CS358: Applied Algorithms

Mini-Midterm 1: 3SUM (due 3/11/20)

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 sam@cs.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 and notes, 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] a sort algorithm taking $O(n \log n)$ time and $O(\frac{n}{B} \log_{M/B} \frac{n}{B})$ I/Os, and a balanced binary search tree with $O(\log n)$ worst-case height.

Comments 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. Both problems will be broken down into approximately-equal points for each question; the code will be worth about half of Problem 1.

[1] As we discussed in class, $O(1)$ is only expected; for simplicity you may assume it is worst case.

1 Problem 1: 3SUM (70 points)

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 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 "tiling"-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 is very easy to implement and 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 inline in the function, 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:

- Create 2^s "buckets" for A (let's call them $B_A = 0, \ldots, 2^s - 1$)
- Create 2^s buckets for B (let's call them $B_B = 0, \ldots, 2^s - 1$)
- Create 2^s buckets for C (let's call them $B_C = 0, \ldots, 2^s - 1$)
- For $i = 1$ to n, add $A[i]$ to bucket $B_A =$ `hash3`$(A[i])$.
- For $i = 1$ to n, add $B[i]$ to bucket $B_B =$ `hash3`$(B[i])$.
- For $i = 1$ to n, add $C[i]$ to bucket $B_C =$ `hash3`$(C[i])$.
- For each bucket B_A of A, and each bucket B_B of B:
  - Let B_C be the bucket of C satisfying $B_C = (B_A + B_B) \ \% \ 2^s$.
  - Run the simple 3SUM algorithm with lists B_A, B_B, and 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 with lists B_A, B_B, and B_C.

A "bucket" is just a list of items from a given list that share the same hash value. (For example, $B_A = 0$ refers to a list of all items $A[i]$ such that `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

Problem 1. Please describe your implementation at a high level. What techniques and data structures did you use? What optimizations did you use? Determine the value of `SHIFT` that results in the best performance of your code, and state what value that is.

I am not looking for your code to necessarily hit certain benchmarks. Rather, discuss how to use the optimizations we talked about in class on this problem. This can include techniques to avoid expensive operations, algorithmic speedups, or efficient data structure ideas (for example, an efficient way to store buckets.) Then, implement them and discuss if they appear to improve your code. As a rule of thumb, you should look to include and/or discuss about two good optimizations.

Solution.

Problem 2. 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 $O(n/2^s)$ elements.[3] In the External Memory model, how many I/Os does this algorithm take? (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 3. What value should you use for `SHIFT` to minimize the number of I/Os 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.

NEXT PROBLEM IS ON THE FOLLOWING PAGE

[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 Problem 2: 4SUM (70 points)

What if we had 4 arrays A, B, C, and D, each of n elements? We want to find i, j, k, ℓ such that $A[i] + B[j] = C[k] + D[\ell]$. This problem seems similar to 3SUM, but the techniques we use to solve it will be substantially different.

Problem Description

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

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

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

GOAL: Let the four integers output by the algorithm be i, j, k, and ℓ. The goal is to have i, j, k, ℓ satisfy $A[i] + B[j] = C[k] + D[\ell]$.

Questions

Please note that you do not need to implement any algorithms for this problem.

Problem 4. Design an $O(n^2)$ time algorithm to solve this problem.[5] You do not need to implement it. Be sure to give a clear solution: I should be able to implement your ideas exactly based on your description.

Hint: This will look somewhat different than the solution in Part 1—instead you will use one of the algorithmic techniques we learned about in class.

Solution.

Problem 5. How much space does your $O(n^2)$-time algorithm take?

Solution.

Problem 6. Describe an algorithm for 4SUM that requires $O(n)$ space. What is its running time? Is this space-efficient version likely to be superior in practice? Please discuss briefly why you would or would not think it is.

Hint 1: This space improvement will likely come at the expense of a worse running time.
Hint 2: This question is not going to have a clever divide and conquer solution like Hirshberg's. Keep it simple—this problem is intended to have a simple answer!

[5] Note that this is smaller than $O(n^2 \log n)$. An $O(n^2 \log n)$ solution will be worth nearly full credit. An $O(n^3)$ solution will be worth partial credit.

Solution.

Problem 7. We discussed in class that sorting can improve practical running times for large-scale search problems. How could a sorting step be incorporated into your algorithm? What would be the advantages and disadvantages of incorporating a sort?

Solution.