# **1-Fundamentals (analysis)**

# **Run-time Analysis of Algorithms**

The goal of this assignment is to understand how to perform time complexity analysis of algorithms. For each of the following fragments of code, determine the number of times op() is called as a function of the input size n. Express your answer in terms of the Big-Theta Θ notation.

You might need a quick math refresher on logarithms and summations (e.g., https://www.tug.org/texshowcase/cheat.pdf)

*Remember that when studying order of growth, we drop constant coefficients.*

### Single Loops

for (i=10; i<n+5; i=i+2)

op();

for (i=1; i<n; i= i\*2)

op();

for (i=n; i>1; i=i/2)

op();

for (i=0; i\*i < n; i++)

op();

### *Hints*

Θ(n): (n+5-10)/2

Θ(log n): The number of steps needed to get from 1 to *n* by doubling or from *n* to 1 by halving is log2 n. The base is not important when using the order of growth notation, since logs with different bases are a constant factor away from each other.

Θ(log n): The number of steps needed to get from 1 to *n* by doubling or from *n* to 1 by halving is log2 n. The base is not important when using the order of growth notation, since logs with different bases are a constant factor away from each other.

Θ(sqrt n)

### Nested Independent Loops

for (i=0; i<n; i++)

for (j=0; j < 100; j++)

op();

for (i=0; i<n; i++){

for (j=0; j < n; j++)

op();

for (j=1; j<=n; j=j\*2)

op();

}

### *Hints*

Θ(n): op() is called 100\*n, but we ignore the constant coefficient

Θ(n2): At each iteration of the outer loop, the first inner loop runs n times, then the second inner loop runs log n times. Therefore op() is called times n\* (n + log n), which is in the order of Θ(*n*2)

### Nested Dependent Loops

for (i=1; i<=n; i++)

for (j=1; j <=i; j++)

op();

for (i=1; i<=n; i++)

for (j=1; j <=i; j++)

for (k=1; k<=i; k++)

op();

for (i=1; i<=n; i++)

for (j=1; j <=i; j++)

for (k=1; k<=j; k++)

op();

for (i=1; i<=n; i=i\*2)

for (j=1; j <=i; j++)

op();

for (i=1; i<=n; i++)

for (j=1; j <i; j=j\*2)

op();

### *Hints*

Θ(n2): The inner loop performs 1 iteration when i=1, 2 iterations when i=2 , etc. Therefore, op() is called 1+2+3 …+ n times. This can be represented as a summation: \sum\_{i=1}^{n} I = n\*(n-1)/2

Θ(n3): When i=1, op() is performed i\*i=1, when i=2, op() is performed i\*i=2\*2=4, etc. Therefore, op() is called 12+22+32 …+ n2 times. This can be represented as a summation: \sum\_{i=1}^{n} i2 = 1/3 n3

Θ(n3): The innermost loop performs k=1 iteration when j=1, k=2 iterations when j=2 , etc. Therefore, op() is called \sum\_{j=1}^{i} j = i\*(i-1)/2. Now I can take all the values from 1 to n, so I have a double summation: \sum\_{i=1}^{n} \sum\_{j=1}^{i} j = \sum\_{i=1}^{n} i\*(i-1)/2 (solve to get n3)

Θ(n): The inner loop performs i times each time. Since i doubles at each iteration, this is 1+2+4+8+16 … This is \sum\_{k=0}^{log n} 2k = 2 lg n+1 ~ 2n

Θ(n log n): The inner loop performs log i times for each i. Therefore op() is called log 1+ log 2+ log 3 + … log n = log (1 x 2 x … n) = log (n!) ~ n log n (Stirling approximation)

### Recursion

void f (int n) {

if (n==0) return;

op();

f(n-1);

}

void f (int n) {

if (n==0) return;

op();

f(n/2)

}

void f (int n) {

if (n==0) return;

for (i=0; i<n; i++)

op();

f(n-1);

}

void f (int n) {

if (n==0) return;

op();

f(n/2);

f(n/2);

}

void f (int n) {

if (n==0) return;

op();

f(n-1);

f(n-1);

}

### *Hints*

Θ(n): op() is executed once in each of the recursive calls for n, n-1, …, 1. These are exactly n calls.

Θ(log n): op() is executed once in each of the recursive calls for n, n/2, n/4, …1. These are log n calls.

Θ(n2): op() is executed n times in each of the recursive calls, so that means n + (n-1) + (n-1) + … + 1. This is order of n2.

Θ(n): op() is executed once in each recursive call. How many recursive calls do we have? f(n) 🡪 f(n/2)+f(n/2) 🡪 f(n/4)+f(n/4) + f(n/4)+f(n/4) 🡪 … 20+21+22+…+2log n ~ n

Θ(2n): same as above, op() is called once per recursive call, but how many recursive calls do I have? f(n) 🡪 f(n-1)+f(n-1) 🡪 f(n-2)+f(n-2) + f(n-2)+f(n-2) 🡪 … 20+21+22+…+2n ~ 2n