CSCE310: Data Structures and Algorithms

www.cse.unl.edu/~choueiry/S01-310/

Acknowledgment: Most of the material presented here and throughout the course is borrowed (and freely modified) from the course notes of Dr. Charles Cusack (cusack@cse.unl.edu) and other courses on the Web.

Berthe Y. Choueiry (Shu-we-ri)
Ferguson Hall, Room 104
choueiry@cse.unl.edu, Tel: (402)472-5444

Goal:

Understanding of

- **computational** problems
- algorithms for solving them

Algorithms specified in **pseudocode**

Two examples: insertion sort, merge sort (divide+conquer)

Algorithm (El-Kowarizimi, 9th century Persian mathematician)

A tool for solving well-specified computational problems

Algorithm

- well-defined computational procedure
- takes a value or set of values as input
- produces a value, or set of values as output

Example: divide (1347, 8)

Procedure: divide

Input: 1347, 8

Output: result 168, remainder 3

Sorting problem

Fundamental operation in CS

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_3 \rangle$
Output: A permutation/reodering $\langle a'_1, a'_2, \ldots, a'_3 \rangle$ of the input such that $a'_1 \leq a_2 \leq \ldots \leq a'_3 \rangle$

Example:
Input: Instance of the sorting problem, $\langle 31, 41, 59, 26, 41, 58 \rangle$
Output: $\langle 26, 31, 41, 41, 58, 59 \rangle$

Many sorting algorithms exist, differ in performance

Correct algorithm

- for every input instance, halts/terminates with the correct output
- i.e., solves the computational problem

Incorrect algorithm

- does not halt for some instances, or
- halts with an incorrect output

Note: sometimes useful, if the error rate can be controlled

Specification of an algorithm

precise description of the computational procedure to be followed

It can be

- In English, pseudocode, C, etc.
- computer program (software)
- hardware design (hardware implementation)

Example: `Insertion-sort`

Good algorithm for small number of elements

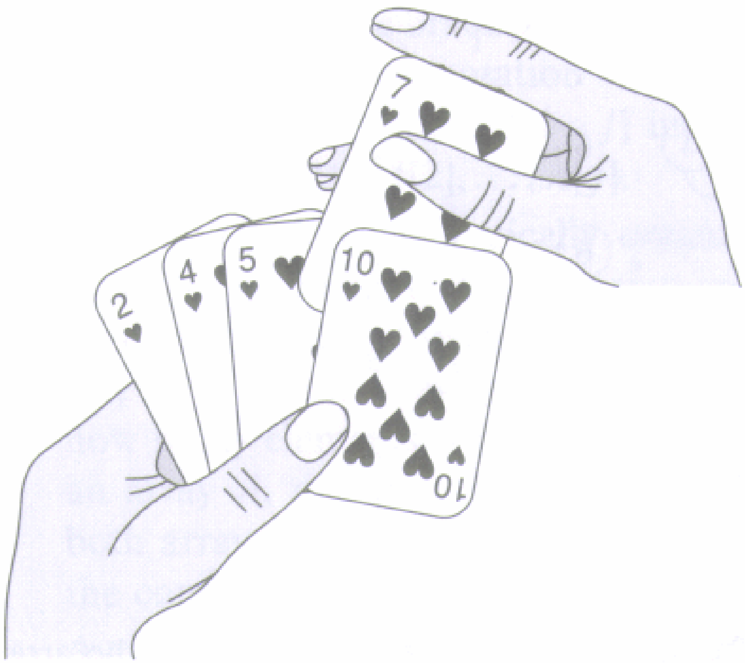


Figure 1.1 Sorting a hand of cards using insertion sort.

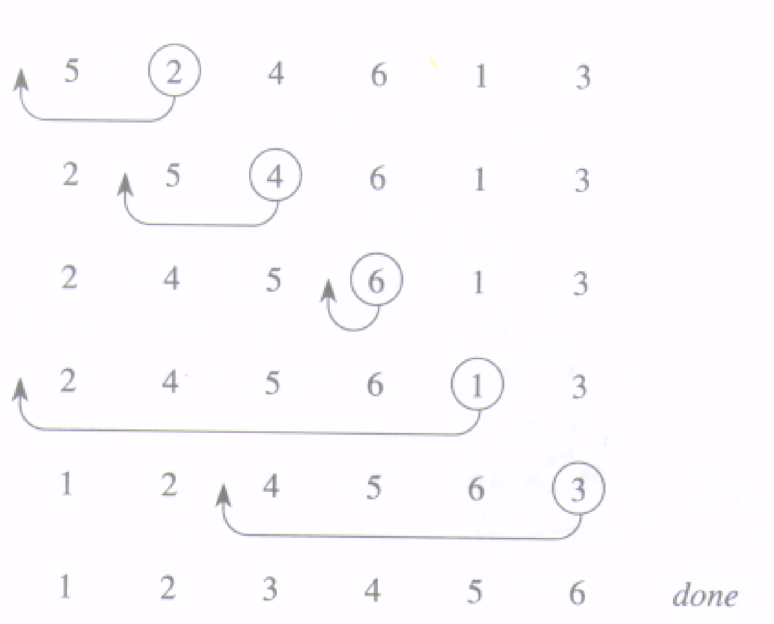
Suppose you are playing cards.
Cards piled on the table,
Start with an empty hand,
Take one card at at time,
Insert it in the correct position by comparing it to existing cards, from right to left (or left to right)

Pseudocode: procedure `Insertion-sort`

Insert algorithm

- Input parameter: array $A[1..n]$, of the sequence to sort
- $length[A] = n$
- Elements are sorted in place, rearranged within the array
- At the end of the procedure, array contains the sorted sequence

Execute for $A = \langle 5, 2, 4, 6, 1, 3 \rangle$ on board



5 2 4 6 1 3

2 5 4 6 1 3

2 4 5 6 1 3

2 4 5 6 1 3

1 2 4 5 6 3

1 2 3 4 5 6 *done*

Pseudocode: procedure `Insertion-sort`

Insert algorithm

j: current card

$A[1..j-1]$ currently sorted hand

$A[j+1..n]$ cards still on table

line 2: $A[j]$ is picked out

lines 4–7: then all elements (j-1) are shifted right

line 8: picked element is inserted

Pseudocode conventions (I)

- Indentation indicates block structure (while, for, if-then-else)
  one could use begin-end to define block
- Looping constructs: while, for, repeat
  Conditional constructs: if-then-else, when, unless
- Comment: $\triangleright$
- Assignment: $i \leftarrow j$. Multiple assignments: $i \leftarrow j \leftarrow e$
- Variables are local to procedures: i, j, key
- Array elements accessed by `Array-name[position]`
  Subarray $A[1..j] = A[1], A[2], \ldots , A[j]$

Pseudocode conventions (II)

- Compound data organized into objects.
  Objects have attributes/fields
  Field accessed by `field-name[object-name]`
  Example: object, A, an array. Attribute: *length*.
  It is accessed with $length[A]$

- A variable representing an object treated as a **pointer** to the data representing the object.
  Example: $y \leftarrow x$, x, y are pointers to the same location. If you set a field of x to a value, you are setting also the field of y.

- Parameters to procedures are passed by **value**.
  Procedure makes it own copy of parameter, the change is not visible to calling routine.
  However, for objects, the pointer is copied, not the fields.
  Changes to the field are visible to calling procedure.

Analyzing algorithms: predict resources needed

Sometimes: memory, communication bandwidth, logic gates, etc.
Usually: computational time

Model of implementation technology: generic one-processor **random-access machine** (RAM):
— instructions executed one after another, serially
— no concurrent operations

Expression of resources needed:
— simple to write and manipulate
— shows important characteristics of an algorithm's resource requirements
— ignores tedious details

Example: `Insertion-sort`

Time needed depends on input:

- size: 3 elements vs. 1000 elements
- ‘state of input:’ sorted, nearly sorted, etc.

Input size: running time

’State of input:’ best, average, worst case.

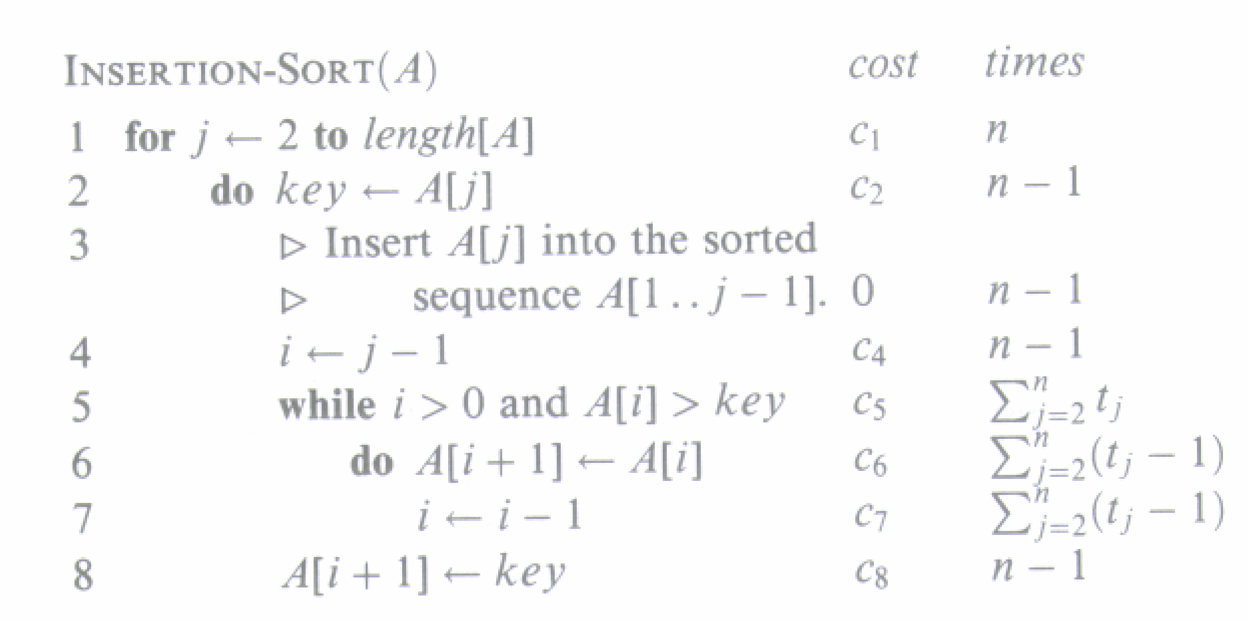
Input size:

- Usually: number of items in input
  1 number: size of array
  2 numbers: number of vertices and edge in a graph
- Sometimes: total number of bits needed to represent input

Running time: number of steps, operations executed

- line i of code: constant cost c_i
- 1 line calling a subroutine: constant time
  but executing subroutine may cost more

Pseudocode: procedure `Insertion-sort`



```
INSERTION-SORT($A$)                                  cost    times
1  for $j \leftarrow 2$ to $length[A]$                $c_1$   $n$
2        do $key \leftarrow A[j]$                     $c_2$   $n-1$
3           ▷ Insert $A[j]$ into the sorted
            ▷       sequence $A[1\,..\,j-1]$.        0       $n-1$
4           $i \leftarrow j-1$                        $c_4$   $n-1$
5           while $i > 0$ and $A[i] > key$            $c_5$   $\sum_{j=2}^{n} t_j$
6                do $A[i+1] \leftarrow A[i]$          $c_6$   $\sum_{j=2}^{n}(t_j-1)$
7                   $i \leftarrow i-1$                $c_7$   $\sum_{j=2}^{n}(t_j-1)$
8           $A[i+1] \leftarrow key$                   $c_8$   $n-1$
```

line 1, 2, 3, 4, 8: straightforward

t_j: number of times while-loop executed for a j

$T(n) = ..$ *on board*

Procedure `Insertion-sort`

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

- Best case: A sorted, $t_j = 1$.

  $T(n) = c_1 n + (c_2 + c_4 + c_5 + c_8)(n-1)$

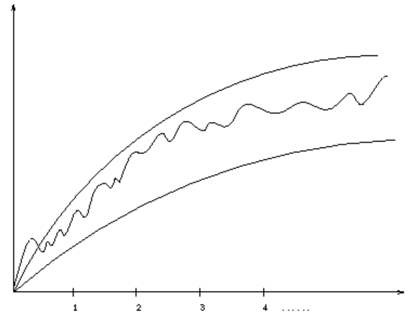
  Running time is a linear function of n.

- Worst case: A in reverse order, we have to move all the sorted elements, $t_j = j$

  Knowing that $\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$ and $\sum_{j=2}^{n}(j-1) = \frac{n(n-1)}{2}$
  (derive from $\sum_{j=1}^{n} = n(n+1)/2$)
  $T(n) = an^2 + bn + c$

  Running time is a quadratic function of n.

Running time



Usually: we concentrate on worst-case running time

- Worst-case is an upper bound for any input.
  A guarantee that algorithm won't run longer
- For some algorithms, worst-case is frequent.
  Example: searching a database, worst-case occurs when item is not in Database. Not uncommon in many applications.
- Average case, often 'as bad' as worst case.
  Example: in `Insertion-sort`, on average we need to move half the elements in $A[1..j-1]$, so $tj = j/2$, average case remains quadratic in n

Order of growth

`Insertion-sort`: worst-case: $T(n) = an^2 + bn + c$
with a, b, c depending on $c_1, c_2, \ldots, c_8$

For order of growth:

- we consider only the highest-degree term:

  the growth of the terms of lower degrees can be neglected with respect to the growth of the term of the highest degree.

  worst-case: an^2

- we ignore the constant coefficient (less significant for large input) worst-case running time $\Theta(n^2)$

Exercise: 1.2-5

Comparing two algorithms:

One is more efficient than another if the worst-case running time has a lower order of growth
Example: $\Theta(n^2)$, $\Theta(n^3)$

This may not hold for small input, but it does for large ones
Example: $\Theta(n^2)$, $\Theta(n^3)$ vs. $\Theta(2^n)$

(Side note: An algorithm is said to be efficient, if the order of growth of its worst-case running time is polynomial)

Designing algorithms

- Incremental approach
  example: `Insertion-sort`
- Divide and conquer (by recursive calls)
  example: merge sort

Divide and conquer

The algorithm calls itself recursively one or more times to deal with closely related subproblems (divide).

Three steps:

1. **Divide**: the problem into similar subproblems, but smaller in size
2. **Conquer**: the subproblems by solving them recursively. However, is subproblem is small enough, solve it in a straightforward manner
3. **Combine**: the solutions to subproblems into a solution to initial problem

Example: `Merge-sort`

1. **Divide**: n-element sequence to be sorted into two subsequences of $n/2$ elements each
2. **Conquer**: sort the two subsequences by calling recursively `Merge-sort`
3. **Combine**: merge the two sorted subsequences to produce the sorted answer

We will divide until we get sequences of length 1,
at this point, all sequences (of length 1) are sorted

Key operation: merging of two sorted sequences in the combine step

Merging in `Merge-sort`: `Merge`(A, p, q, r)

Input: A: array, p, q, r: indices in the array $p \leq q < r$
We assume $A[p..q]$ and $A[q+1..r]$ already sorted
Output: $A[p..r]$ sorted

Analogy: two piles P_1, P_2 of n cards, face up on the table
each of them is sorted, smallest cards on top,
you need to generate one pile that is sorted, face down on the table

Basic step: look on cards on top of P_1 and P_2
choose the smaller of the two
and move it face down on the output pile

Repeat basic step: each step takes constant time, we perform at most n basic steps, $\Theta(?)$

`Merge`(A, p, q, r) is called as a subroutine in `Merge-sort`

`Merge-sort`(A, p, r)

Insert merge sort

To sort $A = \langle A[1], A[2], \ldots, A[n] \rangle$,
we call `Merge-sort`$(A, 1, n)$ with $n = length[A]$

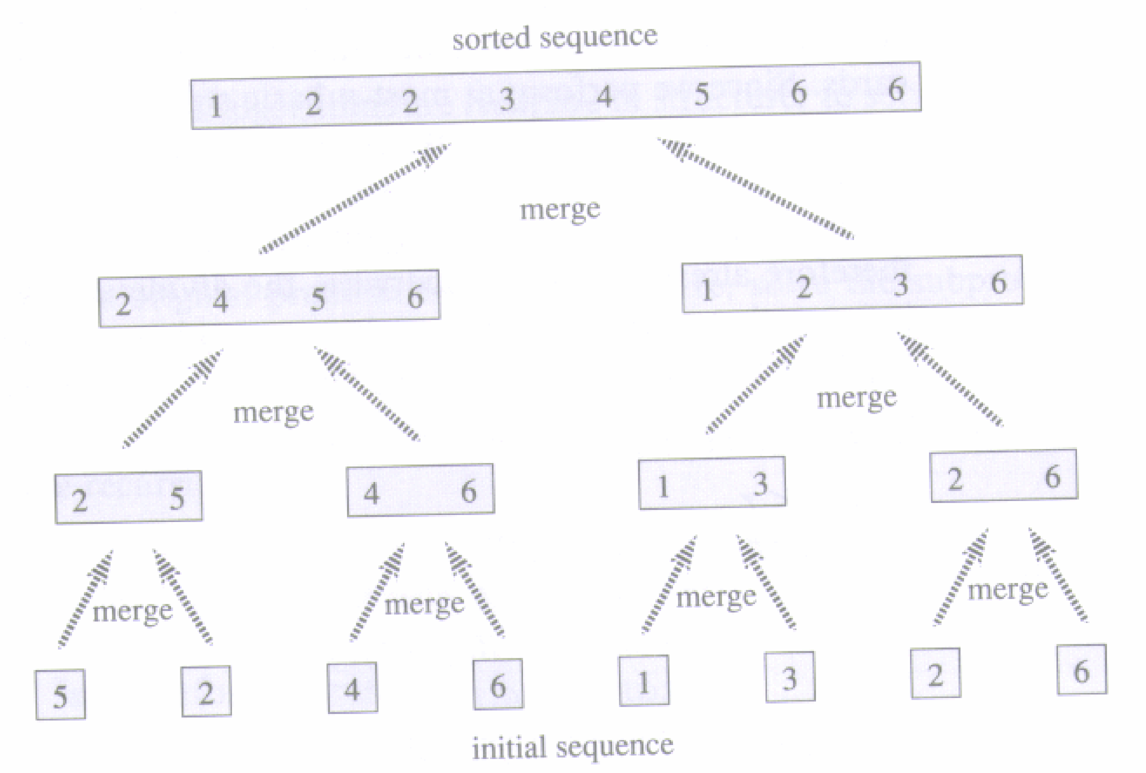
Analysis of: `Merge-sort`

When n is a power of 2, `Merge-sort`

merges pairs of 1-item sequences to form sorted 2-item sequences,

which it merges to form sorted 4-item sequences, etc.

until it merges two $n/2$-item sequences



Analysis of Divide-&-Conquer algorithms (I)

Running time of is determined by **recurrence equation** (recurrence)

Recurrence equation describes the running time on problem of input size n in terms of running time on smaller inputs

Recurrence relations is then manipulated to provide bounds on performance of algorithm

Analysis of Divide-&-Conquer algorithms (II)

Let: n: input size, $T(n)$ is running time

- If a (sub)problem is small enough $n \leq c$,
  solution is straightforward, takes constant time $\theta(1)$
- Suppose: Original problem divided into a subproblems with size subproblem = size of original/b
- $D(n)$ time to divide subproblem
- $C(n)$ time to combine solutions

Recurrence:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

Analysis of `Merge-sort`:

Assume: input size n, power of 2

Divide: computes the middle of a subarray

takes constant time $\Theta(1)$

Conquer: $2T(n/2)$

Combine: `Merge` $\Theta(n)$

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) + \Theta(1) = 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

Chapter 4: $T(n) = \Theta(n \lg n)$

For large n, `Merge-sort` outperforms `Insertion-sort`

Growth of Functions

By counting the cost of steps in an algorithm, we compute running time ($T(n)$), a numerical function whose domain is $\mathbb{N}$

we can compute worst-case, average-case, best-case running-time

we can compute the order of growth, e.g., of the worst-case

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

Determining the **exact** running time $T(n)$ is too complicated

If n is large (large inputs),

— multiplicative factors and

— lower-order terms

are dominated by the effect of n and can be ignored

Order of growth of worst-case running time of `Insertion-sort` was...

Order of growth of an algorithm:

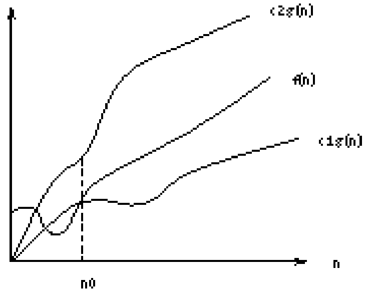
- gives a simple characterization of its efficiency
- allows us to compare algorithms

One algorithm is said to be more efficient than another when its worst-case running time has a lower order of growth.

Asymptotic efficiency of an algorithm:
For n large enough to make **only** the order of growth relevant

Definition: $\Theta(g(n))$

$\Theta(g(n))$ is a set of functions



$\Theta(g(n)) = \{f(n)$: there exist positive constants c_1, c_2, n_0 such that
$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\ \forall n \geq n_0\}$

Note: $g(n)$ and any $f(n)$ in the set, are nonnegative $\forall n \geq n_0$

To prove that $f(n)$ is in the set, we have to find: $c_1 \geq 0$, $c_2 \geq 0$, $n_0 \geq 0$

Formally: $f(n) \in \Theta(g(n))$, but we write $f(n) = \Theta(g(n))\}$
$g(n)$ is an asymptotically tight bound to $f(n)$

Example:

Prove that $\frac{1}{2}n^2 - 3n = \Theta(n^2)$

Find c_1, c_2, n_0 such that $c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2 \ \forall n \geq n_0$

Divide by n^2: $c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$

Consider: $\frac{1}{2} - \frac{3}{n} \leq c_2$

Choose $c_2 = ...$

Consider: $c_1 \leq \frac{1}{2} - \frac{3}{n}$,

choose $n_0 = ...$ such that $c_0 \leq 0$

choose c_1

Many choices exist, we need to find *one*