

When we say

“the algorithm A runs in quadratic time.”,

we mean

“the algorithm A is in $O(n^2)$.”

that is (mostly),

“the worst case $T(n)$ of A is in $O(n^2)$ ”.

1 Insertion Sort: Example

Here we use an example to show that even for inputs of a given size n , an algorithm's running time $T(n)$ may depend on exactly which input of that size is given. For the insertion example (sort a deck of cards), let us consider the following operations as steps:

- remove a card from the table;
- insert a card to the indexed position;
- move a card one position to its right;
- comparison of the values of two cards, or checking the whether the current pointer is out of the boundary (i.e., left-most position reached).

A deck of five cards in the order $\langle 2, 5, 4, 1, 3 \rangle$ would require 27 steps ($T(5) = 27$). The maximal of $T(5)$ is 35, where the deck of cards is $\langle 5, 4, 3, 2, 1 \rangle$. The minimal of $T(5)$ is 15, where the deck of cards is $\langle 1, 2, 3, 4, 5 \rangle$. $T(5)$ is between 15 and 35 for any case (e.g., try $\langle 1, 3, 2, 5, 4 \rangle$).

2 Time Complexity Analysis for Algorithms/Programs

2.1 Definitions

For an algorithm A , let $t(x)$ represents the number of steps (running time) taken by A on input x , where the size of x is n , we define the **worst-case running time** of A on input of size n to be maximum time/steps taken by A on all inputs of size n :

$$T(n) = \max\{t(x) : x \text{ is an input of size } n\};$$

The **best-case running time** of A on input of size n to be minimum time/steps taken by A on all inputs of size n :

$$T(n) = \min\{t(x) : x \text{ is an input of size } n\}.$$

2.2 Textbook Examples

For examples in this section, worse-case is the same as the best-case.

Example 1. (Textbook Example 3.9)

The statement $a \leftarrow 2$ takes constant time i.e., the statement is in $O(1)$.

Example 2. (Textbook Example 3.10)

Let assignments, increments, arithmetic operations, logical comparisons, all are considered as steps. Take the following program:

program	steps

1: <code>sum <- 0</code>	1
2: <code>for i <- 1 to n</code>	$1+(n+1)+n = 2n+2$
3: <code>sum <- sum + 14</code>	$n+n = 2n$

The running time is

$$T(n) = 1 + 2n + 2 + 2n = 4n + 3 \in O(n).$$

Note: the sum of steps needed for line 2 and line 3 is $2n + 2 + 2n = (4n + 2)$, which is referred in our next example.

Example 3. (Textbook Example 3.11)

program	steps

1: <code>sum <- 0</code>	1
2: <code>for i <- 1 to n</code>	$1+(n+1)+n = 2n+2$
3: <code>for j <- 1 to i</code>	
4: <code>sum <- sum + 3</code>	

We know that the j loop in line 3 and line 4 of the above example iterates with respect to i not to n . Hence, For each i from 1 to n , line 3 and line 4 takes $4i + 2$ time. Hence in total we have

$$\sum_{i=1}^n (4i + 2) = 2n + \frac{4n(n+1)}{2} = 2n^2 + 4n$$

The running time is

$$T(n) = 1 + (2n + 2) + 2n^2 + 4n = 2n^2 + 4n + 3 \in O(n^2).$$

2.3 Example: ListSearch

Consider the following example program that searches a singly-linked list for a specific key k . Remember that $length(L)$ is the length of the list, $head(L)$ refers to the head of the list, and a node z contains two parts: its key $key(z)$ and a pointer $next(z)$ pointing to a node next to it in the list.

```
ListSearch(L, k)
1: z <- head(L)
2: while (z != NIL and key(z) != k)
3:   z <- next(z)
4: return z
```

Hence, $n = \text{length}(L)$ and one comparison is considered as one step. What is $t(L, k)$, the number of comparisons performed by *ListSearch* on input (L, k) ?

$$t(L, k) = \begin{cases} 2i & \text{if } k \text{ occurs in } L \text{ at index } i \text{ of } L \\ 2n + 1 & \text{if } k \text{ does not occur in } L \end{cases}$$

(Since two comparisons are performed for every iteration of the loop, and the loop goes through i iterations when k occurs in position i of L ; the extra comparison when $k = 0$ is because of the last test for $z \neq \text{NIL}$ when we reach the end of the list.

In this case, we immediately get that for the worst-case, $T(n) = 2n + 1$ since that is the maximum value of $t(L, k)$ for all lists L that contain n elements.) What is the best-case running time of *ListSearch* on inputs (L, k) such that $\text{length}(L)$ is n ? From the definition of $t(L, k)$, the best scenario is the k occurs first in the list, in which case the running time is 2 (independent of n).

Average-case running time

In order to perform average-case analysis, we need to know some basics of probability theory, in particular the concepts of *sample space*, *probability distribution*, and *the expected value* of a random variable. You are good enough however if you have no problem understanding the experiment of rolling a balanced die once. The random variable X is then defined as the number obtained from rolling the die once. The sample space $S(X)$ for X consists of equally likely outcomes (i.e., the probability is $1/6$ for each number ranging from 1 to 6). Hence we have a uniform probability distribution here for X . $E(X)$, the expected value of X , is defined as

$$1 \cdot 1/6 + 2 \cdot 1/6 + 3 \cdot 1/6 + 4 \cdot 1/6 + 5 \cdot 1/6 + 6 \cdot 1/6 = 21/6 = 3.5$$

Let S_n be the sample space of all inputs of size n . To compute the average-case running time of an algorithm A , we must specify a probability distribution over S_n , i.e., we specify how likely each input is. $t(x)$ is now a random variable. Now we can define $T(n)$ as follows,

$$T(n) = E[t(x) : x \in S_n] = \sum_{x \in S_n} t(x) \cdot \text{Pr}(x).$$

For *ListSearch* what is S_n ? All we need is for S_n to contain one input for each possible behavior of the algorithm. We can define S_n as follows:

$$S_n = \{(L, k) : L = [1, 2, \dots, n], k \text{ is one of } 0, 1, 2, \dots, n\},$$

Since this set of inputs represents every possible behavior of the algorithm. That is,

$$S_n = \left\{ \begin{array}{l} k \text{ occurs in position 1 of } L, \\ k \text{ occurs in position 2 of } L, \\ \dots \quad \dots \quad \dots \quad \dots \quad \dots, \\ k \text{ occurs in position } n \text{ of } L, \\ k \text{ does not occur in } L \end{array} \right\}.$$

(For example, the behaviour of the algorithm on input “ $L = \{76, 9, 15\}$, $k = 15$ ” is exactly the same as “ $L = \{1, 2, 3\}$, $k = 3$ ”.)

Without a particular application to determine the probability distribution, we simply assume a uniform distribution, i.e., each input is equally likely (with $Pr = \frac{1}{n+1}$), then,

$$\begin{aligned}
 T(n) &= E[t(L, k) : (L, k) \in S_n] \\
 &= \sum_{(L, k) \in S_n} \left(t(L, k) \cdot Pr(L, k) \right) \\
 &= \sum_{k=0}^n t(L, k) \cdot \frac{1}{n+1} \\
 &= \frac{1}{n+1} (t(L, 0) + \sum_{k=1}^n t(L, k)) \\
 &= \frac{1}{n+1} \cdot (2n+1) + \frac{1}{n+1} \sum_{k=1}^n 2k \\
 &= \frac{2n+1}{n+1} + \frac{2}{n+1} \cdot \frac{n(n+1)}{2} = n + \frac{2n+1}{n+1}.
 \end{aligned}$$

Notice that this is consistent with our intuition: if we search for elements in a list when the elements are equally likely to be anywhere in the list, we expect that on average we have to look through approximately half the list. And in fact, $T(n)$ is approximately equal to n , which is half of $T(n)$, the worst-case. The fact that $T(n)$ is slightly larger than n is caused by the extra case when k is not in the list, which shifts the average slightly up.

Example 4.