Finding Time Complexity for Recurrences:

Identify the number of times a recursive call\_\_\_\_\_\_\_\_\_\_\_\_\_and what the new \_\_\_\_\_\_\_\_\_\_\_\_\_ is

Local time complexity is\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Note:

* c is generally used as a \_\_\_\_\_\_\_\_\_\_\_\_\_
* c \_\_\_\_\_\_ Θ(1)
* n  \_\_\_\_\_\_   cn  \_\_\_\_\_\_  Θ(n)

int foo(int N){

 int a,b,c;

 if(N<=3) return 1500; // Note N<=3

 a = 2\*foo(N-1);

 // a = foo(N-1)+foo(N-1);

 printf("A");

 b = foo(N/2);

 c = foo(N-1);

 return a+b+c;

}

Base case: T( \_\_ ) = \_\_\_\_\_\_\_\_\_\_

Recursive case: T( \_\_ ) = \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

T(N) gives us the Time Complexity for foo(N). We need to solve it (find the closed form)

void bar(int N){

 int i,k,t;

 if(N<=1) return;

 bar(N/5);

 for(i=1;i<=5;i++){

 bar(N/5);

 }

 for(i=1;i<=N;i++){

 for(k=N;k>=1;k--)

 for(t=2;t<2\*N;t=t+2)

 printf("B");

 }

 bar(N/5);

}

Base case: T( \_\_ ) = \_\_\_\_\_\_\_\_\_\_

Recursive case: T( \_\_ ) = \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Solve T(N)

Let N be the number of elements to process in this call. N = right-left+1

int binary\_search**(**int A**[],** int left**,** int right**,** int v**){**

 int m **=** left**+(**right**-**left**)/**2**;**

 **if** **(**left **>** right**)** **return** **-**1**;**

 **if** **(**v **==** A**[**m**])** **return** m**;**

 **if** **(**v **<** A**[**m**])**

 **return** binary\_search**(**A**,** left**,** m**-**1**,** v**);**

 **else**

 **return** binary\_search**(**A**,** m**+**1**,** right**,** v**);**

**}**

Recurrence: base case: T( ) = T ( ) = \_\_\_\_\_\_\_\_

 recursive case: T( ) = \_\_\_\_\_\_\_\_\_\_\_\_\_\_

Draw TC tree. Use it to find TC. TC = \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

 SC = \_\_\_\_\_\_\_\_\_\_\_

Merge\_sort**(**A**,** le**,** r**)** //N = ri-le+1

 **if** **(**le**>=**ri**)** **return**

 **else**

 m **=** floor**(**le**+(**ri**-**le**)/**2**)**

 Merge sort**(**A**,** le**,** m**);**

 Merge\_sort**(**A**,** m**+**1**,** ri**);**

 Merge**(**A**,** le**,** m**,** ri**);**

Recurrence: base case: T( ) = T ( ) = \_\_\_\_\_\_\_\_

 recursive case: T( ) = \_\_\_\_\_\_\_\_\_\_\_\_\_\_

Merge**(**A**,** le**,** m**,** ri**)**

 1 n1**=**m**-**le**+**1**+**1 // +1 for inf

 2 n2**=**ri**-**m**+**1 // +1 for inf

 3 let L**[**n1**],** R**[**n2**]** be arrays

 4 **for** j**=**0 to n1**-**2

 5 L**[**j**]=**A**[**le**+**j**]**

 6 **for** j**=**0 to n2**-**2

 7 R**[**j**]=**A**[**m**+**1**+**j**]**

 8 L**[**n1**]** **=** inf

 9 R**[**n2**]** **=** inf

10 j**=**0**,**

11 i**=**0

12 **for** k**=**le to ri

13 **if** L**[**i**]** ≤ R**[**i**]**

14 A**[**k**]=**L**[**i**]**

15 i**++**

16 **else**

17 A**[**k**]** **=** R**[**j**]**

18 j**++**

// pseudocode

// - indentation => instruction group in {}

// - loops: for k=le to ri means for(k=le; k<=ri; k++)