Analysis of Algorithms
CS 477/677

Instructor: Monica Nicolescu

Lecture 6

Methods for Solving Recurrences

- *Iteration method*
- *Substitution method*
- *Recursion tree method*
- **Master method**

Master method

- "Cookbook" for solving recurrences of the form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

where, $a > 0$, $b > 1$, and $f(n) > 0$

Idea: compare $f(n)$ with $n^{\log_b a}$

- $f(n)$ is asymptotically smaller or larger than $n^{\log_b a}$ by a polynomial factor n^{ε}
- $f(n)$ is asymptotically equal with $n^{\log_b a} \lg^k n$ $(k \geq 0)$

Master method

- "Cookbook" for solving recurrences of the form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

where, $a > 0$, $b > 1$, and $f(n) > 0$

Case 1: if $f(n) = O(n^{\log_b a - \varepsilon})$ for some $\varepsilon > 0$, then: $T(n) = \Theta(n^{\log_b a})$

Case 2: if $f(n) = \Theta(n^{\log_b a} \lg^k n)$, (for some $k \geq 0$) then:

$$T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$$

Case 3: if $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some $\varepsilon > 0$, and if

$af(n/b) \leq cf(n)$ for some $c < 1$ and all sufficiently large n, then:

regularity condition

$$T(n) = \Theta(f(n))$$

Examples

$$T(n) = 3T(n/4) + n\lg n$$

$a = 3$, $b = 4$, $\log_4 3 = 0.793$

Compare $f(n) = n\lg n$ with $n^{0.793}$

$f(n) = \Omega(n^{\log_4 3+\varepsilon})$

Case 3: check regularity condition:

$$3(n/4)\lg(n/4) \le (3/4)n\lg n = c\, f(n),\ c=3/4$$

$\Rightarrow T(n) = \Theta(n\lg n)$

Examples

$$T(n) = 2T(n/2) + n\lg n$$

$a = 2$, $b = 2$, $\log_2 2 = 1$

- Compare $f(n) = n\lg n$ with n

$\Rightarrow f(n) = \Theta(n\lg^1 n) \Rightarrow$ Case 2

$\Rightarrow T(n) = \Theta(n\lg^2 n)$

Examples

$$T(n) = 2T(n/2) + n/\lg n$$

$a = 2$, $b = 2$, $\log_2 2 = 1$

- Compare $f(n) = n/\lg n$ with n

$\Rightarrow f(n) = O(n) \Rightarrow$ seems like Case 1 applies

- f(n) must be polynomially smaller by a factor of n^{ε}
- In this case it is only smaller by a factor of $\lg n$

The 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 $a_1' \leq a_2' \leq \cdots \leq a_n'$

Why Study Sorting Algorithms?

- There are a variety of situations that we can encounter
  - Do we have randomly ordered keys?
  - Are all keys distinct?
  - How large is the set of keys to be ordered?
  - Need guaranteed performance?
  - Does the algorithm sort in place?
  - Is the algorithm stable?

- Various algorithms are better suited to some of these situations

Stability

- A STABLE sort preserves relative order of records with equal keys

Sort file on first key:

Aaron	4	A	664-480-0023	097 Little
Andrews	3	A	874-088-1212	121 Whitman
Battle	4	C	991-878-4944	308 Blair
Chen	2	A	884-232-5341	11 Dickinson
Fox	1	A	243-456-9091	101 Brown
Furia	3	A	766-093-9873	22 Brown
Gazsi	4	B	665-303-0266	113 Walker
Kanaga	3	B	898-122-9643	343 Forbes
Rohde	3	A	232-343-5555	115 Holder
Quilici	1	C	343-987-5642	32 McCosh

Sort file on second key:

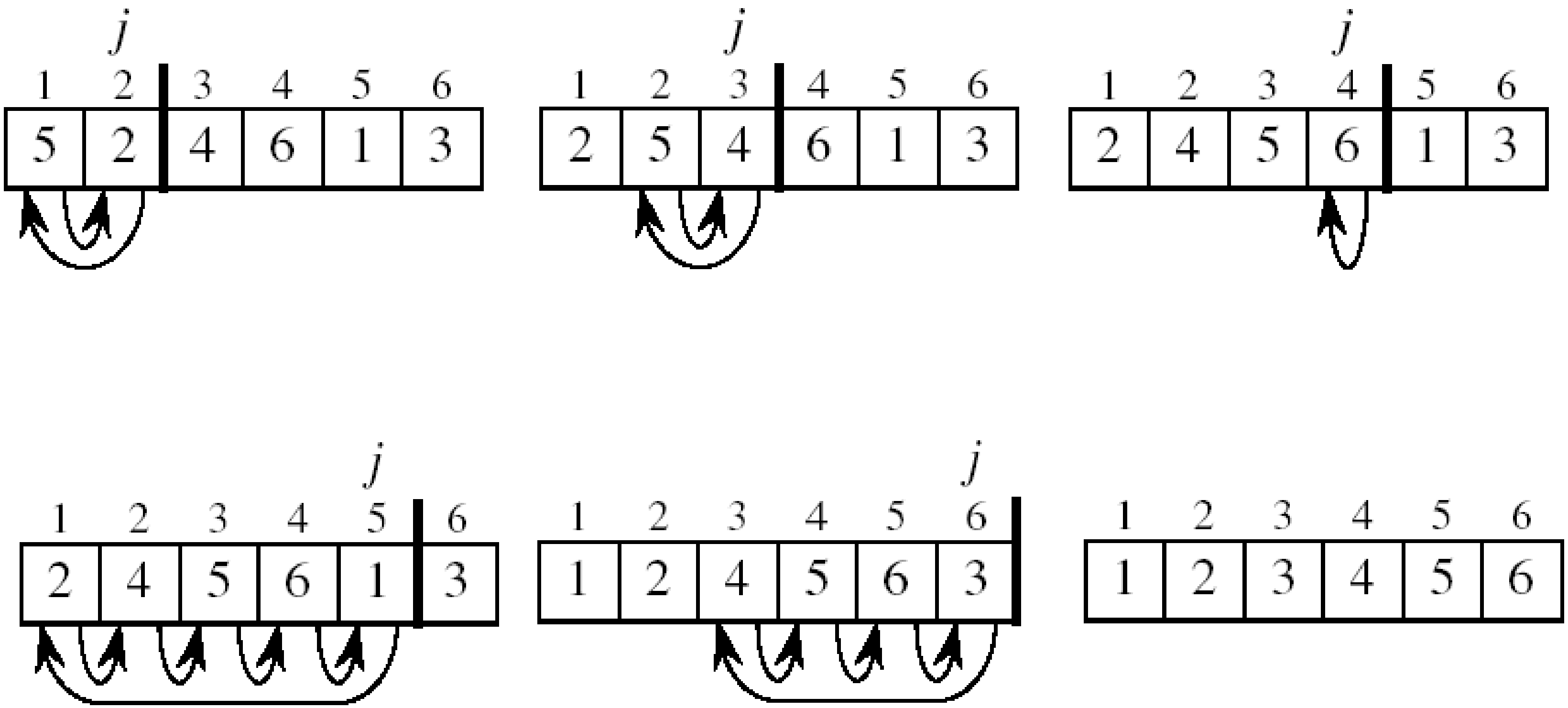
Fox	1	A	243-456-9091	101 Brown
Quilici	1	C	343-987-5642	32 McCosh
Chen	2	A	884-232-5341	11 Dickinson
Kanaga	3	B	898-122-9643	343 Forbes
Andrews	3	A	874-088-1212	121 Whitman
Furia	3	A	766-093-9873	22 Brown
Rohde	3	A	232-343-5555	115 Holder
Battle	4	C	991-878-4944	308 Blair
Gazsi	4	B	665-303-0266	113 Walker
Aaron	4	A	664-480-0023	097 Little

Records with key value 3 are not in order on first key!!

Insertion Sort

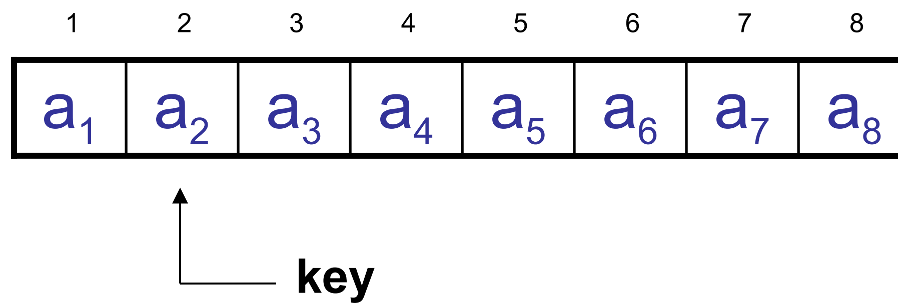
- Idea: like sorting a hand of playing cards
  - Start with an empty left hand and the cards facing down on the table
  - Remove one card at a time from the table, and insert it into the correct position in the left hand
    - compare it with each of the cards already in the hand, from right to left
  - The cards held in the left hand are sorted
    - these cards were originally the top cards of the pile on the table

Example


j
1 2 3 4 5 6
5 2 4 6 1 3
j
1 2 3 4 5 6
2 5 4 6 1 3
j
1 2 3 4 5 6
2 4 5 6 1 3
j
1 2 3 4 5 6
2 4 5 6 1 3
j
1 2 3 4 5 6
1 2 4 5 6 3
1 2 3 4 5 6
1 2 3 4 5 6

INSERTION-SORT

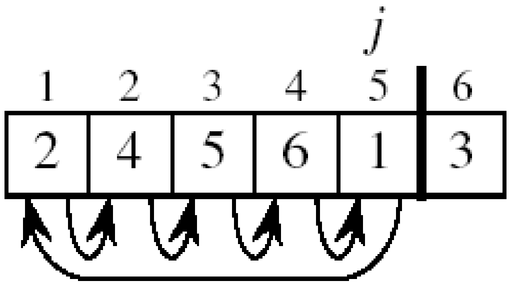
Alg.: INSERTION-SORT(*A*)




```
for j ← 2 to n
        do key ← A[ j ]
           ▷ Insert A[ j ] into the sorted sequence A[1 . . j -1]
           i ← j - 1
           while i > 0 and A[i] > key
                do A[i + 1] ← A[i]
                   i ← i - 1
           A[i + 1] ← key
```

- Insertion sort – sorts the elements in place

Loop Invariant for Insertion Sort



Alg.: INSERTION-SORT(*A*)

for j ← 2 **to** n

do key ← *A*[j]

Insert *A*[j] into the sorted sequence *A*[1 . . j -1]

i ← j - 1

while i > 0 and *A*[i] > key

do *A*[i + 1] ← *A*[i]

i ← i – 1

A[i + 1] ← key

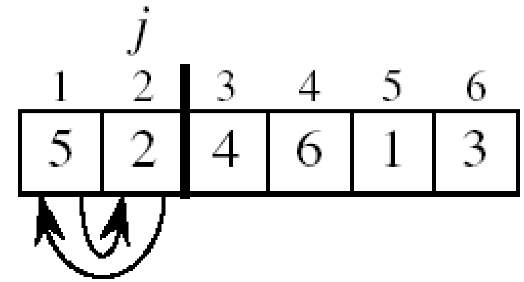
Invariant: at the start of each iteration of the for loop, the elements in A[1 . . j-1] are in sorted order

Proving Loop Invariants

- Proving loop invariants works like induction
- **Initialization (base case):**
  - It is true prior to the first iteration of the loop
- **Maintenance (inductive step):**
  - If it is true before an iteration of the loop, it remains true before the next iteration
- **Termination:**
  - When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct

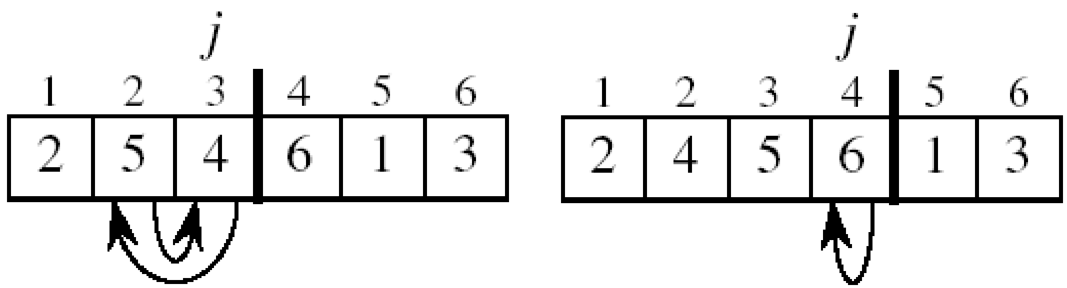
Loop Invariant for Insertion Sort

- **Initialization:**
  - Just before the first iteration, j = 2: the subarray *A*[1 . . j-1] = *A*[1], (the element originally in *A*[1]) – is sorted



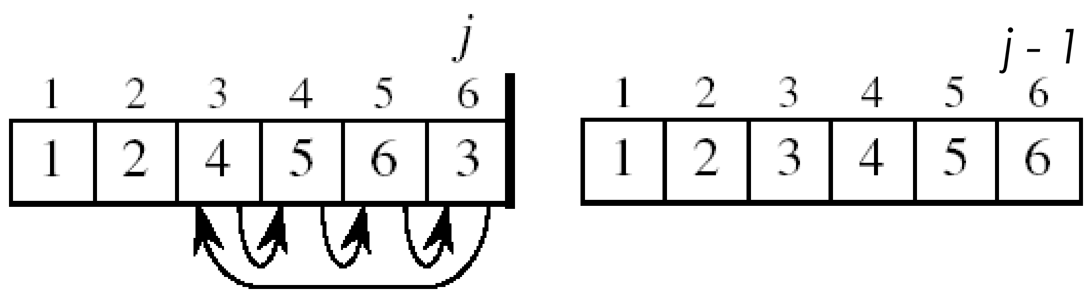
Loop Invariant for Insertion Sort

- **Maintenance:**
  - the **while** inner loop moves *A*[j -1], *A*[j -2], *A*[j -3], and so on, by one position to the right until the proper position for **key** (which has the value that started out in *A*[j]) is found
  - At that point, the value of **key** is placed into this position.



Loop Invariant for Insertion Sort

- **Termination:**
  - The outer **for** loop ends when $j = n + 1 \Rightarrow j-1 = n$
  - Replace n with $j-1$ in the loop invariant:
    - the subarray $A[1 \, . \, . \, n]$ consists of the elements originally in $A[1 \, . \, . \, n]$, but in sorted order



- The entire array is sorted!

Analysis of Insertion Sort

INSERTION-SORT*(A)* cost times

for j ← 2 **to** n c_1 n

do key ← A[j] c_2 n-

▷ Insert A[j] into the sorted seq. A[1 . . j -1] 1

i ← j - 1 0 $\sum_{j=2}^{n} t_j$ n-

while i > 0 and A[i] > key 1 $\sum_{j=2}^{n} (t¿¿ j-1)¿$

do A[i + 1] ← A[i] c_4 n- $\sum_{j=2}^{n} (t¿¿ j-1)¿$

i ← i – 1 1

A[i + 1] ← key c_5

c_6

c_7

c_8 n-

Best Case Analysis

- The array is already sorted "**while** i > 0 and A[i] > key"
  - $A[i] \leq key$ upon the first time the **while** loop test is run (when $i = j - 1$)
  - $t_j = 1$
- $T(n) = c_1 n + c_2(n - 1) + c_4(n - 1) + c_5(n - 1) + c_8(n-1) =$

  $(c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$

  $= an + b = \Theta(n)$

Worst Case Analysis

- The array is reversely sorted "while i > 0 and A[i] > key"
  - Always **A[i] > key** in **while** loop test
  - Have to compare **key** with all elements to the left of the **j**-th position ⇒ compare with **j-1** elements ⇒ t_j = **j**

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

$$¿ a n^2 + bn + c$$ a quadratic function of n

- T(n) = Θ(n^2) order of growth in **n^2**

Comparisons and Exchanges in Insertion Sort

INSERTION-SORT*(A)* cost

times

for j ← 2 **to** n

do key ← A[j] c_1 n

▷ Insert A[j] into the sorted sequence A[1 . . j -1] c_2 n-

i ← j - 1 ≈$n^2/2$ comparisons 1

while i > 0 and A[i] > key 0 $\sum_{j=2}^{n} t_j$ n-

do A[i + 1] ← A[i] 1 $\sum_{j=2}^{n} (t¿¿ j-1)¿$

i ← i – 1 ≈$n^2/2$ exchanges c_4 $\sum_{j=2}^{n} (t¿¿ j-1)¿$ n-

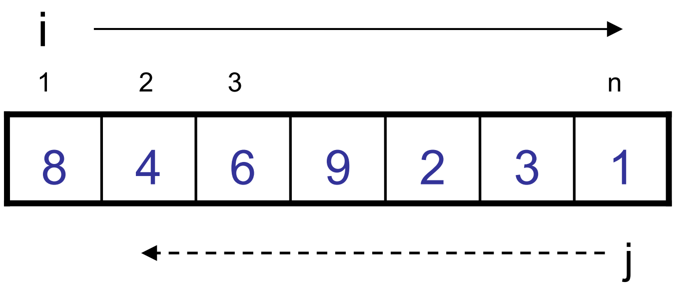
A[i + 1] ← key 1

Insertion Sort - Summary

- Idea: like sorting a hand of playing cards
  - Start with an empty left hand and the cards facing down on the table.
  - Remove one card at a time from the table, and insert it into the correct position in the left hand
- Advantages
  - Good running time for "almost sorted" arrays $\Theta(n)$
- Disadvantages
  - $\Theta(n^2)$ running time in worst and average case
  - $\approx n^2/2$ comparisons and $n^2/2$ exchanges

Bubble Sort

- Idea:
  - Repeatedly pass through the array
  - Swaps adjacent elements that are out of order



- Easier to implement, but slower than Insertion sort

Example

8	4	6	9	2	3	1

i = 1 j

8	4	6	9	2	1	3

i = 1 j

8	4	6	9	1	2	3

i = 1 j

8	4	6	1	9	2	3

i = 1 j

8	4	1	6	9	2	3

i = 1 j

8	1	4	6	9	2	3

i = 1 j

1	8	4	6	9	2	3

i = 1 j

1	8	4	6	9	2	3

i = 2 j

1	2	8	4	6	9	3

i = 3 j

1	2	3	8	4	6	9

i = 4 j

1	2	3	4	8	6	9

i = 5 j

1	2	3	4	6	8	9

i = 6 j

1	2	3	4	6	8	9

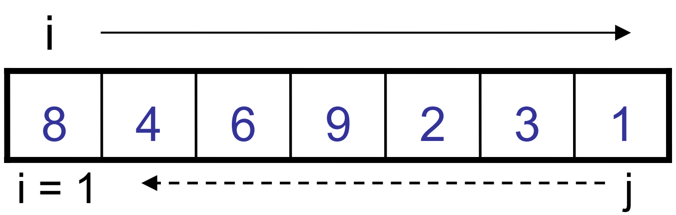
i = 7

j

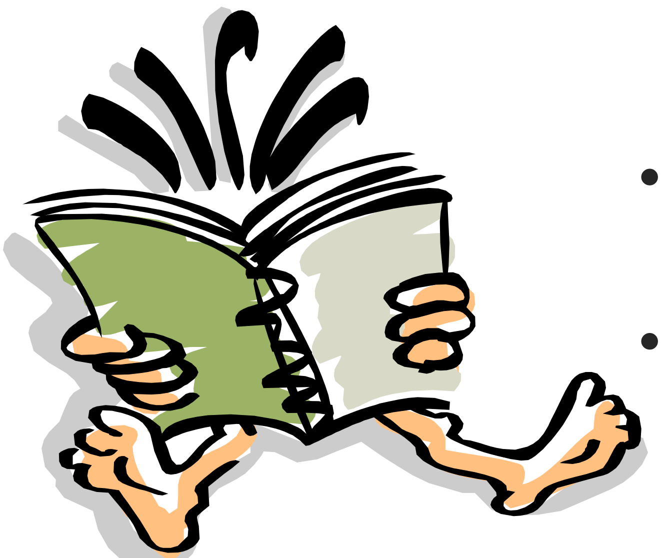
Bubble Sort

Alg.: BUBBLESORT(A)

for i ← 1 **to** length[*A*]
    do for j ← length[*A*] **downto** i + 1
        do if *A*[j] < *A*[j -1]
            then exchange *A*[j] ⟺ *A*[j-1]



Readings



- For this lecture
  - Section 4.5, 2.1, 2.2
- Coming next
  - Section 2.3, 7.1, 7.2