CS106B
Spring 2016 Cynthia Lee

Section 4 (Week 5) Handout

Section problems by Keith Schwarz and Marty Stepp, with edits by Cynthia Lee

1. Backtracking -- partitionable.
Write a function named `partitionable` that takes a vector of `int`s and returns true if it is possible to divide the `int`s into two groups such that each group has the same sum. For example, the vector {1,1,2,3,5} can be split into {1,5} and {1,2,3}. However, the vector {1,4,5,6} can't be split into two.

```
bool partitionable(Vector<int>& nums) { ...
```

2. Big-O Notation.
Below is a simple function that computes the value of m^n when n is a nonnegative integer:

```
int raiseToPower(int m, int n) {
    int result = 1;
    for (int i = 0; i < n; i++) {
        result *= m;
    }
    return result;
}
```

i. What is the big-O complexity of the above function, written in terms of m and n? You can assume that it takes the same amount of time to multiply together any two numbers.

ii. If it takes 1μs to compute `raiseToPower(100, 200)`, about how long will it take to compute `raiseToPower(50, 400)`?

Below is a recursive function that computes the value of m^n when n is a nonnegative integer:

```
int raiseToPower(int m, int n) {
    if (n == 0) return 1;
    return m * raiseToPower(m, n – 1);
}
```

iii. What is the big-O complexity of the above function, written in terms of m and n? You can assume that it takes the same amount of time to multiply together any two numbers.

iv. If it takes 1µs to compute `raiseToPower(100, 200)`, about how long will it take to compute `raiseToPower(50, 400)`? Why can't you give an exact value for the runtime?

It turns out that there is a much faster way to compute m^n when n is a nonnegative integer. The idea is to modify the recursive step as follows:

- If n is an even number, then we can write $n = 2k$. Then $m^n = m^{2k} = (m^k)^2$
- If n is an odd number, then we can write $n = 2k + 1$. Then
  $m^n = m^{2k+1} = m*(m^{2k}) = m*(m^k)^2$

Based on this observation, we can write this recursive function:

```
int raiseToPower(int m, int n) {
    if (n == 0) return 1;
    if (n % 2 == 0) {
        int z = raiseToPower(m, n / 2);
        return z * z;
    } else {
        int z = raiseToPower(m, n / 2);
        return m * z * z;
    }
}
```

v. What is the big-O complexity of the above function, written in terms of m and n? You can assume that it takes the same amount of time to multiply together any two numbers.

vi. If it takes 1µs to compute `raiseToPower(100, 100)`, about how long will it take to compute `raiseToPower(50, 10000)`?

vii. *(Challenge problem)* What happens to the big-O time complexity if you rewrite the function in the following way?

```
int raiseToPower(int m, int n) {
    if (n == 0) return 1;
    if (n % 2 == 0) {
        return raiseToPower(m, n / 2) * raiseToPower(m, n / 2);
    } else {
        return m * raiseToPower(m, n / 2)
                 * raiseToPower(m, n / 2);
    }
}
```

3. Constructors and Destructors.

Constructors and destructors are unusual functions in that they're called automatically in many contexts and usually aren't written explicitly. To help build an intuition for when constructors and destructors are called, trace through the execution of this program and list all times when a constructor or destructor is called.

```
/* Prints the elements of a stack from the bottom
 * of the stack up to the top of the stack.  To
 * do this, we transfer the elements from the
 * stack to a second stack (reversing the order
 * of the elements), then print out the contents
 * of that stack.
 */
void printStack(Stack<int> toPrint) {
    Stack<int> temp;
    while (!toPrint.isEmpty())
        temp.push(toPrint.pop());

    while (!temp.isEmpty())
        cout << temp.pop() << endl;
}

int main() {
    Stack<int> elems;
    for (int i = 0; i < 10; i++)
        elems.push(i);

    printStack(elems);
}
```