Evaluation

- Exams (2) 50%
- Quizzes 10%
- Homework Assignments 30%
- Semester Project 5%
- Class Participation 5%

http://www.cis.fiu.edu/~giri/teach/5407F08.html

https://online.cis.fiu.edu/portal/course/view.php?id=285

Definitions

Algorithm: It 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. It is the sequence of computational steps that transform the input into the output.

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.

Correct Algorithm: for every input instance it halts with the correct output. A correct algorithm solves the given computational problem.

Sorting

- Input is a list of n items that can be compared.
- Output is an ordered list of those n items.
- Fundamental problem that has received a lot of attention over the years.
- Used in many applications.
- Scores of different algorithms exist.
- Task: To compare algorithms
  - On what bases?
    - Time
    - Space
    - Other

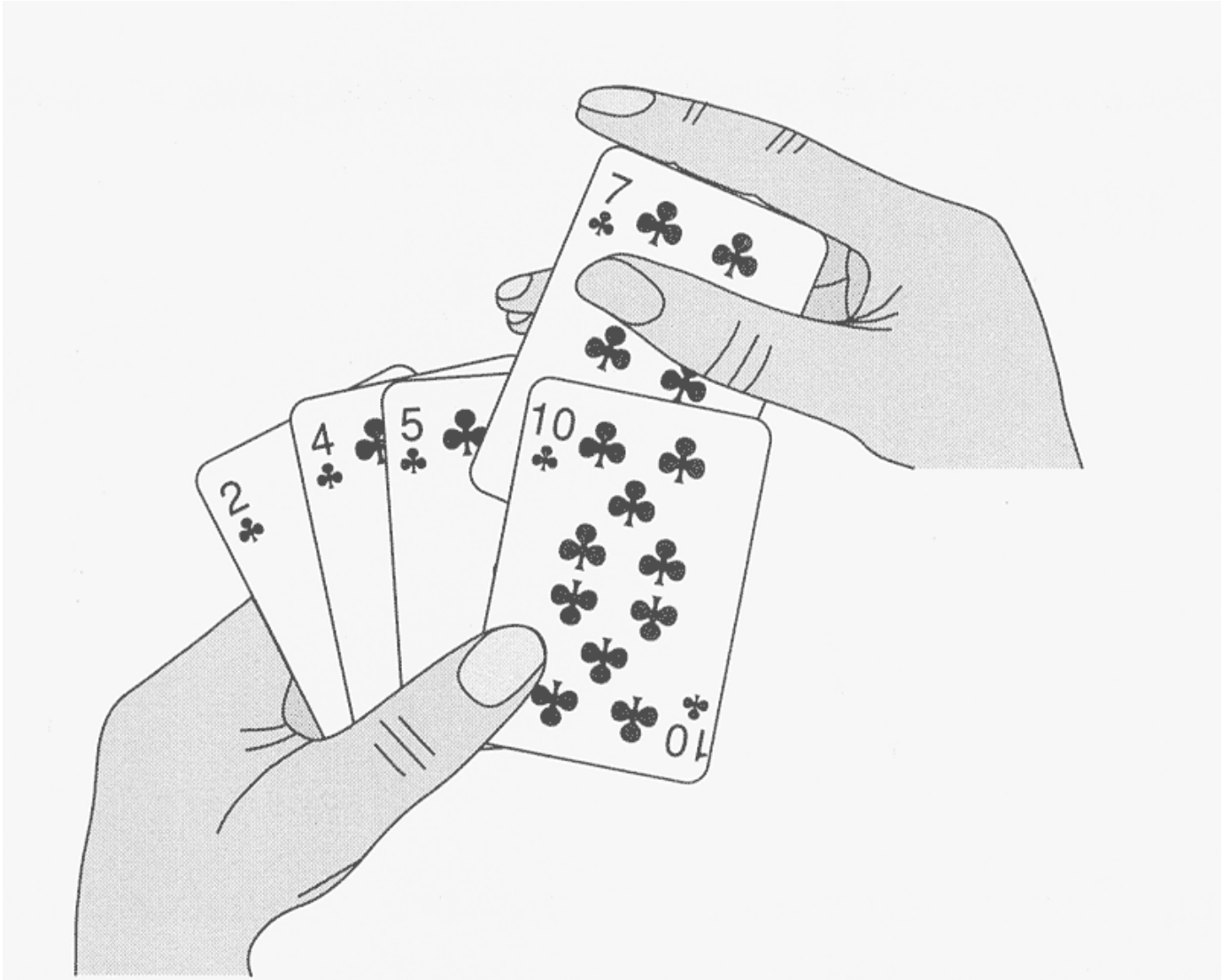


Figure 2.1 Sorting a hand of cards using insertion sort.

Sorting Algorithms

- Number of Comparisons
- Number of Data Movements
- Additional Space Requirements

Sorting Algorithms

- SelectionSort
- InsertionSort
- BubbleSort
- ShakerSort
- MergeSort
- HeapSort
- QuickSort
- Bucket & Radix Sort
- Counting Sort

SelectionSort

Array Position		0	1	2	3	4	5
Initial State		8	5	9	2	6	3
After Iteration 1		2	5	9	8	6	3
After Iteration 2		2	3	9	8	6	5
After Iteration 3		2	3	5	8	6	9
After Iteration 4		2	3	5	6	8	9
After Iteration 5		2	3	5	6	8	9

Psuedocode

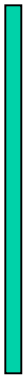
- Convention about statements
- Indentation
- Comments
- Parameters -- passed by value not reference
- And/or are short-circuiting

SelectionSort

```
SELECTIONSORT(array A)
1   N ← length[A]
2   for p ← 1 to N
3         do Compute j, the index of the
             smallest item in A[p..N]
4            Swap A[p] and A[j]
```

SelectionSort

```
SELECTIONSORT(array A)
1  N ← length[A]
2  for p ← 1 to N
         do ▷ Compute j
3           j ← p
4           for m ← p + 1 to N
5                do if (A[m] < A[j])
6                     then j ← m
           ▷ Swap A[p] and A[j]
7           temp ← A[p]
8           A[p] ← A[j]
9           A[j] ← temp
```



SelectionSort: Algorithm Invariants

- iteration k:
  - the k smallest items are in correct location
- NEED TO PROVE THE INVARIANT!!

How to prove invariants & correctness

- **Initialization**: prove it is true at start
- **Maintenance**: prove it is maintained within iterative control structures

- **Termination**: show how to use it to prove correctness

Algorithm Analysis

- Worst-case time complexity
- (Worst-case) space complexity
- Average-case time complexity

SelectionSort

```
SELECTIONSORT(array A)
1  N ← length[A]
2  for p ← 1 to N
         do ▷ Compute j
3           j ← p
4           for m ← p + 1 to N
5                do if (A[m] < A[j])
6                     then j ← m
           ▷ Swap A[p] and A[j]
7           temp ← A[p]
8           A[p] ← A[j]
9           A[j] ← temp
```

$O(n^2)$ time

O(1) space

Solving Recurrence Relations

Page 62, [CLR]

Recurrence; Cond	Solution
$T(n) = T(n-1) + O(1)$	$T(n) = O(n)$
$T(n) = T(n-1) + O(n)$	$T(n) = O(n^2)$
$T(n) = T(n-c) + O(1)$	$T(n) = O(n)$
$T(n) = T(n-c) + O(n)$	$T(n) = O(n^2)$
$T(n) = 2T(n/2) + O(n)$	$T(n) = O(n \log n)$
$T(n) = aT(n/b) + O(n);$ $a = b$	$T(n) = O(n \log n)$
$T(n) = aT(n/b) + O(n);$ $a < b$	$T(n) = O(n)$
$T(n) = aT(n/b) + f(n);$ $f(n) = O(n^{\log_b a - \epsilon})$	$T(n) = O(n)$
$T(n) = aT(n/b) + f(n);$ $f(n) = O(n^{\log_b a})$	$T(n) = \Theta(n^{\log_b a} \log n)$
$T(n) = aT(n/b) + f(n);$ $f(n) = \Theta(f(n))$ $af(n/b) \leq cf(n)$	$T(n) = \Omega(n^{\log_b a} \log n)$

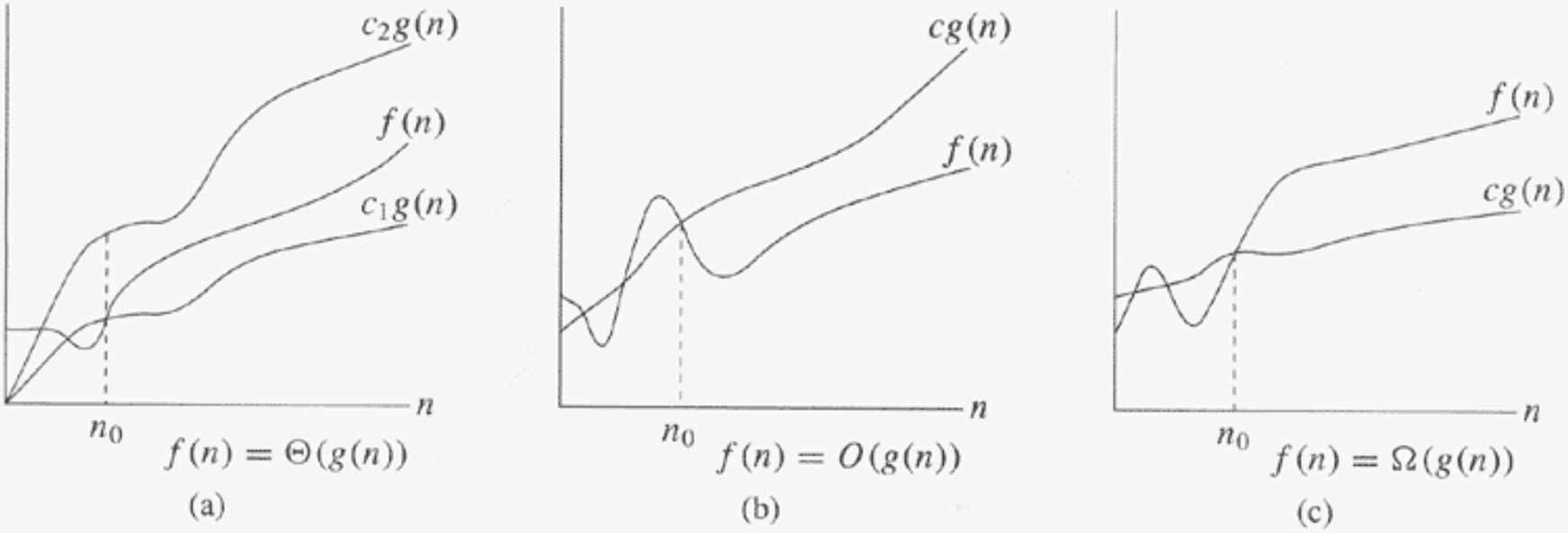


Figure 3.1 Graphic examples of the Θ, O, and Ω notations. In each part, the value of n_0 shown is the minimum possible value; any greater value would also work. **(a)** Θ-notation bounds a function to within constant factors. We write $f(n) = \Theta(g(n))$ if there exist positive constants n_0, c_1, and c_2 such that to the right of n_0, the value of $f(n)$ always lies between $c_1g(n)$ and $c_2g(n)$ inclusive. **(b)** O-notation gives an upper bound for a function to within a constant factor. We write $f(n) = O(g(n))$ if there are positive constants n_0 and c such that to the right of n_0, the value of $f(n)$ always lies on or below $cg(n)$. **(c)** Ω-notation gives a lower bound for a function to within a constant factor. We write $f(n) = \Omega(g(n))$ if there are positive constants n_0 and c such that to the right of n_0, the value of $f(n)$ always lies on or above $cg(n)$.

```
INSERTION-SORT(A)
1  for j ← 2 to length[A]
2       do key ← A[j]
3          ▷ Insert A[j] into the sorted sequence A[1 . . j − 1].
4          i ← j − 1
5          while i > 0 and A[i] > key
6               do A[i + 1] ← A[i]
7                  i ← i − 1
8          A[i + 1] ← key
```

Loop invariants and the correctness of 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	$n-1$

$O(n^2)$ time

$O(1)$ space

InsertionSort: Algorithm Invariant

- iteration k:
  - the first k items are in sorted order.

Figure 8.3

Basic action of insertion sort (the shaded part is sorted)

Array Position	0	1	2	3	4	5
Initial State	8	5	9	2	6	3
After a[0..1] is sorted	5	8	9	2	6	3
After a[0..2] is sorted	5	8	9	2	6	3
After a[0..3] is sorted	2	5	8	9	6	3
After a[0..4] is sorted	2	5	6	8	9	3
After a[0..5] is sorted	2	3	5	6	8	9

Figure 8.4

A closer look at the action of insertion sort (the dark shading indicates the sorted area; the light shading is where the new element was placed).

Array Position	0	1	2	3	4	5
Initial State	8	**5**				
After a[0..1] is sorted	5	8	**9**			
After a[0..2] is sorted	5	8	9	**2**		
After a[0..3] is sorted	2	5	8	9	**6**	
After a[0..4] is sorted	2	5	6	8	9	**3**
After a[0..5] is sorted	2	3	5	6	8	9

```
BUBBLESORT(A)
1  for i ← 1 to length[A]
2       do for j ← length[A] downto i + 1
3            do if A[j] < A[j − 1]
4                 then exchange A[j] ↔ A[j − 1]
```

$O(n^2)$ time

O(1) space

BubbleSort: Algorithm Invariant

- In each pass, every item that does not have a smaller item after it, is moved as far up in the list as possible.
- Iteration k:
  - k smallest items are in the correct location.

Animation Demos

http://cg.scs.carleton.ca/~morin/misc/sortalg/

Comparing $O(n^2)$ Sorting Algorithms

- InsertionSort and SelectionSort (and ShakerSort) are roughly twice as fast as BubbleSort for small files.
- InsertionSort is the best for very small files.
- $O(n^2)$ sorting algorithms are **NOT** useful for large random files.
- If comparisons are very expensive, then among the $O(n^2)$ sorting algorithms, insertionsort is best.
- If data movements are very expensive, then among the $O(n^2)$ sorting algorithms, ?? is best.

Problems to think about!

- What is the least number of comparisons you need to sort a list of 3 elements? 4 elements? 5 elements?
- How to arrange a tennis tournament in order to find the tournament champion with the least number of matches? How many tennis matches are needed?