# Time Complexity

### 001 class notes

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<sub>1iter</sub>(loop\_variable)
  - loop\_variable as a function (expression) of iteration number:
    - k = iter, j=2\*iter,  $k = 3^{iter}$ , v = N-iter
  - table
  - TC of nested loops:

void insertion\_sort(int A[],int N) {

```
int j,k,curr;
for (j=1; j<N; j++) {</pre>
  curr = A[j];
                                   11
// insert curr (A[j]) in the
// sorted sequence A[0...j-1]
                                   //
  k = j - 1;
  while ((k \ge 0) \& (A[k] \ge curr))
     A[k+1] = A[k];
     \mathbf{k} = \mathbf{k} - 1;
  }
  A[k+1] = curr; //____
```

Instructions executed in the while loop , in the WORST case:

• There are 5 basic operations in the while loop:

}

k>=0, A[k]>curr, ()&&(), A[k+1]=A[k], k=k-1

Instruction	Count	Explanation
j=1;		
j <n;< td=""><td></td><td>(N-1) true, 1 false</td></n;<>		(N-1) true, 1 false
curr = A[j];		
k = j-1;		
(k>=0)		
(A[k]>curr)		
88		
A[k+1]=A[k];		
k = k-1;		
A[k+1] = curr;		
j++		
Total (sum of all instructions )		

}

Instructions executed in the while loop , in the WORST case:

• There are 5 basic operations in the while loop:

k>=0, A[k]>curr, ()&&(), A[k+1]=A[k], k=k-1

Instruction	Count	Explanation	
j=1;	1		
j <n;< td=""><td>NN</td><td>(N-1) true 1 false</td><td></td></n;<>	NN	(N-1) true 1 false	
curr = A[j];	N-1	N -1	
k = j - 1;	N-1 Best	N-1 N/ab c+	
(k>=0)	N-1	j72j=1+2+3+N.	-1
(A[k]>curr)	N-1	j = N - N	
& &	N-1	<u>ن</u>	
A[k+1]=A[k];	0	<u>à</u>	
$\mathbf{k} = \mathbf{k} - 1;$	0	à	
A[k+1] = curr;	<i>№ -1</i>	N-1	
j++	N-1	N-1	
Total (sum of all instructions )	$\frac{4dd all}{7(N-1) \neq N+4} = 8N - 6$	$4(N-1)+N+1 + 5(N^2-N)$	
	7	= 5[N2 2] - 4	
		_ 4	

void insertion\_sort(int A[],int N) {

```
int j,k,curr;
for (j=1; j<N; j++) {</pre>
                                  //
  curr = A[j];
// insert curr (A[j]) in the
// sorted sequence A[0...j-1]
                                  //
  k = j - 1;
  while ((k \ge 0) \& (A[k] \ge curr))
     A[k+1] = A[k];
     \mathbf{k} = \mathbf{k} - 1;
  }
  A[k+1] = curr; //
```

Instructions executed in the while loop , in the WORST case:

• There are 5 basic operations in the while loop:

}

k>=0, A[k]>curr, ()&&(), A[k+1]=A[k], k=k-1

Instruction	Count	Explanation
j=1;	1	
j <n;< td=""><td>N</td><td>(N-1) true, 1 false</td></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 <sup>2</sup> + (5/2)N - 3	

void insertion\_sort(int A[],int N) {

```
int j,k,curr;
for (j=1; j<N; j++) {</pre>
                                  //
  curr = A[j];
// insert curr (A[j]) in the
// sorted sequence A[0...j-1]
                                  //
  k = j - 1;
  while ((k \ge 0) \& (A[k] \ge curr))
     A[k+1] = A[k];
     \mathbf{k} = \mathbf{k} - 1;
  }
  A[k+1] = curr; //
```

Instructions executed in the while loop , in the WORST case:

• There are 5 basic operations in the while loop:

}

k>=0, A[k]>curr, ()&&(), A[k+1]=A[k], k=k-1

Instruction	Count	Explanation
j=1;	1	
j <n;< td=""><td>Ν</td><td>(N-1) true, 1 false</td></n;<>	Ν	(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	
88	N(N-1)/2	
A[k+1]=A[k];	N(N-1)/2	
$\mathbf{k} = \mathbf{k} - 1;$	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 <sup>2</sup> + (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<sup>2</sup>) instead of 100N+3N<sup>2</sup>+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<sup>2</sup>+1000** Assume we run this program on a *machine that executes* **10<sup>9</sup>** *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).

Ν	N <sup>2</sup>	3N <sup>2</sup>	100N	1000
10 <sup>4</sup>	Instructions: 10 <sup>8</sup> Time: 0.1sec	Instructions: 3*10 <sup>8</sup> Time: 0.3sec		Instructions: $10^{3}$ Time: $10^{-6}$ sec
10 <sup>9</sup>	Instructions: (10 <sup>9</sup> ) <sup>2</sup> =10 <sup>18</sup> Time: <b>31 yrs</b>	Instructions: 3*(10 <sup>9</sup> ) <sup>2</sup> =3*10 <sup>18</sup> Time: <b>95 yrs</b>		Instructions: 10 <sup>3</sup> Time: <b>10<sup>-6</sup> sec</b>

10<sup>18</sup>/10<sup>9</sup> = 10<sup>9</sup> sec = 10<sup>9</sup> / (60sec\*60min\*24hrs\*365days) = 10<sup>9</sup> / 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
  - NOTE that a term with no variable becomes: 1
    - E.g. 1000 -> 1 b.c. 1000 = 1000\*1 = 1000\*n<sup>0</sup> => the constant is 1000, the function is 1 (n<sup>0</sup>)
- 2. Look at 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 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:  $O(N^2)$

Step 3: Keep the faster growing function: Ordering functions of one variable by their growth

- Motivation:
  - for calculation of O()
  - to be able to compare 2 algorithms
- Notation:  $lg(n) = log_2(N)$
- Order these functions:

 $N, N^{2}, IgN, 50, NIgN, N^{3}, N^{1/2}, Iog_{5}(N)$ 

- Plot them to check
- Place the one you are sure about and leave spaces for the others:

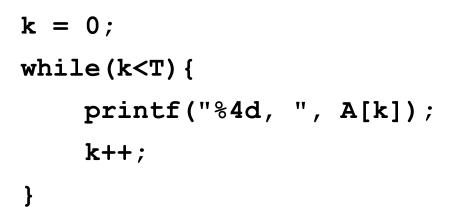
### Arithmetic with O()

- O(1) + O(1) + ... O(1) added T times is  $T^*O(1) = O(T)$
- O(1) + O(1) + O(1) = O(1)
- O(T) + O(T) + O(T) + ... + O(T) added N times => N\*O(T) = O(N\*T)

### Time Complexity for loops

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

<pre>void ex1() {</pre>	// code execution
<pre>int A[7] = {5, 1, 9, 3, 5, 9, 5};</pre>	<b>k=</b> 0;
int $N = 7;$	
int $T = 4;$	
int k;	
$\mathbf{k} = 0;$	
while(k <t){< th=""><th></th></t){<>	
printf("%4d, ", A[k]);	
k++;	
}	
<pre>printf("\n");</pre>	
}	



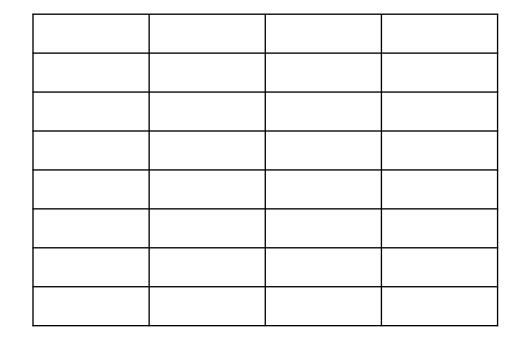
for(j = 0; j<N; j++){ for(v = 0; v<T; v++) { printf(A[v]) } Analyzed for(v) -> O(T) (prev page) Analyze for j

1iter(j)

for(j = 0; j<N; j++){ for(v = 0; v<<mark>j</mark>; v++) { printf(A[v]) } Analyzed for-v -> O(j) (prev page) Analyze for j 1iter(j)

 · · · · · · · · · · · · · · · · · · ·	 

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



Analyzed for-v -> (prev page) Analyze for j

1iter(j)