1e.

The code uses an iterator to traverse thru the vector. On reaching K == 2 we add 5 more elements to it, however our current array does not have enough space for that. What now occurs is the dynamic allocation of a new array with double the space of the original array. The iterator now only remembers memory address of its old positions and fails afterwards as it tries to access memory which is undefined and therefore give random values.

Q3

There is no way for the code to compare two coordinates. the Coord class doesn't have a custom ' > ' or ' < ' operator defined. On calling the insert function , the code checks if the value of the coordinate is bigger than what is already present which does not make sense because a coordinate is just a point with a (x,y) entity representing it.

Q4b

Without the path argument, we would not be able to store the hierarchy in which the path needs to be printed when recursively calling the function. On calling the listAll recursively we will create a new path variable each time which would just print out that submenus name with a separator.

Q5a

const int N = *some value*; ==> O(1)

bool hasCommunicatedWith[N][N]; ==> O(1)

... ==> O(1)

int numIntermediaries[N][N]; ==> O(1)

for (int i = 0; i < N; i++). ==> O(N)

{

numIntermediaries[i][i] = -1; ==> O(1)

for (int j = 0; j < N; j++) ==> O(N)

{

if (i == j) ==> O(1)

continue;

numIntermediaries[i][j] = 0; ==> O(1)

for (int k = 0; k < N; k++) ==> O(N)

{

if (k == i || k == j) ==> O(1)

continue;

if (hasCommunicatedWith[i][k] && ==> O(1) hasCommunicatedWith[k][j])

numIntermediaries[i][j]++; ==> O(1)

}

}

}

Time complexity is O(N^3) because there are 3 nested for loops, all running from 0 to N and the functions called inside have constant time.

Q5b

const int N = some value; **==> 0(1)**

bool hasCommunicatedWith[N][N]; **==> 0(1)**

...

int numIntermediaries[N][N]; **==> 0(1)**

for (int i = 0; i < N; i++) **==> 0(N)**

{

numIntermediaries[i][i] = -1; **==> 0(1)**

for (int j = 0; j < **i**; j++) **==> 0(N)**

{

numIntermediaries[i][j] = 0; **==> 0(1)**

for (int k = 0; k < N; k++) **==> 0(N)**

{

if (k == i || k == j) **==> 0(1)**

continue;

if (hasCommunicatedWith[i][k] && **==> 0(1)** hasCommunicatedWith[k][j])

numIntermediaries[i][j]++; **==> 0(1)**

}

**numIntermediaries[j][i] = numIntermediaries[i][j]; ==> 0(1)**

}

}

Time complexity is O(N^3) because there are 3 nested for loops, all running from 0 to N and the functions called inside have constant time. Assuming worst case scenario where the i takes the value of N and the loop has to run for the entirety of time till it reached N.

Q6a

void concatReverse(const Sequence& seq1, const Sequence& seq2, Sequence& result)

{

Sequence res; **==> 0(1)**

for (int k = seq1.size() - 1; k >= 0; k--) **==> 0(N)**

{

ItemType v; **==> 0(1)**

seq1.get(k, v); **==> 0(N)**

res.insert(res.size(), v); **==> 0(N)**

}

for (int k = seq2.size() - 1; k >= 0; k--) **==> 0(N)**

{

ItemType v; **==> 0(1)**

seq2.get(k, v); **==> 0(N)**

res.insert(res.size(), v); **==> 0(N)**

}

result.swap(res); **==> 0(1)**

}

Insert() has a time complexity of 0(N) and so does get(), swap() however has a time complexity of O(1) and is constant.

Time complexity for concatReverse is O(2N\*N + 2N\*N) = 0(4N^2) which is essentially

**O(N^2)**

We have 2 loops which run from 0 to N and within each loop we have two functions with time complexity of O(N) which add up to give O(2N) and then get multiplied by N respectively for each loop.

Q6b

void Sequence::concatReverse(const Sequence& seq1, const Sequence& seq2)

{

Sequence res;

for (Node\* p = seq1.m\_head->m\_prev; p != seq1.m\_head; p = p->m\_prev)

res.insertBefore(res.m\_head, p->m\_value);

for (Node\* p = seq2.m\_head->m\_prev; p != seq2.m\_head; p = p- >m\_prev)

res.insertBefore(res.m\_head, p->m\_value);

// Swap \*this with res

swap(res);

// Old value of \*this (now in res) is destroyed when function returns.

Time complexity is O(2N) this is because we have two loops which run from 0 to N whereas swap has time complexity of O(1) which is constant. Essentially time complexity of **O(N)**