EX1:

A: → Code

B: Cost Function?

Derivation:

Int i = 0; → +1

Outer while loop contains 5 operations:

- Outer loop comparison While (i > n)
- int j = 0;
- terminating comparison for the inner loop while(j < n)
- cout << endl;
- i++;

Inner while loop contains 4 operations:

- comparison while(j < n)
- cout << "{" << arr[i] << "," << arr[j] << "}";
- j++;
- cout << " ";

terminating comparison for the outer loop while (i < n) → +1

t(cartesianProduct) = 1 + 5 * n + n * (4 * n) + 1 = 4 * n^2 + 5 * n + 2

Answer: $\mathrm{t}(\mathrm{cartesianProduct}) = 4 \cdot \mathrm{n}^2 + 5 \cdot n + 2$

C: Barometer operation?

Any of the following operations:

- comparison while(j < n)
- cout << "{" << arr[i] << "," << arr[j] << "}";
- j++;
- cout << " ";

these are the barometer operations because they are executed the most equally.

D: O-notation?

Answer: $\boldsymbol{O(n^2)}$

EX2:

A: → code

B: Cost function? <u>Note that we consider x as n for simplicity.</u>

int i = 0; → +1

terminating comparison `while (i < x);` → +1

terminating comparison `while (i > 0);` → +1

in the 1st outer while loop, there are 5 operations:

- `while (i < x)` (comparison)
- `int j = 0;`
- `while (j <= i)` (terminating comparison)
- `cout << endl;`
- `i++;`

in the 1st inner while loop, there are 3 operations:

- `while (j <= i)`
- `cout << j << " ";`
- `j++;`

note that to get the average number of iterations in this inner while loop,

when i = 0, there is 1 iteration

when i = 1, there are 2 iterations

…

When i = n-1, there are n interations

Avg iterations = (1 + n) * n / (2 * n) = (n + 1) / 2

Hence, the first loop of the cost function t_1 will be

t_1 =5 * n + n * (3 * (n + 1) / 2)

In the 2nd outer while loop, there are 5 operatons:

- `while (i > 0) (comparison)`
- `i--;`
- `int j = 0;`
- `while (j <= i) (terminating comparison)`
- `cout << endl;`

in the 2nd inner loop, there are 3 operations:

- `while (j <= i) (comparison)`
- `cout << j << " ";`
- `j++;`

for the avg iterations,

when i = n – 1, there are n iterations (0 <= n - 1)

when i = n-2, there are n-1 iterations

…

When i = 0, there are 1 iteration (0 <= 0)

Similar to the above, we get (n+1) / 2 avg iteration

t_2 = 5 * n + n * (3 * (n + 1) / 2)

Totally,

Answer: $\mathbf{t(triangle) = 3 \cdot n^2 + 13 \cdot n + 3}$

OR alternatively if we use x: $\mathbf{t(triangle) = 3 \cdot x^2 + 13 \cdot x + 3}$

C: Barometer operation?

Any of the following operations:

in the 1st inner while loop:

```
- while (j <= i)
- cout << j << " ";
- j++;
```

in the 2nd inner loop:

```
- while (j <= i) (comparison)
- cout << j << " ";
- j++;
```

D: o-notation?

Answer: $\boldsymbol{O(n^2)}$

EX3:

A: → code

<Best case>

B: cost function for the best case.

Best case would occur in the array which is all the same values (all duplicates).

E.g. `int arr[] = { 1,1,1,1,1,1,1,1,1 };`

Derivation:

3 operations outside of while(i < n)

```
- vector<int> result;
- int i = 0;
- while (i < n) (terminating comparison)
```

when i = 0,

in the outer while loop, there are 7 operations without going into the inner while loop:

```
- while (i < n)
- int iResult = 0;
- bool duplicate = false;
- while (iResult < (int)result.size() && !duplicate)
- if (!duplicate)
- result.push_back(arr[i]);
- i++;
```

when i = 1 .. n - 1,

in the outer while loop, there are 10 operations. Inner loop is only 1 iteration.

```
- while (i < n)
- int iResult = 0;
- bool duplicate = false;
- while (iResult < (int)result.size() && !duplicate)
- if (arr[i] == result[iResult])
- duplicate = true;
- iResult++;
- while (iResult < (int)result.size() && !duplicate) (terminating comparison)
- if (!duplicate)
- i++;
```

hence, the cost function for the best case:

t(removeDuplicates, best) = 3 + 7 + (n-1) * 10 = 10 * n

Answer: $\mathbf{t(removeDuplicates, best) = 10 \cdot n}$

C: Barometer operation for the best case.

Any of the following: (these are executed the most number of times)

```
- while (i < n)
- int iResult = 0;
- bool duplicate = false;
- while (iResult < (int)result.size() && !duplicate)
- if (!duplicate)
- i++;
```

Reason: these are the operations that are executed the most. For the inner loop, any statements inside of the inner loop are not executed when i = 0, so they are not executed the most frequent and hence not the barometer operations. Also, only when i = 0, result.push_back is called, so it is also not barometer operation. Other than these, the 6 operations (any of them) I stated will be the barometer operations.

D: O-notation for the best case.

Answer: $O(n)$

<Worst case>

B: cost function for the worst case.

Best case would occur in the array which has no duplicates at all.

E.g. `int arr[] = { 1,2,3,4,5,6,7,8,9 };`

Derivation:

3 operations outside of while(i < n)

```
- vector<int> result;
- int i = 0;
- while (i < n) (terminating comparison)
```

when i = 0,

in the outer while loop, there are 7 operations without going into the inner while loop:

```
- while (i < n)
- int iResult = 0;
- bool duplicate = false;
- while (iResult < (int)result.size() && !duplicate)
- if (!duplicate)
- result.push_back(arr[i]);
- i++;
```

when i = 1 .. n-1,

in the outer while loop, there are 7 operations

```
- while (i < n)
- int iResult = 0;
- bool duplicate = false;
- while (iResult < (int)result.size() && !duplicate) (terminating comparison)
- if (!duplicate)
- result.push_back(arr[i]);
- i++;
```

in the inner while loop, there are 3 operations (if statement never be equal since no duplicates)

```
- while (iResult < (int)result.size() && !duplicate)
- if (arr[i] == result[iResult])
- iResult++;
```

avg iterations is

when i = 1, result.size() = 1, 0 < 1, so 1 iteration

when i = 2, result.size() = 2, 0 < 2, so 2 iterations

..

When i = n-1, result.size() = n - 1, 0 < n - 1, so n - 1 iterations

avg iteration = (1 + n - 1) * (n - 1) / (2 * (n - 1)) = n / 2

hence, the cost function for the worst case is

t(removeDuplicates, worst) = 3 + 7 + (n-1) * 7 + (n-1) * (n / 2 * 3)

Answer: $\mathbf{t(removeDuplicates, worst) = 1.5 \cdot n^2 + 5.5 \cdot n + 3}$

C: Barometer operation for the worst case.

Any of the following:

```
- while (iResult < (int)result.size() && !duplicate)
- if (arr[i] == result[iResult])
- iResult++;
```

Reason: they are inside of inner loop (multiple iterations) that have been executed the most.

D: O-notation for the worst case.

Answer: $\boldsymbol{O(n^2)}$

Again, **example** of best and worst case:

All duplicate (Best): `int arr[] = { 1,1,1,1,1,1,1,1,1 };`

No duplicate (Worst): `int arr[] = { 1,2,3,4,5,6,7,8,9 };`

EX4:

A: → code

B: Cost function.

Note that `rcIndex()` is a constant time, so we count it as single operation and of course regardless of its complexities in a singlestatement.

Let's refer **n** = rows (= columns).

Outside of outer while loop, there are 4 operations:

```
- int columns = rows;
- int* result = new int[rows * columns];
- int r = 0;
- while (r < rows) (terminating comparison)
```

Inside of outer while loop, there are 4 operations:

```
- while (r < rows)
- int c = 0;
- while (c < columns) (terminating comparison)
- r++;
```

Inside of mid while loop, there are 6 operations:

```
- while (c < columns)
- int next = 0;
- int iNext = 0;
- while (iNext < rows) (terminating comparison)
- result[rcIndex(r, c, columns)] = next;
- c++;
```

inside of inner while loop, there are 3 operations:

```
- while (iNext < rows)
- next += m[rcIndex(r, iNext, columns)] * m[rcIndex(iNext, c, columns)];
- iNext++;
```

Hence, the cost function will be:

t(matrixSelfMultiply) = 4 + 4 * n + n * (6 * n + n * (3 * n))

Answer: $\mathbf{t(matrixSelfMultiply)} = \mathbf{3} \cdot \boldsymbol{n}^{\mathbf{3}} + \mathbf{6} \cdot \boldsymbol{n}^{\mathbf{2}} + \mathbf{4} \cdot \boldsymbol{n} + \mathbf{4}$

C: Barometer operation.

Any of the following:

```
- while (iNext < rows)
- next += m[rcIndex(r, iNext, columns)] * m[rcIndex(iNext, c, columns)];
- iNext++;
```

D: O-notation.

Answer: $\boldsymbol{O}(\boldsymbol{n}^{\mathbf{3}})$

EX5:

A: → code

<best case>

B: Cost function for the best case.

Best case example for the selection sort is the array that is already sorted.

e.g. `int arr[] = { 0,1,4,5,7 };`

Derivation:

First, add +1 for the if statement since it will be executed after the function call regardless

- `if (i < n - 1)`

Inside of the if (i < n-1), there are 6 operations. Note that if statement comparison count is already included above.

- `int next = i + 1;`
- `int smallest = i;`
- `while (next < n) (terminating comparison)`
- `int temp = arr[i];`
- `arr[i] = arr[smallest];`
- `arr[smallest] = temp;`

Inside of the while loop, there are 3 operations. Note that for the best case, arr is already sorted and so `(arr[next] < arr[smallest])` will be always false.

- `while (next < n)`
- `if (arr[next] < arr[smallest])`
- `next++;`

The average number of iterations in this inner while loop while (next < n) is:

when i = 0, next = 1, while (1 < n) → n – 1 iteration

when i = 1, next = 2, while (2 < n) → n – 2 iteration

…

when i = n-2, next = n-1, while (n-1 < n) → 1 iteration

(when i = n- 1, `if (i < n - 1)` won't be true, so will go to implicit base case and return ops)

avg iterations = (1 + n - 1) * (n - 1) / (2 * (n - 1)) = n / 2

Note that the number of times the function ssort gets called is:

When i = 0, ssort gets called

When i = 1, ssort gets called

…

When i = n-2, ssort gets called

When i = n-1, ssort don't get called but go to implicit base case

So the ssort will be called n - 1 times

Let's solve for the cost function.

When i = 0 to i = n - 2, if statement becomes true, so the cost is

t(i=0.. n-2) = (n - 1) * (6 + n / 2 * 3 + 1) → I added + 1 for if statement

when i = n-1, if statement becomes false,

t(i = n - 1) = 1 → only when i = n-1

summing up,

t(ssort) = 1 + (n - 1) * (6 + n / 2 * 3 + 1)

Answer: $\mathbf{t(ssort, best) = 1.5 \cdot n^2 + 5.5 \cdot n - 6}$

C: Barometer operation for the best case.

Any of the following:

```
- while (next < n)
- if (arr[next] < arr[smallest])
- next++;
```

since they are executed the most.

D: O-notation for the best case.

Answer: $\boldsymbol{O(n^2)}$

<worst case>

C: Barometer operation for the worst case.

Worst case would happen when the array is in a reversed order.

Any of the following:

```
- while (next < n)
- if (arr[next] < arr[smallest])
- next++;
```

Reason: Inside of if statement, we have `smallest = next;` this statement is not always executed (further detail proof below by actually calculating the cost). Unlike the best case, it is executed but not always (for the best case, it is never executed). Therefore, the above 3 lines of codes are the most executed lines and hence barometer operations. For example, for the case:

e.g. `int arr[] = { 5,4,3,2,1 };`

`smallest = next;` is executed 4 times when i = 0. arr is {5, 4, 3, 2, 1}

`smallest = next;` is executed 2 times when i = 1. arr is {4, 3, 2, 5}

and not anymore. When i =2, arr is {2, 3, 4} now already sorted

D: O-notation for the worst case.

Answer: $\boldsymbol{O(n^2)}$

Again, **examples** of best and worst case:

Array that is already sorted (best): `int arr[] = { 1,2,3,4,5,6,7,8,9 };`

Array that is all duplicate is also best case (best): `int arr[] = { 1,1,1,1,1,1,1,1,1 };`

Array that is in a reversed order (Worst): `int arr[] = { 9,8,7,6,5,4,3,2,1};`

We can actually further prove that the worst case is O(n^2) by actually calculating the cost.

Using the formula:

$$sum = \#\ of\ sequence * \frac{first\ value + last\ value}{2}$$

When n is odd, `smallest = next;` is executed:

$$2\ times + 4\ times + 6\ times + .. + (n-1)\ times = \frac{n-1}{2} \cdot \frac{2 + (n-1)}{2} = \frac{1}{4}n^2 - \frac{1}{4}$$

When n is even, `smallest = next` is executed:

$$1\ times + 3\ times + 5\ times + .. + (n-1)\ times = \frac{n}{2} \cdot \frac{1 + (n-1)}{2} = \frac{1}{4}n^2$$

Hence, the cost function is, adding to the best case for each,

t(ssort, worst, n is even) = 1 + (n - 1) * (6 + n / 2 * 3 + 1) + 1 / 4 * n^2

t(ssort, worst, n is odd) = 1 + (n - 1) * (6 + n / 2 * 3 + 1) + 1 / 4 * n^2 – 1/4

Hence, the exact cost function i

$$\mathbf{t(ssort, worst, n\ is\ even)} = \mathbf{1.75 \cdot n^2 + 5.5 \cdot n - 6}$$

$$\mathbf{t(ssort, worst, n\ is\ odd)} = \mathbf{1.75 \cdot n^2 + 5.5 \cdot n - 6.25}$$

Therefore, $\boldsymbol{O(n^2)}$

EX6:

A: → code

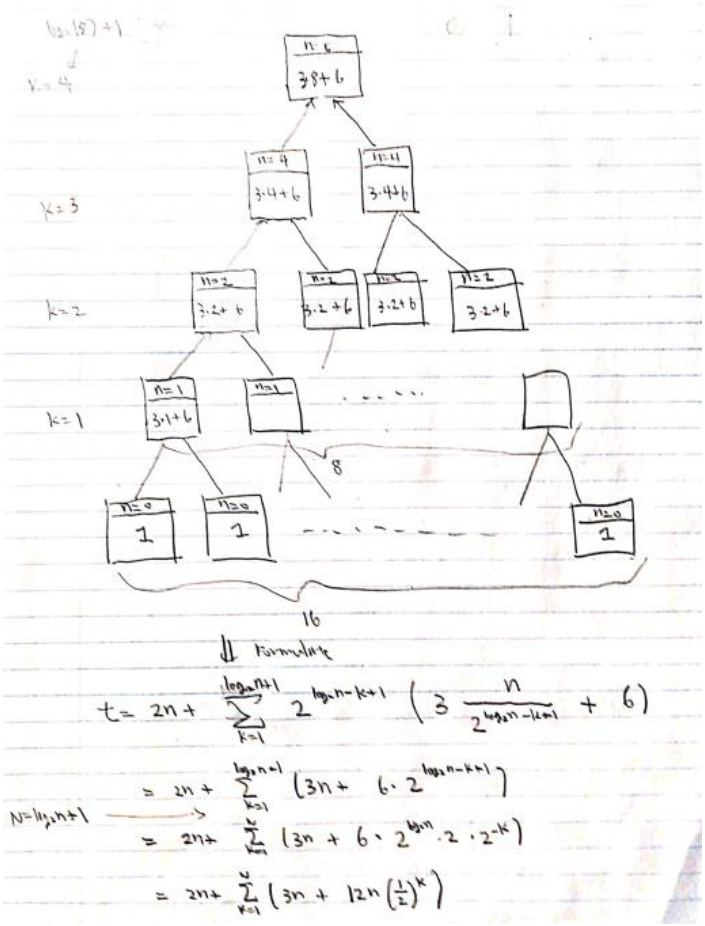
B: Cost function.

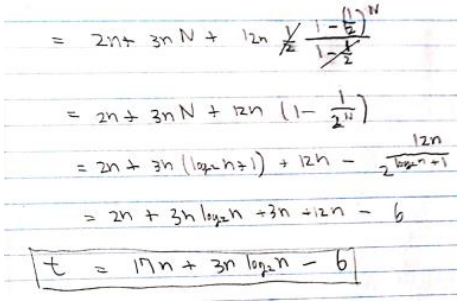
Answer: $\mathbf{t(ssort, best) = 17 \cdot n + 3 \cdot n \cdot log_2(n) - 6}$

Derivation**:**

There are totally 3 * n + 6 operations in a one run of function `pattern`

To derive the cost function, consider the example n = 8, you have the following diagram and formulation.





C: Barometer operation.

Any of the following:

- `while (ast < n)`
- `cout << "* ";`
- `ast++;`

D: O-notation.

Answer: $\boldsymbol{O(n \cdot log_2(n))}$