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_{1iter}(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:

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];                    //______
  // insert curr (A[j]) in the
  // sorted sequence A[0…j-1]
    k = j-1;                        //______
    while ((k>=0) && (A[k]>curr)){
       A[k+1] = A[k];
       k = 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

- Each one of them (e.g. A[k+1] = A[k]) executes:

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

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

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];                        //______
    // insert curr (A[j]) in the
    // sorted sequence A[0…j-1]
    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	1
`j<N;`	N	N (N-1) true, 1 false
`curr = A[j];`	N-1	N-1
`k = j-1;`	N-1	N-1
`(k>=0)`	N-1 (Best)	(Worst) $j: \sum j = 1+2+3+..N-1 = \frac{N^2-N}{2}$
`(A[k]>curr)`	N-1	j
`&&`	N-1	j
`A[k+1]=A[k];`	0	j
`k = k-1;`	0	j
`A[k+1] = curr;`	N-1	N-1
`j++`	N-1	N-1
Total (sum of all instructions)	Add all $7(N-1)+N+1 = 8N-6$	$4(N-1)+N+1+5\frac{N^2-N}{2} = \frac{5N^2}{2}+\frac{5}{2}N-4$

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

- Each one of them (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];                    //______
  // insert curr (A[j]) in the
  // sorted sequence A[0…j-1]
    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)	$1+N+4(N-1)+5*N(N-1)/2 = (5/2)N^2 + (5/2)N - 3$	

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

- Each one of them (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];                    //______
  // insert curr (A[j]) in the
  // sorted sequence A[0…j-1]
    k = j-1;                        //______
    while ((k>=0) && (A[k]>curr)){
       A[k+1] = A[k];
       k = 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

- Each one of them (e.g. A[k+1] = A[k]) executes:

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

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	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	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
   - NOTE that a term with no variable 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. 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 , lgN , 50, $NlgN$, N^3 , $N^{1/2}$, $log_5(N)$
- 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)

```
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");
}
```

```
// code execution
k=0;
```

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


```
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<j; 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)

