Question 1)

Donya Akhavan CS104 - HW1 September 6th

Prof. Slocum

```
struct Node {
    int val;
    Node*  next;
};

Node* llrec(Node* in1, Node* in2)
{
    if(in1 == nullptr) {
        return in2;
    }
    else if(in2 == nullptr) {
        return in1;
    }
    else {
        in1->next = llrec(in2, in1->next);
        return in1;
    }
}
```

Question a: What linked list is returned if llrec is called with the input linked lists in1 = 1,2,3,4 and in2 = 5,6?

Question b: What linked list is return if llrec is called with the input linked lists in1 = nullptr and in2 = 2?

To show work, you can draw a call tree or box diagram of the function calls using some simplified substitution of your choice rather than pointer values (e.g. “p3” for a pointer to a node with value 3). Submit your answers as additional pages in your PDF file. Make sure to show your work and derivations supporting your final answer. The file will be the same as for Q1: **hw1_answers.pdf**. Upload this file to the **hw1** folder.

Question A).

in1 = 1,2,3,4,
in2 = 5,6

This function essentially says if list 1 is empty, return list 2, and if list 2 is empty, return list 1, else (if both lists have at least 1 node left), take the first node of list 1 (in1), make it the head, and call the function again, feeding it these parameters (in2 and in1->next).

Explaining the else segment in more detail: take one node from the first list, in1, then alternate with one node from the second list (in2). Keep alternating until one list runs out. Similar to braiding hair.

a) in1 → 1 → 2 → 3 → 4 , in2 = 5 → 6

llrec (1 , 5) sets 1→next = llrec(5,2)

llrec (5 ,2) sets 5→next = llrec (2,6)

llrec (2 , 6) sets 2→next = llrec(6,3)

llrec (6 ,3) sets 6→next = llrec (3,4)

llrec (3 , 4) sets 3→next = llrec(4, nullptr)

llrec (4, nullptr) returns 4

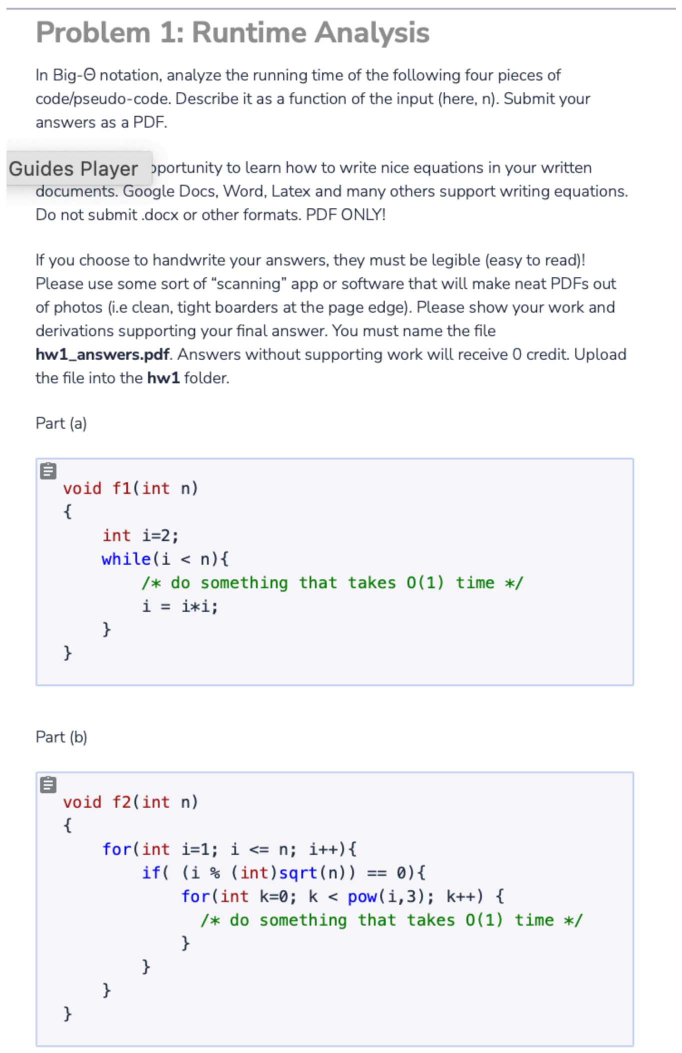
putting those all together, you get

1→5→2→6→3→4 ★

---

b) The first if(in1 == nullptr) gets triggered immediately, & it returns in2, which is simply:

2. ★



Problem 1: Runtime Analysis

In Big-Θ notation, analyze the running time of the following four pieces of code/pseudo-code. Describe it as a function of the input (here, n). Submit your answers as a PDF.

Guides Player pportunity to learn how to write nice equations in your written documents. Google Docs, Word, Latex and many others support writing equations. Do not submit .docx or other formats. PDF ONLY!

If you choose to handwrite your answers, they must be legible (easy to read)! Please use some sort of "scanning" app or software that will make neat PDFs out of photos (i.e clean, tight boarders at the page edge). Please show your work and derivations supporting your final answer. You must name the file **hw1_answers.pdf**. Answers without supporting work will receive 0 credit. Upload the file into the **hw1** folder.

Part (a)

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

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 */
            }
        }
    }
}
```

a) iteration 1 : $i = 2$

iteration 2 : $i = 2^2 = 4$

iteration 3 : $i = 4^2 = 16$

iteration 4 : $i = 16^2 = 256$

...

$i = 2^{(2^k)}$

loop terminates when $i \geq n$,
which means $2^{(2^k)} \geq n$.

★ algorithm squares itself in each iteration leading to very fast growth!

taking logarithms:

$2^k \geq \log_2 (n)$

$k \geq \log_2 (\log(n))$

time complexity: $\Theta(\log\log n)$

a) Math

let $i_0 = 2$; $i_{t+1} = (i_t)^2$. Define $j_t = \log^2 i_t$. Then

1) $j_{t+1} = \log^2 (i_{t+1}) = \log^2(i_t^2) = 2\log^2(i_t^2) = 2\log^2(i_t) = 2j_t \quad j_0 = 1$

Hence $j_t = 2^t$ and

$i_t = 2^{j_t} = 2^{2^t}$

Stopping time T: smallest t with $i_t \geq n$:

$2^{2^t} \geq n \iff 2^T \geq \log^2 n \iff T \geq \log^2 (\log^2 n)$

Conversely, if $t = \lfloor \log_2(\log_2 n) \rfloor - 1$ then $2^{2^t} < n$.

So,

$T = \lceil \log^2 (\log^2 n) \rceil = \Theta(\log\log n)$.

So, $\Theta(\log\log n)$

★ brother is a math major ☺

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 */
            }
        }
    }
}
```

b) • outer loop runs n times (i from 1 to n)

• inner loop executes only when i is divisible by $\sqrt{n}$

• values of i divisible by $\sqrt{n}$: $\sqrt{n}, 2\sqrt{n}, 3\sqrt{n} \ldots, n$

• numbers of such values : $n/\sqrt{n} = \sqrt{n}$

$\sum j = 1 \, (js)^3 = s^3 \sum j^3 = \Theta(n^{3/2}) \cdot \Theta(n^2) = \Theta(n^{7/2})$

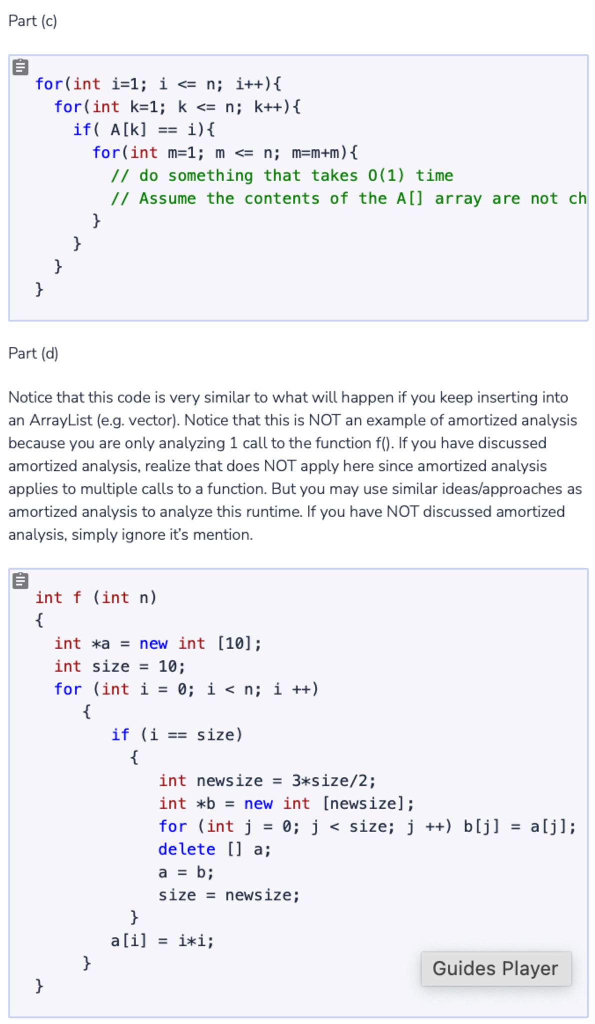
modulo checks add only $O(n)$

time complexity : $\Theta(n^{7/2})$

detailed math :

write $i = js$ with $j = 1 \ldots \lfloor n/s \rfloor = \Theta(\sqrt{n})$

$$\sum_{j=1}^{\Theta(\sqrt{n})} (js)^3 = s^3 \sum_{j=1}^{\Theta(\sqrt{n})} j^3 = \Theta(s^3 \cdot \sqrt{n})^4 = \Theta(n^{3/2} \cdot n^2) = \Theta(n^{7/2})$$



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 ch
      }
    }
  }
}
```

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;
     }
}
```

Guides Player

c) analysis:

- Outer loop runs n times, i from 1 to n
- middle loop runs n times for each i (k from 1 to n)
- inner loop runs n times only when $A[k] == i$.

Each array element $A[k]$ has exactly 1 value, so across all iterations of the outer loop ($i = 1$ to n) each element $A[k]$ will match exactly one value of i.

- condition $A[k]==i$ is checked n^2 times total (n values of $i \times n$ values of k)
- the condition is true exactly n times total (1x for each array element)
- each time the condition is true, the inner loop executes n times
- total inner loop executions: $n \times n = n^2$

for $i = 1..n$ for $k=1..n$ if $(A[k]=i)$ then for $m=1; m \leq n; m{\times}=2$) $\{O(1)\}$

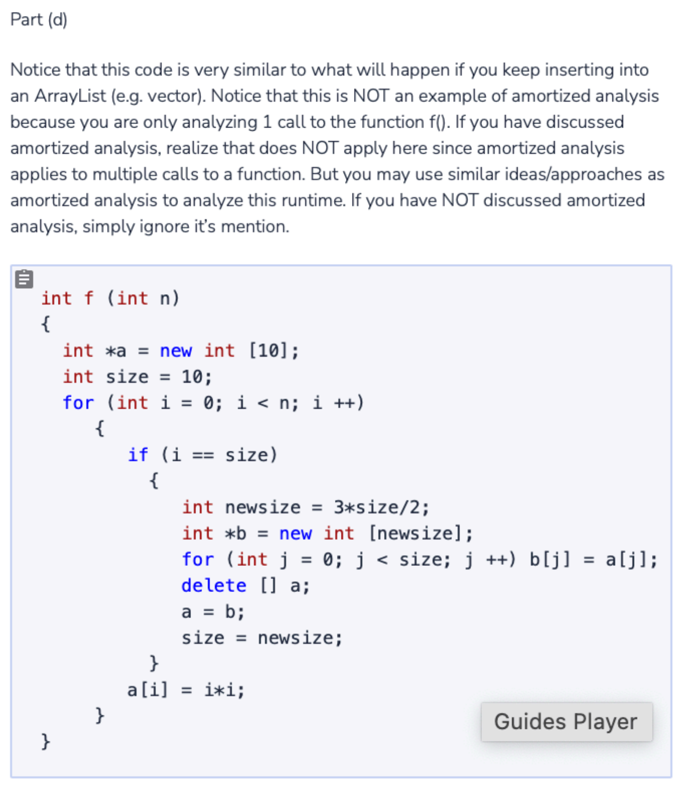
two outer loops always do $n^2 \Rightarrow \Theta(n^2)$

let $I_{i,k} = 1\{A[k]=i\}$ for each fixed k, at most one i matches, so

$\sum_{i=1}^{n} I_{i,k} \leq 1$ and $\sum^{ik} I_{i,k} \leq n$

Each time condition is true, doubling loop costs $\Theta(\log n)$, thus extra work $= O(n \log n)$.

$T(n) = \Theta(n^2) + O(n \log n) = \Theta(n^2)$


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;
     }
}
```

Appending n items takes $\Theta(n)$ time & uses $\Theta(n)$ space.

You only resize when array is full. When that happens, you make a new array 1.5x bigger & copy everything over. As array gets larger, resizes become rarer. You don't pay the copy cost every push, only once in awhile. The total copying over is proportional to n. Every push also does one normal write, which is another n operations total.

Copies + writes = constant × n = $\Theta(n)$

Let $C^0 = 10$ & $C^{t+1} = \lfloor \frac{3}{2} C^t \rfloor$

1. Resizes

$C_r \approx 10\left(\frac{3}{2}\right)^r \geq n \Rightarrow r \approx \left\lceil \log_{\frac{3}{2}} \frac{n}{10} \right\rceil = O(\log n)$

2. Copy across resizing

$$\sum_{t=0}^{r-1} C_t \leq 10 \sum_{t=0}^{r-1} \left(\frac{3}{2}\right)^t = 10 \cdot \frac{\left(\frac{3}{2}\right)^r - 1}{\frac{3}{2} - 1} = 20\left(\left(\frac{3}{2}\right)^r - 1\right) \leq 20\left(\frac{n}{10}\right) = 2n = \Theta(n)$$

3. Regular writes (one per push)

$W_{write} = n$

4. Time & avg cost

$T(n) = W_{write} + W_{copy} + O(r) = n + O(n) + O(\log n) = \Theta(n)$

Amortized per push $= \frac{T(n)}{n} = O(1)$

5. Space

Peak cap $\propto C_r \sim \Theta(n) \rightarrow$ space $= \Theta(n)$

time complexity : $\Theta(n)$