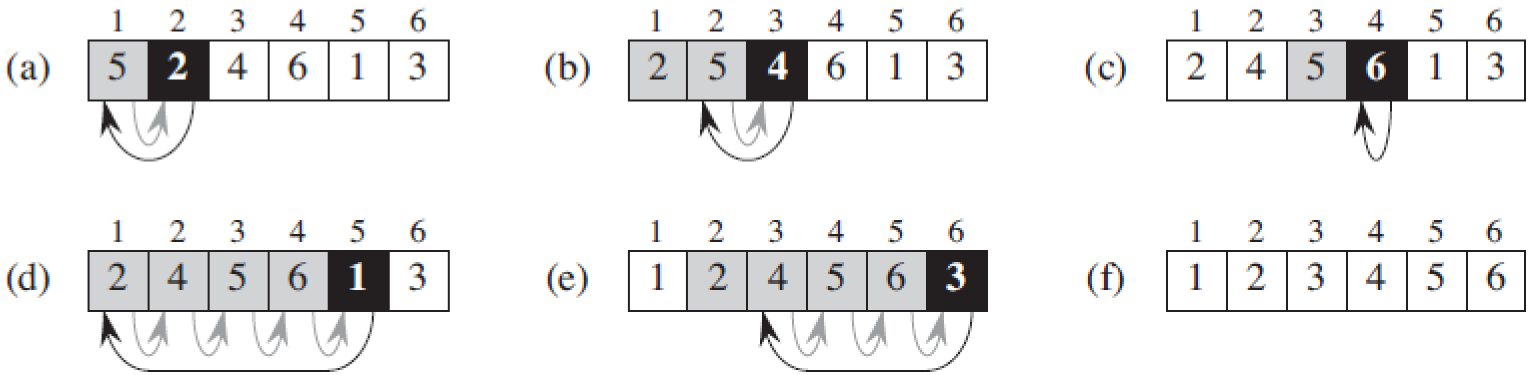
- Space Complexity
    - Space Complexity of an algorithm is total space taken by the algorithm with respect to the input size.

Analyzing Algorithms – Which one is better?

```
largest = a
if b > largest then
   largest = b
end if
if c > largest then
   largest = c
end if
if d > largest then
   largest = d
end if
return largest
```

```
if a > b then
   if a > c then
      if a > d then
         return a
      else
         return d
      end if
   else
      if c > d then
         return c
      else
         return d
      end if
   end if
else
   if b > c then
      if b > d then
         return b
      else
         return d
      end if
   else
      if c > d then
         return c
      else
         return d
      end if
   end if
end if
```

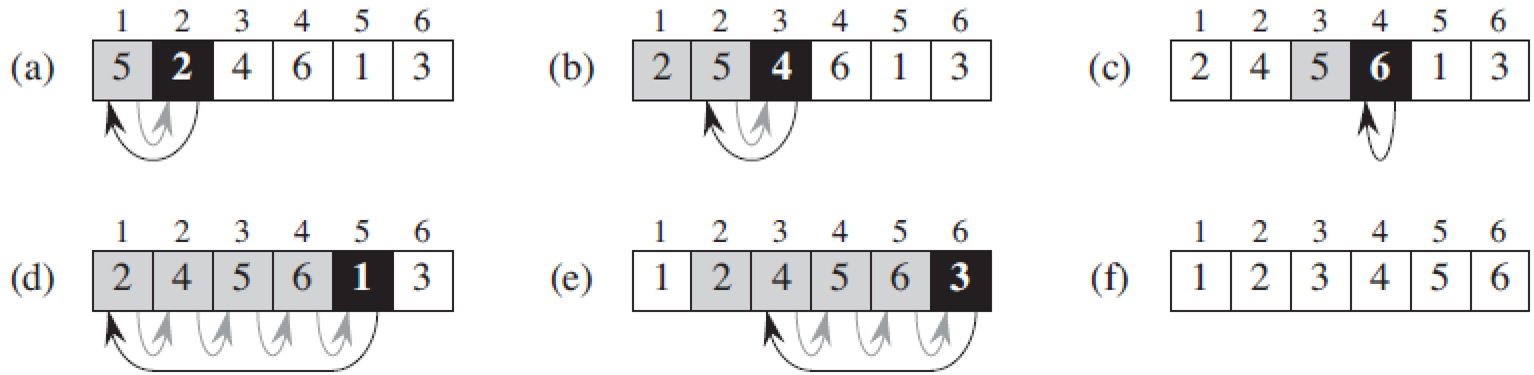
Analyzing Insertion Sort



INSERTION-SORT(A)

```
for j = 2 to A.length
    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
```

Analyzing Insertion Sort



INSERTION-SORT(A)	*cost*	*times*
for $j = 2$ **to** $A.length$	c_1	n
$key = A[j]$	c_2	$n - 1$
// Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	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	$n - 1$

Analyzing Insertion Sort

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

Best Case Running Time (i.e. Array is already Sorted):

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

Worst Case Running Time (i.e. Array is Sorted in Reverse Order)

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

and

$$
\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}
$$

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

Analyzing Insertion Sort

Average Case Running Time

On average, half the elements in A[1 .. j – 1] are less than A[j], and half the elements are greater. On average, therefore, we check half of the subarray A[1 .. j – 1], and so t_j is about j/2. The resulting average-case running time turns out to be a quadratic function of the input size, just like the worst-case running time.

Analyzing Insertion Sort

- Order of Growth
  - We used some simplifying abstractions to ease our analysis of the INSERTION-SORT procedure.

Analyzing Insertion Sort

- Order of Growth
  - We used some simplifying abstractions to ease our analysis of the INSERTION-SORT procedure.

  - First, we ignored the actual cost of each statement, using the constants c_i to represent these costs.

Analyzing Insertion Sort

- Order of Growth
  - We used some simplifying abstractions to ease our analysis of the INSERTION-SORT procedure.

  - First, we ignored the actual cost of each statement, using the constants c_i to represent these costs.

  - Then, we observed that even these constants give us more detail than we really need: we expressed the worst-case running time as $an^2 + bn + c$ for some constants a, b, and c that depend on the statement costs c_i .

Analyzing Insertion Sort

- Order of Growth
  - We used some simplifying abstractions to ease our analysis of the INSERTION-SORT procedure.

  - First, we ignored the actual cost of each statement, using the constants c_i to represent these costs.

  - Then, we observed that even these constants give us more detail than we really need: we expressed the worst-case running time as $an^2 + bn + c$ for some constants a, b, and c that depend on the statement costs c_i .

  - We thus ignored not only the actual statement costs, but also the abstract costs c_i .

Analyzing Insertion Sort

- Order of Growth
  - We shall now make one more simplifying abstraction: it is the rate of growth, or order of growth, of the running time that really interests us.

Analyzing Insertion Sort

- Order of Growth
  - We shall now make one more simplifying abstraction: it is the rate of growth, or order of growth, of the running time that really interests us.

  - We therefore consider only the leading term of a formula (e.g., an^2), since the lower-order terms are relatively insignificant for large values of n.

Analyzing Insertion Sort

- Order of Growth
  - We shall now make one more simplifying abstraction: it is the rate of growth, or order of growth, of the running time that really interests us.
  - We therefore consider only the leading term of a formula (e.g., an^2), since the lower-order terms are relatively insignificant for large values of n.
  - We also ignore the leading term's constant coefficient, since constant factors are less significant than the rate of growth in determining computational efficiency for large inputs.

Analyzing Insertion Sort

- Order of Growth
  - We shall now make one more simplifying abstraction: it is the rate of growth, or order of growth, of the running time that really interests us.
  - We therefore consider only the leading term of a formula (e.g., an^2), since the lower-order terms are relatively insignificant for large values of n.
  - We also ignore the leading term's constant coefficient, since constant factors are less significant than the rate of growth in determining computational efficiency for large inputs.
  - For insertion sort, when we ignore the lower-order terms and the leading term's constant coefficient, we are left with the factor of n^2 from the leading term.

Analyzing Insertion Sort

- Order of Growth
  - We write that insertion sort has a worst-case running time of $\theta(n^2)$ (pronounced "theta of n-squared").

  - We usually consider one algorithm to be more efficient than another if its worst-case running time has a lower order of growth.

  - Due to constant factors and lower order terms, an algorithm whose running time has a higher order of growth might take less time for small inputs than an algorithm whose running time has a lower order of growth.

  - But for large enough inputs, $\theta(n^2)$ algorithm, for example, will run more quickly in the worst case than a , $\theta(n^3)$ algorithm.

Disclaimer

- The presentation is not original and its is prepared from various sources for teaching purpose only.