iteration – time complexity.

Monday, 21 June 2021 12:10 PM

Single loops

```
for(int i=0; i<n; i++)   —(n+1)
{
   some statement;       —(n)
}
```

i: 1, 2, 3, ..., k

Stop ⇒ i≥n ⇒ k≥n ⇒ k=n

∴ loop runs for n times

→ f(n) = 2n+1 ⇒ O(n) time complexity.

```
for(int i=n; i>1; i--)   —(n+1)
{
   some statement;       —(n)
}
```

f(n) = 2n+1 ⇒ O(n) time complexity.

```
for(int i=0; i<n; i=i+2)   —(n/2 + 1)
{
   some statement;         —(n/2)
}
```

f(n) = (n/2+1)+(n/2) = n+1 ⇒ O(n) time complexity.

```
for(int i=0; i<√n; i++)   —(√n + 1)
{
   some statement;        —(√n)
}
```

∴ f(n) = 2√n + 1 ⇒ O(√n) time complexity.

```
for(int i=0; i<n; i=i*2)   —(log₂n + 1)
{
   some statement;         —(log₂n)
}
```

i: 1, 1×2=2, 2×2=2², 2²×2=2³, ..., 2^k

Stop ⇒ i≥n ⇒ 2^k ≥ n ⇒ k ≥ log₂n ⇒ loop runs for log₂n times

∴ f(n) = 2log₂n + 1

∴ O(log₂n) time complexity

```
for (int i=n; i>=1; i=i/2)   —O(log₂n + 1)
{
   some statement;           —O(log₂n)
}
```

i: n, n/2, n/4, ..., n/2^k

Stop ⇒ i<1 ⇒ n/2^k < 1 ⇒ n ≤ 2^k ⇒ k ≥ log₂n

∴ loop runs for log₂n times

→ f(n) = 2log₂n + 1 ⇒ O(log₂n) time complexity.

```
int p=0
for(int i=1; p<=n; i++)   —√n + 1
{
   p=p+i;                 —√n
}
```

i	p
1	0+1=1
2	1+2=3
3	1+2+3=6
4	1+2+3+4=10
⋮	⋮
k	1+2+3+...+k = k(k+1)/2

Stop ⇒ p>n

⇒ k(k+1)/2 > n ⇒ k > √n

⇒ k² > n ⇒ loop runs to √n times

∴ f(n) = 2√n + 1 ⇒ O(√n) time complexity

```
i=0;                 —1
while(i<n)           —n+1
{
   some statement;   —n
   i++;              —n
}
```

f(n) = 3n+2 ⇒ O(n) time complexity.

```
a=1;                 —1
while(a<b)           —log₂(b)+1
{
   some statement;   —log₂(b)
   a=a*2;            —log₂(b)
}
```

a: 1, 1×2=2, 2×2=2², ..., 2^k

Stop ⇒ a≥b ⇒ 2^k = b ⇒ k = log₂b

∴ f(b) = 3log₂(b)+2

∴ f(n) = 3log₂(n)+2

∴ O(log₂(n)) time complexity.

```
i=n
while(i>1)
{
   some statement;
   i=i/2;
}
```

O(log₂n) time complexity.

→ if while loop is given best is to convert it into equivalent for loop and then find time complexity of 'for loop'.

Two unnested loops

```
for(int i=0; i<n; i++)      } O(n)
{
   some statement;
}
for(int i=0; i<m; i++)      } O(m)
{
   some statement;
}
```

∴ O(n+m) time complexity ≈ O(2n) ≈ O(n) time complexity

```
p=0
for(i=1; i<n; i=i*2)        } O(log₂n)
{
   p++;
}
for(j=1; j<p; j=j*2)        } O(log p), but p = log₂n ⇒ O(log log₂n)
{
   some statement;
}
```

i	p
1	0
2	1
2²	2
⋮	⋮
2^k	k

Stop ⇒ i>=n ⇒ 2^k = n ⇒ k = log₂n = p

∴ O(log₂n + log log₂n) ≈ O(log log₂n)

Two nested loops

```
for(i=0; i<n; i++)          —n+1
{
   for(j=0; j<m; j++)       —m+1
   {
      some statement;       —m
   }
}
```

→ n

```
for(i=0; i<n; i++)          —n+1
{
   for(j=0; j<m; j++)       —n×(m+1)
   {
      some statement;       —n×m
   }
}
```

∴ f(n) = (n+1) + n(m+1) + nm ⇒ O(n·m) time complexity.

```
for(i=0; i<n; i++)          —n+1
{
   for(j=1; j<m; j=j*2)     —log₂m + 1
   {
      some statement;       —log₂m
   }
}
```

→ n

```
for(i=0; i<n; i++)          —n+1
{
   for(i=1; j<m; i=i*2)     —(log₂m+1)(n)
   {
      some statement;       —(log₂m)(n)
   }
}
```

∴ f(n) = (n+1) + 2n log₂m + 1 ⇒ O(n log₂m) time complexity.

```
for(i=0; i<n; i++)          —(n+1)
{
   for(j=0; j<i; j++)       —n times
   {
      some statements;
   }
}
```

i	j	no. of times inner loop executed
0	—	0
1	0	1
2	0, 1	2
3	0, 1, 2	3
⋮	⋮	⋮
n	0, 1, ..., n-1	n

∴ total no. of times inner loop executed = 0+1+2+...+n = n(n+1)/2

```
for(i=0; i<n; i++)          —(n+1)
{
   for(j=0; j<i; j++)       —n(n+1)/2 + 1
   {
      some statements;      —n×(n+1)/2
   }
}
```

cause we calculated inner loop got executed this much

∴ f(n) = (n+1) + n(n+1) + n(n+1)/2 ≈ O(n²) time complexity.

Time complexities in order

O(1) < O(log₂n) < O(√n) < O(n) < O(n log₂n) < O(n²) < O(n² log n) < O(n³) < O(n⁴) < O(2ⁿ) < O(2ⁿ × n) < O(2ⁿ × n²) < O(n!) = O(nⁿ) < ... < 3ⁿ < 4ⁿ < ... < nⁿ

To decide which algorithm is better than other, we use two parameter, namely, time complexity and space complexity.

Space complexity deals which how much space does a program take in your memory. But now since space is not a concern anymore, we do not pay attention to it.

Time complexity deals with how much time does a program takes to run. The lesser time your program takes to run, the better it is. It has been represented in three form i.e. Ω(), and Θ(). O() is said to be perfect while Ω() is lower bound and Θ() is upper bound.

n	log(n)	n	n log n	n²	n³	2ⁿ
4	2	4	8	16	64	16
16	4	16	64	256	4096	65536
128	7	128	896	16384	2097152	3.40×10³⁸

Hence we can say following given below series should always be yours preference → O(n² log n)

O(1) > O(log n) > O(n) > O(n log n) > O(n²) > O(n³) > ...

↳ polynomial function

> O(√n) > ... > O(2ⁿ) > O(3ⁿ) > ... > O(mⁿ)

exponential functions