Q1:

a)

i) natural size metric: **n**

ii) Basic operation is addition

```
int sum = 0;
for (int i = 0; i < arr.length; i++) {
    sum = sum + arr[i];
}
```

basic operation is just addition

iii) Since you are just summing up the numbers, **the basic operation count will be the same for inputs of the same size**. The best, worst, and average case running time (or number of operations) are all the same.

b)

i) natural size metric: n

ii) Basic operation is multiplication

```
int fact = 1;
for (int i = n; i ≥ 1; i--) {
    fact = fact * i;
}
```

basic operation is multiplication

iii) **The basic operation count will be the same for the same size input**, since you are just multiplying the numbers. The Best, Worst, and Average case running times are the same for the same size input.

c)

i) natural size metric: n

ii) The basic operation is the **comparison**, when the index of the array is compared with the maxVal. This will be executed every iteration of the for loop, while the other assignment operation will not.

ii)

```
int maxVal = A[0];

for (int i = 1; i < n-1; i++){
    if(A[i] > maxval){
        maxval = A[i];
    }
}
```

the most basic operating is comparing the value, since this will be executed everytime

iii) **The Basic Operation count will be the same for same size inputs**. Even if the algorithm finds the max element before the last index, it will still iterate for the rest of the array. This is how the algorithm is designed.

<u>Q2:</u>

a)

<u>$n(n+1)$ and $2000n^2$</u>

$n(n+1) = n^2 + n = \mathbf{O(n^2)}$

$2000n^2 = \mathbf{O(n^2)}$

These functions have the same order of growth.

b)

<u>$100n^2$ and $0.01n^3$</u>

$100n^2$ = $\mathbf{O(n^2)}$
$0.01n^3$ = $\mathbf{O(n^3)}$

The order of growth of the first function is smaller than the second function.

c)

<u>$\log_2(n)$ and $\ln(n)$</u>

$\log_2(n)$ = $\mathbf{O(\log(n))}$
$\ln(n) = \log_e(n)$ = $\mathbf{O(\log(n))}$

These functions have the same order of growth.

d)

d) $\log_2^2 n$ and $\log_2(n^2)$

$\hookrightarrow (\log_2 n)^2$ $\hookrightarrow O(\log(n))$

$\hookrightarrow O(\log^2(n))$

$\hookrightarrow$ The first function has a smaller order of growth than the second function e)

<u>2^{n-1} and 2^n</u>

$2^{n-1} = 2 \times 2^n =$ **$O(2^n)$**
$2^n =$ **$O(2^n)$**

These functions have the same order of growth.

f)

<u>(n-1)! and n!</u>

n! = n(n-1)(n-2)(n-3) ...
(n-1)! = (n-1)(n-2)(n-3) …

n(n-1)! > (n-1)!
As you can see, n! is greater because it’s a higher order polynomial.

<u>Q3)</u>

$$2^{2n} = (2^2)^n = 4^n$$

$$5\lg(n+100)^{10} < \ln^2(n) < \sqrt[3]{n} < 0.001n^4 + 3n^3 + 1 < 3^n < 2^{2n} < (n-2)!$$

<u>Q4)</u>

a) The algorithm finds the max and min elements in the array and returns their difference
b) We consider the **comparisons** as the basic operation because they are executed every iteration, even if the condition is not true, the algorithm must check if it's true.
c) For one comparison it would be (n-1) times since the for loop executes for (n-1) times. But there are 2 comparisons in the for loop so the number of executions is **2(n-1)**.

d) The efficiency class of this algorithm would then be **O(n)**, since you just drop the constant
e) One improvement I would suggest is to first sort the array with a good sorting algorithm that has a very low order of growth, and then we can simply access the first and last elements of the array in constant time (O(1)) and return their difference.

Q6

Q6) a) Chaining

numbers

12	44	13	88	23	94	11	39	20	16	5

$h(i) = (3i + 5) \% 11$

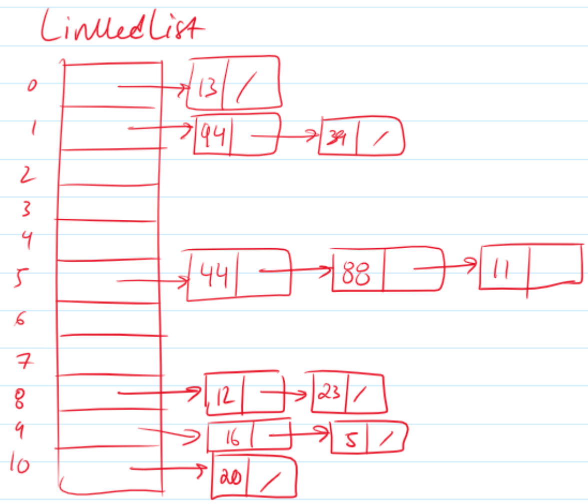
$h(12) = (36 + 5) \% 11$

$= 41 \% 11$

$= 8$

$h(12) = 8$

$h(44) = 5$



b) Lineer probing

$h(i) = (3i + 5) \% 11$

$h'(i) = (h(x) + f(i)) \% 11$

numbers

12	44	13	88	23	94	11	39	20	16	5

Array

0	13
1	94
2	39
3	16
4	5
5	44
6	88
7	11
8	12
9	23
10	20

$h(12) = 8$

$h(44) = 5$

$h(13) = 0$

$h(88) = 5$

$h'(88) = (5 + 0) \% 11$
$= 5$

c) <u>Double Hashing</u>

$h(i) = (3i + 5) \% 11$

$h_2(k) = 7 - (k \% 7)$

numbers

12	44	13	88	23	94	11	39	20	16	5

Array

0	13
1	94
2	23
3	88
4	39
5	44
6	11
7	5
8	12
9	16
10	20