HOMEWORK-2
Total Points: 85

1. [10 points] Write an implementation of the `mul2` routine that is used in the matrix multiplication based solution for Fibonacci.

```
void mul2(int A[2][2], int B[2][2])
{
  int C[2][2];
  C[0][0] = (A[0][0] * B[0][0]) +  (A[0][1] * B[1][0]);
  C[0][1] = (A[0][0] * B[0][1]) +  (A[0][1] * B[1][1]);
  C[1][0] = (A[1][0] * B[0][0]) +  (A[1][1] * B[1][0]);
  C[1][1] = (A[1][0] * B[0][1]) +  (A[1][1] * B[1][1]);
  A[0][0] = C[0][0];
  A[0][1] = C[0][1];
  A[1][0] = C[1][0];
  A[1][1] = C[1][1];
}
```

**Up to 4 points for partially correct answers.**

2. [15 points] Modify the four algorithms for Fibonacci numbers discussed in class (see below) to compute the “nth Fibonacci number % 10000” instead of the “nth Fibonacci number”.

   - Algorithm-1: `fib(n) = fib(n-1) + fib(n-2)`
   - Algorithm-2: `fib(n)` returns `(fib(n), fib(n-1))`
   - Algorithm-3: The iterative solution
   - Algorithm-4: using matrix multiplication

   Run all these four algorithms for various numbers of n, and answer the following questions.

   - What is the runtime of Algorithm-1 for n = 44?
   - What are the runtimes and the return values of `fib(n)` (i.e., the “nth Fibonacci number % 10000”) for the other three algorithms when n is 1 lakh, 5 lakhs, 1 million, 100 million, 1000 million, and 2000 million?

| Problem Size | Return Values of all Algorithms | Algorithm-1 Runtime | Algorithm-2 Runtime | Algorithm-3 Runtime | Algorithm-4 Runtime |
|---|---|---|---|---|---|
| 44 | | 3000 ms | | | |
| 1 Lakh | 6875 | | 6 ms | 4 ms | 2 ms |
| 5 Lakhs | 3125 | | 9 ms | 6 ms | 2 ms |
| 1 Million | 6875 | | SEGFAULT | 7 ms | 2 ms |
| 100 Million | 6875 | | SEGFAULT | 227 ms | 2 ms |
| 1000 Million | 6875 | | SEGFAULT | 2229 ms | 2 ms |
| 2000 Million | 3125 | | SEGFAULT | 4457 ms | 2 ms |

**Deduct 2 points if the runtime of Algorithm-1 for n= 44 is less than 1000 ms.**
**Deduct 4 points if any return value is incorrect.**
**Deduct 4 points if they didn't encounter a segmentation fault for the 2000 Million case using Algorithm-2.**
**Deduct 4 points if the time taken by Algorithm-3 for 2000 Million is not around twice the time needed for 1000 Million.**
**Deduct 4 points if the time taken by Algorithm-4 for 2000 Million is not around the same as the time for 1000 Million.**
**Total points can't be negative after all deductions. Make it zero in that case.**

You don't need to submit your modified algorithm. You can use an implementation similar to the following one to print the runtime of the `fib` routine.

```
#include <stdio.h>
#include <sys/time.h>

int fib(int n) {
   if (n == 0 || n == 1)
      return n;
   return (fib(n-1) + fib(n-2)) % 10000;
}

int main() {
  struct timeval start;
  struct timeval end;
  unsigned long t;
  int r;
```

```
  gettimeofday(&start, 0);
  r = fib(44);
  gettimeofday(&end, 0);

  t = ((end.tv_sec * 1000000) + end.tv_usec) -
      ((start.tv_sec * 1000000) + start.tv_usec);
  printf("r:%d\n", r);
  printf("elapsed time: %lf milliseconds\n", t/1000.0);
  return 0;
}
```

3. [25 points] Consider the following caching algorithm for Fibonacci numbers, as discussed in class.

```
int cache[1000] = {0};
int num_calls = 0;

int fib(int n) {
  num_calls++;
  if (cache[n] != 0)
    return cache[n];
  if (n == 0 || n == 1)
    return n;
  int r = (fib(n-1) + fib(n-2)) % 10000;
  cache[n] = r;
  return 0;
}
```

This algorithm works only when the value of n is less than 1000. In this algorithm, we save the return values of all Fibonacci numbers between 2 and n. Therefore, if we use a larger value of n, we might need a lot of memory for the cache. But as we know that at a given point, only the value of `fib(n-2)` is needed to avoid the recomputation. So we can achieve similar performance with a cache of size one that stores the previously computed value corresponding to `fib(n-2)`. The skeleton of the proposed algorithm is shown below. In this code, the implementations of BLOCK-1 and BLOCK-2 are missing. Write the implementation of BLOCK-1 and BLOCK-2 in such a way that the modified algorithm will make the same number of recursive calls for n = 900 as it will make for the above algorithm with the cache size 1000. The cache_entry contains two fields, `"key"` and `"val"`. `"key"` contains an integer value, and `"val"` contains the “Fibonacci number % 10000” corresponding to `"key"`.

```
struct cache_entry {
  int key;
  int val;
};

int num_calls = 0;

struct cache_entry cache[1];

int fib(int n) {
  num_calls++;

  /* BLOCK-1: add some code here */

  if (n == 0 || n == 1)
    return n;
  int r1 = fib(n-1);
  int r2 = fib(n-2);
  int r = (r1 + r2) % 10000;

  /* BLOCK-2: add some code here. */
  return r;
}
```

- Provide the implementations of BLOCK-1 and BLOCK-2.
- What is the runtime of your modified algorithm when n is 1 lakhs, 5 lakhs, and 1 million?

```
BLOCK-1
if (cache[0].key == n) {
  return cache[0].val;
}
BLOCK-2
cache[0].key = n-1;
cache[0].val = r1;
```

**Give zero if they are not storing the value of r1 in the cache at BLOCK-2. Deduct 10 points if they are not checking and returning a cached value in BLOCK-1.**

**The runtime for 1 lakh should be similar to the result for algorithm-2 in question 2. However, a segmentation fault is also a valid solution. Deduct 5 points if they didn't report any runtime or segmentation faults, and total points >= 5.**

4. [10 points] Write an implementation of the `cmp_string` routine that takes two character strings (in this case, a character string is a word in an English dictionary) as input and returns 0, -1, or 1 when the first argument is equal to, less than, or greater than the second argument, respectively. Notice that a character string is a sequence of chars in a char array that terminates with '\0'. Your implementation should ignore the case (i.e., dog and DoG are the same strings). You are not allowed to use any library function. The prototype of the cmp_string is the following:

```
int cmp_string(char str1[], char str2[]);
```

```
char get_uppercase(char c) {
   if (c >= 'a' && c <= 'z')
     return c - ('a' - 'A');
   else
     return c;
}

int cmp_string(char str1[], char str2[]) {
   int i = 0;
   while (get_uppercase(str1[i]) == get_uppercase(str2[i])
         && str1[i] != '\0') {
     i++;
   }
   int diff = get_uppercase(str1[i]) - get_uppercase(str2[i]);
   if (diff == 0)
      return 0;
   else if (diff > 0)
      return 1;
   else
     return -1;
}
```

**The valid answers may be slightly different too. Give max 4 points if the answer is partially correct.**

5. [25 points] Extend the Towers of Hanoi problem discussed in the class to use four towers instead of three towers. In your modified solution, for all n > 2, the number of moves must be less than the number of moves needed for three towers. Let's say the prototype of the new move function is:

```
void move(int n, char src_t[], char dst[], char tmp1[], char tmp2[]);
```

The goal is to move n discs from the src to dst using tmp1 and tmp2 as temporaries.

- Write your extended algorithm.

  **[10 Points for the correct answer to this part]**

```
void move(int n, char src_t[], char dst[], char tmp1[], char tmp2[]) {
  // deduct five points if two base cases either (n == 1 and n == 2) or (n == 0 and n == 1) are not handled
  if (n == 1) {
     printf("moving from %s to %s\n", src_t, dst);
     return;
  }
  if (n == 2) {
      printf("moving from %s to %s\n", src_t, tmp2);
      printf("moving from %s to %s\n", src_t, dst);
      printf("moving from %s to %s\n", tmp2, dst);
      return;
  }
  // Give zero if the first argument is not n-2 or the function call below  is missing
  move(n-2, src_t, tmp1, dst, tmp2);

  // deduct five points if three print statements are not present in the solution
  printf("moving from %s to %s\n", src_t, tmp2);
  printf("moving from %s to %s\n", src_t, dst);
  printf("moving from %s to %s\n", tmp2, dst);

  // Give zero if the first argument is not n-2 or the function call below is missing
  move(n-2, tmp1, dst, src_t, tmp2);
}
```

- Write the sequence of function calls and their arguments that will take place when we invoke move in your extended implementation as `move`(4, “T1”, “T4”, “T2”, “T3”).

  **[2 points for the correct answer to this part.]**

  **Give zero in this part if the number of calls is not three.**
  n:4 src:T1 dst:T4 tmp1:T2 tmp2:T3
  n:2 src:T1 dst:T2 tmp1:T4 tmp2:T3
  n:2 src:T2 dst:T4 tmp1:T1 tmp2:T3

- What is the output of your algorithm when we invoke move in your extended implementation as `move`(4, “T1”, “T4”, “T2”, “T3”)?

  **[2 points for the correct answer to this part.]**

  **Give zero in this part if the number of moves is not 9.**

  moving from T1 to T3
  moving from T1 to T2
  moving from T3 to T2
  moving from T1 to T3
  moving from T1 to T4
  moving from T3 to T4
  moving from T2 to T3
  moving from T2 to T4
  moving from T3 to T4

- Write the sequence of function calls and their arguments that will take place when we invoke the move routine in the solution for three towers as `move`(4, “T1”, “T3”, “T2”).

  **[10 points for the correct answer to this part.]**
  **Give zero in this part if the answer is not exactly matching with the following sequence of function calls and the corresponding arguments.**

  n:4 src:T1 dst:T3 tmp:T2
  n:3 src:T1 dst:T2 tmp:T3
  n:2 src:T1 dst:T3 tmp:T2
  n:1 src:T1 dst:T2 tmp:T3
  n:1 src:T2 dst:T3 tmp:T1
  n:2 src:T3 dst:T2 tmp:T1
  n:1 src:T3 dst:T1 tmp:T2
  n:1 src:T1 dst:T2 tmp:T3
  n:3 src:T2 dst:T3 tmp:T1
  n:2 src:T2 dst:T1 tmp:T3
  n:1 src:T2 dst:T3 tmp:T1
  n:1 src:T3 dst:T1 tmp:T2
  n:2 src:T1 dst:T3 tmp:T2
  n:1 src:T1 dst:T2 tmp:T3
  n:1 src:T2 dst:T3 tmp:T1

- What is the output of your algorithm when we invoke the move routine in the solution for three towers as `move`(4, “T1”, “T3”, “T2”)?

  **[1 point for the correct answer to this part.]**

**Give zero in this part if the answer is not exactly matching with the following sequence of statements.**

moving from T1 to T2
moving from T1 to T3
moving from T2 to T3
moving from T1 to T2
moving from T3 to T1
moving from T3 to T2
moving from T1 to T2
moving from T1 to T3
moving from T2 to T3
moving from T2 to T1
moving from T3 to T1
moving from T2 to T3
moving from T1 to T2
moving from T1 to T3
moving from T2 to T3