Monsoon-2023

An Introduction to Program Design

CS2002D Program Design

Lecture 2 & 3

Why Sorting?

- Database search - Binary search is more efficient.
- Computational geometry, Computer graphics.
- Comparison based sorting - elements are compared and rearranged to perform sorting.
- How to analyse the run-time of a sorting algorithm?
- Which sorting algorithm is the best?

Sorting : Problem Definition

Input: A sequence of n numbers <a1,a2,a3,a4,...,an>

Output: A permutation (reordering) of <a1',a2',a3',a4',...,an'> such that a1'<= a2'<=a3' <=a4'<=.....<=an'

Insertion Sort

- Example
    - Arranging cards while playing card game

Insertion Sort

- INSERTION_SORT(Integer_Array A)
    - Input: Array A, Output: Elements of A in sorted order
- In i^{th} iteration, the first i elements are sorted. (Incremental Approach)
- Insert i^{th} element to the sorted list of elements at positions 1 through i (to its left).

How do we write the algorithm??

- What is the input of our algorithm?
- Input: array of elements 5, 7, 2, 3, 7, 10 . We pass the input array to the function insertion sort
- Keep 5 as such in the first position of the array, assuming that it is sorted by itself
- Take 7, compare it with 5, place it there itself
- Take 2, we put in a temporary variable (say *key*), we copy 7 to 2's position in the array, now the array will be 5,7,7,3,7,10
- Do we have to place *key* (2) in the first 7's position and make the array as 5,2,7,3,7,10 ?

Design of Insertion Sort Algorithm

- Input: array of elements 5, 7, 2, 3, 7, 10 .
- Compare key with 5 and make the array 5,5,7,3,7,10
- How many loops we should have?
- One loop for sure which goes from 2 to n
- Another loop which goes from position of *key* element to 1

Insertion Sort - Pseudocode

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
```

Insertion Sort

INSERTION_SORT(A)

```
1. for j=2 to A.length
2.    key = A[ j ];
4.    i = j-1
5.    while i > 0 and A[ i ] > key
6.          A[ i+1 ]=A[ i ]
7.          i=i-1
8.    A[ i+1 ]= key
```

index	1	2	3	4	5	6	7
value	6	4	8	10	2	14	12

j=2

key = 4

i=1

Insertion Sort

INSERTION_SORT(A)

```
1. for j=2 to A.length
2.    key = A[ j ];
4.    i = j-1
5.    while i > 0 and A[ i ] > key
6.          A[ i+1 ]=A[ i ]
7.          i=i-1
8.    A[ i+1 ]= key
```

index	1	2	3	4	5	6	7
value	6	**4**	8	10	2	14	12

j=2

key = 4

i=1

1>0 and 6>4

Insertion Sort

INSERTION_SORT(A)

1. for j=2 to A.length
2. key = A[j];
4. i = j-1
5. while i > 0 and A[i] > key
6. **A[i+1]=A[i]**
7. i=i-1
8. A[i+1]= key

index	1	2	3	4	5	6	7
value	**6**	**6**	8	10	2	14	12

j=2

key = 4

i=1

1>0 and 6>4

Insertion Sort

INSERTION_SORT(A)

1. for j=2 to A.length
2. key = A[j];
4. i = j-1
5. while **i > 0 and A[i] > key**
6. A[i+1]=A[i]
7. i=i-1
8. A[i+1]= key

index	1	2	3	4	5	6	7
value	**6**	**6**	8	10	2	14	12

j=2

key = 4

i=0

0>0 and A[0]>4

Insertion Sort

INSERTION_SORT(A)

1. for j=2 to A.length
2. key = A[j];
4. i = j-1
5. while i > 0 and A[i] > key
6. A[i+1]=A[i]
7. i=i-1
8. **A[i+1]= key**

index	1	2	3	4	5	6	7
value	**4**	**6**	8	10	2	14	12

j=2

key = 4

i=0

Insertion Sort

INSERTION_SORT(A)

1. for j=2 to A.length
2. key = A[j];
4. i = j-1
5. while **i > 0 and A[i] > key**
6. A[i+1]=A[i]
7. i=i-1
8. A[i+1]= key

index	1	2	3	4	5	6	7
value	4	6	**8**	10	2	14	12

j=3

key = 8

i=2

2>0 and 6>8

Insertion Sort

INSERTION_SORT(A)

1. for j=2 to A.length
2. key = A[j];
4. i = j-1
5. while i > 0 and A[i] > key
6. A[i+1]=A[i]
7. i=i-1
8. **A[i+1]= key**

index	1	2	3	4	5	6	7
value	4	6	**8**	10	2	14	12

j=3

key = 8

i=2

A[3] = 8

Insertion Sort

INSERTION_SORT(A)

1. for j=2 to A.length
2. key = A[j];
4. i = j-1
5. while **i > 0 and A[i] > key**
6. A[i+1]=A[i]
7. i=i-1
8. A[i+1]= key

index	1	2	3	4	5	6	7
value	4	6	8	**10**	2	14	12

j=4

key = **10**

i=3

3>0 and 8>10

Insertion Sort

INSERTION_SORT(A)

```
for j=2 to A.length
    key = A[ j ];
    i = j-1
    while i > 0 and A[ i ] > key
        A[ i+1 ]=A[ i ]
        i=i-1
    A[ i+1 ]= key
```

index	1	2	3	4	5	6	7
value	4	6	8	10	2	14	12

j=4

key = **10**

i=3

A[4] = 10

Insertion Sort

INSERTION_SORT(A)

1. for j=2 to A.length
2. key = A[j];
4. i = j-1
5. while **i > 0 and A[i] > key**
6. A[i+1]=A[i]
7. i=i-1
8. A[i+1]= key

index	1	2	3	4	5	6	7
value	4	6	8	10	**2**	14	12

j=5

key = **2**

i=4

4>0 and 10>2

Insertion Sort

INSERTION_SORT(A)

1. for j=2 to A.length
2.   key = A[j];
4.   i = j-1
5.   while i > 0 and A[i] > key
6.   **A[i+1]=A[i]**
7.   i=i-1
8.   A[i+1]= key

index	1	2	3	4	5	6	7
value	4	6	8	10	**10**	14	12

j=5

key = **2**

i=4

Insertion Sort

INSERTION_SORT(A)

```
1. for j=2 to A.length
2.     key = A[ j ];
4.     i = j-1
5.     while i > 0 and A[ i ] > key
6.             A[ i+1 ]=A[ i ]
7.             i=i-1
8.     A[ i+1 ]= key
```

index	1	2	3	4	5	6	7
value	4	6	8	10	10	14	12

j=5

key = **2**

i=3

3>0 and 8>2

Insertion Sort

INSERTION_SORT(A)

1. for j=2 to A.length
2. key = A[j];
4. i = j-1
5. while i > 0 and A[i] > key
6. **A[i+1]=A[i]**
7. i=i-1
8. A[i+1]= key

index	1	2	3	4	5	6	7
value	4	6	8	**8**	10	14	12

j=5

key = **2**

i=3

Insertion Sort

INSERTION_SORT(A)

```
for j=2 to A.length
    key = A[ j ];
    i = j-1
    while i > 0 and A[ i ] > key
        A[ i+1 ]=A[ i ]
        i=i-1
    A[ i+1 ]= key
```

index	1	2	3	4	5	6	7
value	4	6	8	8	10	14	12

j=5

key = **2**

i=2

2>0 and 6>2

Insertion Sort

INSERTION_SORT(A)

1. for j=2 to A.length
2. key = A[j];
4. i = j-1
5. while i > 0 and A[i] > key
6. **A[i+1]=A[i]**
7. i=i-1
8. A[i+1]= key

index	1	2	3	4	5	6	7
value	4	6	**6**	8	10	14	12

j=5

key = **2**

i=2

Insertion Sort

INSERTION_SORT(A)

1. for j=2 to A.length
2. key = A[j];
4. i = j-1
5. while **i > 0 and A[i] > key**
6. A[i+1]=A[i]
7. i=i-1
8. A[i+1]= key

index	1	2	3	4	5	6	7
value	4	6	6	8	10	14	12

j=5

key = **2**

i=1

1>0 and 4>2

Insertion Sort

INSERTION_SORT(A)

1. for j=2 to A.length
2. key = A[j];
4. i = j-1
5. while i > 0 and A[i] > key
6. **A[i+1]=A[i]**
7. i=i-1
8. A[i+1]= key

index	1	2	3	4	5	6	7
value	4	**4**	6	8	10	14	12

j=5

key = **2**

i=1

Insertion Sort

INSERTION_SORT(A)

1. for j=2 to A.length
2. key = A[j];
4. i = j-1
5. while **i > 0 and A[i] > key**
6. A[i+1]=A[i]
7. i=i-1
8. A[i+1]= key

index	1	2	3	4	5	6	7
value	4	4	6	8	10	14	12

j=5

key = **2**

i=0

0>0 and A[0]>2

Insertion Sort

INSERTION_SORT(A)

```
1. for j=2 to A.length
2.    key = A[ j ];
4.    i = j-1
5.    while i > 0 and A[ i ] > key
6.          A[ i+1 ]=A[ i ]
7.          i=i-1
8.    A[ i+1 ]= key
```

index	1	2	3	4	5	6	7
value	2	4	6	8	10	14	12

j=5

key = **2**

i=0

A[1]=2

Insertion Sort

INSERTION_SORT(A)

1. for j=2 to A.length
2.   key = A[j];
4.   i = j-1
5.   while **i > 0 and A[i] > key**
6.   A[i+1]=A[i]
7.   i=i-1
8.   A[i+1]= key

index	1	2	3	4	5	6	7
value	2	4	6	8	10	**14**	12

j=6

key = **14**

i=5

5>0 and 10>14

Insertion Sort

INSERTION_SORT(A)

1. for j=2 to A.length
2.   key = A[j];
4.   i = j-1
5.   while i > 0 and A[i] > key
6.   A[i+1]=A[i]
7.   i=i-1
8.   **A[i+1]= key**

index	1	2	3	4	5	6	7
value	2	4	6	8	10	**14**	12

j=6

key = **14**

i=5

A[6]=14

Insertion Sort

INSERTION_SORT(A)

```
1. for j=2 to A.length
2.     key = A[ j ];
4.     i = j-1
5.     while i > 0 and A[ i ] > key
6.         A[ i+1 ]=A[ i ]
7.         i=i-1
8.     A[ i+1 ]= key
```

index	1	2	3	4	5	6	7
value	2	4	6	8	10	14	**12**

j=7

key = **12**

i=6

6>0 and 14>12

Insertion Sort

INSERTION_SORT(A)

1. for j=2 to A.length
2. key = A[j];
4. i = j-1
5. while i > 0 and A[i] > key
6. **A[i+1]=A[i]**
7. i=i-1
8. A[i+1]= key

index	1	2	3	4	5	6	7
value	2	4	6	8	10	14	**14**

j=7

key = **12**

i=6

Insertion Sort

INSERTION_SORT(A)

```
for j=2 to A.length
    key = A[ j ];
    i = j-1
    while i > 0 and A[ i ] > key
        A[ i+1 ]=A[ i ]
        i=i-1
    A[ i+1 ]= key
```

index	1	2	3	4	5	6	7
value	2	4	6	8	10	14	14

j=7

key = **12**

i=5

5>0 and 10>12

Insertion Sort

INSERTION_SORT(A)

1. for j=2 to A.length
2. key = A[j];
4. i = j-1
5. while i > 0 and A[i] > key
6. A[i+1]=A[i]
7. i=i-1
8. **A[i+1]= key**

index	1	2	3	4	5	6	7
value	2	4	6	8	10	**12**	14

j=7

key = **12**

i=5

A[6]=12

Insertion Sort

INSERTION_SORT(A)

```
for j=2 to A.length
    key = A[ j ];
    i = j-1
    while i > 0 and A[ i ] > key
        A[ i+1 ]=A[ i ]
        i=i-1
    A[ i+1 ]= key
```

index	1	2	3	4	5	6	7
value	6	4	8	10	2	14	12

Number of comparisons?

Insertion Sort

INSERTION_SORT(A)

```
1. for j=2 to A.length
2.     key = A[ j ];
4.     i = j-1
5.     while i > 0 and A[ i ] > key
6.             A[ i+1 ]=A[ i ]
7.             i=i-1
8.     A[ i+1 ]= key
```

index	1	2	3	4	5	6	7
value	1	2	3	4	7	8	9

Number of comparisons?

Insertion Sort

INSERTION_SORT(A)

```
1. for j=2 to A.length
2.    key = A[ j ];
4.    i = j-1
5.    while i > 0 and A[ i ] > key
6.          A[ i+1 ]=A[ i ]
7.          i=i-1
8.    A[ i+1 ]= key
```

index	1	2	3	4	5	6	7
value	16	14	12	10	8	6	4

Number of comparisons?

Insertion Sort - Visualization

- What is the worst case input for insertion sort?
- How many “copy” operations (steps 6, 8) are performed in this example?

How do we prove that INSERTION SORT(A) is correct?

- We observe the algorithm critically and try to understand
- We observe that :
  - The index *j* indicates:
    - The current number/card being inserted
  - At the beginning of each iteration of *for* loop indexed by *j:*

    A [1 .. j-1] is sorted and *A [j+1..n]* is remaining
- **Elements A [1.. j-1] are the elements originally in positions 1 through j-1, but now in sorted order**
- We state these properties of A[1 .. j-1] formally as a **loop invariant**

Loop Invariant of INSERTION SORT(A)

- **At the start of each iteration of the for loop of lines 1-8, the subarray A [1 ... j-1] consists of the elements originally in A [1 ... j-1], but in sorted order.**
- We use loop invariant (a property of the algorithm) to prove that the algorithm is correct
- We must show three things about a loop invariant:
  - Initialization
  - Maintenance
  - Termination

Correctness of Insertion Sort

- We must show three things about a loop invariant :
  - **Initialization:** The loop invariant is true prior to the first iteration of the loop
  - **Maintenance:** If the loop invariant is true before an iteration of the loop, it remains true before the next iteration
  - **Termination:** When the loop terminates, the loop invariant gives us a useful property that helps us to show that the algorithm is correct

Loop invariant Vs mathematical induction

- Invariant holds before the first iteration ~ base case of mathematical induction
- Invariant holds from iteration to iteration ~ inductive step
- Third property ie. The termination property differs from math induction
  - In math induction, we use the inductive step infinitely, whereas here we stop the induction when the loop terminates

Proof – Correctness of Insertion sort

- **Initialization**: Loop invariant trivially holds before the first loop iteration. A [1] is sorted by itself.
- **Maintenance:**
  - *for* loop works by moving A[j-1], A[j-2], A[j-3] and so on by one position to the right and finds the proper position for A[j] and inserts the value of A[j].
  - Hence, A[1..j-1] consists of the elements originally in

    A[1..j-1], but in sorted order.
  - Incrementing j for the next iteration preserves the loop invariant

Proof: Correctness of Insertion sort - contd.

Termination:

- *for* loop terminates when $j > A.length = n$ ie. $j = n+1$
- Substituting $n+1$ in the wording of loop invariant, the subarray A[1..n] consists of originally in A[1..n], but in sorted order

- Observe that A[1..n] is the entire array and since the entire array is sorted, the algorithm is correct.
- After proving Insertion sort is correct, we have to prove insertion sort is efficient.
- For that we analyse insertion sort

Analyzing the algorithms

- **What do we mean by “analyzing the algorithms”**
- Analyzing the algorithms means predicting the resources the algorithm uses
- What are the resources?
- **Computational time and Memory for storage**
- **Why do we analyze algorithms?**
- Analyzing several algorithms for a particular problem results in the most efficient algorithm in terms of computational time/memory

Analyzing the algorithms - Contd

- A bench mark/model of the implementation technology and the resources of that technology and their costs
- We assume a generic one processor **Random Access Machine** (RAM) model of computation
    - We use RAM as an implementation technology
    - Our algorithms will be implemented as computer programs
    - Instructions are executed one after another, with no concurrent operations

RAM model

- Assume a realistic RAM

- RAM contains instructions commonly found in real computers such as:
  - Arithmetic :eg: add, subtract, multiply, divide, remainder, floor, ceil
  - Data movement : eg: load, store, copy
  - Control : conditional & unconditional branch, subroutine call and return

RAM model - Contd

- **Data types** : integer and floating point
- Usually we do not concern ourselves on the precision of the value unless precision is very crucial
- We assume a limit on the size of each word of data
  - Eg: when working with inputs of size *n*, we assume that integers are represented by c lg *n* bits for some constant c >=1
    - We require c >=1, so that each word can hold the value of n
    - We restrict c to be a constant so that the word size does not grow arbitrarily

Analysis of algorithm

- The **time taken** by an algorithm **grows with the input size**
- **Running time** of an algorithm is described as **a function of its input**
- We will formalize **"input size" and "running time**"
- **Input size**
  - Depends on the problem being studied
  - eg: for sorting problem, it is the number of elements
  - eg: for multiplying two numbers, it is the total number of bits

Running Time of instructions

....

$a = 0$

.....

$b = a$

....

$z = x + y$

....

return x

- Each statement is translated to a set of primitive operations or steps
- Running time depends on the number of primitive operations

Primitive Operations

Pseudocode statement: $z = x + y$

Translated code (for a hypothetical machine)

load r_1, x // loads contents of memory location x to register r_1

load r_2, y // loads contents of memory location y to register r_2

add r_1, r_2 // adds contents of r_1 and r_2, stores result in r_1

store z, r_1 // moves data in r_1 to memory location z

- $z = x + y$ required 4 machine instructions
- Number of steps different for different types of statements

Running Time of an Algorithm

- Running time on a particular input is the number of primitive operations or steps executed
- A constant amount of time for each line in the pseudocode
- The i^{th} line takes time c_i where c_i is a constant

Random-Access Machine (RAM) model

- Random-Access Machine (RAM) model
  - Single Processor Machine model
  - Instructions for arithmetic, data movement, transfer of control-each taking a constant amount of time
  - Instructions executed one after the other, no concurrent operations

Example

SAMPLE(a, b)

$c = a - b$

$temp = a$

$a = b$

$b = temp$

return c

Running Time $= c_1 + c_2 + c_3 + c_4 + c_5$

Example

ARRAY-SUM(A)

```
sum = 0
for i = 1 to A.length
    sum = sum + A[i]
return sum
```

Running Time = ???

Example

```
ARRAY-SUM(A)
sum = 0
for i = 1 to A.length
    sum = sum + A[i]
return sum
```

Running Time = $c_1 + c_2(n+1) + c_3 n + c_4$ // $A.length$ is n

Pseudocode of Linear Search

LINEAR SEARCH(A, key)

```
found = 0
for i = 1 to A.length
        if A[ i ] = key
            found = 1
            return i
if found = 0
    return 0
```

Best Case of Linear Search

- **Best Case input of Linear Search** : The element to be searched is in the first position of the list
  - Eg: A= 1, 4, 2, 7, 10, 5 & key = 1
- How do we **analyse the linear search in the best case**?
- Step 1: Cost : c_{1} , Times : 1
- Step 2: Cost : c_{2} , Times : 1
- Step 3: Cost : c_{3} , Times : 1
- Step 4: Cost : c_{4} , Times : 1
- Step 5: Cost : c_{5} , Times : 1
- $T(n) = c_1 + c_2 + c_3 + c_4 + c_5$

LINEAR SEARCH(A,key)

```
1. found = 0
2. for i = 1 to A.length
3.             if A[ i ] = key
4.                     found = 1
5.         return i
6. if found = 0
7.     return 0
```

Worst Case of Linear Search

- **One of the Worst Case input of Linear Search :** The element to be searched is in the last position of the list
  - Eg: A = 1, 4, 2, 7, 10, 5 & key = 5
- How do we analyse the linear search in the worst case – successful search?
- Step 1: Cost : c_1 , Times : 1
- Step 2: Cost : c_2 , Times : n
- Step 3: Cost : c_3 , Times : n
- Step 4: Cost : c_4 , Times : 1
- Step 5: Cost : c_5 , Times : 1
- $T(n) = c_1 + n * c_2 + n * c_3 + c_4 + c_5$

LINEAR SEARCH(A,key)

```
1. found = 0
2. for i = 1 to A.length
3.              if A[ i ] = key
4.                     found = 1
5.          return i
6. if found = 0
7.     return 0
```

Analysis of Worst Case of Linear Search- Unsuccessful search

- **One of the Worst Case input of Linear Search** : The unsuccessful search analysis ie. the element is not present
  - Eg: A = 1, 4, 2, 7, 10, 5 & key = 0
- Step 1- Cost : c_1, Times : 1
- Step 2- Cost : c_2, Times : n+1
- Step 3- Cost : c_3, Times : n
- Step 4- Cost : c_4, Times : 0
- Step 5- Cost : c_5, Times : 0
- Step 6- Cost : c_6, Times : 1
- Step 7- Cost : c_7, Times : 1
- $T(n) = c_1 + (n+1)* c_2 + n* c_3 + c_6 + c_7$

LINEAR SEARCH(A,key)

```
1. found = 0
2. for i = 1 to A.length
3.          if A[ i ] = key
4.               found = 1
5.      return i
6. if found = 0
7.   return 0
```

INSERTION-SORT(A)

for j = 2 to A.length	c_1
key = A[j];	c_2
// Insert A[j] into the sorted sequence A[1… j-1]	
i = j-1	c_3
while i > 0 and A[i] > key	c_4
do A[i+1] = A[i]	c_5
i = i-1	c_6
A[i+1] = key	c_7

Insertion Sort - Analysis

While loop within for loop

- **For/while loop** : test is executed one time more than the loop body
- **Let t_j be the number of times** the while loop in line 5 is executed
- Since it is within a for loop, for each j = 2,3,…,n, where n = A.length, **total number of times while loop executed is $\Sigma_{j=2 \text{ to } n}\ t_j$**

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]		
i = j-1	c_3	$n - 1$
while i > 0 and A[i] > key	c_4	$\Sigma_{j=2 \text{ to } n} t_j$
A[i+1] = A[i]	c_5	$\Sigma_{j=2 \text{ to } n} (t_j-1)$
i = i -1	c_6	$\Sigma_{j=2 \text{ to } n} (t_j-1)$
A[i+1] = key	c_7	$n - 1$

Running time of an algorithm

- Sum of the running times for each statement executed
  - a statement that takes a cost of $\mathbf{c_i}$ to execute and is executed **n times**, **contribute $\mathbf{c_i}$ * n** to the total running time

T(n): running time of IS : sum of the products of the cost and times

T(n) = ?

What do you think is the best case for IS?

Input:
1,2,3,4,5,6,7,8,9,10

Input:
10,9,8,7,6,5,4,3,2,1

Best case of IS – Already sorted array

- For each j = 2,3,….,n, we know that **A[i] <= key** in line 5, i has its initial value of j – 1

i.e A[1] <= 2, for j = 2, A[2] <= 3, for j = 3, …..

- **Condition is FALSE and the body of the while loop will not be executed**
- Condition alone will be executed, therefore,

$t_j = 1$, for j = 2,3,…,n

- Best case running time:

$$T(n) = c_1 n + c_2 (n-1) + c_3 (n-1) + c_4 (n-1) + c_7 (n-1)$$

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

Best case of IS - Linear function

$T(n) = (c_1 + c_2 + c_3 + c_4 + c_7)n – (c_2 + c_3 + c_4 + c_7)$

$=$ a n + b, where a and b depend on the statement costs c_i

It’s a linear function of n

Worst case of IS - reverse sorted

Input: 10,9,8,7,6,5,4,3,2,1

Compare each element A[j] with each element in the entire sorted subarray A[1… j-1]

i.e for j = 2, A[2] will be compared with A[1]

Resultant array: 9,10,8,7,6,5,4,3,2,1

for j = 3, A[3] will be compared with A[2] and A[1]

Resultant array: 8, 9,10,7,6,5,4,3,2,1

What is the value of tj in the worst case?

5.while i > 0 and A[i] > key c_4 $\Sigma_{j=2 \text{ to } n}\ t_j$

6. A[i+1] = A[i]

What is t_j for the worst case?

$t_j = j$, for $j = 2, 3, \ldots, n$

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

and

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

$T(n) = ?$

Insertion Sort - Worst case running time

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

Worst case running time

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

$T(n) = a\, n^2 + b\, n + c$, for constants a, b and c that depends on the statement costs c_i

T(n) quadratic function of n

Worst case running time

- Longest running time for any input of size n
- Upper bound on the running time for any input
- Provides a guarantee that the algorithm will not take more than the specified value
- Worst case occurs fairly often – Searching a

Bubble Sort- Visualization

Which among the two

takes more swaps?

Thank You !!!

$$\text{Cost of successful search} = \sum_{t=1}^{\log n+1} \frac{1}{n} \cdot t \cdot 2^{t-1} = \frac{1}{n} \sum_{t=1}^{\log n+1} \frac{d}{d2}(2^t)$$

$$= \frac{1}{n} \; \frac{d}{d2} \sum_{t=1}^{\log n+1} (2^t) = \frac{1}{n} \; \frac{d}{d2} 2^{\log n+2} - 1 = \frac{1}{n} \; 4 \log n \cdot 2^{\log n-1} = \frac{1}{2n} 4 \log n \cdot n^{\log 2}$$

Reference:

http://iiitdm.ac.in/old/Faculty_Teaching/Sadagopan/pdf/ADSA/new/average-analysis.pdf