Design and Analysis of Algorithms

Bilal Khalid Dar

Properties of O(n)

Properties of the O notation

- Constant factors may be ignored
  - $\forall\ k > 0,\ kf$ is $O(f)$
- Higher powers grow faster
  - n^r is $O(n^s)$ if $0 \le r \le s$
- Fastest growing term dominates a sum
  - If f is $O(g)$, then $f + g$ is $O(g)$
  - $eg\ \ an^4 + bn^3$ is $O(n^4)$
- Polynomial's growth rate is determined by leading term
  - If f is a polynomial of degree d,
    then f is $O(n^d)$

Properties of the O notation

Important!

- f is $O(g)$ is transitive
  - If f is $O(g)$ and g is $O(h)$ then f is $O(h)$
- Product of upper bounds is upper bound for the product
  - If f is $O(g)$ and h is $O(r)$ then fh is $O(gr)$
- Exponential functions grow faster than powers
  - n^{k} is $O(b^{n})$ $\forall$ $b > 1$ *and* $k \geq 0$
    e.g. n^{20} *is* $O(1.05^{n})$
- Logarithms grow more slowly than powers
  - $\log_b n$ is $O(n^{k})$ $\forall$ $b > 1$ *and* $k > 0$
    e.g. $\log_2 n$ *is* $O(n^{0.5})$
- All logarithms grow at the same rate
  - $\log_b n$ is $O(\log_d n)$ $\forall$ $b, d > 1$

Order of growth

- We usually consider one algorithm to be more efficient than another if its worst case running time has a lower order of growth.
- Due to constant factors and lower order terms, an algorithm whose running time has a higher order of growth might take less time for small inputs than an algorithm whose running time has a lower order of growth. But for large enough inputs, $\Theta(n^2)$ algorithm, for example, will run more quickly in the worst case than $\Theta(n^3)$ algorithm

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

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

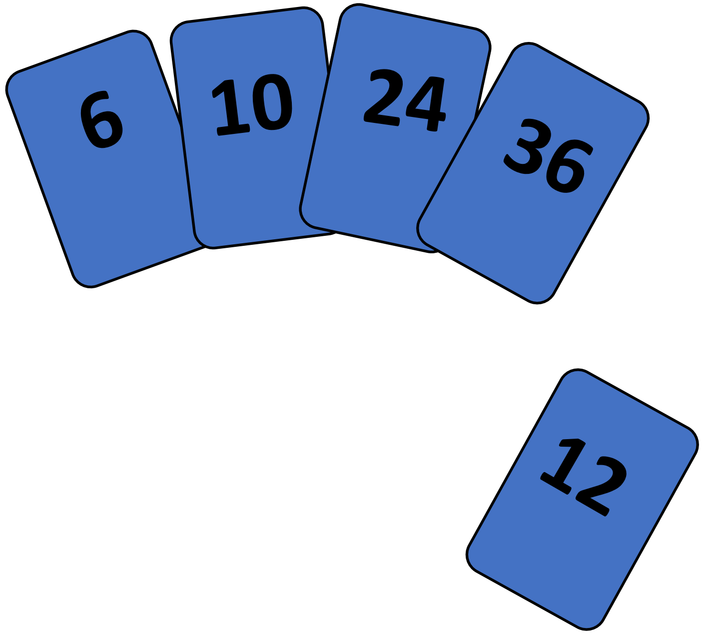
Insertion Sort

- 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

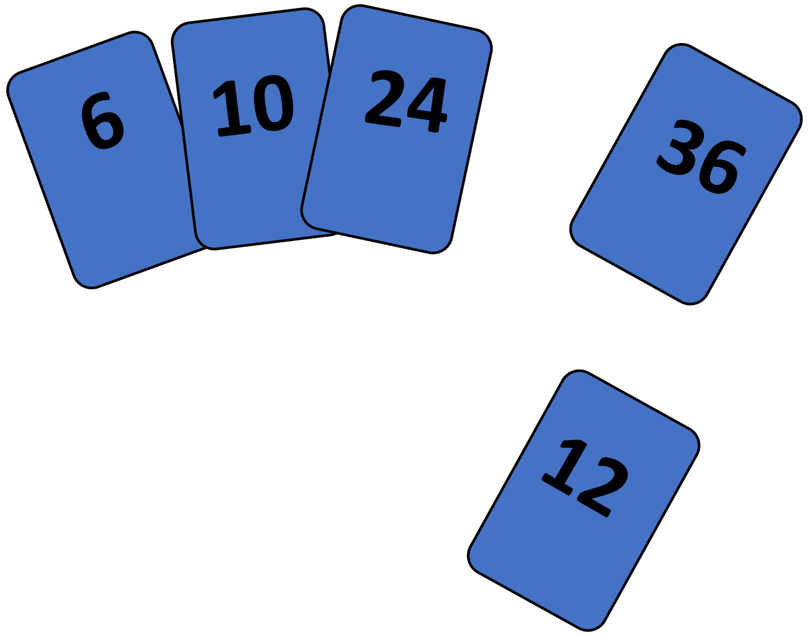
- The list is divided into two parts: sorted and unsorted
- In each pass, the following steps are performed
  - First element of the unsorted part (i.e., sub-list) is picked up
  - Transferred to the sorted sub-list
  - Inserted at the appropriate place
- A list of n elements will take at most n-1 passes to sort the data

Insertion Sort

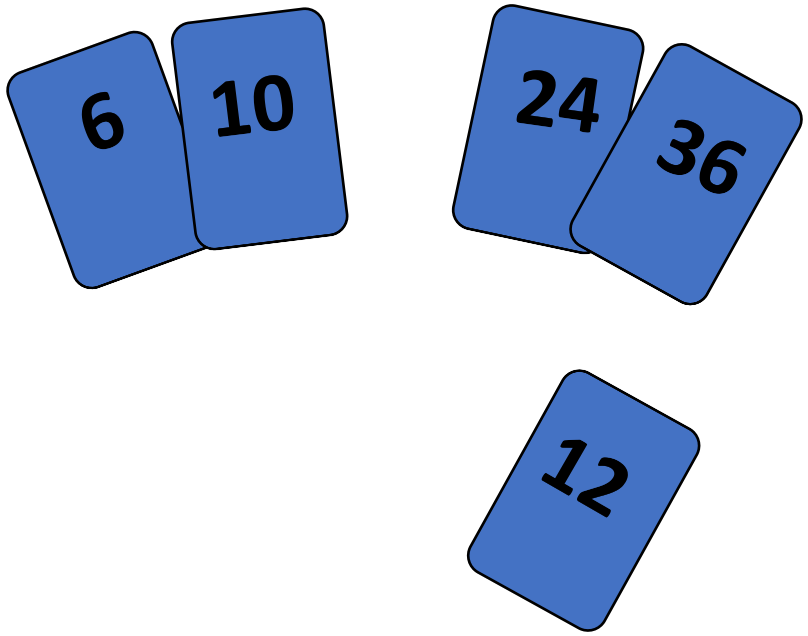


To insert 12, we need to make room for it by moving first 36 and then 24.

Insertion Sort



Insertion Sort



Example of Insertion Sort

8 2 4 9 3 6

Example of Insertion Sort

8 2 4 9 3 6

Example of Insertion Sort

8 2 4 9 3 6

2 8 4 9 3 6

Example of Insertion Sort

8	2	4	9	3	6
2	8	4	9	3	6

Example of Insertion Sort

8	2	4	9	3	6
2	8	4	9	3	6
2	4	8	9	3	6

Example of Insertion Sort

8 2 4 9 3 6

2 8 4 9 3 6

2 4 8 9 3 6

Example of Insertion Sort

8	2	4	9	3	6
2	8	4	9	3	6
2	4	8	9	3	6
2	4	8	9	3	6

Example of Insertion Sort

8	2	4	9	3	6
2	8	4	9	3	6
2	4	8	9	3	6
2	4	8	9	3	6

Example of Insertion Sort

8	2	4	9	3	6
2	8	4	9	3	6
2	4	8	9	3	6
2	4	8	9	3	6
2	3	4	8	9	6

Example of Insertion Sort

8	2	4	9	3	6
2	8	4	9	3	6
2	4	8	9	3	6
2	4	8	9	3	6
2	3	4	8	9	6

Example of Insertion Sort

8	2	4	9	3	6	
2	8	4	9	3	6	
2	4	8	9	3	6	
2	4	8	9	3	6	
2	3	4	8	9	6	
2	3	4	6	8	9	*done*

Insertion Sort

input array

5 2 4 6 1 3

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

left sub-array

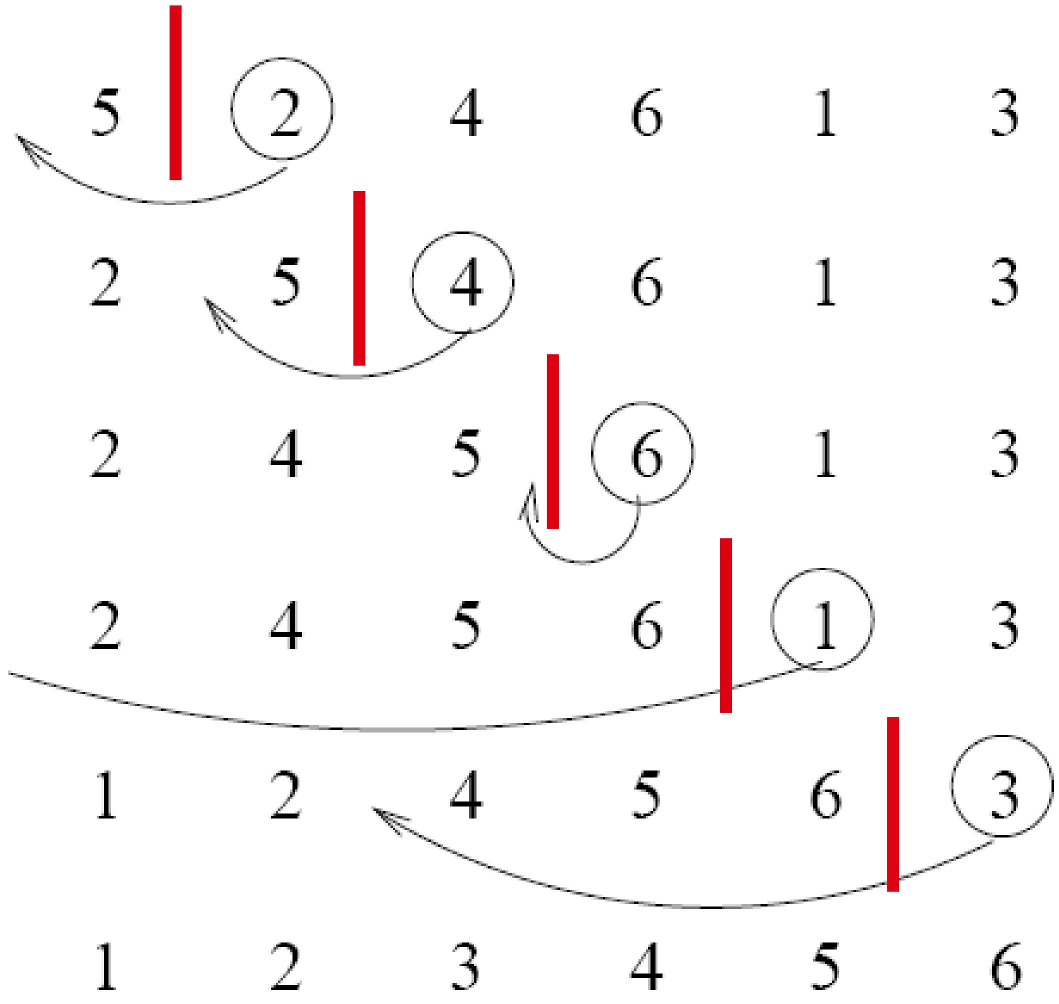
right sub-array

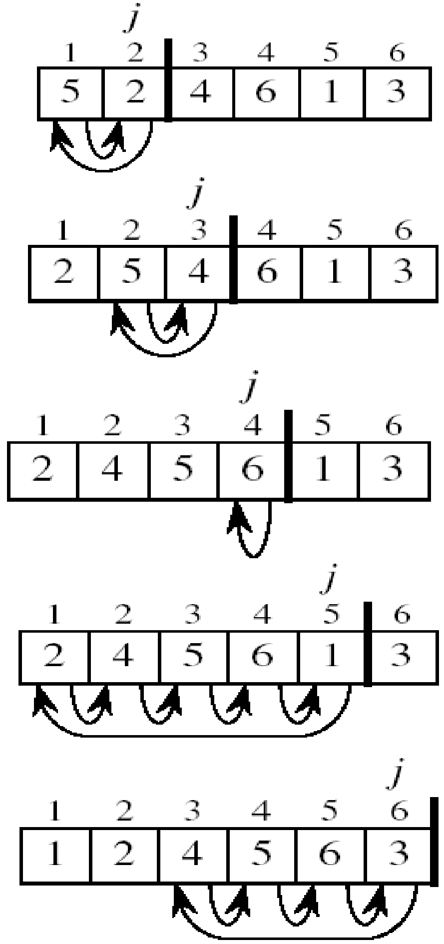
2 5 | 4 6 1 3

sorted

unsorted

Insertion Sort





Best Case

1	2	3	4	5

Worst Case

5	4	3	2	1

Running time of Insertion Sort

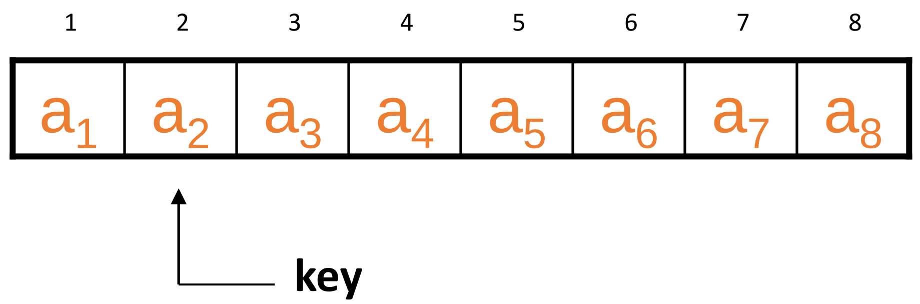
- The running time depends on the input: an already sorted sequence is easier to sort.
- Parameterize the running time by the size of the input,
  - short sequences are easier to sort than long ones.
- Generally, we seek upper bounds on the running time, because everybody likes a guarantee.

Kinds of Analyses

- Worst-case: (Usually)
  - T(n) = (maximum time of algorithm) on any input of size n.
- Average-case (Sometimes):
  - T(n) = (expected time of algorithm) over all inputs of size n.
  - Need assumption of statistical distribution of inputs
- Best-case: (bogus)
  - Cheat with a slow algorithm that works fast on some input

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

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] ≤ key upon the first time the **while** loop test is run

    (when $i = j - 1$)

  - $t_j = 1$

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

$$= n \bullet (c_1 + c_2 + c_4 + c_5 + c_8)$$

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

$$= c_9 n + c_{10}$$

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

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	≈ $n^2/2$ comparisons	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	≈ $n^2/2$ exchanges	c_7	$\sum_{j=2}^{n} (t_j - 1)$
A[i + 1] ← key		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:

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

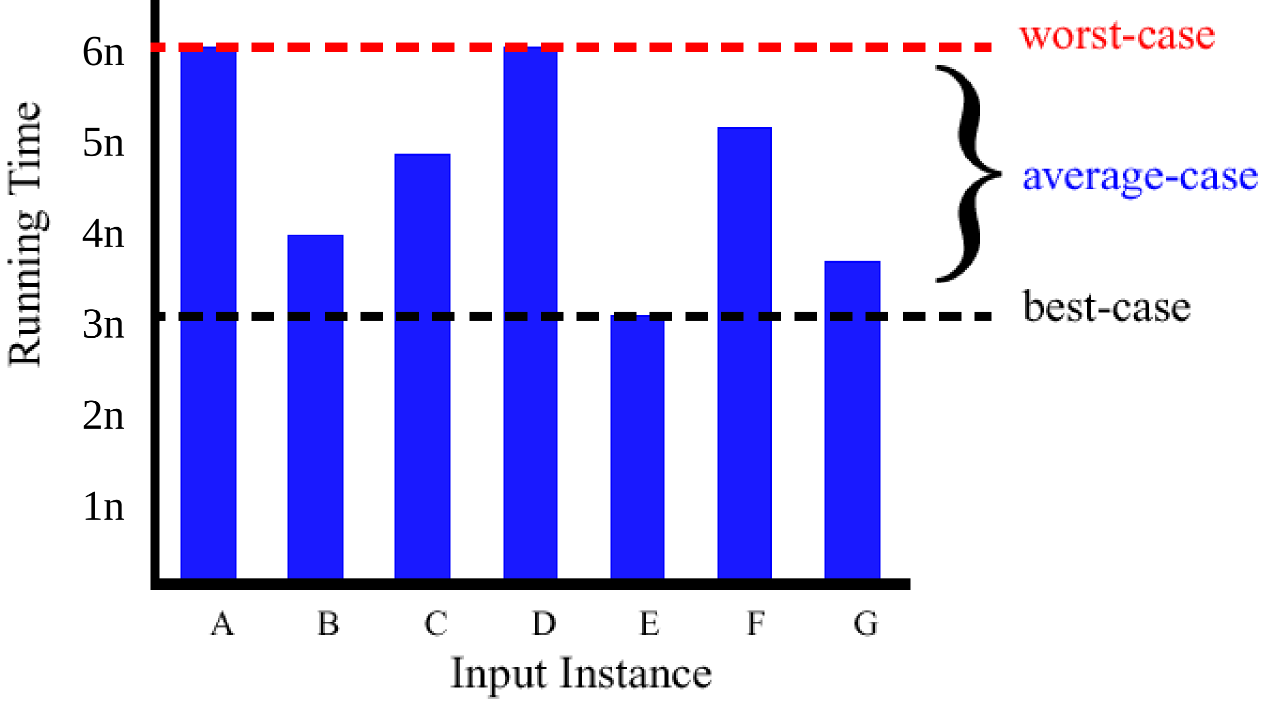
$$= an^2 + bn + c$$ a quadratic function of n

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

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

Best/Worst/Average Case (2)

- For a specific size of input *n*, investigate running times for different input instances:



Insertion Sort Analysis

- Is insertion sort a fast sorting algorithm?
  - Moderately so, for small n
  - Not at all, for large n
  - sorting "almost sorted" lists

Reference

- Introduction to Algorithms
- Chapter # 2
  - Thomas H. Cormen
  - 3rd Edition

- https://visualgo.net/bn/sorting