Part (a)

Problem 3 — Runtime Analysis

CSCI 104 — Devin C. Martin

```
void f1(int n)
{
    int i=2;
    while(i < n){
        /* do something that takes O(1) time */
        i = i*i;
    }
}
```

→ loop condition

→ each iteration i is squared (i=i*i)

→ the # of iterations can be expr as the # of times i requires to be squared in order to exceed n

→ as a result, we can denote this as K, therefore, $i^k >= n$

→ extracting the log base i on each side, we get $k = \log_i(n)$

→ with that being said, the time complexity is $O(\log_i(n))$ where i is the base of the logarithm

→ the loop will perform until i exceeds or equals n. furthermore, the loop will execute for k iterations where k is the lowest int such that 2^k is >= n

→ In Big-O notation, this is exemplified as $O(\log_2(n))$, where $\log_2$ reps the log base 2.

→ this ultimately gives us a logarithmic time complexity of $O(\log_i(n))$ where i is the base of the logarithm

Part (b)

```
void f2(int n)
{
    for(int i=1; i <= n; i++){
        if( (i % (int)sqrt(n)) == 0){
            for(int k=0; k < pow(i,3); k++) {
              /* do something that takes O(1) time */
            }
        }
    }
}
```

→ the worst case for the inner loop stems from the runs for pow(i, 3) time. for the outer loop, it stems from runs for n iterations. As a result, the time complexity is $O(n * \max(pow(1,3), pow(2,3), \dots pow(n,3)))$ $= O(n^4$

→ this is in part due to many things:

→ for one, the outer loop(i) runs from i = 1 to n, for each iteration, the condition (i % int(sqrt(n))) == 0 is verified as it relies on the value of i and n. furthermore, int(sqrt(n)) shows the int part of the sqrt of n. If i, its bitwise, % w/ the int part of the sqrt of n is zero then the inner-loop runs. On the other hand, it doesn't.

→ for two, the inner loop(k) runs for pow(i, 3) iterations. This exemplifies i raised to the power of 3. As a result, the # of iterations of the inner loop varies based on the value of i

→ the runs for pow(1, 3) play a hand in the worse case, but it occurs when the inner loop runs for the max # of iterations. for each iteration of the outer loop, the inner loop runs for pow(i, 3) times;

→ for finding the overall worst-case complexity, we review the max value of pow(i, 3) for all values of i from 1 to n

→ the time complexity is given by:

$O(n \cdot \max(pow(1,3), pow(2,3), \dots, pow(n,3)))$

the max value

when $i = n$ $\Rightarrow$ the max value of pow(i, 3) $\Rightarrow$ pow(n, 3)

$\rightarrow$ as ~~a~~ result, the complexity

is $O(n \cdot pow(n, 3)) \Rightarrow O(n^4) \Rightarrow$ bcuz pow(n, 3)

takes over for the linear term. The overall time complexity grows asymptocially as the 4th power of the input size n becomes the worst case

Question C

is below

Part (c)

```
for(int i=1; i <= n; i++){
  for(int k=1; k <= n; k++){
    if( A[k] == i){
      for(int m=1; m <= n; m=m+m){
        // do something that takes O(1) time
        // Assume the contents of the A[] array are not changed
      }
    }
  }
}
```

→ this code involves nested loops that iterate over n, n & log 2(n) times

→ to begin, the outer loop executes from i=1 to n resulting in a time complexity of O(n)

→ Next, we'll review the middle loop as it runs from k=1 to n, adding a factor of O(n) to the time complexity once again

→ lastly, for the innermost loop, it runs for m=1, m=m+m until m exceeds n. The loop doubles m in each iteration making the # of iterations decided by the # of times m can be double to meet or go over n. This is examplified through log base 2 of n ($\log_2(n)$)

→ Upon reviewing the conditions of each loop, specifically the inner loop of (if (A[k] == 1)) then we can consider this O(1) bcuz it is a constant - time operation

→ to bring this all together we can multiply the time complexities of each loop which is:

$O(n) * O(n) * O(\log_2(n)) * O(1)$

which results in $O(n^2 \log_2(n))$ => overall time complexity

Part (d)

Notice that this code is very similar to what will happen if you keep inserting into an ArrayList (e.g. `vector`). Notice that this is NOT an example of amortized analysis because you are only analyzing *1* call to the function `f()`. If you have discussed amortized analysis, realize that does NOT apply here since amortized analysis applies to *multiple* calls to a function. But you may use similar ideas/approaches as amortized analysis to analyze this runtime. If you have NOT discussed amortized analysis, simply ignore it's mention.

```
int f (int n)
{
  int *a = new int [10];
  int size = 10;
  for (int i = 0; i < n; i ++)
     {
        if (i == size)
          {
             int newsize = 3*size/2;
             int *b = new int [newsize];
             for (int j = 0; j < size; j ++) b[j] = a[j];
             delete [] a;
             a = b;
             size = newsize;
          }
        a[i] = i*i;
     }
}
```

→ In the code provided there is a singular loop that runs for n iterations. In each iteration, a conditional is present for the resizing of the array a. This takes O(size) time, where size is the current size of the array. Due to the array being resized when i achieves the current size of the array then that will result in the total time complexity being the sum of sizes from 10 to n; examplified as $O(10 + 11 + \ldots + n) \Rightarrow$ this being $O(n^2)$

→ this is the case do to many factors

→ for one, the array a is initialized w/ size 10

→ for two, the function executes a loop that runs n times starting at i=0 to n-1

→ w/ in the loop, the statement if (i == size) checks the conditions, including, when i reaches the current size of the array, in which case, a resizing operations occurs. from this, the array a is duplicated into a new array b w/ a new size as well. This new size stems from (3 * size / 2). In this case of the array a being duplicated, the original is then deleted, and the pointer a is redirected to the new array b; resulting in the size variable being modified to the new size

→ evaluating the function

→ there is zero resizing during the first iteration of the loop although the size is originally 10

→ on the second go round, if i goes to 10, there is one resizing operation starting at size 10 all the way to the new size

→ lastly, with the third roundabout, if i goes to 11, then there is another resizing done starting on of the new size to a new size and so on

→ therefore, the compiled time spent resizing results in the sum of sizes from 10 to n. bringing this to a sum examplified by:

$10 + 11 + 12 + \ldots + n$

→ furthermore, the consecutive int's sum can be compiled through the arithmetic series formula:

formula: $S = 2n \cdot (n+1) - 2m \cdot (m-1)$

implementation: $S = 2n \cdot (n+1) - 210 \cdot 11$

→ this results in the overall time complexity of the function being $O(n^2)$