Time complexity of loops

Alexandra Stefan

Time complexity of function definition vs function call

```
// Assumes array nums has at least T elements.
// Calculate the TC for the count
int count(int nums[], int T, int V)
{
  int count = 0;
  for(int k=0; k<T; k++) {
    if ( nums[k] == V )
      count++;
  }
  return count;
}
TC1iter(k) = O(1)+O(1) + O(1) +O(1) = O(1)
```

```
// Write the TC for each function call below
/* Assume all variables exist and have good
values */

 count(nums, N, val);    // O(________)

 count(nums, X, N);      // O(________)

 count(nums, N+M, val); // O(________)

 count(nums, N*N, val); // O(________)

 count(nums, 1000, val); // O(________)

 count(arr, X, N);        // O(________)
```

Time complexity of function definition vs function call

```
// Assumes array nums has at least T elements.
// Calculate the TC for the count
int count(int nums[], int T, int V)
{
  int count = 0;
  for(int k=0; k<T; k++) {
    if ( nums[k] == V )
      count++;
  }
  return count;
}
TC1iter(k) = O(1)+O(1) + O(1) +O(1) = O(1)
```

K	TC1iter(k) = O(1)
0	O(1)
1	O(1)
2	O(1)
3	O(1)
T-1	O(1)

Time Complexity for Function **Definitions** and Function **Calls**

TC for a function DEFINITION can ONLY depend on the data passed as argument E.g. for

```
int count(int nums[], int T, int V){
    …
}
```

In the TC expression, O, we can only have:

- variable names that are parameters
  - E.g. O(T+V) is ok because both `T` and `V` are parameters.
  - E.g. O(N) is wrong because there is no N in the list of parameters
- variables that represent a number (typically integer)
  - E.g. size of the array, size of a data record, max value in an array
  - It is wrong to say O(nums^2) because nums is an entire array. What is nums^2?
    - Remember that TC in the end should give a quantity.
- DEFINED "new" variables
  - I can use names that are not among the function parameters, but I have clearly define them with respect to the input data. E.g. I can say "*O(N) where N is the number of elements in nums*". Here N is not a parameter of `count`, but I have defined what N represents with respect to `nums`, and N is an integer.

Solve the examples with strings from the next page

The TC for a function CALL is based on the time complexity of the function definition and the arguments passed when called.

E.g. given the TC for a function definition in terms of its signature:

`int search(int nums[], int S, int V)` // has TC **O(S^2)**

when called, we replace the S with whatever data is passed in the function call (pay special attention to constants-see example below) . E.g.

We can find the TC of function calls:

`r=search(nums, M, 7);` // O(**M^2**)

`r=search(nums, d, S);` // O(**d^2**), not S^2

`r=search(nums, T+X, 7);` // O(**(T+X)^2**)

`r=search(nums, 39,7);` // O(1) 1, not 39

To understand why we simply substitute the corresponding argument variables in the TC, here is a discussion at the Assume that the Θ(S^2) time complexity of the above function comes from the detailed instruction count: 7S^2 + 12S + 9 . When the function is called with M for S (e.g. in `search(arr1, M, 7)`) the detailed count becomes 7M^2 + 12M + 9 which is Θ(M^2)

What is the time complexity (TC) of the function definitions below?

```
/* Assumes text is a good string (NULL terminated).
   text could have been allocated either dynamic
(with malloc/calloc) or static (with [])   */


int count_char(char * text){
    int count = 0;
    for(int k=0; k<strlen(text); k++) {
        if ( text[k] == 'A' ){
            count++;
        }
    }
    return count;
}
```

```
/* Assumes animal is a good string (NULL terminated).
   animal could have been allocated either dynamic
(with malloc/calloc) or static (with [])   */


void print_ct_sheep(int X, char * animal){
    printf("X=%d, N=%d\n", X, N);
    for (int k=0; k<X; k++){
        printf("%3d %s\n", k, animal);
    }
    printf("\n");
}
```

What is the time complexity (TC) of the function definitions below?

```
/* Assumes text is a good string (NULL terminated).
   text could have been allocated either dynamic
(with malloc/calloc) or static (with [])   */


int count_char(char * text){
    int count = 0;
    for(int k=0; k<strlen(text); k++) {
        if ( text[k] == 'A' ){
            count++;
        }
    }
    return count;
}
```

```
/* Assumes animal is a good string (NULL terminated).
   animal could have been allocated either dynamic
(with malloc/calloc) or static (with [])   */


void print_ct_sheep(int X, char * animal){
    printf("X=%d, N=%d\n", X, N);
    for (int k=0; k<X; k++){
        printf("%3d %s\n", k, animal);
    }
    printf("\n");
}
```

What is the time complexity (TC) of the function definitions below?

```
/* Assumes text is a good string (NULL terminated).
   text could have been allocated either dynamic
(with malloc/calloc) or static (with [])   */


int count_char(char * text){
    int count = 0;
    for(int k=0; k<strlen(text); k++) {
        if ( text[k] == 'A' ){
            count++;
        }
    }
    return count;
}
```

```
/* Assumes animal is a good string (NULL terminated).
   animal could have been allocated either dynamic
(with malloc/calloc) or static (with [])   */


void print_ct_sheep(int X, char * animal){
    printf("X=%d, N=%d\n", X, N);
    for (int k=0; k<X; k++){
        printf("%3d %s\n", k, animal);
    }
    printf("\n");
}
```

Practice

Solve: $1 + 5 + 5^2 + 5^3 + 5^4 + \cdots + 5^p$, *where* $5^p = N$

Review log and exponent and closed form (solutions) for common summations

$a^p = N \Rightarrow p = \log_a N$ $(Proof: apply\ \log_a\ on\ both\ sides: \log_a(a^p) = \log_a N \Rightarrow p = \log_a N)$

Other useful equalities with log: $a^{\log_a N} = N$, $\log_a(a^p) = p$

$$\boldsymbol{a^{\log_b N} = N^{\log_b a}}$$

$$\log_a N = \frac{\log_b N}{\log_b a}$$ (Change of log basis)

*The **closed form** is the solution for the summation. It is an expression that is equivalent to the summation, but does NOT have any $\sum$ in it.*

We need the closed form in order to find O for that summation.

$$\sum_{k=1}^{N} k = 1 + 2 + 3 + 4 + \cdots + (N-1) + N = \boldsymbol{\frac{N(N+1)}{2}} = O(N^2)$$

$$\sum_{k=1}^{N} k^2 = 1 + 2^2 + 3^2 + 4^2 + \cdots + +N^2 = \boldsymbol{\frac{N(N+1)(2N+1)}{6}} = O(N^3)$$

$$\sum_{k=0}^{N} a^k = 1 + a + a^2 + a^3 + a^4 + \cdots + a^N = \boldsymbol{\frac{a^{N+1} - 1}{a - 1}} = O(a^N)\ when\ a > 1;$$

Practice

Solve: $1 + 5 + 5^2 + 5^3 + 5^4 + \cdots + 5^p$, *where* $5^p = N$

$$= \sum_{k=0}^{p} 5^k = 1 + 5 + 5^2 + 5^3 + 5^4 + \cdots + 5^p = \frac{\mathbf{5^{p+1} - 1}}{\mathbf{5 - 1}} = O(5^p) = O(N)$$

Review

Techniques for solving summations (useful for O or dominant term calculations)

Note that some of these will NOT compute the EXACT solution for the summation

Terminology: summation term, summation variable. E.g. in $\sum_{k=1}^{N} kS$ *, (kS)-summation term,* k *–summation variable*

Independent case (term in summation does not have the variable of the summation).

$$\sum_{\boldsymbol{k}=1}^{\boldsymbol{N}} \boldsymbol{S} = S + S + S + \cdots . + S = \boldsymbol{NS} \quad (= S\sum_{k=1}^{N} 1 = SN)$$

Pull constant in front of summation: $\sum_{k=1}^{N}(Sk) = 1S + 2S + 3S \ \ldots + NS = S\sum_{k=1}^{N} k = S\frac{N(N+1)}{2} = O(SN^2)$

Break summation in two summations

$$\sum_{k=1}^{N}(kS + k^2) = \sum_{k=1}^{N} kS + \sum_{k=1}^{N} k^2 = S\sum_{k=1}^{N} k + \sum_{k=1}^{N} k^2 = \mathrm{S}\frac{N(N+1)}{2} + \frac{N(N+1)(2N+1)}{6} = O(SN^2 + N^3)$$

Drop lower order term from summation term. E.g. $10k$ *is lower order compared to* k^2*:*

$$\sum_{k=1}^{N}(10k + k^2) = \sum_{k=1}^{N} k^2 = \frac{N(N+1)(2N+1)}{6} = O(N^3)$$

Use approximation by integrals for increasing or decreasing $f(k)$ $-$ *REMOVED (not required)*

$$\sum_{S}^{N} f(k) = \Theta\big(F(N) - F(S)\big) \ (where\ F\ is\ the\ antiderivative\ of\ f)$$

Terms and notation

- input size
  - number of items
    - e.g. size of array
  - two numbers
    - if 2 arrays are processed, size of each array
    - for a graph: number of edges and number of vertices
  - number of bits needed to represent input value
    - e.g. when finding is a number is prime
- runtime of a program = number of primitive instructions executed
  - a primitive instruction executes in constant time
  - a function call is NOT a primitive instruction
  - notation: T(n)
  - it is often a polynomial function
  - given in terms of input size
- order or growth/growth rate/asymptotic behavior
  - use the leading/dominant term of the polynomial
  - $10\mathbf{n^2}-70n+1000 = O(\mathbf{n^2})$
  - behavior as n goes to infinity
- time complexity (TC) of a program = order of growth of its runtime
  - E.g. $O(n^2)$
- space complexity (SC) of a program = amount of space a program uses, **excluding the space used for input and output**
  - given in in O notation
  - E.g.: O(1), O(n), O(lgn)
- **lg**(n) = $\log_2(n)$ (log base 2 of n)

TC for Loops

- Loops terminology:
  - Loop variable – variable that controls the loop (i/j/k/…).
  - Loop repetitions – "how many times the loop repeats" : the loop control condition evaluates to true and the loop body and header instructions execute
  - One iteration
  - TC_{1iter} - time complexity (as O) for all instructions executed in 1 iteration of the loop, including fct calls.
- TC_{1iter}
  - Show the loop variable as an argument, e.g. TC_{1iter} (j)
  - Typically it is based on the body of the loop (instructions between { }).
  - but check the loop condition and the variable update as well. They may include function calls and the dominant term may come from there. (Most often these are simple instructions, e.g.: i<N, i++ .)
- Loop repetitions – approximate number:
  - e.g. $\log_2 N$ instead of $1 + \lfloor \log_2 N \rfloor$
- Change of variable
  - needed if loop variable does not take consecutive values
- Summation – **must be used if TC_{1iter} depends on loop variable**. (dependent case, e.g. $TC_{1iter}(j)=j^2$)
- Summation method (the general method) handles all loop cases correct.

TC - general

- Notation:
  - $\lg = \log_2 \quad e.g. \ lgN = \log_2 N$
- **O(1) – constant time complexity** (does not depend on the data size).
  - Note: **use 1, NOT any other number**. E.g. O(1), not O(35)
- **for time complexity in general, give the WORST case**. E.g. if a loop can terminate earlier (e.g. because it finds a value and returns), when asked for the time complexity, assume the loop repeats as much as possible (give the worst case TC). Here you may want to (or be asked to) discuss the special cases:
  - Best case (when it repeats the least number of times)
  - Average case (what is the average over multiple executions/runs) and
  - Worst case (the worst behavior for one execution).
- TC calculation for large code:
  - from small to large code pieces;
  - "simplify" at every step
- TC may be discussed in terms of a specific operation. E.g.: "How many data comparisons does insertion sort do?" In that case you compute the TC for ONLY the code that executes and includes the comparisons.
- Data size and Pseudo linear time complexity.
  - One integer value, N, needs only lgN bits to be stored. If TC is Θ(N) => it is $\Theta(2^{\lg N})$ exponential
- O – arithmetic
  - O(N) + O(S) = O(N+S)
  - **$O(N) + O(S+N^2) + O(U)$** = $O(N+S+N^2+U)$ **= $O(S + N^2 + U)$**
  - O(N)* O(k) = O(Nk)
  - In a summation "pull O out": $\sum_{k=1}^{N} O(k) = O\left(\sum_{k=1}^{N} k\right) = O\left(\frac{N(N+1)}{2}\right) = O(N^2)$

Change of variable

```
for(i=1; i<=N; i=i+1){   // i takes consecutive values
...
}

for(i=0; i<=N; i=i+5){    // i does NOT take consecutive values
... //Θ(i) code
}
for(i=1; i<=N; i=i*2){    // i does NOT take consecutive values
... //Θ(i) code
}
```

See also:

```
for(i=N; i>=1; i=i/2){
... //Θ(i) code
}
for(i=N; i<=0; i=i-3){
... //Θ(i) code
}
for(i=1; i<=N; i=i+i){
... //Θ(i) code
}
```

When the variable in a loop does not take consecutive values, it becomes harder to calculate the following (and be confident in your answer):

1. How many iterations the loop does
2. The time complexity for the entire loop

We will **use a change of variable to write the original loop variable as a function of another variable that DOES take consecutive values**. E.g. i = 7x where x takes consecutive values 0,1,2,3…, N/7 .
Another way to look at this. Most common loops that appear in code, generate values from either an arithmetic series or a geometric one. With the change of variable we identify the series.

Change of variable – Math/programming - Arithmetic Series

```
for(i=0; i<=N; i=i+5){   // easy case
...
}
```

+5 +5 +5 +5

Values of i: 0, 5, 10, 15, 20,....., i , ... $i_{last} \le N$

5*0, 5*1, 5*2, 5*3, 5*4, ..., 5x ,... , 5p

Values of x: 0, 1, 2, 3, 4,..., x, ..., p

$\Rightarrow$ i = 5x

$\Rightarrow$ 5p = i_{last} = N (we used use = instead of ≤ to make math easy)

$\Rightarrow$ p = N/5

Each table on the right has ONE ROW for EACH ITERATION of the corresponding loop on the left.
It shows:
- i as a function of a variable x where x takes consecutive values
- what values x takes

x	i=f(x)=5x	i
0	5*0	0
1	5*1	5
2	5*2	10
3	5*3	15
...	...	...
x	5*x	i
...	...	...
p	5*p	i_{last} ≤N

```
for(i=s; i<=(3*N+7); i=i+d){   // general case
...
}
```

+d +d +d +d

Values of i: s, s+d, s+2d, s+3d, s+4d,....., s+x*d , ... s+p*d=i_{last} ≤ (3N+7)

Values of x: 0, 1, 2, 3, 4,..., x, ..., p

$\Rightarrow$ i = s+x*d

$\Rightarrow$ s+p*d = i_{last} = (3N+7) (we used use = instead of ≤ to make math easy)

$\Rightarrow$ p = (3N+7-s)/d

x	i = f(x) = s+xd
0	s
1	s+d
2	s+2d
3	s+3d
...	...
x	i
...	...
p	i_{last} =s+dp ≤(3N+7)

Change of variable – Math/programming - Geometric series

```
for(i=1; i<=N; i=i*2){   // easy case
...
}
```

*2 *2 *2 *2

Values of i: 1, 2, 4, 8, 16,....., i , … $i_{last} \le N$

2^0, 2^1, 2^2, 2^3, 2^4,…, 2^x,… , 2^p

Values of x: 0, 1, 2, 3, 4,…, x, …, p

$\Rightarrow i = 2^x$

$\Rightarrow 2^p = i_{last} = N$ (we used use = instead of ≤ to make math easy)

$\Rightarrow p = \log_2 N$ (b.c. $2^p = N$)

Each table on the right has ONE ROW for EACH ITERATION of the corresponding loop on the left.
It shows:
- i as a function of a variable x where x takes consecutive values
- what values x takes

x	$i = 2^x$
0	2^0
1	2^1
2	2^2
3	2^3
…	…
x	2^x
…	…
p	$2^p = i_{last} \le N$

```
for(i=s; i<=(5*N-3); i=i*d){   // general case
...
}
```

*d *d *d *d

Values of i: $s*d^0$, $s*d^1$, $s*d^2$, $s*d^3$, $s*d^4$,…, $s*d^x$,… , $s*d^p = i_{last} \le (5N-3)$

Values of x: 0, 1, 2, 3, 4, …, x, …, p

$\Rightarrow i = s*d^x$

$\Rightarrow s*d^p = i_{last} = (5N-3)$ (we used use = instead of ≤ to make math easy)

$\Rightarrow p = \log_d \frac{5N-3}{s}$

x	$i = s*d^x$
0	$s*d^0$
1	$s*d^1$
2	$s*d^2$
3	$s*d^3$
…	…
x	$s*d^x$
…	…
p	$s*d^p = i_{last} \le$ **(5N-3)**

The Four Cases of Time Complexity Calculations for Loops

(or use summation always and no special cases)

Example 1

```
// Assume int linear_search(int* ar, int T, int v) has TC O(T)

for(i=1; i<=N; i=i+1){
  res = linear_search(nums1,M,7); // O(M)
  printf("index = %d\n", res);
}
```

i=e, $p=i_{last}$ <=N => p=N

$$\sum_{e=1}^{p} O(M) = p * M = N * M = O(NM)$$

Because i takes consecutive values, we can directly write the summation in terms of i (instead of in terms of e):

$$\sum_{i=1}^{N} O(M) = N * M = O(NM)$$

Iteration of loop	e	i	$TC_{1iter}(i) = O(M)$	Sample detailed count:
1st	1	1	M	7M + 12
2nd	2	2	M	7M + 12
3rd	3	3	M	7M + 12
…		…	…	…
ith	e	i	M	7M + 12
…		…	…	…
(N-1)th		N-1	M	7M + 12
Nth	N	N	M	7M + 12

Total: M+M+M+…+M+…M = N*M = NM

Adding all the terms in the TC_{1iter} column gives the time complexity for all the code.

NM is the final result.

Common error: take this result (NM) and multiply it by N again. WRONG!

Check that when adding the detailed count, we get the same time complexity

Example 2

```
// Assume int linear_search(int* ar, int T, int v) has TC O(T)

for(k=1; k<=N; k=k+1){
  res = linear_search(nums1,k,7); // O(k)
  printf("index = %d\n", res);
}
```

k=e, $p=k_{last}$ <=N => p=N

$$\sum_{e=1}^{p} O(e) = \frac{p(p+1)}{2} = \frac{N(N+1)}{2} = O(N^2)$$

Because k takes consecutive values, we can directly write the summation in terms of k (instead of in terms of e):

$$\sum_{k=1}^{N} O(k) = \frac{N(N+1)}{2} = O(N^2)$$

Iteration of loop	e	k	$TC_{1ter}(k) = O(k)$	Sample detailed count:
1st	1	1	1	7*1 + 12
2nd	2	2	2	7*2 + 12
3rd	3	3	3	7*3 + 12
…		…	…	…
kth	e	k	k	7*k + 12
…		…	…	…
(N-1)th		N-1	N-1	7*(N-1) + 12
Nth	N	N	N	7*N + 12

Total: 1+2+3+…+k+…N = N*(N+1)/2 = $\Theta(N^2)$

Adding all the terms in the TC_{1iter} column gives the time complexity for all the code.

Common error: take this result and multiply it by N again. WRONG!

Check that when adding the detailed count, we get the same time complexity

Example 3

```
// Assume int linear_search(int* ar, int T, int v) has TC O(T)
for(t=1; t<=N; t=t*2){
  res = linear_search(nums1,M,7); // O(M)
  printf("index = %d\n", res);
}
```

Because t does NOT take consecutive value, we need change of var:

$t=2^e$, $2^p=t_{last}<=N \Rightarrow 2^p=N \Rightarrow p=\log_2(N)$

$$\sum_{e=0}^{p} \boldsymbol{O(M)} = (p+1) * O(M) = (log_2(M)+1) * O(M)$$
$$= O(Mlog_2(M))$$

Iteration of loop	e	t	$TC_{1iter}(t)$ = O(M)	Sample detailed count:
1st	0	1 (=2^0)	M	7*M + 12
2nd	1	2 (=2^1)	M	7*M + 12
3rd	2	4 (=2^2)	M	7*M + 12
…		…	…	…
	e	t (=2^e)	M	7*M + 12
…		…	…	…
pth	p-1		M	7*M + 12
last (p+1)th	p =$\log_2$N	t_{last} (=2^p) $2^{p=t}_{last}$ ≤N	M	7*M + 12

Total: M+M+M+…+M+…+M = M*$\log_2$N = Θ(M*$\log_2$N)

Adding all the terms in the TC_{1iter} column gives the time complexity for all the code.

Common error: take this result and multiply it by N or $\log_2$N. WRONG!

Check that when adding the detailed count, we get the same time complexity

Example 4

```
// Assume int linear_search(int* ar, int T, int v) has TC O(T)

for(i=1; i<=N; i=i*2){
  res = linear_search(nums1,3*i,7); // O(i)
  printf("index = %d\n", res);
}
```

Because i does NOT take consecutive value, need change of var:

$i=2^x$, $2^p=i_{last}<=N$ => $2^p=N$ => $p=\log_2(N)$

$$\sum_{x=0}^{p} O(2^x) = \frac{2^{p+1}-1}{2-1} = 2*2^p-1 = 2*2^{\log_2 N}-1 = 2N-1$$

Closed form: $2N-1$ => O(**N**) ****This O cannot have i,x p, or T in it.**

(Note that here if summation is not user or change of variable is skipped, you may get the wrong answer: 1+2+3+..+N = $O(N^2)$)

Iteration of loop	x	i	$TC_{1iter}(i) = O(i)$	Sample detailed count:
1st	0	1 (=2^0)	1	7*(3*1) + 12
2nd	1	2 (=2^1)	2	7*(3*2) + 12
3rd	2	4 (=2^2)	4	7*(3*4) + 12
…		…	…	…
	x	i (=2^x)	2^x	7*(3*2^x) + 12
…		…	…	…
	p-1		2^{p-1}	7*(3*2^{p-1}) + 12
last	p= $\log_2 N$	i_{last} (=2^p) $2^p=i_{last} \le N$	2^p	7*(3*2^p) + 12

Total: 1+2+4+8+16+…+2^x+…+2^p =…= 2N-1 = O(N)

Adding all the terms in the TC_{1iter} column gives the time complexity for all the code.

Common error: take this result and multiply it by N or $\log_2 N$ again. WRONG!

Check that when adding the detailed count, we get the same time complexity

Good example with i^2 in TC_{1iter} (i)

```
// assume that the time complexity of foo(int t, int M) is O(t²M)
for(i=0; i<N; i=i+4)
    foo(i,M);
```

In the call to foo, i was passed for t => use **$O(i^2M)$**, NOT $O(t^2M)$

for-i: $TC_{1iter}(i) = O(i^2M)$

change of var. i: 0,4, 8, 12, … i=4x, $i_{last} = 4p = N \Rightarrow p = N/4$

$$\sum_i O(i^2M) = \sum_{x=0}^{N/4}[(4x)^2M] = 16M\sum_{x=0}^{N/4} x^2 = 16M\frac{\frac{N}{4}(\frac{N}{4}+1)(2\frac{N}{4}+1)}{6} = O(N^3M)$$

$O(N^3M)$

replace i^2 with $(4x)^2$

$$\sum_{k=1}^{N} k^2 = \frac{N(N+1)(2N+1)}{6}$$

Time complexity of nested loops – special cases solution

- Solve the innermost loop and use its time complexity in the calculation of the TC_{1iter} () of the next outer loop and so on. Note that we solve t first, then k, then i.

```
for(i=1; i<=N; i++)
    for(k=1; k<=M; k=k*2)
        for(t=0; t<=i; t=t+3)
            printf("C");
```

Assume i++ is replaced with i=i*2 .
What is the TC for the entire code?

for-t: $TC_{1iter}(t) = O(1)$
t: 0,3,6,9,… t = 3e, 3p=i => p = i/3 repetitions

} i/3*O(1) = O(i)

for-k: $TC_{1iter}(k) = O(i)$
k: 1,2,4,8,… $k = 2^x$, $2^p = M$ => $p = \log_2 M$ repetitions

} $\log_2 M * O(i) = O(i * \log_2 M)$

for-i: $TC_{1iter}(i) = O(i * \log_2 M)$
change of variable NOT needed i: 1 to N

$$\sum_{i=1}^{N} O(i \log_2 M) = \log_2 M \sum_{i=1}^{N} i = \log_2 M \frac{N(N+1)}{2} = O(N^2 \log_2 M)$$

} $O(N^2 \log_2 M)$

Time Complexity of Sequential Loops

```
// Write the O (Theta) time complexity
for (i=left; i<=right; i++)
    printf("%d, ", A[i]);
for (i=0; i<p; i++) {
    for (k=0; k<r; k++)
        printf("B");
}
```

O(M)
where M = right-left+1

O(pr)

O(M+pr)
where M = right-left+1

The actual size of the data being processed by the first loop is right-left+1.

In applications (e.g. binary search or merge sort) this quantity often results in a fraction (e.g. half) of the amount of data in the previous iteration.

If multiple variable appear in the time complexity, there may be more than one dominant term.

Example 1 the time complexity of the above code

Example 2: $27n^4m + 6n^3 + 7nm^2 + 100 = O(\mathbf{n^4m + nm^2})$

Time complexity of nested loops – keep multiplication constant for dominant term

- Solve the innermost loop and use its time complexity in the calculation of the TC_{1iter} () of the next outer loop and so on. Note that we solve t first, then k, then i.

```
for(i=1; i<=N; i++)
    for(k=1; k<=M; k=k*2)
        for(t=0; t<=i; t=t+5)
            printf("C");
```

for-t: $TC_{1iter}(t)$ = **3 instructions** (from: t<=i, printf("C"); t=t+5)

t: 0,5,10,15,… t = 5e, 5p=i => p = i/5 repetitions

} i/5*3= O(**3i/5**)

for-k: $TC_{1iter}(k)$ = O(**3i/5**)

k: 1,2,4,8,… $k = 2^x$, 2^p=M => p = $\log_2 M$ repetitions

} $\log_2 M * O(3i/5) = O((\mathbf{3/5}) * i * \log_2 M)$

for-i: $TC_{1iter}(i) = O(\mathbf{(3/5)} * i * \log_2 M)$

change of variable NOT needed i: 1 to N

$$\sum_{i=1}^{N} O(\left(\frac{\mathbf{3i}}{\mathbf{5}}\right) \log_2 M) = \left(\frac{\mathbf{3}}{\mathbf{5}}\right) \log_2 M \sum_{i=1}^{N} i = \left(\frac{3}{5}\right) \log_2 M \frac{N(N+1)}{2} = O(\left(\frac{3}{5}\right)\left(\frac{1}{2}\right) N^2 \log_2 M)$$

} $O(\left(\frac{3}{10}\right) N^2 \log_2 M)$

The multiplication constant for the dominant term is: $\frac{3}{10}$

Insertion Sort Time Complexity

i	Inner loop time complexity: Best : 1	Worst: i	Average: i/2
1	1	1	1
2	1	2	2/2
…	…		
N-2	1	N-2	(N-2)/2
N-1	1	N-1	(N-1)/2
Total	(N-1)	[N * (N-1)]/2	[N * (N-1)]/4
Order of magnitude	**O(N)**	**O(N^2)**	**O(N^2)**
Data that produces it.	Sorted	Sorted in reverse order	Random data

```
void insertion_sort(int A[],int N){
  int i,k,key;
  for (i=1; i<N; i++)
  key = A[i];
  // insert A[i] in the
  // sorted sequence A[0…i-1]
  k = i-1;
  while (k>=0) and (A[k]>key)
     A[k+1] = A[k];
     k = k-1;
  }
  A[k+1] = key;
}
```

Insertion sort is adaptive

=> O(N^2)

'Total' instructions in worst case:

T(N) = (N-1) + (N-2) + … 2 + 1 =

= [N * (N-1)]/2 -> **O(**N^2)

Note that the N^2 came from the summation, NOT because 'there is an N in the inner loop' (NOT because N * N).

O will be explained in detail later. It says that the algorithm take at most order of N^2.

See the Khan Academy for a discussion on the use of O(N2): https://www.khanacademy.org/computing/computer-science/algorithms/insertion-sort/a/insertion-sort

```c
#include <stdio.h>
int linear_search(int ar[], int S, int val);
int main(){
    char fname[100];
    int mx_size, N, i, M, val;
    FILE *fp;
    printf("Enter the filename: ");
    scanf("%s", fname);
    fp =fopen(fname, "r");
    if (fp == NULL){
        printf("File could not be opened.\n");        return 1;
    }
    fscanf(fp, "%d %d", &mx_size, &N);
    int nums1[mx_size];
   //int* nums1 = malloc(mx_size * sizeof(int));
    // read from file and populate array
    for (i = 0; i < N; i++) {
        fscanf(fp, "%d ", &nums1[i]);
    }
    int found = linear_search(nums1, N, 67);
    printf("\nreturned index:%d\n", found);
    // read the second line of numbers from file and search for them in nums1
    fscanf(fp, "%d", &M);
    for(i=0; i<M; i++){
        fscanf(fp, "%d", &val);
        found = linear_search(nums1, N, val);
        printf("\nreturned index:%d\n", found);
    }
    fclose(fp);
    return (EXIT_SUCCESS);
}
```

```c
// Assumes array ar has S elements.
int linear_search(int ar[], int S, int val){
    printf("\n Searching for %d ...\n",val);
    for(int i=0; i<S; i++) {
        printf("%4d|", ar[i]);
        if ( ar[i] == val )
            return i;
    }
    printf("\n");
    return -1;
}
```

```c
// Assumes array nums has at least T elements.
int count(int nums[], int T, int V){
    int count = 0;
    for(int k=0; k<T; k++) {
        if ( nums[k] == V )
            count++;
    }
    return count;
}
```

Sample data file 1:

```
30 5
3 7 10 19 20
10
67 20 -3 9 7 1 22 13 8 6
```

Sample data file 2:

```
10 7
5 10 13 20 26 30 37
4
67 22 -3 10
```