

Algorithm Performance Prediction ("big O)

- In this subdiscipline of Computer Science, we are interested in *predicting* (not measuring) the *variation* in running time (or sometimes, space requirements) of an algorithm *as the amount of data varies*.
- Here we are not interested in the absolute running time (in seconds, say), but rather in the *variation* of running time.
- Suppose an algorithm takes K seconds to treat 1000 data items. If we double the data to 2000, will the running time be $2K$, or $4K$, or $10K$, or what?
- This variation is sometimes called the *growth rate* of an algorithm.
- "Big O" stands for "order of magnitude". We are interested in predictive estimation, not exact measurement.

Common Algorithm Growth Rates

- $O(1)$, or constant
- $O(\log N)$, or logarithmic (the logarithm is usually taken to the base 2)
- $O(N)$, or linear (directly proportional to N)
- $O(N \log N)$, (usually just called $N \log N$)
- $O(N^2)$, or quadratic (proportional to the square of N)

N	1 (constant)	log N	N log N	N ²
1	1	0	0	1
2	1	1	2	4
4	1	2	8	16
8	1	3	24	64
16	1	4	64	256
32	1	5	160	1024
64	1	6	384	4096
128	1	7	896	16384
256	1	8	2048	65536
512	1	9	4608	262144
1024	1	10	10240	1048576
2048	1	11	22528	4194304
4096	1	12	49152	16777216
8192	1	13	106496	67108864
16384	1	14	229376	268435456
32768	1	15	491520	1073741824

Estimating Algorithm Growth Rates

- Sequence, or writing one statement below another
- Decision, or the well-known if-then or if-then-else
- Loop, including counting loops and while loops
- Subprogram call

Some Java Control Structures

(a) Sequence

```
Temp = A;  
A = B;  
B = Temp;
```

(b) Decision

```
if (x > Max)  
{  
    Max = x;  
}
```

(c) If-then-else

```
if (x = y)  
{  
    Max = x;  
}  
else  
{  
    Max = y;  
}
```

(d) Counting loop

```
int x = 0;
for (int i = p; i <= q; i++)
{
    x = x + i;
}
```

(e) While loop

```
while (x > 0)
{
    y = y + 3;
    x = x / 2;
}
```

Two Simple Counting Loops

(a) Trip count is constant

```
for (int counter = 1; counter < 5; counter++)  
{  
    // something with O(1) performance  
}
```

(b) Trip count depends on N

```
for (int counter = 1; counter < N; counter++)  
{  
    // something with O(1) performance  
}
```

Three Doubly Nested Loops

(a) A double counting loop

```
for (int outerCounter = 1; outerCounter < N; outerCounter++)
{
    for (int innerCounter = 1; innerCounter < N; innerCounter++)
    {
        // something with O(1) performance
    }
}
```

(b) Another double counting loop

```
for (int outerCounter = 1; outerCounter < N; outerCounter++)
{
    for (int innerCounter = 1;
         innerCounter < outerCounter; innerCounter++)
    {
        // something with O(1) performance
    }
}
```

(c) Yet another double counting loop

```
for (int outerCounter = 1; outerCounter < N; outerCounter++)  
{  
    for (int innerCounter = outerCounter;  
         innerCounter < N; innerCounter++)  
    {  
        // something with  $O(1)$  performance  
    }  
}
```

Two Multiplicately Controlled Loops

(a) Control is multiplied by 2

```
int Control = 1;
while (Control <= N)
{
    // something with O(1) performance
    Control = 2 * Control;
}
```

(b) Control is divided by 2

```
int Control = N;
while (Control >= 1)
{
    // something with O(1) performance
    Control = Control / 2;
}
```

Two N log N Loop Structures

```
int Control = 1;
for (int Counter = 1; Counter <= N; Counter++)
{
    while (Control <= N)
    {
        // something with O(1) performance
        Control = 2 * Control;
    }
}
```

```
int Control = N;
while (Control >= 1)
{
    for (int Counter = 1; Counter <= N; Counter++)
    {
        // something with O(1) performance
    }
    Control = Control / 2;
}
```