Time Complexity

CSE 3318

Alexandra Stefan

Overview

- Exact instruction count
- TC (Time complexity) :
  - motivation,
  - O() notation,
  - meaning,
  - calculation for case of a single variable
- TC for loops
  - 1iter
  - TC_{1iter}(loop_variable)
  - table
  - TC of nested loops

Exact instruction count is a sum of terms (often a polynomial)

```
void insertion_sort(int A[],int N){
  int j,k,curr;
  for (j=1; j<N; j++){
    curr = A[j];                    //______
    k = j-1;                        //______
    while ((k>=0) && (A[k]>curr)){
      A[k+1] = A[k];
      k = k-1;
    }
    A[k+1] = curr; //______
}
```

Detailed instruction count for WORST case of insertion sort

Instruction	Count	Explanation
`j=1;`	1	
`j<N;`	N	(N-1) true, 1 false
`curr = A[j];`	N-1	
`k = j-1;`	N-1	
`(k>=0)`	N(N-1)/2	
`(A[k]>curr)`	N(N-1)/2	
`&&`	N(N-1)/2	
`A[k+1]=A[k];`	N(N-1)/2	
`k = k-1;`	N(N-1)/2	
`A[k+1] = curr;`	N-1	
`j++`	N-1	
Total (sum of all instructions)	$1+N+4(N-1)+5*N(N-1)/2 =$ $(5/2)N^2 + (5/2)N - 3$	

In the WORST case each instruction of the while loop (e.g. A[k+1] = A[k]) executes:

$$1+2+3+4+..+(N-2)+(N-1) = ((N-1)*N)/2 = N(N-1)/2$$

Exact instruction count is a sum of terms (often a polynomial)

```
void insertion_sort(int A[],int N){
  int j,k,curr;
  for (j=1; j<N; j++){
    curr = A[j];                    //______
    k = j-1;                        //______
    while ((k>=0) && (A[k]>curr)){
       A[k+1] = A[k];
       k = k-1;
    }
    A[k+1] = curr; //______
}
```

Instruction	Count	Explanation
`j=1;`	1	
`j<N;`	N	(N-1) true, 1 false
`curr = A[j];`	N-1	
`k = j-1;`	N-1	
`(k>=0)`	N(N-1)/2	
`(A[k]>curr)`	N(N-1)/2	
`&&`	N(N-1)/2	
`A[k+1]=A[k];`	N(N-1)/2	
`k = k-1;`	N(N-1)/2	
`A[k+1] = curr;`	N-1	
`j++`	N-1	
Total (sum of all instructions)	1+N+4(N-1)+5*N(N-1)/2 = $(5/2)N^2 + (5/2)N - 3$	

TC (time complexity)

- Algorithm performance for large data size (goes to infinity)
- Looks at dominant term in that expression
  - focuses on N^2
  - instead of $(5/2)N^2 + (5/2)N - 3$
- Notation: O() (and a few other symbols)
- Motivation

Why use $O(N^2)$ instead of $100N+3N^2+1000$

The table below will help understand why TC focuses on the dominant term instead of the exact instruction count.

Assume an exact instruction count for a program gives: **$100N+3N^2+1000$**
Assume we run this program on a ***machine that executes 10^9 instructions per second***.

Compute the time for each term in the summation

(Review: Sample time calculation: 10000 instructions will take: $10000/10^9 = 10^{-5}$ seconds)

Values in table are approximations (not exact calculations).

N	N^2	$3N^2$	100N	1000
10^4 (small)	Instructions: 10^8 Time: 0.1sec	Instructions: $3*10^8$ Time: 0.3sec	Instructions: 10^6 Time: 0.001sec	Instructions: 10^3 Time: 10^{-6} sec
10^9 (large)	Instructions: $(10^9)^2=10^{18}$ Time: **31 yrs**	Instructions: $3*(10^9)^2=3*10^{18}$ Time: ***95 yrs***	Instructions: $100*10^9= 10^{11}$ Time: 100sec = ***1.6 min***	Instructions: 10^3 Time: ***10^{-6} sec***

$10^{18}/10^9 = 10^9$ sec = 10^9 / (60sec*60min*24hrs*365days) = 10^9 / 31536000 = about **31yrs**

You can also plot these functions, add or remove terms and see which terms determine the shape.

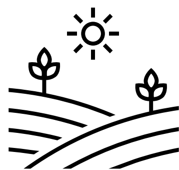
How to find the dominant term O(__?__)
(case with only one variable)

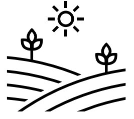
1. Remove multiplication constants
   - A term with no variable does NOT disappear. It becomes **1**
     - E.g. **1000 -> 1** b.c. 1000 = 1000*1 = 1000*N^0 => the constant is 1000, the function is 1 (N^0)
2. View each term as a separate function
3. Keep the function(s) that grow faster than the others
   - more cases later when we look at expressions with 2 or more variables
4. Write it in O(____)

Example: $100N + 3N^2 + 1000$ = O(___)

1. Remove mult constants: $N + N^2 + 1$ (note we still keep 1 for 1000)
2. Look at terms as fcts: N, N^2, 1 (e.g.: $f(N) = N$, $g(N) = N^2$, $h(N) = 1$)
3. Keep the faster growing one: N^2
4. Fill in O: $O(N^2)$

Ordering functions by their growth

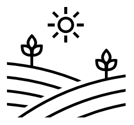




Ordering functions by their growth

- Motivation:
  - calculation of O
  - comparing 2 algorithms

- Notation: $lg(n)$ for $log_2(N)$



Ordering functions by their growth -

- To compare 2 functions:
  - **plot** them (e.g. use this tool) or
  - *** use ratio**: take their ratio, simplify, and compare the remaining functions.
    - Preferred method
    - You can compute the limit as N goes to infinity

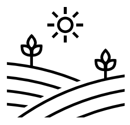
- Compare the functions in each pair using their **ratio**

a) $N^2 \quad ? \quad N^3 \qquad \frac{N^2}{N^3} =$

b) $N\sqrt{N} \quad ? \quad N \qquad \frac{N\sqrt{N}}{N} =$

c) $N\sqrt{N} \quad ? \quad N^2 \qquad \frac{N\sqrt{N}}{N^2} =$

d) $lgN \quad ? \quad log_3 N \qquad \frac{lgN}{\log_3 N} =$



Ordering functions by their growth - Solution

- To compare 2 functions:
  - **plot** them (e.g. use this tool) or
  - *** use ratio**: take their ratio, simplify, and compare the remaining functions.
    - Preferred method
    - You can compute the limit as N goes to infinity

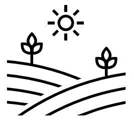
- Compare the functions in each pair using their **ratio**

a) $N^2 < N^3$ $\quad \frac{N^2}{N^3} = \frac{1}{N}$ $\quad$ *use* $\mathbf{1}$ ***grows slower than N*** $\quad$ *or* $\quad \lim\limits_{N\to\infty} \frac{1}{N} = 0 \quad \Rightarrow$ *top grows slower*

b) $N\sqrt{N} > N$ $\quad \frac{N\sqrt{N}}{N} = \frac{\sqrt{N}}{1}$ $\quad$ *use* $\sqrt{\mathbf{N}}$ ***grows faster than 1*** $\quad$ *or* $\quad \lim\limits_{N\to\infty} \frac{\sqrt{N}}{1} = \infty \quad \Rightarrow$ *top grows faster*

c) $N\sqrt{N} < N^2$ $\quad \frac{N\sqrt{N}}{N^2} = \frac{1}{\sqrt{N}}$ $\quad$ *use* $\mathbf{1}$ ***grows slower than*** $\sqrt{\mathbf{N}}$ $\quad$ *or* $\quad \lim\limits_{N\to\infty} \frac{1}{\sqrt{N}} = 0 \quad \Rightarrow$ *top grows slower*

d) $lgN \approx log_3 N$ $\quad \frac{lgN}{\log_3 N} = \frac{lgN}{\frac{lgN}{lg3}} = \frac{lgN}{1} * \frac{lg3}{lgN} = lg3$ $\quad$ *use* ***lg3 is a constant (no N)*** $\quad$ *or* $\quad \lim\limits_{N\to\infty} \frac{lgN}{\log_3 N} = lg3 \quad \Rightarrow$ *same growth*



Ordering functions by their growth

Order in **increasing order of growth** the functions within each group.

a) polynomial: N , N^2, $N^{1/2}$, N^3, $N^{1/3}$, $N^{0.1}$, $N^{0.001}$ ______________________________

b) logarithmic: $\log_3 N$, $\log_7 N$, $\lg N$, $\log_7(N^2)$ ______________________________

* True / False : *All log functions (that differ in only the base) have the same growth.*

c) poly vs log: N, $\lg N$, $N^{1/2}$, $N^{0.001}$ ______________________________

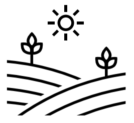
* Select correct answer: *Any log function grows slower / faster than a polynomial function*

d) mix: N , N^2 , $\lg N$, 50, $N\lg N$, N^3 , $N^{1/2}$, $\log_5(N)$ ______________________________

e) exponential: 3^N, 2^N, 5^N, $(1/2)^N$ ______________________________

f) mix with $N!$, N^N : $N!$, N^{100} , N^N , 100^N , N^3 , 3^N ______________________________

High level (use “names”): __________ __________ __________ __________ _____ _____



Ordering functions by their growth

Order in **increasing order of growth** the functions within each group.

a) polynomial: N , N^2, $N^{1/2}$, N^3, $N^{1/3}$, $N^{0.1}$, $N^{0.001}$ ____________________________________

b) logarithmic: $\log_3 N$, $\log_7 N$, $\lg N$ ____________________________________

True / False : *All log functions (that differ in only the base) have the same growth rate.*

True / False : $\log_7(N)$ and $\log_7(N^2)$ *have the same growth rate.*

a) poly vs log: N, $\lg N$, $N^{1/2}$, $N^{0.001}$ ____________________________________

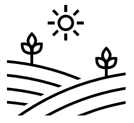
Select correct answer: A*ny log function grows slower / faster than a polynomial function*

b) mix: N , N^2 , $\lg N$, 50, $N\lg N$, N^3 , $N^{1/2}$, $\log_5(N)$ ____________________________________

c) exponential: 3^N, 2^N, 5^N, $(1/2)^N$ ____________________________________

d) mix with $N!$, N^N : $N!$, N^{100} , N^N , 100^N , N^3 , 3^N ____________________________________

High level (use "names"): __________ __________ __________ __________ _____ _____



Ordering functions by their growth - Solution

Order in **increasing order of growth** the functions within each group.

a) polynomial: N , N^2, $N^{1/2}$, N^3, $N^{1/3}$, $N^{0.1}$, $N^{0.001}$ _ $N^{0.001}$, $N^{0.1}$, $N^{1/3}$, $N^{1/2}$, N , N^2, N^3 _____

b) logarithmic: $\log_3 N$, $\log_7 N$, $\lg N$ _ SAME growth _ ($\log_7 N$, $\log_3 N$, $\lg N$) _

True / ~~False~~ : *All log functions (that differ in only the base) have the same growth rate.*

True / ~~False~~ : $\log_7(N)$ and $\log_7(N^2)$ *have the same growth rate.*

a) poly vs log: N, $\lg N$, $N^{1/2}$, $N^{0.001}$ __ $\lg N$, $N^{0.001}$, $N^{1/2}$, N _______________

Select correct answer: A*ny log function grows* ***slower*** */ ~~faster~~ than a polynomial function*

b) mix: N , N^2 , $\lg N$, 50, $N\lg N$, N^3 , $N^{1/2}$, $\log_5(N)$ _ 50, $\log_5(N)$, $\lg N$, $N^{1/2}$, N, $N\lg N$, N^2, N^3 ,__

c) exponential: 3^N, 2^N, 5^N, $(1/2)^N$ __ $(1/2)^N$, 2^N , 3^N , 5^N _______________

d) mix with $N!$, N^N : $N!$, N^{100} , N^N , 100^N , N^3 , 3^N __ N^3 , N^{100} , 3^N , 100^N , $N!$, N^N __________

High level (use "names"): constant , logarithmic , polynomial , exponential , $N!$, N^N

- $3^{\lg N}$? N^2 "$3^{\lg N}$ grows slower than N^2." True / False

- $5^{\lg N}$? N^2 "$5^{\lg N}$ grows slower than N^2." True / False

"False polynomial"

$c^{lg(N)} = N^{lg(c)}$ *for any constant c*

- E.g. $7^{\lg(N)} = N^{\lg(7)}$
- Proof: applying lg on both sides results in two equal terms:

$$lg(c^{lg(N)}) = lg(N^{lg(c)}) \quad =>$$

$$lg(N) * lg(c) = lg(c) * lg(N)$$

- This equality helps identify "false exponentials".

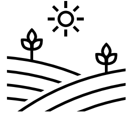
E.g. $3^{lg(N)}$ *may look like an exponential growth but is really polynomial:* $N^{lg(3)}$ *.*

- $3^{\lg N} < N^2$ "$3^{\lg N}$ grows slower than N^2." **True** / ~~False~~

$$3^{\lg N} = N^{\lg 3} = N^{\log_2(3)} < N^2 \quad \text{(b.c. } \log_2 3 < 2\text{)}$$

- $5^{\lg N} > N^2$ "$5^{\lg N}$ grows slower than N^2." ~~True~~ / **~~False~~**

$$5^{\lg N} = N^{\lg 5} = N^{\log_2(5)} > N^2 \quad \text{(b.c. } \log_2 5 > 2\text{)}$$



Ordering functions by their growth

Order the functions below in **increasing order of growth**

Order: N, 500, 4^N, $\lg N$, N^N, $\log_7(N^3)$, $(2/3)^N$, $N^{0.01}$, $N\lg N$, N^3, $N!$, 1 , $5^{\lg N}$, N^2

____ , ____ , ____ , ____ , ____ , ____ , ____ , ____ , ____ , ____ , ____ , ____ , ____ , ____

High level (use “names”) review: _________ _________ _________ _________ ____ ____

Summations

Arithmetic with O()

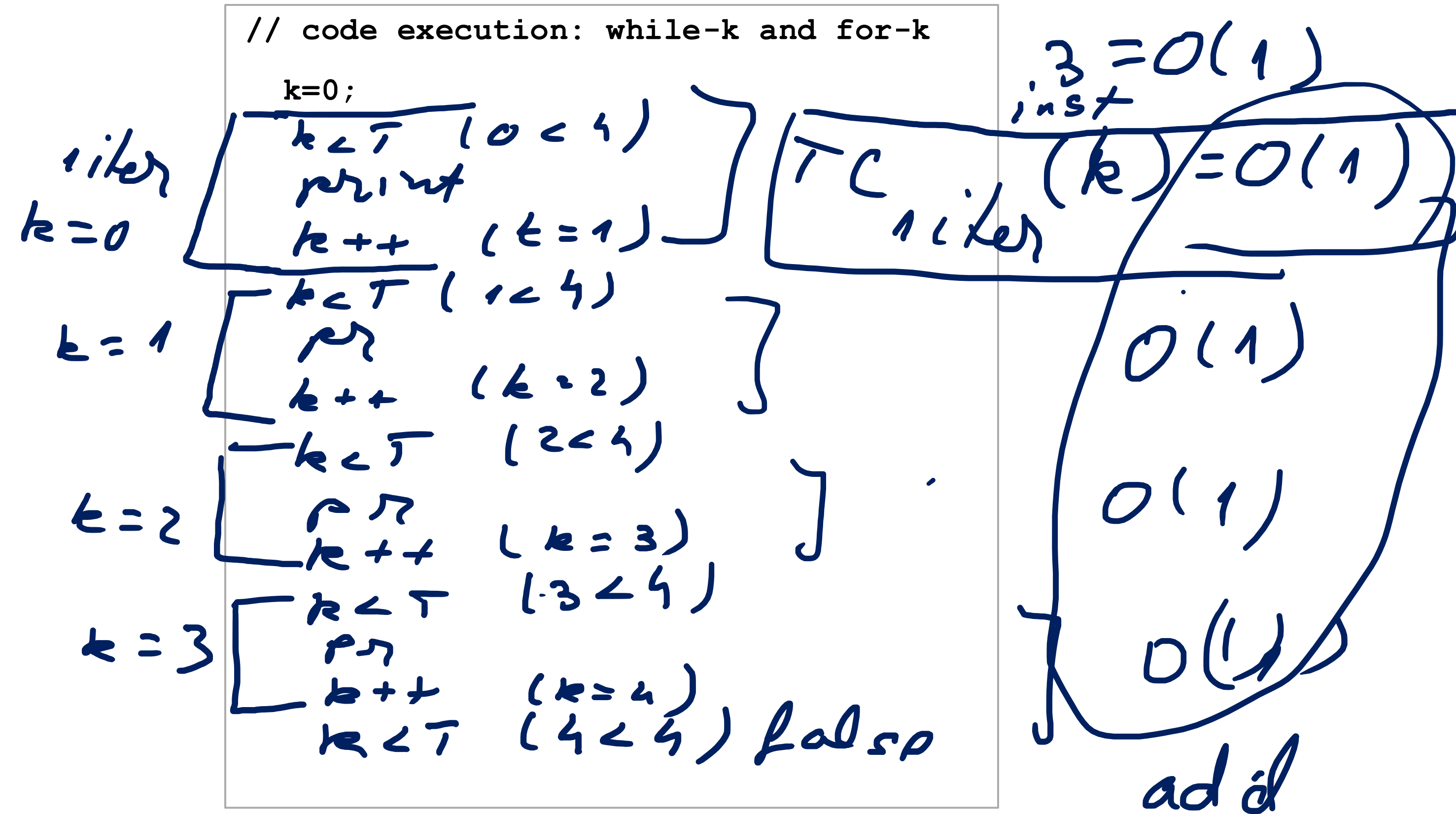
- O(1) + O(1) + … O(1) added T times is **O(T)** . Justification:
  - Added a variable, T, number of times
- O(1) + O(1) + O(1) = **O(1)** . Justification:
  - Added a fixed/constant number of times (3)
- O(T) + O(T) + O(T) + … + O(T) added N times is **O(NT) .** Justification:
  - **N*O(T) = O(NT)** or
  - **O(T + T +… +T) = O(NT)**
  - Keep the variable names provided. Do not use N instead of T or vice versa.
    - WRONG to use N instead of T or vice versa.
    - Wrong answers: $O(N^2)$, $O(T^2)$

Time Complexity for loops

Loop execution and
1iter (code executed in one loop iteration)

```
void ex1() {
    int A[7] = {5, 1, 9, 3, 5, 9, 5};
    int N = 7;
    int T = 4;
    int k;

    k = 0;
    while(k<T){
        printf("%4d, ", A[k]);
        k++;
    }
    printf("\n");
    // for( k = 0; k<T; k++){
    //      printf("%4d, ", A[k]);
    // }
}
```



```
k = 0;
while(k<T){
    printf("%4d, ", A[k]);
    k++;
}
```

Using a table to solve TC of loops

Steps:

0. Fill out column for iter (r)
1. Fill in the loop variable name in the 2 blanks in table header
2. Fill out LV column
3. Write LV as a function of r .
    1. This is not needed if k has the same values as r (or reversed).
    2. If needed, use this function in last row to solve for r_{last}.
4. Compute TC_{1iter}(loop_var)
    1. identify code in 1 iter (iter = iteration)
    2. compute its TC
    3. fill in the blanks in TC_{1iter}(___) = O(___) (top, rightmost cell)
5. Fill out the TC_{1iter} column.
    1. Careful if loop variable is in O( )
    2. May need to use the function from step 3
6. Compute the sum of all values in the rightmost column

iter (r)	Loop_var (LV) ___	LV = ?(r)	TC_{1iter}(___) = O(___)
0			
1			
2			
..			
r			
…			
r_{last} = _			
TC_{loop} = sum of all values in rightmost column (final answer for this loop)			

Last value of k for which the loop condition evaluates true.

Use k_{last} , r_{last} and the formula for k as a function of r to compute r_{last}.

Ex 1 - Worksheet

```
k = 0;
while(k<T){
    printf("%4d, ", A[k]);
    k++;
}
```

Code executed in 1 iteration of while-**k** loop:

```
  (k<T)              ->   O(1)
   printf(one int)   -> O(1)
   k++               -> O(1)
=> TC1iter(k)  = O(1)+O(1)+O(1)  = O(1)
```

iter (r)	Loop var (LV) ___	LV = ?(r)	TC_{1iter}(___) = O(___)
0			
1			
2			
..			
r			
…			
r_{last} = ____			

Tc_{loop} = sum of all values in rightmost column

=

Final answer

Last value of loop variable for which the loop condition evaluates true.

Use k_{last} , r_{last} and the formula for k as a function of r to compute r_{last}.

Ex 1 - Solution

Loop variable is **k**

```
k = 0;
while(k<T){
    printf("%4d, ", A[k]);
    k++;
}
```

Code executed in 1 iteration of while-**k** loop:

```
  (k<T)              ->   O(1)
  printf(one int)    -> O(1)
  k++                -> O(1)
=> TC1iter(k)  = O(1)+O(1)+O(1)  = O(1)
```

iter (r)	Loop var (LV) **k**	LV = ?(r)	TC_{1iter}(**k**) = O(1)
0	0		1
1	1		1
2	2		1
..	…		…
r	k	k = r	1
…	…		…
r_{last} = (T-1)	T-1	(T-1)=k_{last}= r_{last}	1

Tc_{loop} = sum of all values in rightmost column
= 1+1+1+… +1 (T times)
= T
= **O(T)**

Final answer

Last value of k for which the loop condition, (k<T) evaluates true.

Use k_{last} , r_{last} and the formula for k as a function of r to compute r_{last}.

Ex. 2

```
for(j = 0; j<N; j++){
        for(v = 0; v<T; v++) {
                printf(A[v])
        }
}
```

Analyzed for(v) -> O(T) (same as prev page)

Analyze for j

1iter(j)

Ex. 2 - Solution

```
for(j = 0; j<N; j++){
        for(v = 0; v<T; v++) {
                printf(A[v])
        }
}
```

Analyzed for(v) -> O(T) (same as prev page)

Analyze for j

1iter(j)

Ex. 3

```
for(j = 0; j<N; j++){
        for(v = 0; v<j; v++) {
                printf(A[v])
        }
}
```

Analyzed for-v -> O(_) (prev page)

Analyze for j

1iter(j)

Ex. 3 - Solution

```
for(j = 0; j<N; j++){
        for(v = 0; v<j; v++) {
                printf(A[v])
        }
}
```

Analyzed for-v -> O(_) (prev page)

Analyze for j

1iter(j)

Ex. 4

```
for( t = 1 ; t<N ; t=t*3){
        for(v = C; v>=1; v--) {
                printf(A[v])
        }
}
```

1iter(j)

Ex. 4 - Solution

```
for( t = 1 ; t<N ; t=t*3){
        for(v = C; v>=1; v--) {
                printf(A[v])
        }
}
```

1iter(j)

Ex. 4 (review from last lecture)

```
for( t = 1 ; t < N ; t = t*3){
    for( v = C; v >= 1; v-- ) {
        printf(A[v])
    }
}
```

r	Loop variable	= ?(r)	TC_{1iter}(__) = O(___)
0			
1			
2			
3			
…			
r			
…			
r_{last}=			
TC_{loop}=			

r	Loop variable	= ?(r)	TC_{1iter}(__) = O(___)
0			
1			
2			
3			
…			
r			
…			
r_{last}=			
TC_{loop}=			

What is the TC for each of these code pieces?

Use $TC_{for\text{-}t}$ = O(j) and $TC_{for\text{-}v}$ = O(pval)

```
for( j = 1 ; j <= N ; j *= 2){
    for( t = 0; t < j; t++ ) {
        printf("A");
    }
}
```

r	Loop variable	= ?(r)	TC_{1iter}(__) = O(_____)
0			
1			
2			
3			
…			
r			
…			
r_{last}=			

TC_{loop}=

```
for( pval=1, k=1 ; k<=N ; k++, pval*=2){
    for( v = 0; v < pval; v++ ) {
        printf("A");
    }
    // pval *= 2;
}
```

r	Loop variable	= ?(r)	TC_{1iter}(__) = O(______)
0			
1			
2			
3			
…			
r			
…			
r_{last}=			

TC_{loop}=

Geometric series

general term

last term

$$S_p = \sum_{k=0}^{p}(a^k) = 1 + a + a^2 + a^3 + a^4 + \ldots + \boldsymbol{a^k} + \ldots + a^p =$$

$$= \frac{a^{p+1}-1}{a-1} = \frac{a*a^p-1}{a-1} = \frac{a*lastTerm-1}{a-1}$$

- Solve:
  - $1 + 5 + 25 + 125 + \ldots + 5^k + \ldots + 5^6$ = ______________ (number of terms: __________)
  - $1 + 3 + 9 + 27 + \ldots + 3^k + \ldots + 3^9$ = ______________ (number of terms: __________)
  - $1 + 2 + 4 + 8 + \ldots + 2^k + \ldots + 2^{10}$ = ______________ (number of terms: __________)
  - $1 + 2 + 4 + 8 + \ldots + 2^k + \ldots +$ **1024** = ______________ (number of terms: __________)
  - $1 + 3 + 9 + 27 + \ldots + 3^k + \ldots +$ **729** = ______________ (number of terms: __________)
- Pay attention to the form in which the last term in the summation is given: 2^{10}, or 1024.
  - Do you know the exponent, p, for the last term, or do you know the value, v, of the term ?
  - If $a^p = v$ then $p = \log_a(v)$.

Geometric series

$$S_p = \sum_{k=0}^{p}(a^k) = 1 + a + a^2 + a^3 + a^4 + \ldots + a^k + \ldots + a^p =$$

$$= \frac{\boldsymbol{a^{p+1} - 1}}{\boldsymbol{a - 1}} = \frac{a * a^p - 1}{a - 1} = \frac{\boldsymbol{a * lastTerm - 1}}{\boldsymbol{a - 1}}$$

- Two programs, A, and B, process an array of size N. Fill in the answers in O() for each.
  - TC of program A is: $1 + 2^1 + 2^2 + 2^3 + 2^4 + 2^5 + \ldots + 2^k + \ldots + \mathbf{2^N}$ = O(_________)
  - TC of program B is: $1 + 2^1 + 2^2 + 2^3 + 2^4 + 2^5 + \ldots + 2^k + \ldots + \mathbf{N}$ = O(_________)

Geometric series

$$S_p = \sum_{k=0}^{p}(a^k) = 1 + a + a^2 + a^3 + a^4 + \ldots + a^k + \ldots + a^p =$$

$$= \frac{a^{p+1}-1}{a-1} = \frac{a*a^p-1}{a-1} = \frac{a*lastTerm - 1}{a-1}$$

- Two programs, A, and B, process an array of size N. Fill in the answers in O() for each.
  - TC of program A is: $1 + 2^1 + 2^2 + 2^3 + 2^4 + 2^5 + \ldots + 2^k + \ldots + \mathbf{2^N} = (2^{N+1}-1) / (2-1) = 2^{N+1}-1 = O(2^N)$
  - TC of program B is: $1 + 2^1 + 2^2 + 2^3 + 2^4 + 2^5 + \ldots + 2^k + \ldots + \mathbf{N} = 2N-1 = O(N)$
- E.g. if N == 64 :
  - A: $1 + 2^1 + 2^2 + 2^3 + 2^4 + 2^5 + 2^6 + \ldots + 2^k + \ldots + 2^{63} + 2^{64} = (2^{65}-1) / (2-1) = 2^{65}-1$
  - B: $1 + 2 + 4 + 8 + 16 + 32 + 64 = 1 + 2^1 + 2^2 + 2^3 + 2^4 + 2^5 + 2^6 = 2*64 – 1 = 128 – 1 = 127$
    - Note that $2^6 = 64$

TC of functions

- TC of function definition
- TC of function call

Time complexity of function definition

```
// 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)
```

r	Loop variable	= ?(r)	TC_{1iter}(__) = O(____)
0			
1			
2			
3			
…			
r			
…			
r_{last}=			

TC_{loop}=

Time complexity of function definition vs function call

```
// Assume this function has TC  O(T)
int count(int nums[], int T, int V)   // O(T)
```

```
// 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

```
// Assume this function has TC  O(T)
int count(int nums[], int T, int V)   // O(T)
```

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

 count(nums, N, val);   // O(__ N ___)

 count(nums, X, N);     // O(__ X ___)

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

 count(nums, N*N, val); // O(__ N^2 __)

 count(nums, 1000, val); // O(__ 1 __)

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

Time Complexity for Function **Definitions**

TC for a function DEFINITION can ONLY depend on the data passed as argument. We assume no global or external variables.

E.g. for

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

In the O() for the TC expression 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.
- "new" variables
  - names that are not among the function parameters, but we have clearly defined with respect to the input data.
  - E.g. "*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.

Time Complexity for Function **Calls**

The TC for a function CALL is based on

- the time complexity of the function definition and
- the arguments passed in that call

Sample problem:

The line below indicates that the function `search` has TC $O(S^2)$.

```
int search(int nums[], int S, int V);  // O(S²)
```

Fill in the TC of each line of code below:

```
r=search(nums, M, 7);    // O(_____)

r=search(nums, d, S);    // O(_____)

r=search(nums, T+X, 7);  // O(_____)

r=search(nums, 39,7);    // O(______)
```

Practice problems - Interesting cases

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

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


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

```
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 below?

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

Math review

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

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

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

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

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

*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.

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

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

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

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).

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

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

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

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

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

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

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