

Chapter 28 – Performance Analysis

1. What is Performance Analysis?

In order to be accessed, data is stored in a data structure. Often, there is a choice of data structures that can be used to store the data. Almost any data structure can be accessed by more than one algorithm. Frequently, more than one algorithm performs the same action.

In order to choose between data structures and algorithms, the efficiency of competing data structures (in terms of space required) and competing algorithms (in terms of time used) must be compared. From this chapter on, each algorithm presented will be briefly discussed in terms of its efficiency.

2. *A Priori* Analysis

The best test of a data structure or algorithm is to measure its efficiency in use. This is called *a posteriori* (after the fact) analysis. Unfortunately, this requires programmer time and resources that are usually unavailable. Since the choice of a data structure or algorithm is generally made before a program is written, *a priori* (before the fact) analysis is necessary. The rest of the chapter discusses forms of *a priori* analysis.

3. Frequency Count Analysis

A frequency count of statements executed is the most direct form of a priori analysis of the time used by an algorithm. Each statement in a program adds the value of 1 to the frequency count each time it is executed. The major failing of frequency count analysis of an algorithm written in a high level programming language is that each statement in such a programming language does not generate the same number of binary instructions when compiled, nor do the binary instructions each take the same time to execute.

A frequency count of data structure elements used is the most direct form of a priori analysis of data structures. Each element in a data structure adds the value of 1 to a frequency count analysis of a data structure. The major failing of frequency count analysis of a data structure is that data elements in a high level programming language program are not all equal in space used.

Even though frequency count analysis produces a very specific looking result, remember that it really produces an approximation. In fact, all forms of *a priori* analysis are approximations. These approximations are tolerated because of they produce a *good enough* result that allows competing algorithms and data structures to be compared before programmer time is spent coding a solution. The assumption made in a frequency count analysis is that these differences will average out. Since they are approximations, the results can apply equally over solutions to be produced for all possible hardware.

Frequency Count Example 1

Here is a program fragment that will be used for a frequency count analysis:

```
cout << "Enter your name: ";
string name;
cin >> name;
```

The frequency analysis of this fragment is as follows:

```
cout << "Enter your name: ";           // add 1 to the time count
string name;                          // add 1 to the space count
cin >> name;                           // add 1 to the time count
```

Notice that the data declaration code is not counted in the time count. This example has a frequency count of 2 for time and a frequency count of 1 for space.

In the previous example, since the actual space the string object will take up will not be set until data is entered, it could be argued that the space added should be more than one. In fact, since the data will tend to an average size if enough are entered, adding 1 for each data item input will be an effective approximation, as long as the same approximation is used in the analysis of all competing data structures.

Frequency Count Example 2

Here is another program fragment for analysis, this one uses a loop:

```
for(int i=0; i<5; i++) {
    cout << i;
}
```

The frequency count analysis of this fragment is:

```
int sum=0;                               // add 1 to the time count, add 1 to the space count
for(int i=0; i<5; i++) {                 // add 6 to the time count, add 1 to the space count
    sum += i;                             // add 5 to the time count
}
cout << sum;                             // add 1 to the time count
```

This example has a frequency count for time of 13 and a frequency count for space of 2.

Note that compound statements symbols are not counted. Note that the for statement requires 1 more iteration than the body of the for loop. This is so that the for variable (i) can reach 5 and trigger the loop halt.

Frequency Count Example 3

Interesting things begin to happen when the number of repetitions in a loop or number of elements in a data structure is unknown.

In this example, as the number of elements in use in the array grows, the number of repetitions required to sum all of the populated elements increases in a 1-to-1 relationship. Typically, the letter n is used to represent such growth.

Here is a fragment that repeats an unknown number of times:

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

In this example, a is an integer array with elements that have been given values in such a way that the number of elements that are to be used (n) is not knowable in advance (when the program is written). Here is the analysis of this algorithm.

```
creation of array a           // add n to the space count
int sum=0;                    // add 1 to the time count, add 1 to the space count
for(int i=0; i<n; i++) {      // add n+1 to the time count, add 1 to the space count
    sum += a[i];              // add n to the time count
}
cout << sum;                  // add 1 to the time count
```

This is the summary of the time analysis for the previous example:

$$\begin{array}{r}
 1 \\
 + n + 1 \\
 + n \\
 + \quad 1 \\
 \hline
 2n + 3
 \end{array}$$

Therefore, the algorithm and data structures have the frequency counts of $n+1$ for space and $2n+3$ for time.

Frequency Count Example 4

Here is an example that populates the elements of an array with integers read from a file. The number of integers is unknown.

```

ifstream filein;
filein.open( "datafile.dat",ios::in);
int a[max];
int j, k=0;
filein >> j;
while (filein) {
    a[k] = j;
    k++;
    filein >> j;
}
filein.close();

```

In the analysis of this example, the number of integers in the file is represented by n and the maximum size of the array a is represented by m . Analysis is almost always performed using the *worst case* assumptions that n will be very large and m will be very large. In effect, n and m will be indistinguishable from one another. With that in mind, the size of the array a and the number of integers in the file can both be represented by n .

Here is the analysis of the above example fragment:

```

ifstream filein;           // 1 to space
filein.open("datafile.dat", ios::in); // add 1 to time
int a[max];               // add n to space
int j, k=0;               // add 1 to time, 2 to space
filein >> j;              // add 1 to time
while (filein) {         // add n+1 to time
    a[k] = j;            // add n to time
    k++;                 // add n to time
    filein >> j;         // add n to time
}
filein.close();          // add 1 to time

```

This gives the following analysis of time and space

<u>Time</u>	<u>Space</u>
1	1
+ 1	+ n
+ 1	+ <u>2</u>
+ n + 1	n + 3
+ n	
+ n	
+ n	
+ 1	
4n + 5	

Frequency Count Example 5

Algorithms involving nested loops are very common. The following is a fragment that uses a nested for-loop:

```
int b;
for (int x=0; x<5; x++)
    for (int y=0; y<4; y++) {
        b = x * y;
        cout << b;
    }
```

Since there are three simple variables, space use is 3. The only complication in analyzing this fragment for time is the nested for-y-loop, which will be performed in its entirety each time the for-x-loop makes one pass. The number of times that the for-y-loop will be started from its beginning conditions (the number of passes through the body of the for-x-loop) will have to be used to multiply the time count of the statements in the for-y-loop. The frequency count analysis of this fragment is as follows:

```
int b;                // add 1 to space
for (int x=0; x<5; x++) // add 1 to space, 6 to time
    for (int y=0; y<4; y++) { // add 1 to space, 5 times 5 to time
        b = x * y;           // add 4 times 5 to time
        cout << b;           // add 4 times 5 to time
    }
```

This gives:

<u>Time</u>	<u>Space</u>
6	1
+ 25	+ 1
+ 20	+ <u>1</u>
<u>+ 20</u>	3
71	

Frequency Count Example 6

If, in the previous example fragment, the upper limit of the for-x-loop and the for-y-loop are represented by variables, the problem of frequency analysis becomes marginally more complex. Again, if the *worst case* assumption is used, both the for-x and for-y loops will have a big enough upper limit, so both upper limits can be represented by n . The fragment and the analysis of the fragment becomes:

```
int b;                // add 1 to space
for (int x=0; x<n; x++) // add 1 to space, n + 1 to time
    for (int y=0; y<n; y++) { // add 1 to space, n times n + 1 to time
        b = x * y;           // add n times n to time
        cout << b;           // add n times n to time
    }
```

This gives:

<u>Time</u>	<u>Space</u>
$n + 1$	1
$+ n^2 + n$	$+ 1$
$+ n^2$	$+ 1$
$+ n^2$	$+ 1$
<hr style="width: 100%; border: 0.5px solid black;"/>	<hr style="width: 100%; border: 0.5px solid black;"/>
$3n^2 + 2n + 1$	3

Notice that one loop dependent on n gives a result in the form of $(an + b)$, where the largest order of magnitude is 1. Two nested loops dependent on n give the result in the form $(an^2 + bn + c)$, where the largest order of magnitude is 2. In fact, the pattern continues. With a triply nested set of loops that depend on a limit of n , the result is in the form of $(an^3 + bn^2 + cn + d)$, where the largest order of magnitude is 3. This pattern continues as nesting of loops continues.

4. Big O Notation

Except for the provisos that the analysis of the competing data structures and algorithms are approximations of the requirements of the data structures, frequency count analysis of algorithms and data structures is a fairly good if labor intensive way to compare algorithms and data structures. Because it is labor intensive, frequency count analysis has the same drawback of actually writing competing code and then selecting the best. Because it is labor intensive, frequency count analysis is seldom used to evaluate algorithms and data structures. Instead, frequency count analysis has been presented here as the necessary background material to understand Omicron (or Big O) notation in the analysis of data structures and algorithms.

Big O notation is used to represent the worst case growth of an algorithm in time or a data structure in space when they are dependent on n , where n is *big enough*.

The concept of *big enough* takes effect when n is large enough that the differences between the expressions produced by frequency count analysis of competing algorithms becomes, for all practical purposes, completely dependent on the size of n . In other words, if two competing algorithms produced the following results for time:

$$5n^2 + 3n + 16$$

$$3n^2 + 9n + 7$$

A *big enough* n means that the differences produced by adding in the constants 16 and 7 are negligible, the differences produced by multiplying n by 3 and 9 are negligible, and even the effects produced by multiplying n^2 by 5 and 3 are negligible. A *big enough* n means that n^2 will so completely dominate the growth curve of both polynomials that n^2 is all that is needed to compare these two algorithms. In fact, a *big enough* n means the largest magnitude of n is all that is needed to *a priori* rate an algorithm or data structure. This means that the results of both examples are n^2 and are, for all practical purposes, identical.

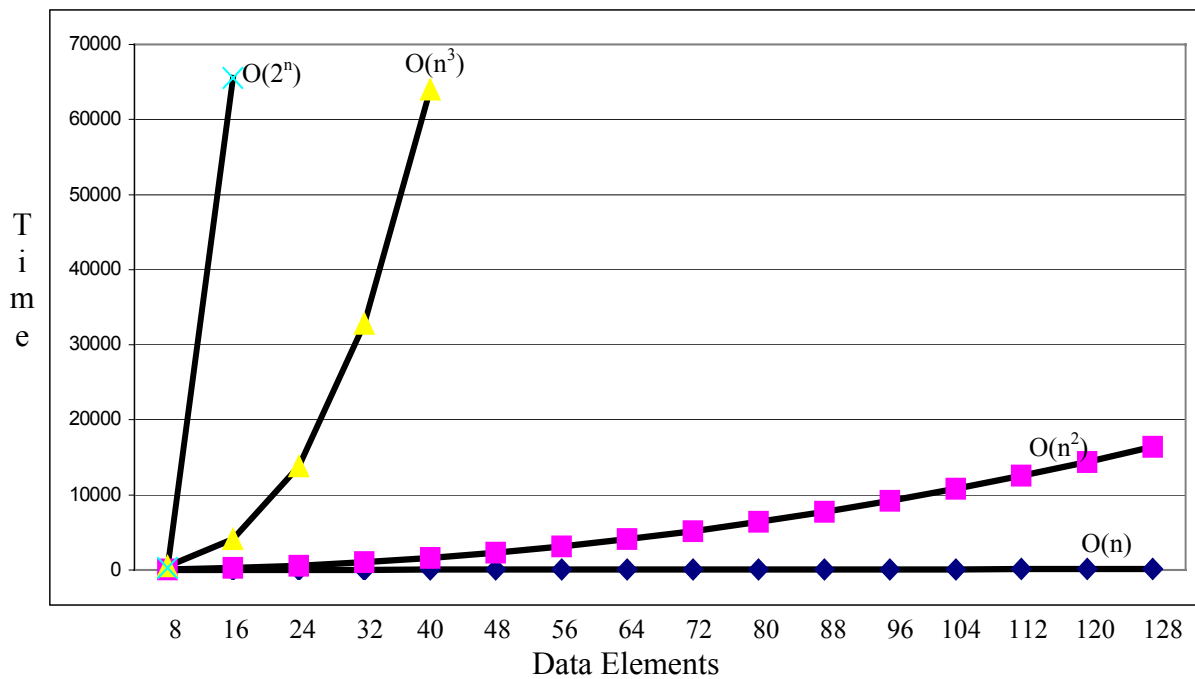
If we accept the premise that the largest magnitude of n is a *good enough* approximation to evaluate an algorithm or data structure, then all that needs to be analyzed for time in an algorithm dependent on n is the most frequently occurring statement. All that needs to be analyzed for space is the most frequently occurring increase in the data structure. By accepting this premise, the comparison of algorithms and data structures becomes less precise, but the rating produced is *good enough* to produce a usable rating for an acceptable amount of work.

The worst case analysis of the growth rate of an algorithm in time or data structure in space that is produced with the above premise is written

$$O(\text{largest magnitude using } n)$$

and is pronounced *Omicron-expression* or *Big O-expression*.

For example, if an algorithm produced a rating of in time of n^3 , it would be written $O(n^3)$. Here is a graph of the curves of common Omicron results:



Thus far, we have used data structures that at worst $O(n)$ and algorithms that are at worst $O(n^2)$ in terms of their growth based on n .

Exercises

Select several of the programs from chapters 20 through 27 that you have written and perform frequency and Big O notation analysis on them. Chose programs with loops, arrays and dynamic arrays.