CS358: Applied Algorithms

Mini-Midterm 1: 3SUM (due 10/6/21)

Instructor: Sam McCauley

Instructions

All submissions are to be done through github, as with assignments. This process is detailed in the handout "Handing In Assignments" on the course website. Answers to the questions below should be submitted by editing this document. All places where you are expected to fill in solution are marked in comments with "FILL IN." The mini-midterm will not have a leaderboard or any automated testing.

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

This is a mini-midterm (as on the syllabus). This means that **all work must be done alone.** Please do not discuss solutions with any other students, even at a high level. You should not look up answers, hints, or even code libraries on the internet. This midterm was designed to be completed with no external resources, beyond those explicitly linked in the midterm. (Class resources, such as slides, notes, and your previous assignment submissions, are of course acceptable, as are basic resources such as looking up debugging information.)

You may assume the running times of any algorithms and data structures you learned about in CSCI 256 or this class. For example, you may assume a dictionary implemented with a hash table that requires $O(1)$ lookup time and $O(n)$ space,[1] or a sort algorithm taking $O(n \log n)$ time and $O(\frac{n}{B} \log_{M/B} \frac{n}{B})$ cache misses.

A comment on grading: remember that we have four of these over the course of the semester, and (while important) this midterm is only worth 20% of your final grade.

[1] As we'll see in the next part of the course, $O(1)$ is only expected, but for simplicity you may assume it is worst case.

1 Problem 1: 3SUM

Problem Description

INPUT: The input consists of three lists A, B, and C, each consisting of n 64-bit (signed) integers.

Each instance will appear on five lines in the input file. The first line will consist solely of the integer n. The next three lines will each contain n 64-bit integers (possibly negative), written as normal decimal text (same as in Assignment 1). The last line will contain the expected output.

OUTPUT: The output consists of three integers, each between 0 and $n-1$.

GOAL: Let the three integers output by the algorithm be i, j, and k. The goal is to output i, j, k such that $A[i] + B[j] = C[k]$. All instances given to you have exactly one triple of integers satisfying this.

Thus, if $n = 2$, and $A = \{1, 5\}$, $B = \{8, 37\}$, and $C = \{2, 13\}$, then the correct answer would be $i = 1$, $j = 0$, $k = 1$ because $A[1] + B[0] = 5 + 8 = 13 = C[1]$.

Algorithm Description

Simple algorithm

Let's begin with a relatively simple $O(n^2)$ time solution. **Implementing this algorithm is not sufficient for full credit.**

First, sort A and B. Then, for each entry k of C, we repeat the following steps:

- Set $i = 0$ and set $j = n - 1$.
- While $i < n$ and $j \geq 0$:
  - if $A[i] + B[j] = C[k]$, return the original positions[2] of i, j, and k.
  - if $A[i] + B[j] < C[k]$, increment i.
  - if $A[i] + B[j] > C[k]$, decrement j.

This process takes $O(n)$ time for each k (because we only need to scan through A and B once). Repeating for all n values of k, we obtain $O(n^2)$ time.

Cache-efficient algorithm

The following is a beautiful way to solve 3-SUM using a "blocking"-like method. As in Assignment 2, it is possible that an optimized version of the simple solution will be fast enough to pass the tests (and may even be faster). **You are required to implement the algorithm given here.** (The simple 3-SUM algorithm described above is fairly easy to implement and will be worth only a very small amount of partial credit.)

The algorithm uses the following hash function, parameterized by a constant `SHIFT`:

[2] That is to say, you want to return the position these elements were in before they were sorted.

```
uint64_t hash3(uint64_t value){
        return (uint64_t)(value * 0x765a3cc864bd9779) >> (64 - SHIFT);
}
```

You may implement `SHIFT` in whatever way you want (you can hardcode a constant using `#define`, or store it as a constant variable, or pass it as an argument to `hash3`, or keep it as a global variable—all of these are fine). `SHIFT` should never be more than 64. You do not need to use variables of type `uint64_t` (that is, you can change the type if you want).

Note that if you set `SHIFT` to a value s, then `hash3` will output a value between 0 and $2^s - 1$.

The final algorithm proceeds as follows (for readability, we use s to denote the value of `SHIFT`):

- Create 2^s "buckets" for A (let's call them $BktA[0], BktA[2], \ldots, BktA[2^s - 1]$)
- Create 2^s buckets for B (let's call them $BktB[0], BktB[2], \ldots, BktB[2^s - 1]$)
- Create 2^s buckets for C (let's call them $BktC[0], BktC[2], \ldots, BktC[2^s - 1]$)
- For $i = 0$ to $n - 1$, add $A[i]$ to bucket $BktA[\texttt{hash3}(A[i])]$.
- For $i = 0$ to $n - 1$, add $B[i]$ to bucket $BktB[\texttt{hash3}(B[i])]$.
- For $i = 0$ to $n - 1$, add $C[i]$ to bucket $BktC[\texttt{hash3}(C[i])]$.
- For each bucket b_A of A (i.e. $b_A \in \{0, \ldots, 2^s - 1\}$) and each bucket b_B of B (i.e. $b_B \in \{0, \ldots, 2^s - 1\}$):
  - Let b_C be the bucket of C satisfying $b_C = (b_A + b_B) \ \% \ 2^s$.
  - Run the simple 3SUM algorithm using $BktA[b_A]$, $BktB[b_B]$, and $BktC[b_C]$.
  - Let b_C be the bucket of C satisfying $b_C = (b_A + b_B + 1) \ \% \ 2^s$.
  - Run the simple 3SUM algorithm using $BktA[b_A]$, $BktB[b_B]$, and $BktC[b_C]$.

A "bucket" is just a collection of items from a given array that share the same hash value. (For example, $BktA[0]$ refers to a list of all items $A[i]$ such that $\texttt{hash3}(A[i]) = 0$.) You may store them however you want—but bear in mind that they should be accessible in a way that's cache-efficient!

The % operator is intended to mean modulo (as in C) in the above.

I am happy to answer questions you may have about this algorithm. It was originally described in the paper "Subquadratic Algorithms for 3SUM" by Baran, Demaine, and Pătraşcu, which can be found here: https://people.csail.mit.edu/mip/papers/3sum/3sum.pdf (see Section 3.2 for the final algorithm; the hash is described in Section 2.1).

Questions

Code (50 points). Implement the cache-efficient algorithm for 3SUM described above. (Note that the simpler $O(n^2)$-time solution is probably something you want to implement first—it's useful for testing, and will be used as a subroutine in your algorithm anyway.)

Problem 1 (10 points)**.** Let s be the value of `SHIFT` used in your code. This means that there are 2^s buckets.

Assume that each of the 2^s buckets contains $\Theta(n/2^s)$ elements.[3] In the External Memory model, how many cache misses are incurred by this algorithm? (Please give your answer in terms of s, n, M, and B. If you need to make any assumptions—such as $n > B$—please state them explicitly.)

Solution.

Problem 2 (5 points)**.** Using your answer from Problem 1, what value should you use for `SHIFT` to minimize the number of cache misses incurred by the algorithm in the external memory model?[4] Please give your answer in terms of n, M, and B. (You may not need to use all three.)

Solution.

Problem 3 (10 points)**.** Let's investigate the performance of your algorithm when `SHIFT` $= 4$.

First, compare the running time of your algorithm with `SHIFT` $= 4$ to its performance when using the naive approach. Describe your experiment in detail. How do the running times differ? (Do they?)

Then, use `cachegrind` to simulate the number of cache misses incurred by your algorithm with `SHIFT` $= 4$ to the number of cache misses when using the naive approach. Describe your experiment in detail. How do the number of cache misses differ (answer this for both L1 and LL cache)?

Solution.

[3] You may have noticed that in practice this is not true—there are likely a small number of buckets that are quite a bit bigger. We're ignoring those buckets for the sake of simplifying our analysis.

[4] This value is likely to deviate significantly from what you found experimentally—here I am asking for the theoretical prediction alone.

2 One More Two Towers Question

In this section there is one more question about the Two Towers problem, as we looked at in Assignment 1.

Problem 4 (10 points)**.** Let's say you only have enough space to store a lookup table for S subsets,[5] where $S \leq 2^{n/2}$. Describe an algorithm for the two towers problem (as defined in Assignment 1) that requires $O(n2^n/S)$ time and only uses a lookup table for S subsets.

You may assume that S is a power of 2 for simplicity.

Solution.

3 Moving Two Towers

So far we've looked for awhile at how to *build* two towers of similar sizes from an array of numbers. In this problem, we're going to play a game, *moving* blocks between the two towers.

You are given as input an array of n block areas (as in Assignment 1), as well as four further numbers `start`, `end`, `threshold`, and k.

In a *move*, you may move a single block from one tower into the other tower. However, a move is only legal if the two towers in the resulting instance have height that differs by at most `threshold`.

`start` and `end` are instances of the two towers problem. For the sake of simplicity, assume that these follow the same format we used in Assignment 1: they are unsigned integers where a 1 in position i indicates that i is in the smaller tower.[6] Let's assume that in both `start` and `end` the towers have height that differs by at most `threshold`.

In this problem, you want to design an algorithm that determines if it is possible to get from `start` to `end` in exactly k legal moves. You do not need to output the moves themselves; the algorithm only needs to determine if it is possible.

Problem 5 (5 points)**.** Design an $O(n^k)$-time algorithm for this problem.[7] How much space does it take?

Solution.

[5] Note that if each subset requires $O(n)$ space, then I am saying that we have $O(nS)$ space in total to work with.

[6] In a real machine, you probably need to assume $n \leq 64$ for this to make sense— to keep things simple, let's ignore that here.

[7] You may notice that if k is very large, a $O(2^n)$ time algorithm is superior. Let's assume that k is small enough that this doesn't matter. (In particular, let's assume that $k < n/2 \log_2 n$).

Problem 6 (10 points)**.** Design an $O(n^{k/2})$-time algorithm for this problem using a meet-in-the-middle approach. How much space does it take?

Hint: Do not divide the array of blocks into two parts. Instead, notice the $k/2$—we're dividing the *solution* (of length exactly k) into two parts.

Solution.