1)

| Sequence | Function type | |
|---|---|---|
| 1 | Constant | $n^0$ |
| 2 | Log-logarithmic | - |
| 3 | Logarithm | $2^{lglgn} + \sqrt[3]{n}$, $lg(n+10)^2$ |
| 4 | Polylogarithmic | - |
| 5 | Radicals | - |
| 6 | Linear | $\sum_{k=1}^{n} k$ , $2^{lgn}$, |
| 7 | Linearithmic | 5nlgn + 20n |
| 8 | Polynomial-Quadratic | $16^{lgn}$ |
| 9 | Polynomial-Cube | $n^3$ |
| 10 | Exponentiation | $3^{2n}$ |
| 11 | Factorial | $(n^3+3)!$ |

2)

```
static int doIt(int n) {
        count <- 0                ----------O(1)
        j <- 0            ------------O(1)
        for i  0 to n-1 {      -----------O(n)
                j = i     -------------O(1) *  O(n)
                while j != 0 {      -----------O(lg n) * O(n)
                        if j%2 = 0 ----------O(1)  * O(lgn) * O(n)
                                count = count + 1   ---------O(1)   * O(lg n) * O(n)
                        }
                        j = j / 2  ----------O(1)   * O(lg n) * O(n)
                }
        }
        return count   -------O(1)
}
```

Calculation for doIt(int n) function:

1 + 1 + n + n + nlgn + nlgn + nlgn + nlgn = 3n lgn + 2n + 2

```
static int myMethod (int n) {
```

```
    sum <- 0            -----O(1)
    for i <- 1 to n {               ------- O(n)
        sum = sum + doIt(i)    --------O(n)*(O(3n log n)  + O(2n) + O(2)))
    }
    return 1;    ----O(1)
}
```

Calculation for myMethod (int n):

1 + n + n*(3n log n  + 2n + 2) + 1 = 1 + n + 3n^2 log n + 2n^2 + 2n +1

= O(n^2 log n)

3)

a)

```
function random_sort(array, new_array, max) {
    if (array.length == 0 or array.length == none) {
        return max
    }

    else {
        index <- random integer given the range from 0 to array.length
        random_int <- array[index]
        //Add random int to new array
        new_array[new_array.length()] <- random_int

        if random_int > max {
            max <- random_int
        }

        //for loop to shift all numbers to left to override the chosen index
        for i <- index to array.length()-1 {
            array[index] <- index + 1
        }

        //for loop to exclude last item
        for i <- 0 to array.length()-2 {
            dummy_array <- array[i]
        }
        array <- dummy_array
        return random_sort(array, new_array, max)
    }

}

//main function to drive the other algorithm
function main() {
    array = {1,2,3,4,5}
    new_array = {}
```

max = 0
random_sort(array, new_array,max)
}

b)

Worst-case complexity

| Algorithm | Calculation |
|---|---|
| function random_sort(array, new_array, max) { | |
| if (array.length == 0 or array.length == none) { | 1 |
| return max | 1 |
| } | |
| else { | 1 |
| index <- random integer given the range from 0 to array.length | 1 |
| random_int <- array[index] | 1 |
| new_array[new_array.length()] <- random_int | 1 |
| if random_int > max { | 1 |
| max <- random_int | 1 |
| } | |
| for i <- index to array.length()-1 { | n |
| array[index] <- index + 1 | n*1 |
| } | |
| for i <- 0 to array.length()-2 { | n*1 |
| dummy_array <- array[i] | n*1 |
| } | |
| array <- dummy_array | 1 |
| return random_sort(array, new_array, max) | 1 |
| } | |

| | |
|---|---|
| } | |

calculation = 11 + 4n

| Algorithm | Calculation |
|---|---|
| function main() { | |
| array = {1,2,3,4,5} | 1 |
| new_array = {} | 1 |
| max = 0 | 1 |
| random_sort(array, new_array,max) | n * (11 + 4n) |
| } | |

calculation = 3 + 11n + 4n^2
time complexity for worst case is Θ(n^2).

Best Case

| Algorithm | Calculation |
|---|---|
| function random_sort(array, new_array, max) { | |
| if (array.length == 0 or array.length == none) { | 1 |
| return max | 1 |
| } | |
| else { | 1 |
| index <- random integer given the range from 0 to array.length | 1 |
| random_int <- array[index] | 1 |
| new_array[new_array.length()] <- random_int | 1 |
| if random_int > max { | 1 |
| max <- random_int | 1 |
| } | |
| for i <- index to array.length()-1 { | n |

| | |
|---|---|
| array[index] <- index + 1 | n*1 |
| } | |
| for i <- 0 to array.length()-2 { | n*1 |
| dummy_array <- array[i] | n*1 |
| } | |
| array <- dummy_array | 1 |
| return random_sort(array, new_array, max) | 1 |
| } | |
| } | |

calculation = 11 + 4n

| Algorithm | Calculation |
|---|---|
| function main() { | |
| array = {1,2,3,4,5} | 1 |
| new_array = {} | 1 |
| max = 0 | 1 |
| random_sort(array, new_array,max) | n * (11 + 4n) |
| } | |

calculation = 3 + 11n + 4n^2
time complexity for best case is Θ(n^2).

**Average-case**

| Algorithm | Calculation |
|---|---|
| function random_sort(array, new_array, max) { | |
| if (array.length == 0 or array.length == none) { | 1 |
| return max | 1 |
| } | |

| | |
|---|---|
| else { | 1 |
| index <- random integer given the range from 0 to array.length | 1 |
| random_int <- array[index] | 1 |
| new_array[new_array.length()] <- random_int | 1 |
| if random_int > max { | 1 |
| max <- random_int | 1 |
| } | |
| for i <- index to array.length()-1 { | n |
| array[index] <- index + 1 | n*1 |
| } | |
| for i <- 0 to array.length()-2 { | n*1 |
| dummy_array <- array[i] | n*1 |
| } | |
| array <- dummy_array | 1 |
| return random_sort(array, new_array, max) | 1 |
| } | |
| } | |

calculation = 11 + 4n

| Algorithm | Calculation |
|---|---|
| function main() { | |
| array = {1,2,3,4,5} | 1 |
| new_array = {} | 1 |
| max = 0 | 1 |
| random_sort(array, new_array,max) | n * (11 + 4n) |
| } | |

calculation = 3 + 11n + 4n^2

time complexity for average case is Θ(n^2).

4)
a)
$T(n) = 4T(n / 2) + n^2 + n$
so, a=4, b=2, c=2, $f(n)=n^2 + n$

$a / b^c = 4/ 2*2$
= 1 (apply case 2)

<u>Case 2:</u>
$f(n) \in \Theta(n^{\lg_b a})$
$n^2 + n \in \Theta(n^{\lg_2 4})$
$n^2 + n \in \Theta(n^2)$, this is true
$\therefore \Theta(n^{\lg_2 4}\lg n) = \Theta(n^{\lg_2 4}\lg n) = \Theta(n^2\lg n)$

b)
$T(n) = 16T(n / 4) + n$
a=16, b=4, c=1, f(n)=n

$a / b^c = 16/ 4*1$
= 4 >1 (apply case 1)

Case 1:
$f(n) \in O(n^{\lg_b a-\varepsilon})$ for some $\varepsilon > 0$.
$n \in O(n^{\lg_4 16-\varepsilon})$ for some $\varepsilon > 0$.
$n \in O(n^{2-\varepsilon})$ for some $\varepsilon > 0$.

let $\varepsilon = 1$.
$n \in O(n^{2-1})$
$n \in O(n)$
$\therefore$
$T(n) = \Theta(n^{\lg_2 4})$
$= \Theta(n^2)$

c)
$T(n) = 16T(n/4) + n^3$
a=16, b=4, c=3, $f(n)=n^3$

$a / b^c = 16/64$
= 0.25<1

Case 3:
$f(n) \in O(n^{\lg_b a+\varepsilon})$ for some $\varepsilon > 0$.
$n^3 \in \Omega(n^{\lg_4 16+\varepsilon})$
$n^3 \in \Omega(n^{2+\varepsilon})$

when $\varepsilon$ = 1>0,
$n^3 \in \Omega(n^{2+1})$
$n^3 \in \Omega(n^3)$

$a * f(n/b) \leq C * f(n)$
$16 * f(n/4) \leq C * n^3$
$16 * (n/4)^3 \leq C * n^3$
$16 / 64 \leq C$
$1 / 4 \leq C$
C >= 0.25
so C < 1
$\therefore T(n) = \Theta(n^3)$

d)
Since this equation is a reduce and conquer algorithm,
a=1, b=1, $f(n) = n^4$
If a=1 then $T(n) = O(n^{k+1})$ or $O(n * f(n))$
$T(n) = O(n^5)$

e)
$T(n) = 2T(n/ 2) + n \lg n$
$a = 2$, $b = 2$, $c = 1$, $f(n) = n \lg n$
$a / b^c = 2 / 2 = 1$, hence we may use test case 2
$n \lg n \in \Theta(n \log_2 2)$
$n \lg n \in \Theta(n)$
Hence, master theorem cannot be applied. Expansion and substitution will be used instead.
$T(n) = 2T(n/2) + n \lg n$
$T(n) = 2 [2T(n/2^2)+ n/2 * \lg n/2] + n\lg n$
$T(n)= 2^2T(n/2^2)+ n \lg(n - 1)] + n\lg n$
$T(n) = 2^2T(n/2^2)+ n \lg n - n + n\lg n$
$T(n) = 2^2T(n/2^2)+ 2n \lg n - n$

$T(n) = 2^2T(n/2^2)+ 2n \lg n - n$
$T(n) = 2^2[2T(n/2^3)+ n/2^2 * \lg n/2^2]+ 2n \lg n - n$
$T(n) = 2^3T(n/2^3) + n(\lg n - 2) + 2n \lg n - n$
$T(n) = 2^3T(n/2^3) + n \lg n - 2n + 2n \lg n - n$
$T(n) = 2^3T(n/2^3) + 3n \lg n - 3n$

$T(n) = 2^3T(n/2^3) + 3n \lg n - 3n$
$T(n) = 2^3[2T(n/2^4)+ n/2^3 \lg n/2^3] + 3n \lg n - 3n$
$T(n) = 2^4T(n/2^4) + n(\lg n - 3) + 3n \lg n - 3n$
$T(n) = 2^4T(n/2^4) + n \lg n - 3n + 3n \lg n - 3n$
$T(n) = 2\ 4 + 4n \lg n - 6n$

The recurrence relation can be generalized as $T(n) = 2^k T(n/2^k) + k\, n \lg n - (k(k-1)/2)*n$
--eq(1)
The recursive call will continue and stop when $(n/2^k) = 1$. Solving the equality, we have $n = 2^k$ and hence $k = \lg n$. Substituting $k$ into $eq(1)$, we have

$T(n) = 2^{\lg n} T(n/2^{\lg n}) + \lg n(n \lg n) - (\lg^2 n - \lg n)/2 *n$

$$
\begin{aligned}
T(n) &= 2^{\lg n} T(n/2^{\lg n}) + \lg n(n \lg n) - (\lg^2 n - \lg n)/2 *n \\
&= nT(n/n) + n \lg^2 n - n/2 \lg^2 n + n/2 \lg n \\
&= nc + (2n - n)/2 \lg^2 n + n/2 \lg n \\
&= nc + (½)n \lg^2 n + (½)n \lg n
\end{aligned}
$$

∴ *the running time complexity is* $\boldsymbol{\Theta(n \lg^2 n)}$.