1

a

T(n) = $\log_2 n^2$ + 1 <= cn when n >= n0

2 $\log_2 n$ + 1 <= cn → if c = 2 and n0 = 2 for all n > n0, T(n) = O(n) is true

b

T(n) = $\sqrt{n(n+1)}$ >= cn when n >= n0

→ if c = 1 and n0 = 1 for all n > n0, T(n) = $\Omega(n)$ is true

c

T(n) = n^{n-1}

c1(n^n) <= n^{n-1} <= c2(n^n)

There is no c1 value that makes T(n) grows faster than c1(n^n), T(n) = $\theta(n^n)$ is false

2

$$\lim_{n\to\infty}\left(\frac{n^2}{n^3}\right) = \lim_{n\to\infty}\left(\frac{2\text{n}}{3n^2}\right) = \lim_{n\to\infty}\left(\frac{2}{6\text{n}}\right) = 0$$ → So n^3 grows faster than n^2

$$\lim_{n\to\infty}\left(\frac{n^2}{n^2 logn}\right) = \lim_{n\to\infty}\left(\frac{1}{\text{logn}}\right) = 0$$ → So $n^2 logn$ grows faster than n^2

$$\lim_{n\to\infty}\left(\frac{n^2}{\sqrt{n}}\right) = \infty$$ → So n^2 grows faster than $\sqrt{n}$

$\lim_{n\to\infty}\left(\frac{n^2}{logn}\right)$ = $\lim_{n\to\infty}\left(\frac{2n}{1/n}\right)$ = $\lim_{n\to\infty}(2n^2)$ = ∞ → So n^2 grows faster than logn

$\lim_{n\to\infty}\left(\frac{n^2}{10^n}\right)$ = 0 → So 10^n grows faster than n^2

$\lim_{n\to\infty}\left(\frac{n^2}{2^n}\right)$ = 0 → So 2^n grows faster than n^2

$\lim_{n\to\infty}\left(\frac{n^2}{8^{\log_2 n}}\right)$ = $\lim_{n\to\infty}\left(\frac{n^2}{n^3}\right)$ = 0 → So $8^{\log_2 n}$ grows faster than n^2

$\lim_{n\to\infty}\left(\frac{n^3}{n^2 logn}\right)$ = $\lim_{n\to\infty}\left(\frac{n}{\text{logn}}\right)$ = ∞ → So n^3 grows faster than $n^2 logn$

$\lim_{n\to\infty}\left(\frac{n^3}{\sqrt{n}}\right)$ = ∞ → So n^3 grows faster than $\sqrt{n}$

$\lim_{n\to\infty}\left(\frac{n^3}{logn}\right)$ = $\lim_{n\to\infty}\left(\frac{3n}{1/n}\right)$ = $\lim_{n\to\infty}(3n^2)$ = ∞ → So n^3 grows faster than $logn$

$\lim_{n\to\infty}\left(\frac{n^3}{10^n}\right)$ = 0 → So 10^n grows faster than n^3

$\lim_{n\to\infty}\left(\frac{n^3}{2^n}\right)$ = 0 → So 2^n grows faster than n^3

$\lim_{n\to\infty}\left(\frac{n^3}{8^{\log_2 n}}\right)$ = $\lim_{n\to\infty}\left(\frac{n^3}{n^3}\right)$ = 1 → So $8^{\log_2 n}$ and n^3 grows at same rate

---

$\lim_{n\to\infty}\left(\frac{n^2 logn}{\sqrt{n}}\right)$ = $\lim_{n\to\infty}\left(n^{3/2} logn\right)$ = ∞ → So $n^2 logn$ grows faster than $\sqrt{n}$

$\lim_{n\to\infty}\left(\frac{n^2 logn}{logn}\right)$ = $\lim_{n\to\infty}(n^2)$ = ∞ → So $n^2 logn$ grows faster than logn

$\lim_{n\to\infty}\left(\frac{n^2 logn}{10^n}\right)$ = 0 → So 10^n grows faster than $n^2 logn$

$\lim_{n\to\infty}\left(\frac{n^2 logn}{2^n}\right)$ = 0 → So 2^n grows faster than $n^2 logn$

$\lim_{n\to\infty}\left(\frac{n^2 logn}{8^{\log_2 n}}\right)$ = $\lim_{n\to\infty}\left(\frac{logn}{\mathrm{n}}\right)$ = 0 → So $8^{\log_2 n}$ grows faster than $n^2 logn$

$\lim_{n\to\infty}\left(\frac{\sqrt{n}}{logn}\right)$ = ∞ → So $\sqrt{n}$ grows faster than logn

$\lim_{n\to\infty}\left(\frac{\sqrt{n}}{10^n}\right)$ = 0 → So 10^n grows faster than $\sqrt{n}$

$\lim_{n\to\infty}\left(\frac{\sqrt{n}}{2^n}\right)$ = 0 → So 2^n grows faster than $\sqrt{n}$

$\lim_{n\to\infty}\left(\frac{\sqrt{n}}{8^{\log_2 n}}\right) = \lim_{n\to\infty}\left(\frac{1}{n^{5/2}}\right)$ = 0 → So $8^{\log_2 n}$ grows faster than $\sqrt{n}$

$\lim_{n\to\infty}\left(\frac{logn}{10^n}\right)$ = 0 → So 10^ngrows faster than logn

$\lim_{n\to\infty}\left(\frac{logn}{2^n}\right)$ = 0 → So 2^n grows faster than logn

$\lim_{n\to\infty}\left(\frac{logn}{8^{\log_2 n}}\right) = \lim_{n\to\infty}\left(\frac{logn}{n^3}\right)$ = 0 → So $8^{\log_2 n}$ grows faster than $8^{\log_2 n}$

$\lim_{n\to\infty}\left(\frac{10^n}{2^n}\right) = \lim_{n\to\infty}(5^n)$ = ∞ → So 10^n grows faster than 2^n

$\lim_{n\to\infty}\left(\frac{10^n}{8^{\log_2 n}}\right) = \lim_{n\to\infty}\left(\frac{10^n}{n^3}\right)$ = ∞ → So 10^n grows faster than $8^{\log_2 n}$

$\lim_{n\to\infty}\left(\frac{2^n}{8^{\log_2 n}}\right)$ = $\lim_{n\to\infty}\left(\frac{2^n}{n^3}\right)$ = ∞ → So 2^n grows faster than $8^{\log_2 n}$

ORDER BY GROWTH RATE (FROM FASTER TO SLOWER:

$10^n > 2^n > 8^{\log_2 n} = n^3 > n^2 logn > n^2 > \sqrt{n} > \log n$

3

a

Inside of the for loop is constant time : θ(1)

for loop :

- i is almost doubled for odd cases (half of the for loop). We can say its logn.
- For the other half its linear time.

T(n) = θ(n)

b

First 2 lines in the function : θ(1)

Every line in for loop : θ(1)

for loop :

- For every case, loop runs size of array times : θ(n)

T(n) = θ(n)

c

Constant time : T(n) = θ(1)

d

First line, inside the loop, and return statement are all constant time : θ(1)

for loop :

- Runs n/5 times, θ(n/5) = θ(n)

T(n) = θ(n)

e

Inside the loops are constant time : θ(1)

Outer loop runs n times : θ(n)

Inner loop : θ(logn)

T(n) = θ(nlogn)

f

p_4 runs in all cases so if condition is : θ(n)

Condition:

- For best case, condition is false. Then else part runs : θ(1) + θ(n) = θ(n)
- For worst case, condition is true. Then if part runs : θ(nlogn)

T(n) = O(nlogn)

g

First line, inside of the for loop, and i=i/2 line : θ(1)

While loop : θ(logn)

For loop : θ(n)

T(n) = θ(nlogn)

h

Inner loop is dependent on the variable that while loop iterates through.

Both loops are dependent to each other, we can say:

for loop iterates n times first, then n/2 times, then n/4 times, through to $n/2^{k-1}$ times. If we add them up:

$$n + n/2 + n/4 + n/2^{k-1}$$

So this means that → T(n) = O(n)

i

if (n = 0) return 1

- Constant time

else return n * p_9(n-1) :

- 1 for multiplication + 1 for subtraction + cost of evaluation of p_9(n-1)
- T(n) → cost of evaluation of p_9(n) = $\underline{2}$ + T(n-1)

T(n) = θ(1) if n = 0

T(n) = θ(1) + T(n-1)

- → T(n) = 2 + T(n-1)
- → T(n) = 2 + 2 + T(n-2)
- → …
- → T(n) = k * 2 + T(n-k)
- → k=n-1:
- → T(n) = (n-1)*2 + T(n-(n-1))
- → T(n) = (n-1)*2 + T(1)
- → T(n) = (n-1)*2 + 1
- → T(n) = 2n - 1

T(n) = θ(n)

j

if (n==1) return; → Constant time

$$T(n) = \text{cost of evaluation of p\_10} = \underbrace{1}_{j=n-1} + \underbrace{\underline{n-1}}_{\text{while loop}} + T(n-1)$$

Inside of while loop is constant time and it iterates n-1 times for worst case.

$T(n) = \theta(1)$ if $n = 1$

$T(n) = \theta(1) + O(n) + T(n-1)$

Worst (while iterates n-1 times):

- → $T(n) = 1 + n-1 + T(n-1)$
- → $T(n) = 1 + n-1 + 1 + n-2 + T(n-2)$
- → …
- → $T(n) = k*1 + kn + \ldots + T(n-k)$ → … is constants, not important
- → $k=n-1$:
- → $T(n) = n-1 + n^2 - n + T(1)$
- → $T(n) = n^2$

$T(n) = \theta(n^2)$

Best (while iterates 0 times):

- → $T(n) = 1 + T(n-1)$
- → $T(n) = 1 + 1 + T(n-2)$
- → …
- → $T(n) = k + T(n-k)$ → … is constants, not important
- → $k=n-1$:
- → $T(n) = n+1$

$T(n) = \theta(n)$

All in all: $T(n) = O(n^2)$

4

a

Statement says that running time of A (T(n)) >= O(n^2).

Big-O notation provides an upper bound for a function but here we say that T(n) is bigger or equal to O(n^2). So we can't determine the upper bound of T(n). Also we can't determine the lower bound of T(n) because O(n^2) means that T(n)'s grow rate is less than or equal to grow rate of n^2. So lower bound may be n or something smaller than n^2.

b

I. c1(2^n) <= 2^{n+1} <= c2(2^n)

For c1 = 1, c2 = 1, their grow rates are all equal (constants are not important)

So clause is true.

II. c1(2^n) <= 2^{2n} <= c2(2^n)

There is no c2 value that makes 2^n grows faster than 2^{2n}.

So clause is false.

III. Result must be O(n^4), because θ notation is tight but O notation is not tight. Therefore the result must be not tight.

So clause is false.

5

a

- → $T(n) = 2T(n/2) + n$
- → $T(n) = 2(2T(n/4) + n/2) + n = 4T(n/4) + 2n$
- → $T(n) = 4 (2T(n/8) + n/4) + 2n = 8T(n/8) + 3n$
- → …
- → $T(n) = 2^k(T(n/2^k)) + kn$
- → $k = \log_2 n$:
- → $T(n) = 2^{\log_2 n}(T(1)) + n\log_2 n$ -----> $T(1) = 1$ is given
- → $T(n) = n + n\log n$

So ------> $T(n) = O(n\log n)$

b

- → $T(n) = 2T(n-1) + 1$
- → $T(n) = 2(2T(n-2) + 1) + 1 = 4T(n-2) + 3 = 4T(n-2) + 2^1 + 1$
- → $T(n) = 4 (2T(n-3) + 1) + 3 = 8T(n-3) + 7 = 8T(n-3) + 2^2 + 2^1 + 1$
- → …
- → $T(n) = 2^k(T(n-k)) + 2^{k-1} + 2^{k-2} + \ldots + 2^1 + 1$
- → $k = n$:
- → $T(n) = 2^n T(0) + 2^{n-1} + \ldots + 2^1 + 1$ -----> $T(0) = 0$
- → $T(n) = 1 + 2^1 + \ldots + 2^{n-1} = 2^n - 1$

So ------> $T(n) = O(2^n)$

array = {1, 2, 3, -5, -4, 10, -2, -9, 13, -11, -20, -90, 101, 30, -19, 0, -63, 16, 19, 21}

Number of elements = 20

sum = 11

Running time = 29921986 ns

---

array = {1, 2, 3, -5, -4, 10, -2, -9, 13, -11, -20, -90, 101, 30, -19, 0, -63, 16, 19, 21

,1, 2, 3, -5, -4, 10, -2, -9, 13, -11, -20, -90, 101, 30, -19, 0, -63, 16, 19, 21}

Number of elements = 40

sum = 11

Running time = 62305193 ns

---

array = {1, 2, 3, -5, -4, 10, -2, -9, 13, -11, -20, -90, 101, 30, -19, 0, -63, 16, 19, 21

,1, 2, 3, -5, -4, 10, -2, -9, 13, -11, -20, -90, 101, 30, -19, 0, -63, 16, 19, 21

,1, 2, 3, -5, -4, 10, -2, -9, 13, -11, -20, -90, 101, 30, -19, 0, -63, 16, 19, 21}

Number of elements = 60

sum = 11

Running time = 111452950 ns

---

array = {1, 2, 3, -5, -4, 10, -2, -9, 13, -11, -20, -90, 101, 30, -19, 0, -63, 16, 19, 21

,1, 2, 3, -5, -4, 10, -2, -9, 13, -11, -20, -90, 101, 30, -19, 0, -63, 16, 19, 21

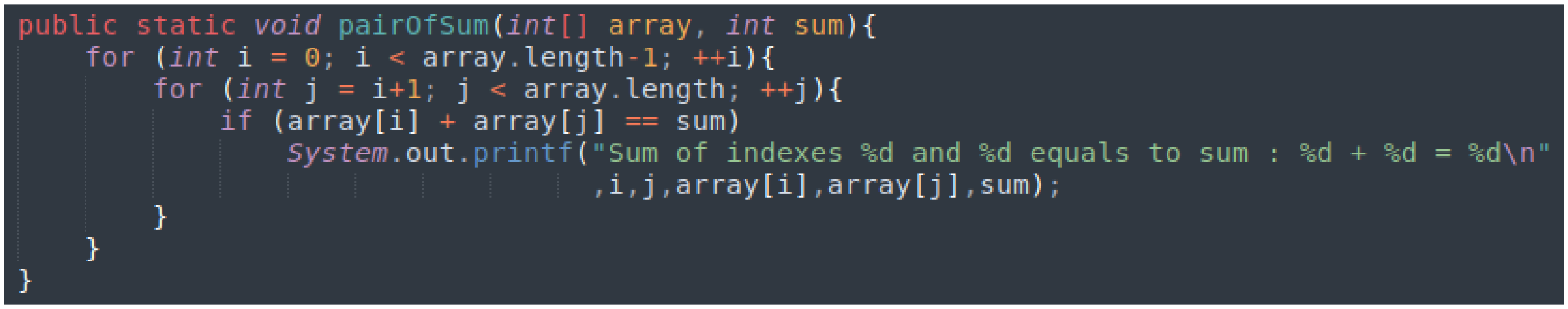
,1, 2, 3, -5, -4, 10, -2, -9, 13, -11, -20, -90, 101, 30, -19, 0, -63, 16, 19, 21

,1, 2, 3, -5, -4, 10, -2, -9, 13, -11, -20, -90, 101, 30, -19, 0, -63, 16, 19, 21}

Number of elements = 80

sum = 11

Running time = 150596043 ns



```java
public static void pairOfSum(int[] array, int sum){
    for (int i = 0; i < array.length-1; ++i){
        for (int j = i+1; j < array.length; ++j){
            if (array[i] + array[j] == sum)
                System.out.printf("Sum of indexes %d and %d equals to sum : %d + %d = %d\n"
                                  ,i,j,array[i],array[j],sum);
        }
    }
}
```

Time complexity of the algorithm:

- → Inside of the loops is constant time
- → Inner for loop is θ(n)
- → Outer for loop is θ(n)

T(n) = $\theta(n^2)$

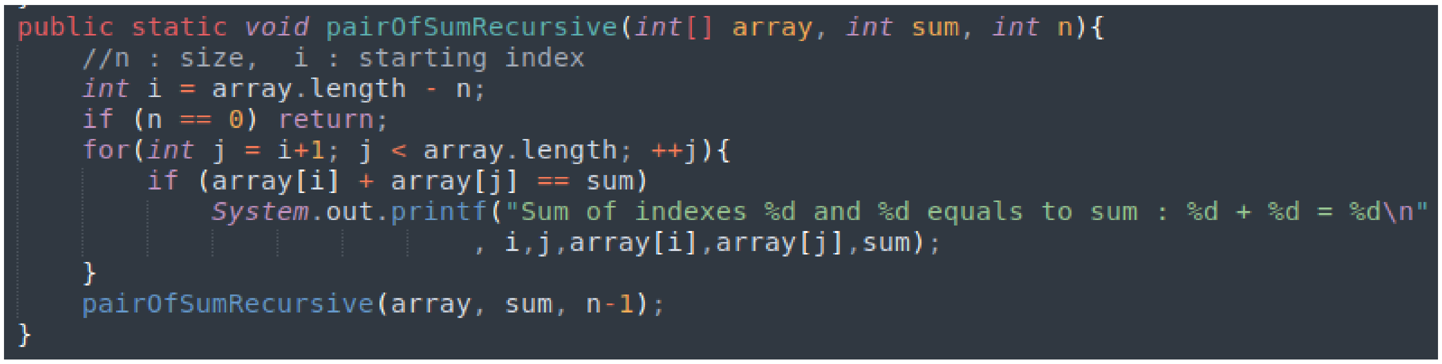
It doesn’t seem like my test results exactly fit with my complexity calculation. Running times don’t increment quadratically while input size increments.

Here, other factors come into play such as computer, compiler, and OS. Although order of the function is still n^2.

7

- → $T(n) = 3 + n + T(n-1) = n + T(n-1)$ → Constants are not important, $T(0) = 1$
- → $T(n) = n + (n-1) + T(n-2)$
- → $T(n) = n + (n-1) + (n-2) + T(n-3)$
- → …
- → $T(n) = n + (n-1) + \ldots + (n-(k-2)) + (n-(k-1)) + T(n-k)$
- → $k = n$:
- → $T(n) = n + (n-1) + \ldots + (2) + (1) + T(0)$
- → $T(n) = n(n+1)/2 + 1 = n^2/2 + n/2 + 1$

So $T(n) = \theta(n^2)$



```java
public static void pairOfSumRecursive(int[] array, int sum, int n){
    //n : size,  i : starting index
    int i = array.length - n;
    if (n == 0) return;
    for(int j = i+1; j < array.length; ++j){
        if (array[i] + array[j] == sum)
            System.out.printf("Sum of indexes %d and %d equals to sum : %d + %d = %d\n"
                                , i,j,array[i],array[j],sum);
    }
    pairOfSumRecursive(array, sum, n-1);
}
```