Sorting and Searching
Asymptotic Complexity Analysis

Module – 1 Review of concepts

Sorting problem

- Input: A sequence of n numbers, a1,a2,…,an
- Output: A permutation (reordering) (a1’,a2’,…,an’) of the input sequence such that a1’≤a2’ ≤… ≤an’
  - Comment: The number that we wish to sort are also known as keys

Insertion Sort

- Efficient for sorting small numbers

- In place sort: Takes an array A[0..n-1] (sequence of n elements) and arranges them in place, so that it is sorted.

It is always good to start with numbers

5	2	4	6	1	3

1	2	3	4	5	6

j=5

Invariant property in the loop:

At the start of each iteration of the algorithm, the subarray a[0...j-1] contains the elements originally in a[0..j-1] but in sorted order

Pseudo Code

- **Insertion-sort(A)**

1. for j=1 to (length(A)-1)
2. do key = A[j]
3. #Insert A[j] into the sorted sequnce A[0...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 //as A[i]<=key, so we place //key on the right side of A[i]

Loop Invariants and Correctness of Insertion Sort

- **Initialization:** Before the first loop starts, j=1. So, A[0] is an array of single element and so is trivially sorted.

- **Maintenance:** The outer for loop has its index moving like j=1,2,…,n-1 (if A has n elements). At the beginning of the for loop assume that the array is sorted from A[0..j-1]. The inner while loop of the jth iteration places A[j] at its correct position. Thus at the end of the jth iteration, the array is sorted from A[0..j]. Thus, the invariance is maintained. Then j becomes j+1.
  - ***Also, using the same inductive reasoning the elements are also the same as in the original array in the locations A[0..j].***

Loop Invariants and Correctness of Insertion Sort

- **Termination:** The for loop terminates when j=n, thus by the previous observations the array is sorted from A[0..n-1] and the elements are also the same as in the original array.

Thus, the algorithm indeed sorts and is thus correct!

Analyzing Algorithms

The RAM Model

- A generic one processor Random Access Machine (RAM) model of computation.
- Instructions are executed sequentially (and not concurrently)
- We have to use the model so that we do not go too deep (into the machine instructions) and yet not abuse the notions (by say assuming that there exists a sorting instruction)

The RAM Model

- Our model has instructions commonly found in real computers:
  - arithmetic (add, subtract, multiply, divide)
  - data movement (load, store, copy)
  - control (conditional and unconditional branch, subroutine call and function)
- Each such instruction takes a constant time

Data types & Storage

- In the RAM model the data types are float and int.
- Assume the size of each block or word of data is so that an input of size n can be represented by word of **clog(n)** bits, $c \geq 1$
- $c \geq 1$, so that each word can hold the value of n.
- c cannot grow arbitrarily, because we cannot have one word storing huge amount of data and also which could be operated in constant time.

Gray areas in the RAM model

- Is exponentiation a constant time operation? NO
- Is computation of 2^n a constant time operation? Well…
- Many computers have a "shift left" operation by k positions (in constant time)
- Shift left by 1 position multiplies by 2. So, if I shift left 2, k times…I obtain 2^k in constant time !
  - (as long as k is no more than the word length).

Some further points on the RAM Model

- **We do not model the memory hierarchy**
  - No caches, pages etc
  - May be necessary for real computers and real applications. But the discussions are too specialized and we do use such modeling when required. As they are very complex and difficult to work with.
  - Fortunately, RAM models are excellent predictors! Still quite challenging. We require knowledge in logic, inductive reasoning, combinatorics, probability theory, algebra, and above all observation and intuition!

Lets analyze the Insertion sort

- The time taken to sort depends on the fact that we are sorting how many numbers
- Also, the time to sort may change depending upon whether the array is almost sorted (can you see if the array was sorted we had very little job).
- So, we need to define the meaning of the **input size** and **running time**.

Input Size

- Depends on the notion of the problem we are studying.
- Consider sorting of n numbers. The input size is the cardinal number of the set of the integers we are sorting.
- Consider multiplying two integers. The input size is the total number of bits required to represent the numbers.
- Sometimes, instead of one numbers we represent the input by two numbers. E.g. graph algorithms, where the input size is represented by both the number of edges (E) and the number of vertices (V)

Running Time

- Proportional to the Number of primitive operations or steps performed.
- Assume, in the pseudo-code a constant amount of time is required for each line.
- Assume that the ith line requires ci, where ci is a constant.
- Keep in mind the RAM model which says that there is no concurrency.

Run Time of Insertion Sort

Steps	Cost	Times
for j=1 to n-1	c_1	n
key=A[j]	c_2	n-1
i=j-1	c_3	n-1
while i>0 and A[i]>key	c_4	$\sum_{j=1}^{n-1} t_j$
do A[i+1]=A[i]	c_5	$\sum_{j=1}^{n-1} (t_j - 1)$
i=i-1	c_6	$\sum_{j=1}^{n-1} (t_j - 1)$
A[i+1]=key	c_7	(n-1)

In the RAM model the total time required is the sum of that for each statement:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=1}^{n-1} t_j + c_5 \sum_{j=1}^{n-1} (t_j - 1) + c_6 \sum_{j=1}^{n-1} (t_j - 1) + c_7 (n\text{-}1)$$

Best Case

- **If the array is already sorted:**
  - *While* loop sees in 1 check that A[i]<key and so while loop terminates. Thus t_j=1 and we have:

$$T(n)=c_1 n + c_2(n-1) + c_3(n-1) + c_4\sum_{j=1}^{n-1} 1 + c_5\sum_{j=1}^{n-1}(1-1) + c_6\sum_{j=1}^{n-1}(1-1) + c_7(n-1)$$

$$=(c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7)$$

The run time is thus a linear function of n

Worst Case: The algorithm cannot run slower!

- If the array is arranged in reverse sorted array:
  - *While* loop requires to perform the comparisons with A[j-1] to A[0], that is t_j=j

$$T(n)=c_1 n + c_2(n-1) + c_3(n-1) + c_4\sum_{j=1}^{n-1} j + c_5\sum_{j=1}^{n-1}(j-1) + c_6\sum_{j=1}^{n-1}(j-1) + c_7(n-1)$$

$$=\left(\frac{c_4}{2}+\frac{c_5}{2}+\frac{c_6}{2}\right)n^2 + \left(c_1+c_2+c_3-\frac{c_4}{2}-\frac{3c_5}{2}-\frac{3c_6}{2}\right)n + (c_5+c_6-c_2-c_3-c_7)$$

The run time is thus a quadratic function of n

Average Case

- Instead of an input of a particular type (as in best case or worst case), all the inputs of the given size are equally *probable* in such an analysis.
  - E.g. coming back to our insertion sort, if the elements in the array A[0..j-1] are randomly chosen. We can assume that half the elements are greater than A[j] while half are less. On the average, thus $t_j=j/2$. Plugging this value into T(n) still leaves it quadratic. Thus, in this case average case is equivalent to a worst case run of the algorithm.
  - *Does this always occur?* NO. The average case may tilt towards the best case also.

Comparison of Sorting Algorithms

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 $a_1' \le a_2' \le \cdots \le a_n'$

Structure of data

- Usually, the numbers to be sorted are part of a collection of data called a record

- Each record contains a key, which is the value to be sorted

example of a record

Key	other data

- Note that when the keys must be rearranged, the data associated with the keys must also be rearranged (time consuming !!)

- Pointers can be used instead (space consuming !!)

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?

- Various algorithms are better suited to some of these situations

Some Definitions

- Internal Sort
  - The data to be sorted is all stored in the computer's main memory.
- External Sort
  - Some of the data to be sorted might be stored in some external, slower, device.
- In Place Sort
  - The amount of extra space required to sort the data is constant with the input size.

Stability

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

Sorted 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:

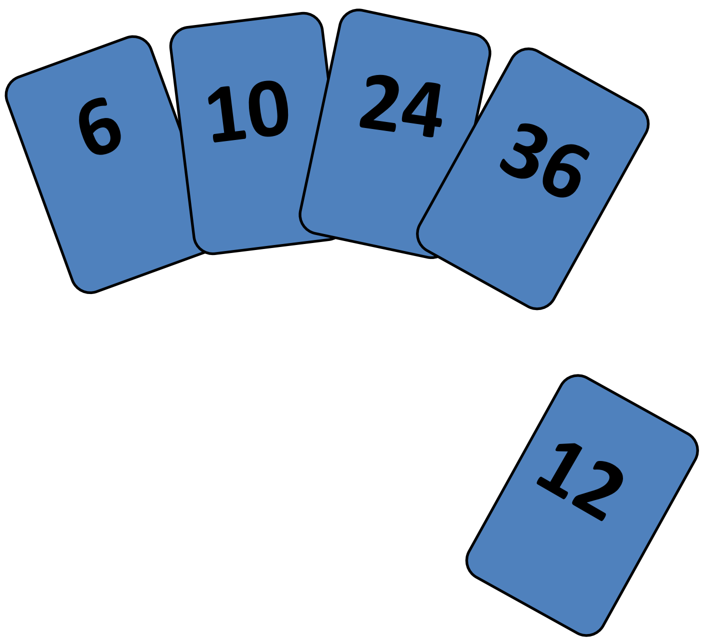
Fox	1	A	243-456-9091	101 Brown
Quilici	1	C	343-987-5642	32 McCosh
Chen	2	A	884-232-5341	11 Dickinson
Kanaga	3	B	898-122-9643	343 Forbes
Andrews	3	A	874-088-1212	121 Whitman
Furia	3	A	766-093-9873	22 Brown
Rohde	3	A	232-343-5555	115 Holder
Battle	4	C	991-878-4944	308 Blair
Gazsi	4	B	665-303-0266	113 Walker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

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

Insertion Sort

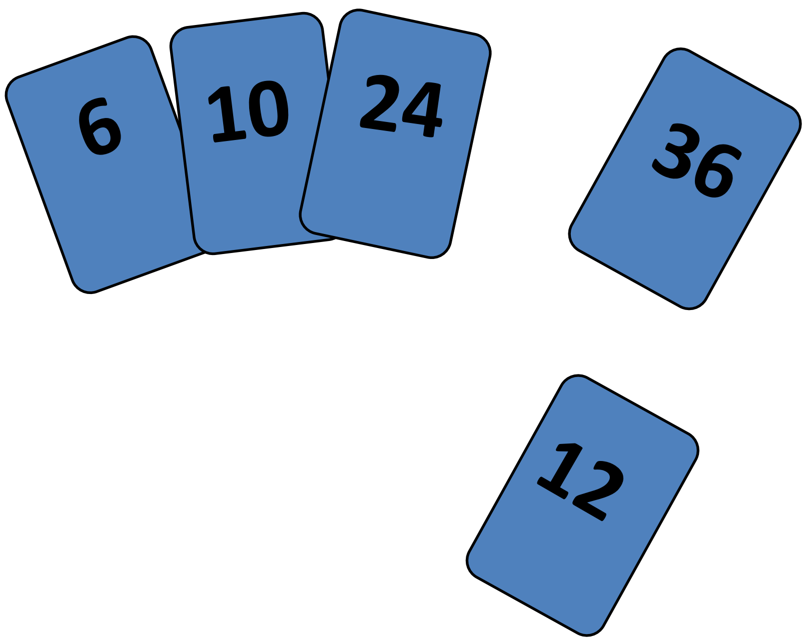
- Idea: like sorting a hand of playing cards
  - Start with an empty left hand and the cards facing down on the table.
  - Remove one card at a time from the table, and insert it into the correct position in the left hand
    - compare it with each of the cards already in the hand, from right to left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

Insertion Sort

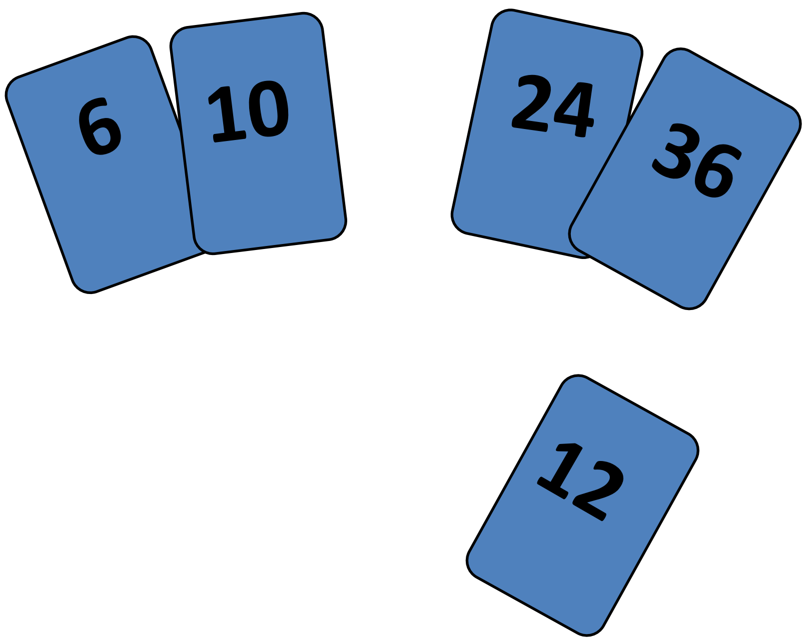


To insert 12, we need to make room for it by moving first 36 and then 24.

Insertion Sort



Insertion Sort



Insertion Sort

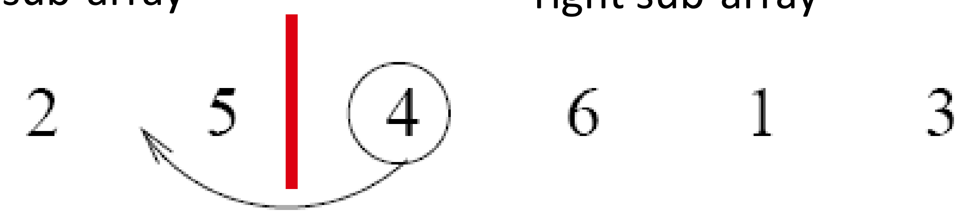
input array

5 2 4 6 1 3

at each iteration, the array is divided in two sub-arrays:

left sub-array

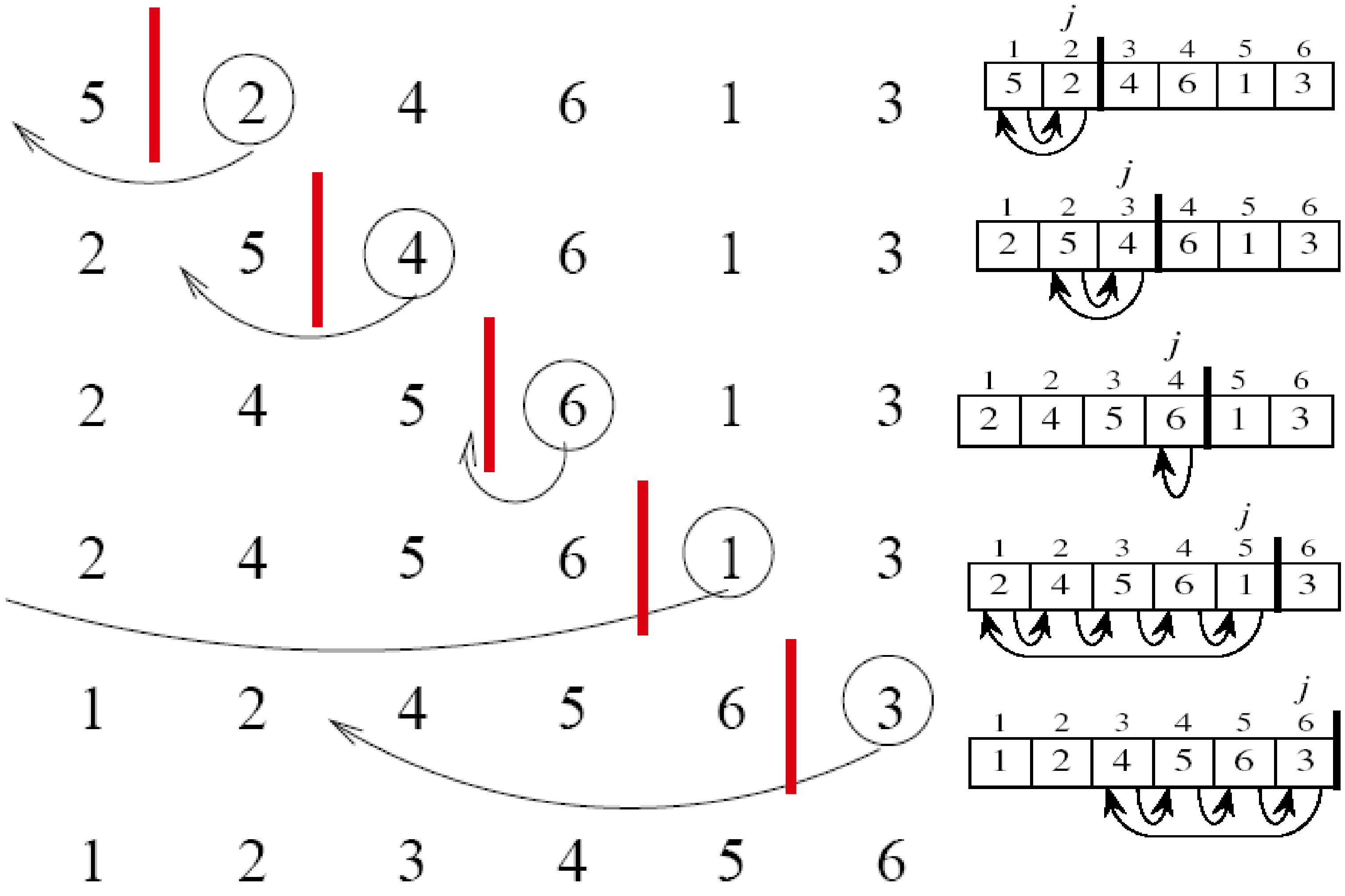
right sub-array



sorted

unsorted

Insertion Sort



INSERTION-SORT

Alg.: INSERTION-SORT(*A*)

1	2	3	4	5	6	7	8
a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8

key

```
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
```

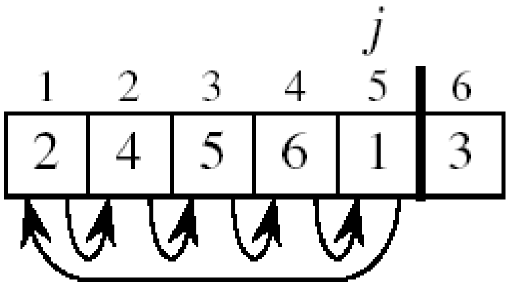
- Insertion sort – sorts the elements in place

Loop Invariant for Insertion Sort

Alg.: INSERTION-SORT(*A*)

for $j \leftarrow 2$ **to** n

do key $\leftarrow A[\,j\,]$



Insert $A[\,j\,]$ into the sorted sequence $A[1\,.\,.\,j-1]$

$i \leftarrow j - 1$

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

do $A[i + 1] \leftarrow A[i]$

$i \leftarrow i - 1$

$A[i + 1] \leftarrow key$

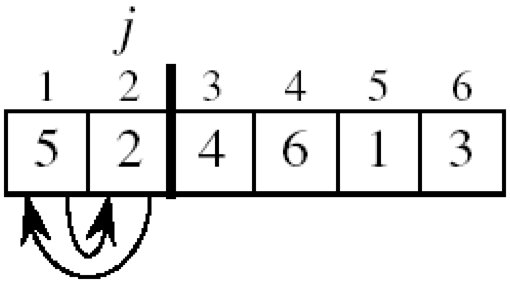
Invariant: at the start of the **for** loop the elements in $A[1\,.\,.\,j-1]$ are in sorted order

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
  - Stop the induction when the loop terminates

Loop Invariant for Insertion Sort

- **Initialization:**

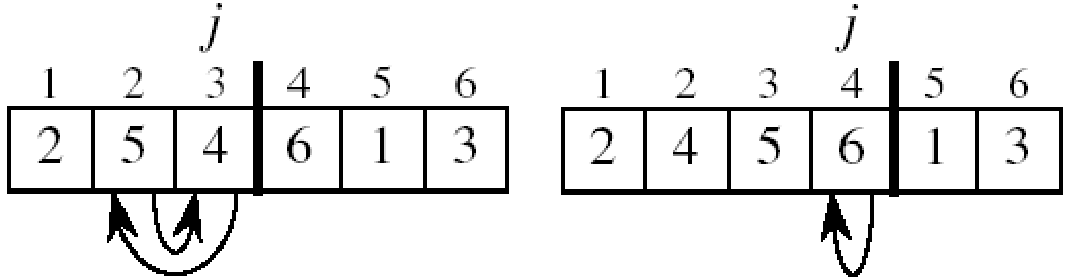


  – Just before the first iteration, j = 2:

    the subarray $A[1\ ..\ j\text{-}1] = A[1]$, (the element originally in $A[1]$) – is sorted

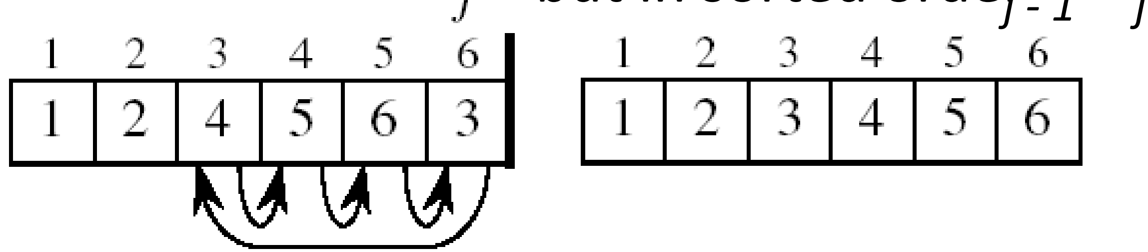
Loop Invariant for Insertion Sort

- **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 or [illegible] but in sorted order



- The entire array is sorted!

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

Analysis of Insertion Sort

INSERTION-SORT*(A)*	cost	times
for j ← 2 **to** n	c_1	n
do 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	$\sum_{j=2}^{n} t_j$
do 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	$n-1$

t_j: # of times the while statement is executed at iteration j

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

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 $i = j - 1$)
  - $t_j = 1$
- $T(n) = c_1n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) = (c_1 + c_2 + c_4 + c_5 + c_8)n + (c_2 + c_4 + c_5 + c_8)$

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

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

Worst Case Analysis

- The array is in reverse sorted order **"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
    $\Rightarrow$ compare with j-1 elements $\Rightarrow$ $t_j = j$

using $\sum_{j=1}^{n} j = \frac{n(n+1)}{2}$ => $\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$ => $\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$ we have:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6\frac{n(n-1)}{2} + c_7\frac{n(n-1)}{2} + c_8(n-1)$$

$$= an^2 + bn + c$$ a quadratic function of n

- $T(n) = \Theta(n^2)$ order of growth in n^2

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

Comparisons and Exchanges in Insertion Sort

INSERTION-SORT*(A)*	cost	times
for j ← 2 **to** n	c_1	n
do 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	$\sum_{j=2}^{n} t_j$
do 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

$\approx n^2/2$ comparisons

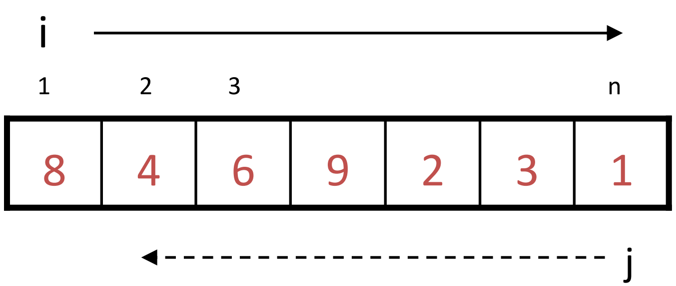
$\approx n^2/2$ exchanges

Insertion Sort - Summary

- 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 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]

      i **then** exchange A[j] ↔ A[j-1]

8	4	6	9	2	3	1

i = 1 ◄- j

Bubble-Sort Running Time

Alg.: BUBBLESORT(A)

for i ← 1 **to** length[A] c_1

do for j ← length[A] **downto** i + 1 c_2

Comparisons: ≈ $n^2/2$ **do if** A[j] < A[j -1] c_3

Exchanges: ≈ $n^2/2$ **then** exchange A[j] ↔ A[j-1] c_4

$$T(n) = c_1(n+1) + c_2 \sum_{i=1}^{n}(n-i+1) + c_3 \sum_{i=1}^{n}(n-i) + c_4 \sum_{i=1}^{n}(n-i)$$

$$= \Theta(n) + (c_2 + c_2 + c_4) \sum_{i=1}^{n}(n-i)$$

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

Thus, $T(n) = \Theta(n^2)$

Selection Sort

- Idea:
  - Find the smallest element in the array
  - Exchange it with the element in the first position
  - Find the second smallest element and exchange it with the element in the second position
  - Continue until the array is sorted
- Disadvantage:
  - Running time depends only slightly on the amount of order in the file

Example

8	4	6	9	2	3	1

1	4	6	9	2	3	8

1	2	6	9	4	3	8

1	2	3	9	4	6	8

1	2	3	4	9	6	8

1	2	3	4	6	9	8

1	2	3	4	6	8	9

1	2	3	4	6	8	9

Selection Sort

8	4	6	9	2	3	1

Alg.: SELECTION-SORT*(A)*

```
n ← length[A]
for j ← 1 to n - 1
    do smallest ← j
        for i ← j + 1 to n
            do if A[i] < A[smallest]
                then smallest ← i
        exchange A[j] ↔ A[smallest]
```

Analysis of Selection Sort

Alg.: SELECTION-SORT*(A)*

	cost	times
$n \leftarrow$ length[A]	c_1	1
for $j \leftarrow 1$ to $n - 1$	c_2	n
do smallest $\leftarrow j$	c_3	$n-1$
for $i \leftarrow j + 1$ to n	c_4	$\sum_{j=1}^{n-1}(n-j+1)$
do if $A[i] < A$[smallest]	c_5	$\sum_{j=1}^{n-1}(n-j)$
then smallest $\leftarrow i$	c_6	$\sum_{j=1}^{n-1}(n-j)$
exchange $A[j] \leftrightarrow A$[smallest]	c_7	$n-1$

$\approx n^2/2$ comparisons

$\approx n$ exchanges

$$T(n) = c_1 + c_2 n + c_3(n-1) + c_4\sum_{j=1}^{n-1}(n-j+1) + c_5\sum_{j=1}^{n-1}(n-j) + c_6\sum_{j=2}^{n-1}(n-j) + c_7(n-1) = \Theta(n^2)$$

Divide-and-Conquer

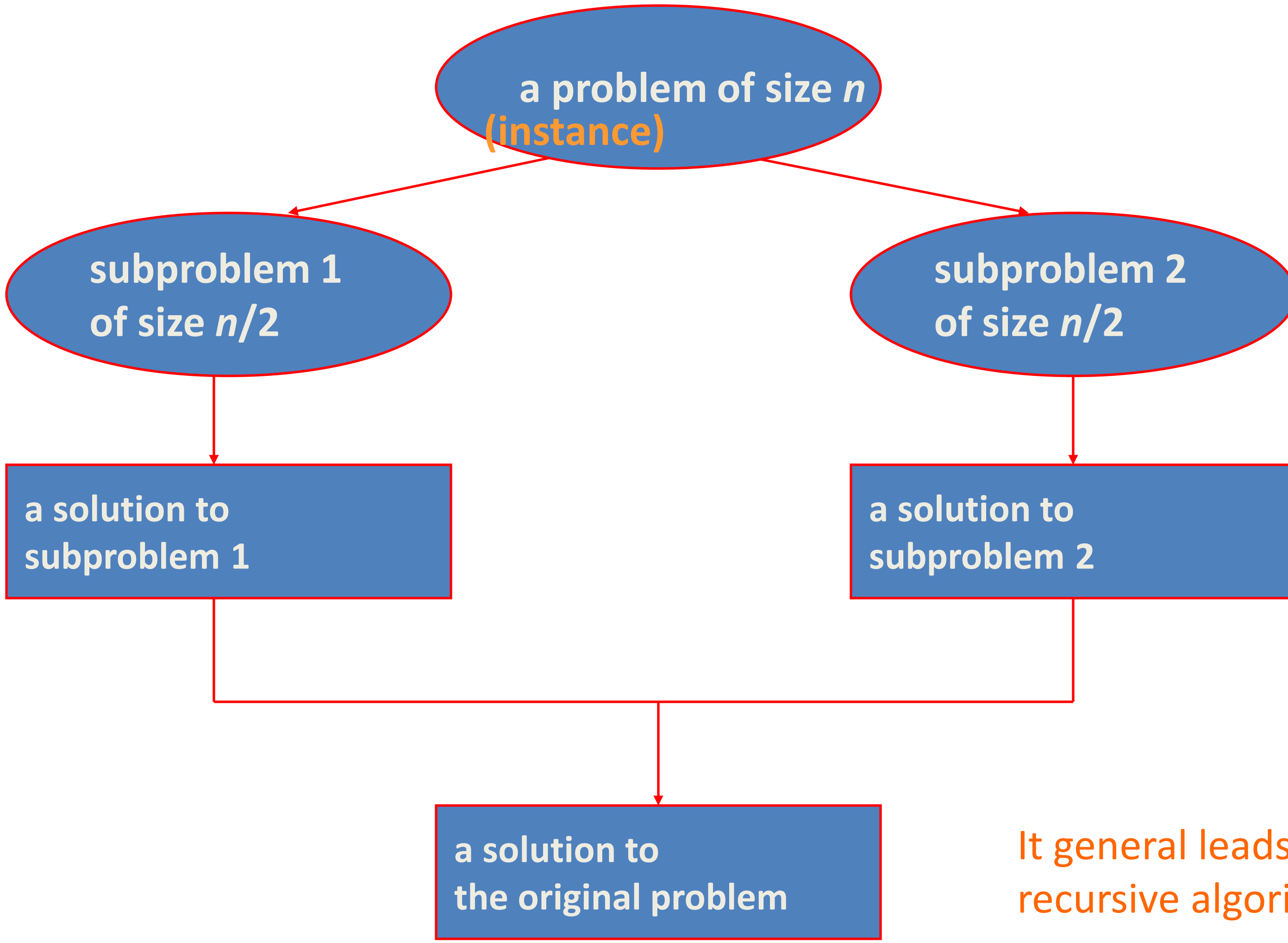
The most-well known algorithm design strategy:

1. Divide instance of problem into two or more smaller instances

2. Solve smaller instances recursively

3. Obtain solution to original (larger) instance by combining these solutions

Divide-and-Conquer Technique (cont.)



It general leads to a recursive algorithm!

Divide-and-Conquer Examples

- Sorting: mergesort and quicksort
- Binary tree traversals
- Binary search (?)
- Multiplication of large integers
- Matrix multiplication: Strassen's algorithm
- Closest-pair and convex-hull algorithms

General Divide-and-Conquer Recurrence

$T(n) = aT(n/b) + f(n)$ where $f(n) \in \Theta(n^d)$, $d \geq 0$

<u>Master Theorem</u>: If $a < b^d$, $T(n) \in \Theta(n^d)$

If $a = b^d$, $T(n) \in \Theta(n^d \log n)$

If $a > b^d$, $T(n) \in \Theta(n^{\log_b a})$

Note: The same results hold with O instead of Θ.

Examples: $T(n) = 4T(n/2) + n \Rightarrow T(n) \in ?$ $\Theta(n^2)$

$T(n) = 4T(n/2) + n^2 \Rightarrow T(n) \in ?$ $\Theta(n^2 \log n)$

$T(n) = 4T(n/2) + n^3 \Rightarrow T(n) \in ?$ $\Theta(n^3)$

Mergesort

- Split array A[0..n-1] into about equal halves and make copies of each half in arrays B and C
- Sort arrays B and C recursively
- Merge sorted arrays B and C into array A as follows:
  - Repeat the following until no elements remain in one of the arrays:
    - compare the first elements in the remaining unprocessed portions of the arrays
    - copy the smaller of the two into A, while incrementing the index indicating the unprocessed portion of that array
  - Once all elements in one of the arrays are processed, copy the remaining unprocessed elements from the other array into A.

Pseudocode of Mergesort

ALGORITHM *Mergesort*$(A[0..n-1])$

//Sorts array $A[0..n-1]$ by recursive mergesort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
if $n > 1$
  copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$
  copy $A[\lfloor n/2 \rfloor..n-1]$ to $C[0..\lceil n/2 \rceil - 1]$
  Mergesort$(B[0..\lfloor n/2 \rfloor - 1])$
  Mergesort$(C[0..\lceil n/2 \rceil - 1])$
  Merge(B, C, A)

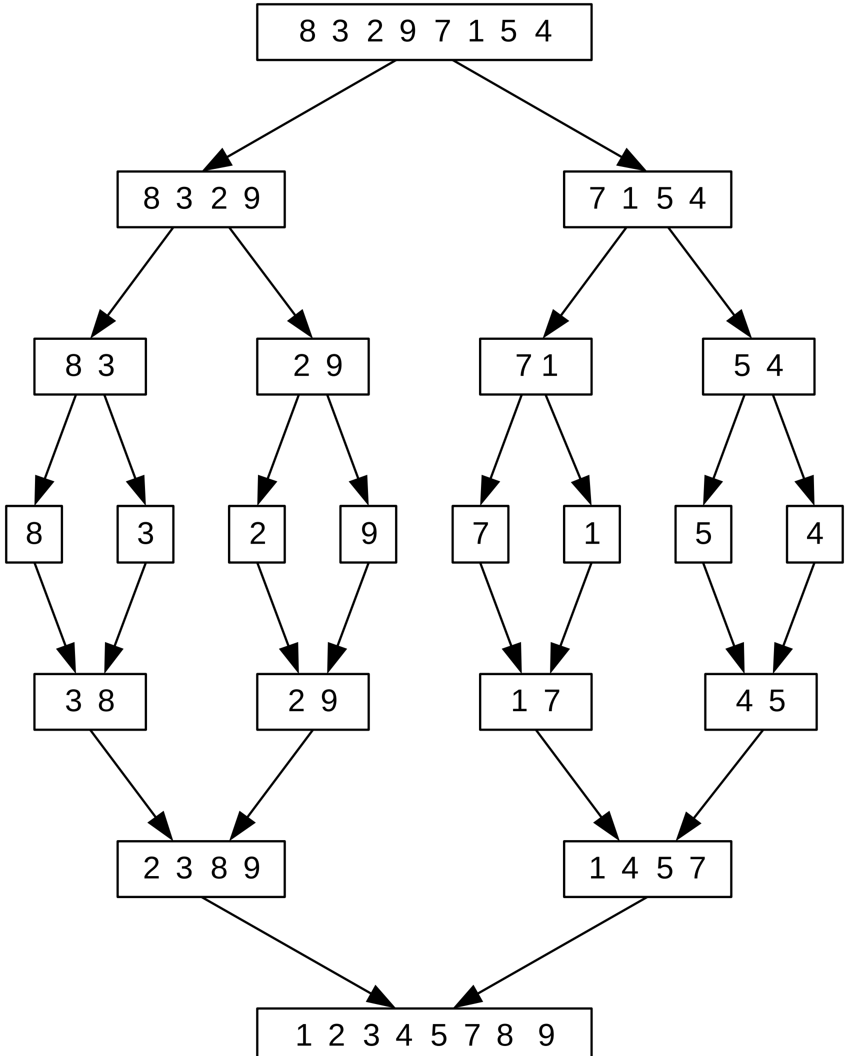
Pseudocode of Merge

ALGORITHM $Merge(B[0..p-1], C[0..q-1], A[0..p+q-1])$

```
//Merges two sorted arrays into one sorted array
//Input: Arrays B[0..p − 1] and C[0..q − 1] both sorted
//Output: Sorted array A[0..p + q − 1] of the elements of B and C
i ← 0; j ← 0; k ← 0
while i < p and j < q do
    if B[i] ≤ C[j]
        A[k] ← B[i]; i ← i + 1
    else A[k] ← C[j]; j ← j + 1
    k ← k + 1
if i = p
    copy C[j..q − 1] to A[k..p + q − 1]
else copy B[i..p − 1] to A[k..p + q − 1]
```

Time complexity: **Θ(*p+q*) = Θ(*n*)** comparisons

Mergesort Example



The non-recursive version of Mergesort starts from merging single elements into sorted pairs.

Analysis of Mergesort

- All cases have same efficiency: $\Theta(n \log n)$

  $T(n) = 2T(n/2) + \Theta(n)$, $T(1) = 0$

- Number of comparisons in the worst case is close to theoretical minimum for comparison-based sorting:

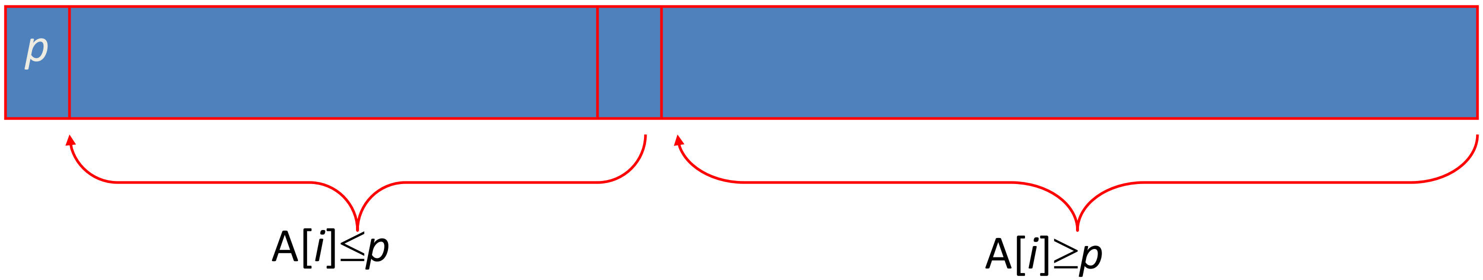
  $$\lceil \log_2 n! \rceil \approx n \log_2 n - 1.44n$$

- Space requirement: $\Theta(n)$ (<u>not</u> in-place)

- Can be implemented without recursion (bottom-up)

Quicksort

- Select a *pivot* (partitioning element) – here, the first element
- Rearrange the list so that all the elements in the first *s* positions are smaller than or equal to the pivot and all the elements in the remaining *n-s* positions are larger than or equal to the pivot (see next slide for an algorithm)



- Exchange the pivot with the last element in the first (i.e., ≤) subarray — the pivot is now in its final position
- Sort the two subarrays recursively

Partitioning Algorithm

Algorithm *Partition*$(A[l..r])$
//Partitions a subarray by using its first element as a pivot
//Input: A subarray $A[l..r]$ of $A[0..n-1]$, defined by its left and right
//   indices l and r $(l < r)$
//Output: A partition of $A[l..r]$, with the split position returned as
//   this function's value
$p \leftarrow A[l]$
$i \leftarrow l; \quad j \leftarrow r + 1$
repeat
    repeat $i \leftarrow i + 1$ **until** $A[i] \geq p$ or $i > r$
    repeat $j \leftarrow j - 1$ **until** $A[j] < p$ or $j = l$
    swap$(A[i], A[j])$
until $i \geq j$
swap$(A[i], A[j])$ //undo last swap when $i \geq j$
swap$(A[l], A[j])$
return j

Time complexity: **Θ(*r-l*)** comparisons

Quicksort Example

5 3 1 9 8 2 4 7

2	3	1	4	5	8	9	7
1	2	3	4	5	7	8	9
1	2	3	4	5	7	8	9
1	2	3	4	5	7	8	9
1	2	3	4	5	7	8	9

Analysis of Quicksort

- Best case: split in the middle — $\Theta(n \log n)$
- Worst case: sorted array! — $\Theta(n^2)$
- Average case: random arrays — $\Theta(n \log n)$

$T(n) = T(n-1) + \Theta(n)$

- Improvements:
  - better pivot selection: median of three partitioning
  - switch to insertion sort on small subfiles
  - elimination of recursion

  These combine to 20-25% improvement

- Considered the method of choice for internal sorting of large files ($n \geq 10000$)

Binary Search

Very efficient algorithm for searching in sorted array:

K

vs

A[0] . . . A[m] . . . A[n-1]

If K = A[m], stop (successful search); otherwise, continue searching by the same method in A[0..m-1] if K < A[m] and in A[m+1..n-1] if K > A[m]

$l \leftarrow 0$; $r \leftarrow n-1$
while $l \leq r$ do
    $m \leftarrow \lfloor (l+r)/2 \rfloor$
    if K = A[m] return m
    else if K < A[m] $r \leftarrow m-1$
    else $l \leftarrow m+1$
return -1