Assignment 02

(1.1) 1) $f(n) = 3n+8$

$3n+8 \leq c\, g(n)$

$3n+8 \leq cn$

$n(c-3) \geq 8$

$n \geq \frac{8}{c-3}$

Since $c > 3$, let $c = 4$

$n \geq \frac{8}{4-3} \Rightarrow n \geq 8$

$\therefore n_0 = 8$

$\therefore 3n+8 \leq 4n$ for $n \geq 8$

$3n+8 = \underline{O(n)}$ when $c = 4$, $n_0 = 8$

2) $f(n) = n^2 - 4n + 7$

$n^2 - 4n + 7 \geq c\, g(n)$

$n^2 - 4n + 7 \geq cn^2$

$n^2 - 4n \geq cn^2$, since 7 is ignored as $n^2 - 4n + 7 \geq cn^2$ and $n^2 - 4n \geq 0$

$1 - \frac{4}{n} \geq c$, let $n = 5$

$1 - \frac{4}{5} \geq c$

$c \leq 1/5$

$\therefore n^2 - 4n + 7 \geq \frac{1}{5} n^2$ for $n \geq 5$

$\therefore n^2 - 4n + 7 = \underline{\Omega(n^2)}$ when $n_0 = 5$, $c = 1/5$

3) $f(n) = 2n + 5$

Upper Bound

$2n + 5 \leq cn$

$5 \leq n(c-2)$

$n \geq \frac{5}{c-2}$ let $c = 3$

$n \geq 5$

$\therefore 2n + 5 \leq 3n$ for $n_0 \geq 5$

$f(n) = O(n)$ when $c = 3$, $n_0 = 5$

Lower Bound

$2n + 5 \geq cn$

let $c = 2$, $n_0 = 1$

$2n + 5 \geq 2n$

$f(n) = \Omega(n)$ with $c = 2$, $n_0 = 1$

Since $2n \leq 2n + 5 \leq 3n$

$f(n) = \underline{\Theta(n)}$ For $c_1 = 2$, $c_2 = 3$, $n_0 = 5$

4) $f(n) = n\log_2 n + 3n$

$n\log_2 n + 3n \leq cn\log_2 n$

$3 + (1 - c)\log_2 n \leq 0$

$\frac{3}{\log_2 n} \leq c - 1$, let $n = 2$ then $\log_2 2 = 1$

$3 \leq c - 1$

$4 \leq c$

$\therefore n\log_2 n + 3n \leq 4n\log_2 n$ for $n \geq 2$

$f(n) = \underline{O(n\log_2 n)}$ when $c = 4$, $n_0 = 2$

5) $f(n) = 4n^2\log n + 2n\log n + 5n$

Upper bound

$4n^2\log n + 2n\log n + 5n \leq cn^2\log n$

Make all terms = to higher order terms

$4n^2\log n + 2n^2\log n + 5n^2\log n = 11n^2\log n$

$\therefore 4n^2\log n + 2n\log n + 5n < 11n^2\log n$ for all $n \geq 2$

$\therefore$ Upper bound = $O(n^2\log n)$ for $c = 11$ and $n_0 = 2$

Lower bound

Ignor, $4n^2\log n + 2n\log n + 5n \geq cn^2\log n$

$4n^2\log n \geq cn^2\log n$ even if we ignore last ~~to~~ two terms

$\therefore$ let $c = 4$, $n_0 = 1$

for all $n \geq 1$, $f(n) \geq 4n^2\log n$

$\therefore f(n) = \Omega\, n^2\log n$

$\therefore 4n^2\log n \leq 4n^2\log n + 2\log n + 5n \leq 11n^2\log n$

$\therefore c_1 = 4$, $c_2 = 11$, $n_0 = 2$

$f(n) = \underline{\Theta(n^2\log n)}$

(1.2) 1) $f(n) = 10$, $g(n) = \log(10)$

$f(n) = \Theta g(n)$

> Both $f(n) = 10$ and $g(n) = \log(10)$ are constants

> Therefore they are Θ of each other

> $10 = \Theta(1)$ and $\log(10) = \Theta(1)$, belong to the same growth rate class

2) $f(n) = \log n^2$, $g(n) = \log n + 5$

$f(n) = \Theta(g(n))$

> $\log n^2 = 2\log n$.

> For large n, constant 5 is negligable

> $2\log n \leq c(\log n + 5)$, let $c = 2$.

$2\log n \leq 2\log n + 10$

$\therefore f(n) = O(g(n))$

> $2\log n \geq \log n + 5$, where $c = 1$

$\therefore f(n) = \Omega(g(n))$

> Therefore $f(n) = \Theta g(n)$, they belong to the same growth rate class.

3) $f(n) = 2n^4 - 3n^2 + 7$, $g(n) = n^5$

$f(n) = O(g(n))$

> Highest degree term in $f(n)$ is n^4, while in $g(n)$, its n^5.

> n^5 grows faster than n^4 , $\therefore n^4$ cannot be a lower bound for n^5

> Therefor $f(n)$ is ~~strictly~~ strictly bounded by $g(n)$

4) $f(n) = \log n$, $g(n) = \log n + \frac{1}{n}$

$f(n) = \Theta g(n)$

> For large n , $1/n$ approaches 0, therefore $g(n)$ approaches to $\log n$

> Therefore both function behave the same as both $f(n)$, $g(n)$ represent $\log n$

> $\log n \leq \log n + 1/n$ for $n \geq 1$, $\therefore f(n) = O(g(n))$

> $\log n + 1/n \leq \log n + 1 \leq 2\log n$, $\therefore g(n) = O(f(n))$

(13) 1) $n^2 = O(2^n)$? = true

For this to be true, $0 \leq n^2 \leq c \cdot 2^n$ should be true where c is a constant and all $n > N$, by the definition of big Oh, such that

Let $c = 1$, $n_0 = 4$, by proof using cases,

for $n_0 = 4$; $16 \leq 16$

$n_0 = 5$; $25 \leq 32$

$n_0 = 10$; $100 \leq 1024$

$\therefore$ $n^2 \leq 2^n$ for $n \geq 4$

$\therefore$ $n = O(2^n)$ with $c = 1$, $n_0 = 4$

2) $n^3 - 3n^2 - n + 1 = O(n^3)$ = true

By definition of Big Oh, there must exist constants $c > 0$, $n > 0$ where $n \to N$; such that $0 \leq n^3 - 3n^2 - n + 1 \leq cn^3$ for all $n > N$.

Wher N is an integer, $N > 0$.

$n^3 - 3n^2 - n + 1 \leq c(n^3)$

divide both sides of the inequality by n^3.

$1 - \frac{3}{n} - \frac{1}{n^2} + \frac{1}{n^3} \leq c$

As $n \to \infty$, $3/n$, $1/n^2$, $1/n^3$ terms approach 0, $\therefore$ $1 - 0 = 1$

Therefore $c = 1$, and any large n_0 value satisfies the inequality

Let $n_0 = 4$

$\therefore$ $n^3 - 3n^2 - n + 1 \leq n^3$, for all $n \geq 4$.

$64 - 48 - 4 + 1 \leq 64$

$13 \leq 64$

$\therefore$ for the instance $c = 1$, $n_0 = 4$, $n^3 - 3n^2 - n + 1 = O(n^3)$

3) $\Theta(n^2) = \Theta(n^2+1)$ → true

From the definition of Big theta (Θ), there must exist constants $C_1 > 0$, $C_2 > 0$ and integer $N > 0$ such that ;

$0 \leq C_1 \cdot n^2 \leq n^2 + 1 \leq C_2 n^2$ for all $n \geq N$

Lower Bound

$f(n) \geq C_1 g(n)$

$n^2 + 1 \geq n^2$ for all $n \geq 0$

$\therefore C_1 = 1$ for this instance

Upper Bound

$f(n) \leq C_2 g(n)$

$n^2 + 1 \leq n^2 + n^2$

$n^2 + 1 \leq 2n^2$

$\therefore C_2 = 2$ for this instance

Therefore let $C_1 = 1$, $C_2 = 2$, $N = 1$,

$n^2 \leq n^2 + 1 \leq 2n^2$

C_1, C_2, N satisfies the equation for this instance.

Therefore $(n^2+1) = \Theta(n^2)$, $\Theta(n^2) = \Theta(n^2+1)$

(1.4) 1) Is $3^n = O(2^n)$? No

From the definition of Big-Oh there must exist constant $c > 0$, and integer $N > 0$, such that $3^n \leq c\,2^n$ for all of $n \geq N$

$$3^n \leq c\,2^n$$
$$\left(\frac{3}{2}\right)^n \leq c$$
$$(1.5)^n \leq c$$

Proof by cases:

$n = 1$; $(3/2)^1 = 1.5$

$n = 10$; $(3/2)^{10} = 57.67$

$n = 20$; $(3/2)^{20} = 3,325.26$

Therefore we see that there exist no 'c' that satisfies the inequality $(3/2)^n \leq c$ for all large n.

Therefore $3^n \neq O(2^n)$

2) Is $\log_3 n = O(\log_2 n)$? Yes

From the definition of Big Oh, there must exist a constant $c > 0$, $n > 0$ and integer $N > 0$ such that $\log_3 n \leq c \log_2 n$ for all $n > N$.

- $\log_3 n = \dfrac{\log_2 n}{\log_2 3}$
- $\log_2 3 \simeq 1.585$ a positive constant.

Therefore $\log_3 n = \left(\dfrac{1}{\log_2 3}\right) \log_2 n$

$$\log_3 n = c \log_2 n, \quad c = 1$$

where $c = 1/\log_2 3$, for all $n \geq 2$

Therefore when $c = 1/\log_2 3$ and $N = 2$, $\log_3 n = O(\log_2 n)$

3) Is $3^n = \Omega(2^n)$? Yes

From the definition of Big-Omega, there must exist a constants $c > 0$, integer $N > 0$ such that, $3^n \geq c\,2^n$ for all $n \geq N$.

$$3^n \geq c\,2^n$$
$$(3/2)^n \geq c$$

Since $3/2 = 1.5$, which is > 1, ~~the~~ For a positive constant c, we can choose N, large enough such that $(3/2)^n \geq c$ for all $n \geq N$.

Let $c = 1$, $N = 1$ for $3^n \geq c\,2^n$

$n=1$; $3^1 = 3 \geq 1 \cdot 2^1 = 2$

$n=2$; $9 \geq 4$.

$n=3$; $27 \geq 8$.

$\therefore$ for all $n \geq 1$, $3^n > 2^n$ is satisfied.

Therefore $3^n = \Omega(2^n)$

4) Is $\log_3 n = \Omega(\log_2 n)$? Yes

From the definition of Big Omega, there must exist a constant $c > 0$, integer $N > 0$ such that $\log_3 n \geq c \log_2 n$ for all $n \geq N$

$$\log_3 n = \frac{\log_2 n}{\log_2 3}$$
$$= \left(\frac{1}{\log_2 3}\right) \log_2 n$$

$\log_3 n \geq c \cdot \log_2 n$

for $n \geq 2$, $c = 1$

$$\log_3 n = c \log_2 n$$

let $c = \frac{1}{\log_2 3}$, ~~$N \geq 2$~~ $N = 2$, then $\log_3 n = \Omega(\log_2 n)$

(02) (1) A)

Size of problem - n

Base case - $n \leq 1$

one simple operation ("return") $\therefore \Theta(1)$

When it keeps going - does 2 operations, subtraction $(n-1)$, multiplication ($n *$ accumulator)

This is all considered $\Theta(1)$ work.

```
int fact_helper (int n, int accumulator) {
if (n <= 1)  ← Θ(1); a comparison
    return accumulator; ; return : Θ(1)
else                                  multiplication : Θ(1)
    return fact_helper (n-1, n * accumulator);
}
              Substraction
                Θ(1)
```

Total non-recursive work $= \Theta(1)$

$$T(\text{size}) = \begin{cases} \text{DirectSolution(cost)} & \text{for size} \leq \text{DirectSolSize} \\ \text{RecursiveCallSum} + \underbrace{t(\text{size})}_{\text{also called non-recursive work}} \end{cases}$$

Here,

$$T(\text{size}) = \begin{cases} \Theta(1) & \text{when } n \leq 1 \\ T(n-1) + \Theta(1) & \text{for } n > 1 \end{cases}$$

$\therefore \quad T(n) = T(n-1) + \Theta(1)$

B)

	depth	nodes per level	T(n)
$t(n) = \Theta(1)$	0	1	$\Theta(1)$
$t(n-1) = \Theta(1)$	1	1	$\Theta(1)$
$t(n-2) = \Theta(1)$	2	1	$\Theta(1)$
⋮			
$T(1) = \Theta(1)$	n-1	1	$\Theta(1)$

c) Total work = work per level × number of levels

$= \Theta(1) \times n$

$= \underline{\Theta(n)}$

The 4th algorithm makes n recursive calls, and work done per call is $\Theta(1)$.

This algorithm has only one recursive call as well. Thus final total work done will be $\Theta(n)$

$$T(n) = T(n-1) + \overbrace{\Theta(1)}^{c}$$

$$= [T(n-2) + c] + c \rightarrow T(n-2) + 2c$$

$$= T(n-3) + 3c$$

⋮

$$T(1) + \cancel{\Theta}(n-1)c$$

$$= T(n-k) + kc \quad \text{let } c = 1$$

$$T(n) = T(n-k) + k$$

Base case $T(0) = 1$ $T(n-k)$ to be $T(c)$, $n = k$.

$\therefore T(n) = T(0) + k$

$T(n) = n \Rightarrow \underline{\Theta(n)}$

② (A) Find (X, low, high)
if (low > high)

```
Find ( X, low, high):
    if low > high:
        return low + 1

    mid = floor((low+high)/2)

    if X[mid] == mid + 1:
        return Find(X, mid+1, high)
    else:
        return Find (X, low, mid-1)

Find (A, 0, n-1)  // main function call
```

(B) Size of problem = n.
Direct Solution Size is when low > high.
Direct Solution Count = $\Theta(1)$

Explanation -
Here we use binary search concept. The algorithm first checks if the middle element satisfies the condition. If all the elements up to the middle element are in correct positions, the missing element is looked for in the right half. But if the middle element doesn't tally, we search for the element in the left half. This is done recursively until base case condition which is (low > high) is met.

(B) $$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 0 \quad (\text{low} > \text{high}) \\ T(n/2) + \Theta(1) & \text{for } n > 0 \end{cases}$$

$T(n) = T(n/2) + c$ where c is a constant

$T(1) = c_1$

	depth	nodes per level	work
$t(n) = \Theta(1)$ or c	0	1	$\Theta(1)$ or c
$t(n/2) = \Theta(1)$ or c	1	1	$\Theta(1)$ or c
$t(n/4) = \Theta(1)$ or c	2	1	$\Theta(1)$ or c
$t(n/2^k) = c$	k	1	$\Theta(1)$ or c

let $\frac{n}{2^k} = 1$ for; $t(1) = 1$

$n = 2^k$

$k = \log_2 n$

Number of levels $= k + 1$

$= \log_2 n + 1$

$\therefore T(n) = \Theta(\log n)$

(a)

```
Median (A, B, n, m):
    if n=0:
        return MedianOneArray (B, m)
    if m=0:
        return MedianOneArray(A, n)

    if n==1 AND m==1:
        return (A[0] + B[0]) /2

    median_A = MedianOneArray(A, n)
    median_B = MedianOneArray(B, m)

    if median_A == median_B:
        return median_A

    if median_A < median_B:
        newA = A[n/2 ... n-1]
        newB = B[0 ... m/2]
        return ~~MedianOneArray~~ Median(newA, newB, size(newA),
                                         size(newB))

    else
        newA = A[0 .... n/2]
        newB = B[m/2 ... m-1]
        return Median(newA, newB, size(newA), size(newB))


MedianOneArray(X, k):
    if k is odd:
        return X[k/2]

    else:
        return (X[k/2+1] + X[k/2]) /2
```

When we are given 2 sorted ~~arrays~~ arrays size n, and m_s, finding the median comes with different cases. Let m_A - median of A, m_B be median of B.

Case 01

$m_A = m_B$, that value is the median of both arrays combined

Case 02.

$m_A < m_B$,

then all elements before m_A in A are smaller than the median and elements after m_B in B are larger than the median.

So for optimisation we discard first half of A and 2nd half of B

Case 03.

$m_A > m_B$

As in Case 02, we discard second half of A here and first half of B.

So basically we eliminate half of the total elements and recurse remaining halfes. Divide and conquer is used.

Find medians in 2 arrays $\rightarrow \Theta(1)$

Compare medians $\rightarrow \Theta(1)$

One recursive call

So total non-recursive calls $= \Theta(1)$

Size $= n+m = N$

Base cases : if $n=0$

if $m=0$

if $n=1$ and $m=1$

thus DirectSolSize $= N \leq 2$

DirectSolution(count $= \Theta(1)$

Recursive call $a = T(N/2)$

$$\therefore T(N) = \begin{cases} \Theta(1) & \text{if } N \leq 2 \\ T(N/2) + \Theta(1) & \text{if } N > 2 \end{cases}$$

	Depth	Node per level	Work
$t(N) = \Theta(1)$ or c	0	1	c
$t(N/2) = c$	1	1	c
$t(N/4) = c$	2	1	c
$\vdots$			
$t(N/2^k) = c$	k	1	c

$t(1) = 1$ for base case.

$\therefore \frac{N}{2^k} = 1$

$k = \log_2 N$

Work per level $= c$

Number of levels $= \log_2 N$

Total work $= c \log N = c \log(m+n)$

$T(n+m) = \Theta(\log(n+m))$

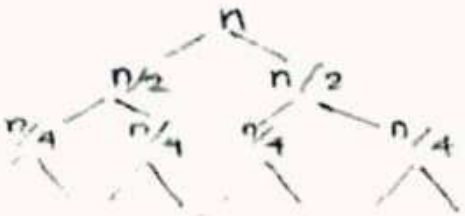
(4)

```
Quicksort ( A, low, high )
    if ( low < high):
        pivotIndex = Partition ( A, low, high )
        Quicksort ( A, low, pivotIndex - 1)
        Quicksort ( A, pivotIndex + 1 , high )

Partition ( A, low, high)
    pivot = A[high]
    i = low - 1
    for j = low to high - 1 :
        if A[j] ≤ pivot :
            i = i + 1
            swap A[i] with A[j]
    swap A[i+1] with A[high]
    return i+1;
```

Here the Partition function rearranges the array so that all the elements less than or equal to the pivot comes before it (to its left) and those greater than it comes to its front (to the right). Thus the pivot is fixed to its proper final position. This is recursively done changing the pivot each time, thus one by one sorting the 2 sub arrays to either side of the pivot of that instance and setting the pivot in its right place. This repeats till the base case is met (low ≥ high) which also handles array of size 0 or 1, which is already sorted.

In an average case, time taken for dividing the array to roughly equal 2 halves of size $n/2$, and partition for partition function to go through the elements swapping them takes $\Theta(n)$. As it works on all n elements.

$\therefore T(n) = \begin{cases} \Theta(1) & \text{when } n \leq 1 \\ 2T(n/2) + \Theta(n) & \text{otherwise.} \end{cases}$



depth	node per level	T(n)
0	1	n
1	2	n
2	4	n
k	2^k	n

Base case : $T(1) = 1$

$T(n/2^k) = 1$

$n/2^k = 1$

$k = \log_2 n$

Number of levels $= \log_2 n + 1$

Work per level $= n$

Total work $= n(\log_2 n + 1) = n\log_2 n + n$

$\therefore$ Average case running time $= \underline{O(n \log_2 n)}$

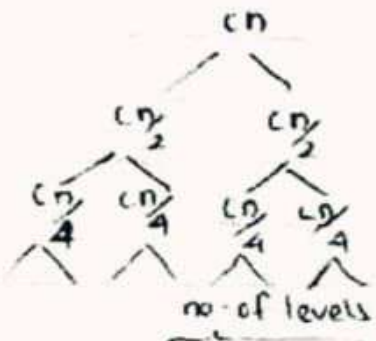
(5) Size = n

DirectSolutionSize = 1 (base case)

DirectSolution(n) = $\Theta(1)$ (return a single element)

```
FFT (a):
  n = len(a)
  if n == 1
      return a ---- Θ(1)

  omega = exp(2*pi * 1/n) ---- Θ(1)

  a_even = [a[2*k] for k in 0 to n/2 - 1] ---- Θ(n)
  a_odd  = [a[2*k+1] for k in 0 to n/2 - 1] ---- Θ(n)

  even_dft = FFT [a_even]   } 2 recursive calls so
  odd_dft  = FFT [a_even]   }   T(n/2) + T(n/2)

  result = [0] * n ---- Θ(n) (creating an array of size n)

  for k in 0 to n/2 - 1                                  } Θ(1)
      wk = omega ^ k                                     }
      result[k]       = even_dft[k] + wk * odd_dft[k]    }
      result[k+n/2] = even_dft[k] - wk * odd_dft[k]      }
  } T(n/2) + Θ(n)

  return result ---- Θ(1)
```

$$T(n) = \begin{cases} c & \text{when } n = 1 \\ 2T(n/2) + cn & \text{otherwise} \end{cases}$$



depth	nodes pr lvl	T(n)
0	1	cn
1	2	cn
2	4	cn
k	2^k	cn

Total work = $(\log_2 n + 1)$ (no. of levels) $\times$ cn (work per level)

Work per = $cn \log_2 n + cn$

$\therefore T(n) = \Theta(n \log n)$

Base case $T(1) = 1$

$T(n/2^k) = 1$

$\frac{n}{2^k} = 1$

$k = \log_2 n$