International Islamic University Chittagong
Department of Computer Science & Engineering
Autumn - 2022

Course Code: CSE-2321
Course Title: Data Structures

Mohammed Shamsul Alam
Professor, Dept. of CSE, IIUC

Lecture – 4

Complexity of Algorithms

Algorithms

A finite set of instructions or logic, written in order, to accomplish a certain predefined task.

- Algorithm is not the complete code or program
- It is just the core logic (solution) of a problem
- Can be expressed either as an informal high level description as pseudo code or using a flowchart.

Characteristics of an Algorithm

- Unambiguous − Algorithm should be clear and unambiguous.
- Input − An algorithm should have 0 or more well defined inputs.
- Output − An algorithm should have 1 or more well defined outputs
- Finiteness − Algorithms must terminate after a finite no. of steps.
- Feasibility − Should be feasible with the available resources.
- Independent − An algorithm should have step-by step directions which should be independent of any programming code.

Algorithm Analysis

An algorithm is said to be efficient and fast, if it takes less time to execute and consumes less memory space.

The performance of an algorithm is measured on the basis of following properties:

1. Time Complexity
2. Space Complexity

Suppose X is an algorithm and **n** is the size of input data, the time and space used by the Algorithm X are the two main factors which decide the efficiency of X.

- Time Factor − The time is measured by counting the number of key operations such as comparisons in sorting algorithm
- Space Factor − The space is measured by counting the maximum memory space required by the algorithm.

Complexity of Algorithms

The **complexity of an algorithm** M is the function *f(n)* which gives the *running time* and/or *storage space* requirement of the algorithm in terms of the size *n* of the input data.

- Frequently, the storage space requirement by an algorithm is simply a multiple of the data size n. So, the term complexity refers to the running time of the algorithm.
- The complexity of an algorithm can be discussed in two cases:

a. *Worst case*: the combination of input data for which the complexity *f* (n) is maximum.

b. *Average case*: the combination of input data for which the complexity *f* (n) is not maximum and not minimum but has an expected value.

Sometimes we discuss about the *Best case* which is the combination of input data for which the complexity *f* (n) is minimum.

Example 2.7

Suppose we are given an English short story TEXT, and suppose we want to search through TEXT for the first occurrence of a given 3-letter word W. If W is the 3-letter word "the," then it is likely that W occurs near the beginning of TEXT, so $f(n)$ will be small. On the other hand, if W is the 3-letter word "zoo," then W may not appear in TEXT at all, so $f(n)$ will be large.

Asymptotic Notations

The main idea of asymptotic analysis is to have a measure of efficiency of algorithms that doesn't depend on machine specific constants, and doesn't require algorithms to be implemented and time taken by programs to be compared.

Asymptotic notations are mathematical tools to represent time complexity of algorithms for asymptotic analysis.

The following 3 asymptotic notations are mostly used to represent time complexity of algorithms.

- O Notation
- Ω Notation
- θ Notation

Rate of Growth

Suppose *M* is an algorithm, and suppose **n** is the size of the input data. Clearly the complexity *f*(n) *of M* increases as n increases. It is usually the rate of increase of *f* (n) that we want to examine. This is usually done by comparing *f* (n) with some standard function, such as

1, $\log_2 n$, n, $n\log_2 n$, n^2, n^3, 2^n, n!

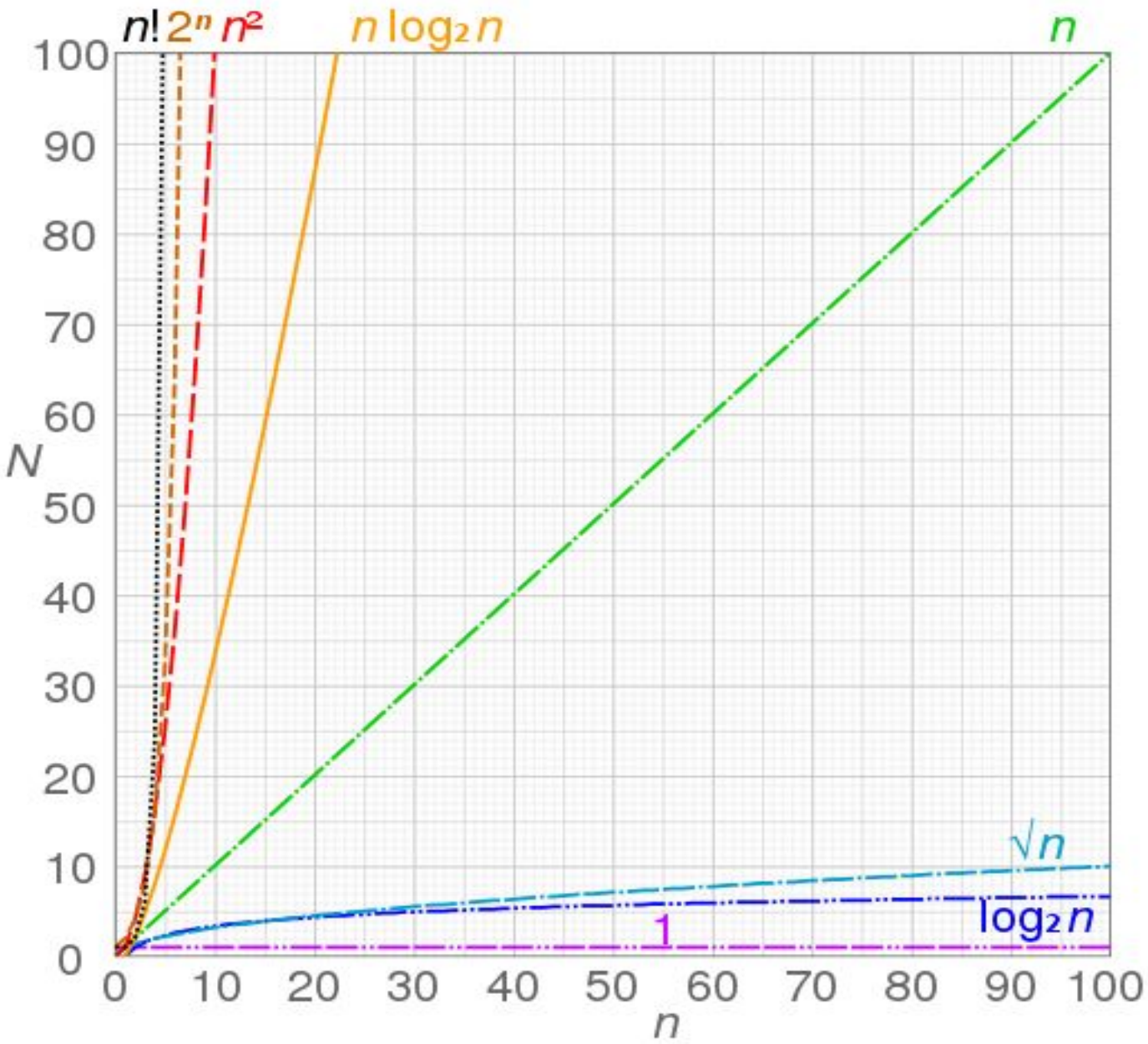
The rates of growth for these standard functions are indicated in the following Table:

n \ g(n)	log n	n	n log n	n^2	n^3	2^n
5	3	5	15	25	125	32
10	4	10	40	100	10^3	10^3
100	7	100	700	10^4	10^6	10^{30}
1000	10	10^3	10^4	10^6	10^9	10^{300}

Fig. 2.6 Rate of Growth of Standard Functions

Observe that the functions are listed in the order of their rates of growth: the logarithmic function $\log_2 n$ grows most slowly, the exponential function 2^n grows most rapidly, and the polynomial functions n^c grow according to the exponent c.

Rate of Growth

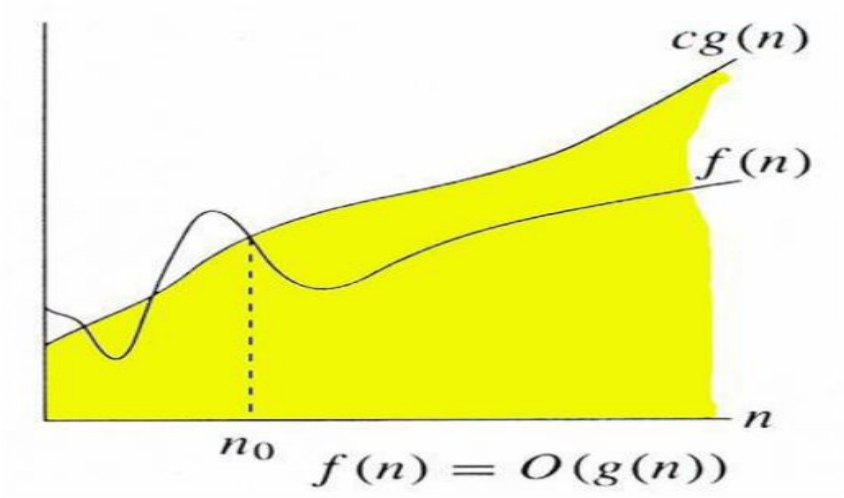


Big O Notation

It would be convenient to have a form of asymptotic notation that means "the running time grows at most this much, but it could grow more slowly." We use "big-O" notation for just such occasions.

Suppose *f* (n) and *g* (n) are functions defined on the positive integers with the property that *f*(n) is bounded by some multiple of *g* (n) for almost all n. That is, suppose there exist a positive integer n_0 and a positive number C such that, for all $n > n_0$, we have, $|f(n)| \leq C\,|g(n)|$

Then we may write, *f* (n) = O (g (n)) which is read as "*f* (n) is the order of *g* (n)".

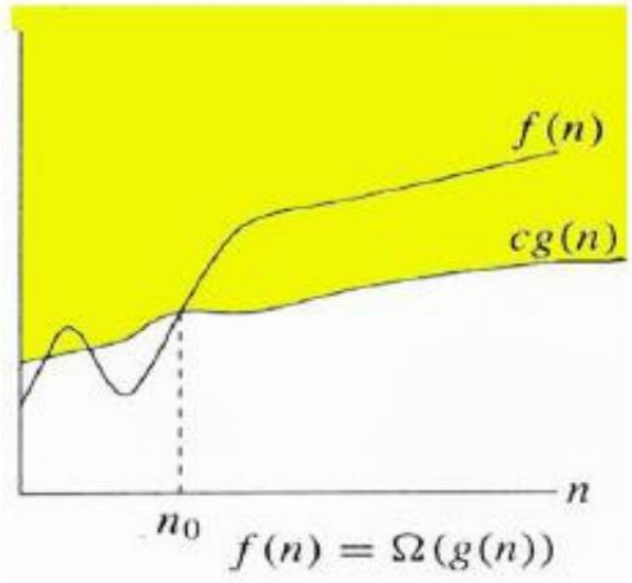


Ω (Omega) notation

Sometimes, we want to say that an algorithm takes *at least* a certain amount of time, without providing an upper bound. We use Ω notation; that's the Greek letter "omega.".

Suppose *f* (n) and *g* (n) are functions defined on the positive integers with the property that *f*(n) is bounded by some multiple of *g* (n) for almost all n. That is, suppose there exist a positive integer n_0 and a positive number C such that, for all $n \geq n_0$, we have, $|f(n)| \geq C\,|g(n)|$

Then we may write, *f* (n) = Ω (g (n)) which is read as "*f* of n is the omega of g of n".



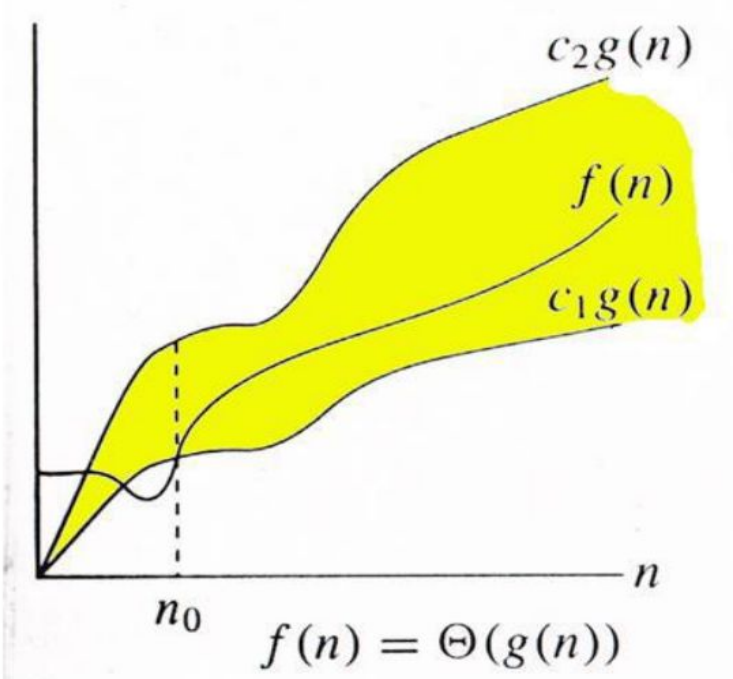
θ (Theta) notation

We use Θ notation to asymptotically bound the growth of a running time to within constant factors above and below.

Suppose *f* (n) and *g* (n) are functions defined on the positive integers with the property that *f*(n) is bounded by some multiple of *g* (n) for almost all n. That is, suppose there exist a positive integer n_0 and two positive numbers C_1 and C_2 such that, for all $n \geq n_0$, we have,

$$C_1 |g(n)| \leq |f(n)| \leq C_2 |g(n)|$$

Then we may write, *f* (n) = Θ (g (n)) which is read as "*f* of n is theta of g of n".



Rate of growth

In algorithms, we focus on estimating the execution time as a function of the inputs. In particular, the main focus is how quickly does the execution time grow as a function of the inputs. For example, f(n)=5n and g(n)=100n+1000 are equivalent functions in terms of growth with respect to n. We simply say both the functions grow linearly. Whereas, f(n)=5n and h(n)=n^2 are not equivalent, since h(n) grows quadratically.

Let's take an example. Take a look at the following pseudo-code.

```
def function1(n):
    a = 0
    for (i = 0; i < n; i += 1)
        for (j = 0; j < n; j += 1)
            a += 1
    return a

def function2(n):
    a = 0
    for (i = 0; i < n; i += 1)
        a += n
    return a

def function3(n):
    return n * n
```

Big O Notation

In the above code, function1 takes time proportional to n^2. In particular, notice that the operation a += 1 executes n^2 times. Similarly, function2 takes time proportional to n and function3 takes constant amount of time to run regardless of the values of n. We write the run-times of function1 as $O(n^2)$, function2 as O(n) and function3 as O(1).

Having a run-time of O(n) means that the run-time is less than or equal to c*n, for some constant value c (doesn't depend on n). For function2 in the above example, assuming variable assignment, addition, comparison are one operation each, we can say that number of operations being done is < 10n. In particular, we do approximately 2n assignments, 2n additions and n comparisons, which is a total of 5n operations (Here, we're treating a += 1 as an addition operation followed by an assignment operation. Don't forget to count operations done on the loop variable i).

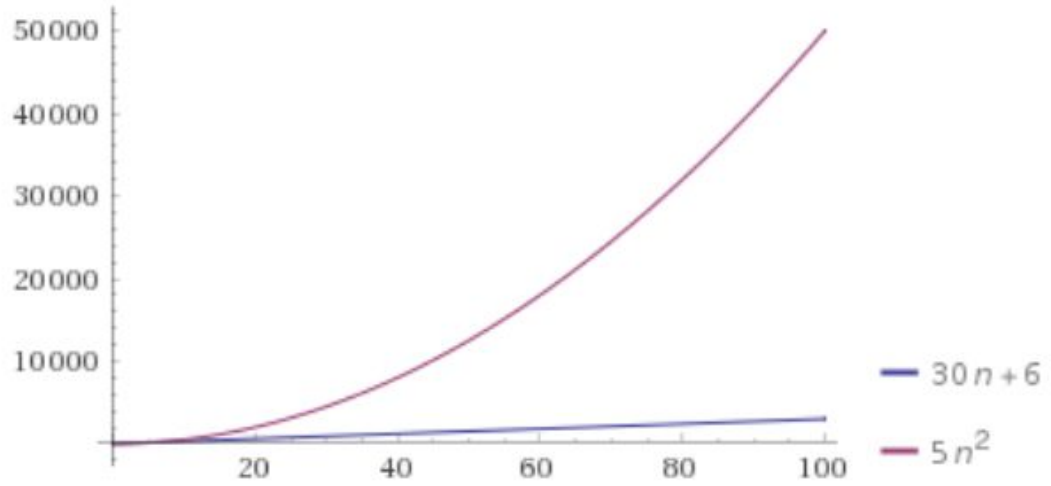
Writing a function in Big-O notation

Converting a function to big-O notation requires two simple rules.

Only consider the highest-order term. (Ignore all lower order-terms.)

Ignore any coefficients or constant factors.

Let's take an example. Suppose the total number of operations being done is $5n^2+30n+6$. In big-O notation, this function is $O(n^2)$. To get this, first we drop the lower order terms (30n and the constant 6), and only keep the highest-order term $5n^2$. Next, we drop the constant factor 5.



In our earlier example, we had f(n)=5n=O(n) and g(n)=100n+1000=O(n). Since both f(n) and g(n) are O(n), we said f(n) is equivalent to g(n). However, since $h(n)=n^2=O(n^2)$, we concluded that h(n) is not equivalent to f(n).

Why do we drop the constant factors?

The big-O notation ignores constants to focus on rate of growth. In the previous example, we can see that when we have a value of n that is 4 times larger, both f(n) and g(n) will become 4 times larger. However, h(n) will become 16 times larger. This is what we mean when we say that big-O notation focuses on how quickly does a function grow.

In practice though, we can't completely ignore constant factors. Usually all we need to keep in mind regarding constants is that if we did 10 times more operations in the part of the code executing the most number of times, then the code will take 10 times longer to execute. In the code examples we saw, this would be doing 10 times more operations inside the innermost for loop. That is, we try to roughly estimate the constant factor for the highest-order term.

However, assigning the constants meaning in an absolute sense is not feasible since there are other constants buried all over the place. For example, different operations - addition vs multiplication vs comparison - take different amounts of time to execute. Each hardware takes different amounts of time to perform different operations, and some hardwares might be capable of combining two operations into one (is a += 1 one operation - increment, or two operations - addition followed by assignment).

Common algorithm complexities and their run-times

Below table lists some of the commonly occurring run-time complexities. The second column indicates what is the largest value of n for which the code would finish running within 1 second (assuming 1 billion operations per second, and constant c to be about 5-10). The third column provides examples of algorithms (many of which we'll learn in this course) with the stated run-time complexity.

Complexity	Max n	Examples
O(log n)	Very Large!	Binary search, Exponentiation by repeated squaring
O(n)	100,000,000	Linear search, Breadth-first search
O(n log n)	5,000,000	Merge-sort, Quick-sort, Heap-sort, Djikstra's algo
O(n^2)	10,000	Bubble-sort
O(n^3)	400	All-pairs shortest paths
O(2^n)	25	Listing all subsets
O(n!)	11	Listing all permutations

Exercise on Big O Notation

Exercise

a.

```
sum = 0;
for( i = 0; i < n; i++)
    for( j = 0; j < n * n; j++)
            sum++;
```

Ans : O(n^3)

b.

```
sum = 0;
for( i = 0; i < n; i++)
    for( j = 0; j < i; j++)
            sum++;
```

Ans : O(n^2)

c.

```
sum = 0;
for( i = 0; i < n; i++)
    for( j = 0; j < i*i; j++)
            for( k = 0; k < j; k++)
                    sum++;
```

Ans : O(n^5)

d.

```
sum = 0;
for( i = 0; i < n; i++)
    sum++;
    val = 1;
for( j = 0; j < n*n; j++)
    val = val * j;
```

Ans : O(n^2)

Space Complexity

Space complexity is the total amount of memory space used by an algorithm/program including the space of input values for execution. So to find space complexity, it is enough to calculate the space occupied by the variables used in an algorithm/program.

- But often, people confuse Space complexity with **Auxiliary space**. Auxiliary space is just a temporary or extra space and it is not the same as space complexity. In simpler terms,

Space Complexity = Auxiliary space + Space use by input values

Why do you need to calculate space complexity?

Similar to Time Complexity, Space complexity also plays a crucial role in determining the efficiency of an algorithm/program. If an algorithm takes up a lot of time, you can still wait, run/execute it to get the desired output. But, **if a program takes up a lot of memory space, the compiler will not let you run it**.

How to calculate Space Complexity of an Algorithm?

Example-1:

```
int sum(int x, int y, int z) {
  int r = x + y + z;
  return r;
}
```

requires 3 units of space for the parameters and 1 for the local variable, and this never changes, so space complexity is **O(1)**.

Example-2:

```
int sum(int a[], int n) {
  int r = 0;
  for (int i = 0; i < n; ++i) {
    r += a[i];
  }
  return r;
}
```

requires N units for a, plus space for n, r and i, so it's **O(N)**.

Complexity of the Linear Search Algorithm

The complexity of linear search algorithm is measured by the number *f(n) of* comparisons required to find ITEM in DATA where DATA contains *n* elements. Two important cases to consider are the average case and the worst case.

Worst Case: Clearly the worst case occurs when ITEM is the last element in the array DATA or is not there at all. In either situation, we have *f* (n) = n.

Accordingly, *f* (n) = n = O (n) is the worst-case complexity of the linear search algorithm.

Average Case: Here we assume that ITEM does appear in DATA and that it is equally likely to occur at any position in the array. Accordingly, the number of comparisons can be any of the numbers 1, 2, 3, ... , n, and each number occurs with probability p = 1/n. Then we can write

$$f(n) = 1.\frac{1}{n} + 2.\frac{1}{n} + \ldots\ldots\ldots\ldots n.\frac{1}{n} = (1+2+3+\ldots\ldots\ldots\ldots n).\frac{1}{n}$$

$$= \frac{n(n+1)}{2}.\frac{1}{n} = \frac{n+1}{2} = \text{O (n)}$$

That is, the average number of comparisons required to find the location of ITEM is approximately equal to half the number of elements in the array.

Complexity of the Binary Search Algorithm

The complexity is measured by the number $f(n)$ of comparisons to locate ITEM in DATA where DATA contains n elements. Observe that each comparison reduces the sample size in half. Hence we require at most $f(n)$ comparisons to locate ITEM where

$$2^{f(n)} > n \text{ or equivalently } f(n) = \log_2 n + 1$$

That is, the running time for the worst case is approximately equal to $\mathbf{\log_2 n}$. One can also show that the running time for the average case is approximately equal to the running time for the worst case.

Example 4.11

Suppose DATA contains 1 000 000 elements. Observe that

$$2^{10} = 1024 > 1000 \quad \text{and hence} \quad 2^{20} > 1000^2 = 1\,000\,000$$

Accordingly, using the binary search algorithm, one requires only about 20 comparisons to find the location of an item in a data array with 1 000 000 elements.

Complexity of the Bubble Sort Algorithm

The number of comparisons *f* (n) in bubble sort does not depend on the primary arrangement of data. Hence, for both worst and average case in bubble sort, there is a constant number of comparisons, which depends only on n, the number of input data.

The number *f* (n) of comparisons in the bubble sort is easily computed. Specifically, there are n - 1 comparisons during the first pass, which places the largest element in the last position; there are n - 2 comparisons in the second step, which places the second largest element in the next-to-last position; and so on. Thus

$$f(n) = (n-1) + (n-2) + \ldots + 2 + 1 = n * (n-1) / 2 = n^2 / 2 + O(n) = O(n^2)$$

In other words, the time required to execute the bubble sort algorithm is proportional to $\mathbf{n^2}$, where n is the number of input items.