CS132 Summer Work

Lydia Shepherd

August 17, 2020

1 Powerset Generation

1.a

The powerset of a set, is the set of all subsets of that set. For example:

$$\mathcal{P}(\{a,b\}) = \{\{\},\{a\},\{b\},\{a,b\}\}$$

The number of elements in the powerset is 2^n where n is the number of elements in the original set. I wrote the following code to generate the powerset of a given set:

```c
/**
 *Function that prints the Power Set of a given set to the
   terminal
 *Parameter: A pointer to the set to find the power set of
 *Parameter: The size of the set passed in
*/
void printPowerSet(char *set, int size){
    int PSet_size = pow(2,size);
    int counter, i;
    for (counter = 0; counter < PSet_size; counter++){
        printf("{");
        for (i = 0; i < size; i++){
            if (counter & (1<<i)){
                printf("%c", set[i]);
            }
        }
        printf("}");
        printf("\n");
    }
}
```

Listing 1: Powerset generation function

This function takes in a pointer to the set and the size of the set that it will print the powerset of. First it calculates the size of the powerset as mentioned above. Then it uses nested for-loops, the outer loop using the powerset size and the inner set using the original set size. It then uses a left shift (<<) to determine whether it should print out one of the elements in the original set. It will only print the corresponding element (ith element) if `counter` and i shifted to the left create a number greater than 0 when the binary AND operator is used (&). This means that when `counter` is 0, it will always print the empty set which is the subset of every set. This ensures that each combination of the elements is printed while avoiding duplicates.

1.b

The asymptotic running time of this algorithm is $n \cdot 2^n$ due to the nested loops one of which runs 2^n times, while the other runs n times.

1.c

To measure the runtime performance of this program, I used the `time.h` library. More specifically, I used the `gettimeofday` function which gets the system's clock time at that moment and records it in a struct. By getting the time before and after the execution of my program, I could find the time in seconds that it took to run.

```c
/**
 * Main function that specifies the set and makes the call to
    print the power set
 * TODO: allow the user to enter the power set themselves
*/
int main(){
    struct timeval start, end;
    gettimeofday(&start, NULL);
    char set[] = {'a','b','c'};
    printPowerSet(set,3);
    gettimeofday(&end, NULL);
    double time_taken = end.tv_sec + end.tv_usec / 1e6 -
   start.tv_sec - start.tv_usec / 1e6; //time in seconds
    printf("time program took %f seconds to execute\n",
   time_taken);
    return 0;
}
```

Listing 2: Main function including the timing data

1.d

Using the timing method described above, I collected 10 points of timing data, one for each size of set from 1 to 10. I used this data to plot a graph of the results:

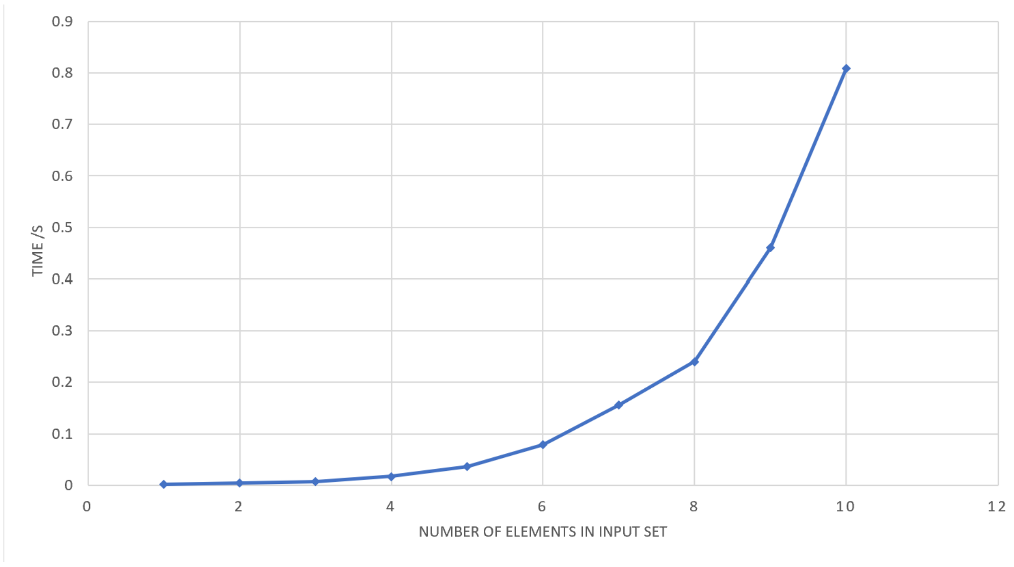

Figure 1: Graph of time against number of elements in the input set.

2 Matched Integers

2.a

The problem was to find the smallest positive integer, x such that $2x$, $3x$, $4x$ and $5x$ contain exactly the same digits. To solve this, I made a function that checks whether two numbers are permutations of each other and then used it to check each multiple of a number. If they were not permutations then the rest of the multiples could be skipped. I applied this to all numbers counting up from 1.
To test if two numbers were permutations, I made an array of 10 integers set to 0 and then went through the first number incrementing the corresponding elements in the array. The I did the same for the second number but decrementing the values. If the final state of the array was all 0s then the two numbers were permutations.

```
/**
 * Function that tests if two given numbers are permutations
   of each other
 * Parameter: long the two numbers to be tested
 * Returns 1 if they are permutations and 0 if not.
*/
int isPerm(long num1, long num2) {
    int digits[10];
    int i;
    for (i = 0; i < 10; i++)       // Init all counts to zero.
        digits[i] = 0;
    while (num1 != 0) {            // Process all digits.
        digits[num1%10]++;         // Increment for least significant digit.
        num1 /= 10;                // Get next digit in sequence.
    }
    while (num2 != 0) {            // Same for num2 except decrement.
        digits[num2%10]--;
        num2 /= 10;
    }
    for (i = 0; i < 10; i++)
        if (digits[i] != 0)        // Any count different, not a permutation.
            return 0;
    return 1;                      // All count identical, was a permutation.
}
```

Listing 3: Code for testing permutations

When testing the multiples, I used a for-loop to test each multiplier. Then I passed it every number up from 1 so that it would stop when it reached the smallest integer that matched the criteria.

```
/**
 * Function that tests each multiple of a number (from 5x to 2x) to see if they are permutations of each other
 * Parameter: the number to test
 * Returns 1 if all multiples are permutations or 0 if not
*/
int isMatched(long number){
    int i;
    int result = 0;
    for (i=5; i>1;i--){                //loop through the multiples to test (5..2)
        long temp = number*i;
        if (!isPerm(number,temp)){   //if it is not a permutation set, return 0
            return 0;
        }

    }
    return 1;                          //otherwise, return 1 since all multiples are permutations
}

```

```c
/**
 * Main function that tests each number in turn until it
   finds the first match then prints it.
*/
int main(){
    long i =1;
    while (!isMatched(i)){
        i++;
    }
    printf("The First Number to contain the same digits is: %
   ld", i);
    return 0;
}
```

Listing 4: Code for testing each integer multiples

3 Pointer Check

3.a

```c
//Function f
int * f (void)
{
int x = 10;
return (&x);
}

//Function g
int * g (void)
{
int * py;
*py = 10;
return py;
}

//Function h
int * h (void)
{
int *pz;
pz = (int *) malloc (sizeof(int));
*pz = 10;
return pz;
}
```

Listing 5: Example pointer functions with dangerous usage

Two of the above functions include dangerous usage of pointers. Functions *f* and *g* are dangerous since they do not allocate memory for the pointer they

are going to return. By using `malloc`, you allocate space on the heap for the pointer and variable so that it cannot be overwritten by another function and has its own valid memory address.
Function *g* could cause a SEG FAULT because the pointer *py* will either try to write 10 to an invalid memory address or corrupt whichever memory address it is mapped to (which will be completely random).
Function *f* allocates a local pointer by returning the address of a local variable in that function. This local pointer will be overwritten on the stack when another function call uses that memory and so it will only work if it is used immediately after the function is called.
The best practice fix for these issues is to allocate memory on the heap as shown in function *h*. Another way to achieve the results of function *f* in a safer way is to use a static variable so that it is not stored on the stack.

Listings