CS358: Applied Algorithms

Assignment 1: Two Towers (due 9/18/24)

Instructor: Sam McCauley

1 Implementation

Instructions

There are two parts to the assignment: code, and a pdf submission.

Code can be submitted by committing and pushing them to gitlab. I strongly suggest that you access `evolene.cs.williams.edu` through a web browser to make sure everything was uploaded as you expected. You may collaborate on code with your classmates, as well as the instructor and TA, but you may not use any LLM assistance.

The pdf should be uploaded to Gradescope for submission and feedback. The pdf should be entirely your own work; you should only use "hands in pockets" discussions with your classmates and the TA; you should not use any online (including LLM) assistance.

Please contact me at srm2@williams.edu if you have any questions or find any problems with the assignment materials.

Problem Description

GOAL: The input represents a set of blocks; the ith integer in the input represents the "area" of the ith block. The height of a block is the square root of its area.

The goal is to partition the blocks into two sets (which we call towers), such that the height of the towers is as close as possible. The value returned should be the blocks that make up the smaller of these two towers.

INPUT: The input to the problem will be an array of at most 64 unsigned 64-bit integers (this array will be given as a pointer of type `uint64_t*`), along with an integer representing how many elements there are in the array.

OUTPUT: The output is a single unsigned 64-bit integer whose bits represent a subset of the blocks in the array. For example, the subset consisting of only the first element in the array would be represented by the integer 1; the subset consisting of the first, third, and fourth elements of the array would be represented by 13.

Testing Parameters

The `main()` method of the testing program (in `test.c`) takes two arguments, each of which is a file containing instances of the two towers problem. An instance of the problem is represented by a sequence of integers (separated by spaces) on one line, and the intended solution (represented as a decimal number) on the second line. You can test your program by first running `make`, and then running `./test.out data.txt` (you should replace `data.txt` with the name of the data file being used). If the solution is incorrect, the program will say so (with some extra information to help debug); if the solution is correct, the program will output the running time in seconds.

There are three instances: a smaller instance, `smallData.txt`, and two larger instances, `data1.txt` and `data2.txt`.

For all instances, the best solution is guaranteed to be at least .0001 larger than the second-best solution (i.e. the smaller tower in the optimal solution is .0001 larger than any other tower smaller than half the height). This guarantee is to help rule out issues with floating point errors. This also means that the smaller tower is strictly smaller than the larger tower—you do not need to worry about tiebreaking.

Provided Code

The provided code is to help you; you can use it any way you wish.

In `test.c`, there is code which reads in the input file, calls your implementation functions, times them, and tests if the answer is correct. You probably don't need to modify or read this.

In `twotowers.c` there is a struct, called `Subset` with two member variables: the first variable is a `uint64_t` storing the subset itself, and the second is a `double` storing the height of the subset.

I have also provided the following functions for you to use in your implementation. These can be found in the files `twotowers.h` and `helper.c`

- `updateBest`: given a new solution, compares to the best found so far and updates if necessary
- `getTotalHeights`: takes the square root to get the height of each block and stores them in an array; returns the total height
- `getHeight`: gets the height of a particular set of blocks
- `compareSlns`: compares two `Subsets`; this function can be used to sort with `qsort`
- `binarySearchTable`: binary searches for the predecessor of a target in a table of `Subsets`
- `merge`: merges two sorted arrays
- `mergeSort`: uses merge sort to sort an array of `Subsets`

Questions

We will compare four implementations for this assignment.

Basic Implementation: First, implement the algorithm as given in class using `qsort` as the sorting function. This algorithm should store all `Subsets` of blue blocks in a table and sort them using `qsort` (remember that I already provided a comparator function for you to use). Then, it should iterate through all subsets of yellow blocks, use `binarySearchTable` to search for the best blue subset for the given yellow blocks, and call `updateBest` to update if it is the best option automatically.

I strongly recommend that you create a function to create the table of subsets, as you'll use it repeatedly throughout the assignment. I left a stub for this purpose, `generateTable`, in `twotowers.c` in the starter code.

Implementation 1. Implement this algorithm in the function `basicTowers()` in `twotowers.c`

Improved Sort Implementation: Second, alter Implementation 1 to use `mergeSort` rather than `qsort`. (This should require very few changes.)

Implementation 2. Implement this algorithm in the function `mergeSortTowers()` in `twotowers.c`

Sort Both: Third, we will do something slightly odd. Let's build *two* tables: one on the blue blocks, and one on the yellow blocks. Then we will sort *both* tables by height. After that, we will iterate through the *table* of yellow blocks (this means we iterate in sorted order by height); for each do the binary search for the best set of blue blocks and update as before.

Implementation 3. Implement this algorithm in the function `sortBothTowers()` in `twotowers.c`

Problem 1 (Extra Credit)**.** Show that in the external memory model, the total cost of all binary search operations is $O(2^{n/2}/B)$. (We will likely go over the external memory model in class on Tuesday.)

Solution. □

Better Sort: Finally, let's work towards a surprising result: for this specific problem, there is a sort function that runs in linear time—it can sort all $2^{n/2}$ subsets in time $O(2^{n/2})$, rather than $O(n2^{n/2})$. Let's break down how this function works.

Our goal is to create a sorted table with all `Subsets` of $n/2$ blocks (let's say the $n/2$ blue blocks; we'll do the same for the yellow blocks). We will do this inductively, by adding one block at a time.

Problem 2. First, explain how to sort all subsets of zero blocks. (Yes, this is trivial, but you'll implement this in the code.)

Solution. □

Now, assume you have sorted all subsets of k blocks, for some $k \geq 0$. Our goal is to obtain two arrays of size 2^k. The first array is the array you already have: it consists of all subsets of k blocks, in sorted order. The second array consists of all subsets of $k+1$ blocks that include the $k+1$st block. In other words, between the two sorted arrays we have all subsets of $k+1$ blocks: the first contains the subsets that do ont have the $k+1$st block, and the second contains the subsets that do.

Problem 3. Explain how to obtain the second array using a single scan through the first array, in $O(2^k)$ time.

Solution. □

Problem 4. Now, explain how to use these two sorted arrays of size $O(2^k)$ to obtain the sorted array of all $k+1$ blocks, in $O(2^k)$ time. (Hint: can we use a function that we already have available?)

Solution. □

Problem 5. Analyze the running time of this algorithm, showing that it requires $O(2^{n/2})$ time to generate the sorted table of all yellow blocks.

Solution. □

Implementation 4. The above gives us an algorithm to generate a sorted table of all subsets: we do the above for $k = 1, 2,$ up to the number of blocks. Implement this algorithm in the function `generateSortedTable()` in `twotowers.c`. Then, use this new algorithm to create a two towers algorithm: alter your `sortBothTowers()` solution to generate the sorted table using `generateSortedTable()`; implement this in the fucntion `betterSortBothTowers()`.

Implementation 5 (Extra Credit)**.** It is possible to compare the arrays without a binary search each time. (Hint: use the fact that both tables are sorted. If we know where the binary search lands for a given yellow subset, what can we say about the next one?) Implement an $O(2^{n/2})$ time algorithm in the function `extraCredit()` for this problem—the implementation is sufficient, you do not need to prove the running time. The implementation should run in at most .90 seconds on input `data2.txt`. Achieving this time likely requires further optimization of the merge implementation in `helper.c`.

2 Comparison

Problem 6. Please fill in below the time each algorithm took on data1.txt and data2.txt.

Hopefully, the times decreased with each successive implementation. The reason why will be a focus of the next couple weeks.

Impl. 1	Impl. 2	Impl. 3	Impl. 4
0	0	0	0

`data1.txt`

Impl. 1	Impl. 2	Impl. 3	Impl. 4
0	0	0	0

`data2.txt`