Cynthia Lee

Section 4 (Week 5) - SOLUTION

1. Backtracking -- partitionable.

```
bool partitionable(Vector<int>& list) {
    return helper(list, 0, 0);
}
bool helper(Vector<int>& rest, int sum1, int sum2) {
    if (rest.isEmpty()) {
        return sum1 == sum2;
    } else {
        int n = rest[0];
        rest.remove(0);
        bool answer = helper(rest, sum1 + n, sum2) ||
                      helper(rest, sum1, sum2 + n);
        rest.insert(0, n);
        return answer;
    }
}
```

2. Big-O Notation.

i. The function has complexity O(*n*). To see this, note that the inner loop runs exactly *n* times, each doing a constant amount of work. Therefore, the overall complexity is O(*n*). This means that there is no dependence on *m*.

ii. Since n has doubled from 200 to 400 and the time complexity is O(*n*), the new runtime should be about twice the runtime as before, so it should take about 2μs.

We can't give an exact value for the runtime because big-O notation ignores lower-order growth terms. These other terms can contribute to the runtime as well for small values of *n*, and might influence the overall runtime.

iii. The runtime is O(*n*). To see this, note that

- `raiseToPower(m, n)` does O(1) work, then calls `raiseToPower(m, n – 1)`.
- `raiseToPower(m, n – 1)` does O(1) work, then calls `raiseToPower(m, n – 2)`
- ...
- `raiseToPower(m, 1)` does O(1) work, then calls `raiseToPower(m, 0)`.
- `raiseToPower(m, 0)` does O(1) work.

This means that there are a total of *n* + 1 calls, each of which does O(1) work. Therefore, the total work done is O(*n*).

iv. As before, the runtime will be around 2μs.

v. The time complexity is O(log n). Note that at each level of the recurrence, n's value goes down by a factor of two. This means that the maximum number of recursive calls can be at most O(log n), since at that point n will have shrunk down to 0 (since we always round down). Each level does only O(1) work, so the total runtime is O(log n).

vi. Note that log 10000 = log 100^2 = 2 log 100. Therefore, we would expect the second call to raiseToPower to take about twice as long as before, giving a runtime of 2μs.

vii. Notice that this function makes two recursive calls at each level. This means that

- There is one recursive call with n at its initial value.
- There are two recursive calls with n around n / 2.
- There are four recursive calls with n around n / 4.
- There are eight recursive calls with n around n / 8.
- ...
- There are 2^k recursive calls with n around $n / 2^k$.

Eventually, this process stops when $k > \log_2 n$. When that happens, the bottom layer will have a total of around n total recursive calls (since $2^k > 2^{\log n} = n$). Each recursive call does a total of O(1) work, so the total amount of work done is equal to the total number of recursive calls, which is

$$1 + 2 + 4 + 8 + \ldots + 2^{\log n}$$

This is the sum of a geometric series. It turns out that this is equal to

$$2^{1 + \log n} - 1 = 2 \cdot 2^{\log n} - 1 = 2n - 1$$

So the total runtime is O(n).

3. Constructors and Destructors.

The ordering is as follows:

- A constructor is called when `elem` is declared in `main`.
- A constructor is then called to set `toPrint` equal to a copy of `elem`.
- A constructor is then called to initialize the `temp` variable in `printStack`.
- When `printStack` exits, a destructor is called to clean up the `temp` variable.
- Also when `printStack` exits, a destructor is called to clean up the `toPrint` variable.
- When `main` exits, a destructor is called to clean up the `elem` variable.