Robert Max Williams

Assignment 1, September 11, 2017

1. Write a recursive function that returns the number of 1 in the binary representation of N. Use the fact that this is equal to the number of 1 in the representation of N/2, plus 1, if N is odd. [1]

```
#include <iostream>
using namespace std;

int count_ones(int x){
    if (x == 0)
        return 0;
    return (x%2) + count_ones(x/2);
}
int main(){
    cout << count_ones(7) << endl;
    return 0;
}
```

The code was very simple and the answer was very elegant. It runs in log(n) time, since doubling the size of the input adds a constant amount to the computation time. It is the same process as I use to convert binary to decimal by hand, so I doubt there are any easy shortcuts.

2. Write the routines with the following declarations:

```
void permute( const string & str );
void permute( const string & str, int low, int high );
```

The first routine is a driver that calls the second and prints all the permutations of the characters in string `str`. If `str` is `"abc"`, then the strings that are output are `abc`, `acb`, `bac`, `bca`, `cab`, and `cba`. Use recursion for the second routine. [2]

How many permutations for a string of length *l*?

```cpp
#include <iostream>
using namespace std;

int permutations = 0;
int calls = 0;
int steps = 0;

void permute(const string &str, int high, int low){
    calls++;

    if (low == high-1)
    {
        cout << str << endl;
        permutations++;
        return;
    }

    for (int i = 0; i < high-low; i++)
    {
        steps++;

        string front = str.substr(0, low);
        string letter = str.substr(i+low, 1);
        string before_letter = str.substr(low, i);
        string after_letter = str.substr(low+i+1);

        permute(front + letter + before_letter + after_letter, high, low+1);

    }

}


void permute(const string &str){
    permute(str, str.length(), 0);
}

int main(){
    permute("abcd");
    cout << "permutations: " << permutations << endl;
    cout << "calls: " << calls << endl;
    cout << "steps: " << steps << endl;
}
```

I kept creating off by one errors, but once I had those sorted out the algorithm is a very good use case for recursive programming. A pure functional version would also be interesting to see, but would likely require more memory.
There are factorial(n) combinations, but the algorithm time is worse than that, since only the final call on each recursive branch finds a unique factorial.
Values for different input sizes are listed below:

Length	1	2	3	4	5	6	7	8
permutations	1	2	6	24	120	720	5040	40320
Function calls	1	3	10	41	206	1237	8660	69281
Ratio	1	1.5	1.6	1.708	1.716	1.718	1.718	1.7182787

*Iterative steps is in the code, but is always one less than function calls, so it is ignored.

The ratio appears to approach a constant, so running time is in fact O(n) =n!

Wow! If only I knew why.

It actually looks eerily familiar. Euler's number minus one is :
e-1 = 2.718281828459045...
Function calls / permutations = 1.7182818...

HMMM….
Uh.
I really want to know why this is like that. I am speechless. I'll come talk to you in class and see if a reason can be determined.

3. Evaluate the following sums:

(a) $\sum_{i=0}^{\infty} \frac{1}{4^i}$

(b) $\sum_{i=0}^{\infty} \frac{i}{4^i}$

This problem totally stumped me. The first one can be solved by a formula we learned in calculus, and equals 4/3.
The second one has no clear solution or simplification. Michael Nester, a peer of mine studying the maths, created a paper showing how he solved it:
https://drive.google.com/open?id=0B7OpIk-vR4yRa3BNS00zRmNyWUZnMmlYNk1tRTUzRVBZYzVv
He used a computer algebra system to generate an equation for partial sums, then verified it by induction and took the limit as i approaches infinity. Doing any of those things inside a computer is far beyond the scope of this class, so my code just prints out the answer with a small explanation. I hope this is what you wanted.

```
#include <iostream>
using namespace std;

int main()
{
    cout << "Sum of 1/4^i from i=0 to inf is 1/(1-1/4) by the sum formula. 1/(1-1/4) = " << 1/(1-0.25) << endl;
    cout << "The sum of i/4^i is vastly more complex to solve. lim as i-> inf of (4^(n+1)-3n-4)/(9*4^n) = " << 9.0/5 << endl;

}
```

4. Write a template [3] class `Collection`, that stores a collection of Objects (in an array), along with the current size of the collection. Provide public functions `isEmpty`, `makeEmpty`, `insert`, `remove`, and `contains`. contains(x) returns true if and only if an Object that is equal to x is present in the collection.[4]

```
template <typename Object>
class Collection
{

    public:

        Collection(){
            makeEmpty();
        }

        Object at(int i){
            if (i >= size){
                throw invalid_argument("out of range");
            }
            else if (i < 0){
                throw invalid_argument("negative index");
            }
            else
                return collection[i];
        }


        bool isEmpty(){
            return (size == 0);
        }

        void makeEmpty(){
            size = 0;
            collection = new Object[size];
        }

        string tostring(){
            stringstream buf;
            buf << "[";

            for (int i = 0; i < size; i++){
                buf << collection[i];

                if (i != size-1)
                    buf << ", ";
            }
            buf << "]";

            return buf.str();
        }

        void append(Object x){
           insert(x, size);
        }
```

```
        void append(Object x){
           insert(x, size);
        }

        void insert(Object x, int i){
            Object * temp;
            temp = new Object[size+1];
            memcpy(temp, collection, i*sizeof(Object));
            memcpy(temp+i+1, collection+i, (size-i)*sizeof(Object));
            temp[i] = x;
            delete [] collection;
            collection = temp;
            size += 1;
        }

        void remove(int i){
            Object * temp;
            temp = new Object[size-1];
            memcpy(temp, collection, i*sizeof(Object));
            memcpy(temp+i, collection+i+1, (size-i-1)*sizeof(Object));
            delete [] collection;
            collection = temp;
            size -= 1;
        }

        bool contains(Object x){
            for (int i = 0; i < size; i++){
                if (collection[i] == x)
                    return true;
            }
            return false;
        }

    private:
        Object* collection;
        int size;

};
```

This does everything it needs to, and more!

The pointer arithmetic is very messy but no constants exceed one, so it passes by me. Helper functions for splicing arrays would be nice, but are really not needed here.

Testing is done further down in the code.

```
:!g++ -o a.out problem4_collection.cpp && ./a.out
[a]
[a, b]
[a, b, c]
[a, x, b, c]
[a, b, c]
foo.contains('b') = 1
foo.contains('x') = 0

[10]
[10, 20]
[10, 20, 30]
[10, 99, 20, 30]
[10, 20, 30]
bar.contains(20) = 1
bar.contains(99) = 0
bar.isEmpty() = 0
```