Problem 3: Simple vector Push Analysis.

Part 1: Mathematical Analysis of Inefficient Push:

```
T * naptr;
try {
    naptr = new T [arraysize +1];
  } catch( bad_alloc) {
    memError();
  }
  for( count = 0, count < arraysize, count ++) {
      naptr[count] = aptr[count];
  }
  naptr[arraySize ++] = val;
  delete [] aptr;
  aptr = 0;
  aptr = naptr;
```

O → represents operations, which equate to clock cycles T(O).

O_B - operations before loop.

O_i - operations during loop.

O_A - operations after loop.

Then:

$$O_B + \sum_{i=0}^{n-1} O_i + O_A. \quad (n \to \text{arraysize})$$

$* \sum_{i=m}^{n} 1 = n-m+1.$ So, $\sum_{i=0}^{n-1} 1 = n-1-0+1 = n.$

$\Rightarrow O_B + (n) O_i + O_A.$

$f(n) =$ 1st order polynomial.

$O_i n + (O_B + O_A) = c' n + c,$

where $c' = O_i$ and $c = O_B + O_A$.

∴ Inefficient Push = $O(n)$ for all $n > 0$,

where $c' > O_i$.

Part 2: Mathematical Analysis of Linked List Push:

```
aptr -> addLst (val);
addLst (val):
    Link * end;
    temp = front;
    do {
        end = temp;
        temp = temp -> linkPtr;
    } while ( temp != NULL);
    Link * add = new Link;
    add -> data = val;
    add -> linkPtr = NULL;
    end -> linkPtr = add.
```

O_B - operations before loop.
O_i - operations during loop.
O_A - operations after loop.

Then: $O_B + \sum_i O_i + O_A$.

Note: To reach the end of our list, the do-while loops exactly n times for a list with n elements.

$\Rightarrow$ $O_B + n(O_i) + O_A = O_i n + (O_B + O_A) = c'n + c$,
where $c' = O_i$ and $c = O_B + O_A$.

$f(n)$ = 1st order polynomial.

$\therefore$ Linked List Push $= O(n)$ for all $n > 0$,
where $c' > O_i$.

⇒ Comparing the Inefficient and Linked List Push:
The inefficient push has more operations that take place before the loop than the linked list push. In the loops, the inefficient push copies values, while the linked list push merely traverses the list. Although they are both O(n), the linked list push is more efficient. It takes less operations, on average, to push the same number of elements.

Part 3: Mathematical Analysis of Efficient Push:

```
if(n == nMax){
    nMax *= 2;
    T *naptr;
    try{
        naptr = new T[nMax];
    }catch(bad_alloc){
        memError();
    }
    for(i=0; i<n; i++){
        naptr[i] = aptr[i];
    }
    naptr[n++] = val;
    delete []aptr;
    aptr = 0;
    aptr = naptr;
}else {
    aptr[n++] = val;
}
```

* Here we have 2 cases:

case 1: $n == nMax$.

O_B - operations before loop.

O_i - operations during loop.

O_A - operations after loop.

Then: $O_B + \sum_{i=0}^{n-1} O_i + O_A$

$= O_B + n\,O_i + O_A = O_i n + (O_B + O_A)$.

$f(n) =$ 1st order polynomial.

$= O_i n + (O_B + O_A)$

$= c' n + c$.

case 2: $n \,!= nMax$.

O_{push} - operations to push a value.

Then: O_{push} is the only operation.

$f(n) =$ constant function.

$= O_{push} = c$.

Considering both cases, there is a probability p tied to case 1. we only execute case 1 if $n == nMax$. As $n \to \infty$, $nMax$ grows exponentially by a factor of 2^k, where k is the number of times case 1 is executed. Given that $nMax$ grows exponentially, the probability p of case 1 executing decays exponentially. It becomes less and less likely for case 1 to execute. As a result, case 2 executes more and more often. As $n \to \infty$, case 2 will execute more and more, and the probability of case 1 executing becomes insignificant, to the extent that it wipes out our n-term in $c'n + c$.

Case 2 executes in constant time. Given that case 2 executes more often as n gets larger, our push function executes in constant time.

$$f(n) = O_{push} = c.$$

$\therefore$ Efficient push $= O(1)$ for all $n > 0$,

where $c > O_{push}$.

Efficient Push $= O(1)$

Inefficient Push $= O(n)$

Linked list Push $= O(n)$.

Efficiency Rankings:

1. Efficient Push,
2. Linked list Push,
3. Inefficient Push,

* Note: I've explained in 1 of the previous pages why the linked list push is more efficient than the inefficient push.

* We concluded that the efficient push was most efficient because its operations were shown to take place in constant time as $n \to \infty$.