15-750: Algorithms in the Real World **CMU, Spring 2023**
HW #1 (out Tuesday Jan 24, 2023) **Due:** Tuesday Feb 7, 2023

Typical homeworks including this one will have some exercises (for practice, not submission, and to teach you ideas you may need to solve problems) and four problems. Please limit yourself to groups of two (or three) people. We adhere to a "whiteboard" collaboration policy: you should discuss in your group on whiteboards (or equivalents) but then should go away and write down your solutions yourself. You must not share written work. **You must write down the names of the person(s) with whom you collaborate.** In general, you should solve the homework problems using only to material we refer you to (or put on the course page), and not books or other online resources. If you have reason to use any resources we point you to (except the course notes), please cite them.

A word of advice: please try to solve the problem by yourself before working with your group. That is the way to improve your problem-solving skills.

Homeworks will be due at 11:59pm on the due date on `gradescope`. Corrections and changes will appear on the course webpage and on Piazza; please check them regularly.

Exercises (just for practice, not for submission)

1. **Asymptotic Notation and Recurrences**

   (a) For each list of functions, order them according to increasing asymptotic growth. Provide a brief argument justifying each successive step in the ordering.

   List 1, fast growing functions: $n!, n^{\lg n}, 2^{3n}, 3^{2n}$.
   List 2, slow growing functions: $2^{\lg n}, \lg(3^n), (\lg n)^5, \lg(n^5)$.

   In this course, we often use $\lg = \log_2$ (i.e., logarithm with base 2) and $\ln = \log_e$. When we say log (without specifying the base), it is usually when we care only about the asymptotics and the base does not matter: it can be set to any constant.

   (b) Solve the following recurrences, giving your answer in $\Theta(\cdot)$ notation. For each of them, assume the base case $T(x) = 1$ for $x \leq 5$. Show your work.

   (i) $T(n) = 3T(n/4) + n$.
   (ii) $T(n) = T(n-2) + n^4$.
   (iii) $T(n) = \sqrt{n}T(\sqrt{n}) + n$.

   You may use the Master theorem from the following handout. Please do not use the one from Wikipedia, or from other sources.

2. **MST T/F**

   For each of the following three statements, decide whether it is true or false. If it is true, give a short explanation. If it is false, give a counterexample.

   (a) Suppose we are given an instance of the Minimum Spanning Tree Problem on a graph G, with edge costs that are all positive and distinct. Let T be the minimum spanning tree for this instance. Now suppose we replace each edge cost $c(e)$ by its square root, $\sqrt{c(e)}$, thereby creating a new instance of the problem with the same graph but different costs. Is T still a minimum spanning tree for this new instance?

(b) Suppose we wish to design a spanning network for which the *most expensive edge* is as cheap as possible. More specifically, we define the *bottleneck edge* of a spanning tree T to be the edge of T with the greatest cost. A spanning tree T of a graph G is a *minimum-bottleneck spanning tree* if there is no spanning tree T' of G with a cheaper bottleneck edge.

Is every minimum-bottleneck spanning tree of G a minimum spanning tree of G?

(c) Is every minimum spanning tree of G a minimum-bottleneck spanning tree of G?

3. **Soring in Linear Time**

Recall that we work in the "Word RAM" model, where you assume that memory is divided into "words" of length w bits. (Think $w = 32$ or $w = 64$.) These words can hold things like integers and pointers to memory locations, and "basic operations" can be done on one or two words in $O(1)$ time.

In this model, suppose you have an input consisting of n words; e.g., an array A of n integers to be sorted. We always assume that w is large enough that $n \leq 2^w$, so that a single word can hold a pointer/index into A.[1] Moreover, we usually assume that $w \leq c \log_2 n$ for an extremely small constant like $c = 1, 2, 3, 4$.[2]

(a) Suppose you are given as input an array of n integers, each of which is either 0 or 1. Describe how to produce a sorted version of this list in $O(n)$ time.

(b) Suppose you are given as input an array of n integers, each of which is between 0 and $n-1$. Describe how to produce a sorted version of this list in $O(n)$ time. Assume you are perfectly happy allocating a couple of additional arrays of size n. (Remark: Given that the input array itself has length n, you probably don't mind allocating a couple more arrays of length n. However, you certainly don't want to go around allocating arrays of length $2^{64}\ldots$)

(c) Suppose you, as a human, are given a deck of slips of paper, and on each paper is a date, given by a year (between 0 and 2000) and a month (between 1 and 12) and a day (between 1 and 30). You want to sort these slips of paper into chronological order. One thing you could try is first sorting them by year (ignoring month and day), then re-sorting by month (ignoring year and day, but not altering the previous ordering of any tied slips), then re-sorting by day (ignoring year and month, but not altering the previous ordering of any tied slips). Does this work? If so, why? If not, does some similar idea work?

(d) Suppose you are now sorting an array of n integers (each fitting into one w-bit word). Assuming, say, $w \leq 3 \log_2 n$, it means that if we write each integer in base-n, it will be at most 3 "digits" long; in other words, it will be like a tuple of 3 numbers between 0 and $n-1$. Explain how such an array may be sorted in $O(n)$ time.

---

[1] This is certainly reasonable if $w = 64$, since 2^{64} is like trillions of terabytes!

[2] This is also reasonable: if $n \leq 65536 = 2^{16}$, that's small enough that any reasonable algorithm will finish in the blink of an eye.

Problems

1. **Triangle Counting Again**

   Here's an algorithm that, given a graph $G = (V, E)$, counts the number of triangles in time $O(m^{1.5})$. (For this problem, assume you have G in adjacency-list format, but moreover you have a data structure that lets you tell in $O(1)$ time whether any edge $\{i, j\}$ exists in E.)

   Fix some parameter $\alpha \geq 1$, and let $H \subseteq V$ be the "heavy" vertices in G, meaning those that have degree at least α. Let $L = V \setminus H$ denote the "light" vertices, with degree less than α.

   Here is the idea of the algorithm:

   - For every triple of vertices in H, check if they form a triangle.
   - For every vertex $u \in L$ and for every pair of edges edges $\{u, v\}, \{u, w\}$ incident to u, check if $\{u, v, w\}$ forms a triangle.

   Fill in the details of how to make a correct algorithm out of this idea, and analyze its running time as a function of n, m, and α. Then, describe how α can be chosen (at the beginning of the algorithm) to obtain an algorithm for triangle-counting that runs in $O(m^{1.5})$ time.

   (Note that this algorithm will be faster than the matrix-multiplication based approach when $m \ll n^2$.)

2. **Dynamic Lists**

   You are asked to implement a *list* data structure, where you may assume the elements in the list are simply integers that fit in a single word of memory. Given a list A, you wish to support the following operations:

   - Read $A[i]$ (retrieve the ith element of the list).
   - Append x to the end of A.
   - "Pop" A (that is, delete the last element of A).

   What makes this a little challenging is you don't know an a priori upper bound on the size of the list. If you knew the list would always have length at most L, you could simply allocate an array of L words (stored contiguously in memory), and all operations would be $O(1)$ time. (Don't worry about things about Popping an empty list or Reading from a spot past the end of the list; it's easy to keep track of the list length and handle such things.)

   In this problem, you will still use memory arrays to solve the problem (which makes Reads $O(1)$ time), but you will do dynamic resizing; this will allow for Append and Pop operations that are *amortized* $O(1)$ time, while at the same time ensuring the amount of memory used to store n items is $O(n)$.

   (a) First, Appends-only. At each time you will have some number "n" of elements in your list, stored in an array of length L, with $L \geq n$. Initially $n = 0$ and $L = 1$. If ever an "Append" occurs with $n = L$, you allocate a new array of length $2L$, copy over the n elements from the old array to the new one, and deallocate the old array. Now you have room to Append in the new array. Assume the cost of all the copying/deallocating is cL for some constant c, and the cost of appending a new element is 1. E.g., a sequence of Appends starting from the beginning will cost $1, c + 1, 2c + 1, 1, 4c + 1, 1, 1, 1, \ldots$.

   Proved that in a sequence of n consecutive Appends, the amortized cost of an Append operation is $O(c)$.

(b) Now we want to handle Pops (more precisely, any mix of Appends and Pops). Of course, if we don't mind being wasteful in terms of space then this is simple: never deallocate any space. But we would like to ensure $L \leq O(n)$ at all times. So if we ever get enough Pops that $n \ll L$, we will want to allocate a new smaller array of size $L' \approx n$, copy the n elements over, and deallocate the old larger array. Assume the cost of this is dL for some constant d.

One strategy is to always ensure that L is equal to "n rounded up to the next higher power of 2". Show that this is a *bad* idea, by showing how to make long sequences of m Appends/Pops that incur a total time cost of $\Omega(m^2)$.

(c) Describe and analyze a different strategy in which we always have $n \leq L \leq 4n$, such that the amortized cost of each Pop/Append operation is $O(c + d)$.

3. **Fast Multiplication**

In this problem we will consider the task of multiplying two n-digit integers A and B. (Just for fun, let's genuinely work with base-10 digits.[3])

You should have in mind a case like $n \geq 1$ million. Thus, at a hardware level, you can't just say "Oh, A and B are stored in one word of memory, so my chip can multiply them in time 1". You should have in mind that A and B are like *strings*: they're stored as (1MB+) arrays of length n in memory, one digit per memory word.

(a) Prove that you can compute $A + B$ in time $O(n)$.

(b) Prove that, given a list of n integers, each of n digits, you can compute their sum in $O(n^2)$ time. (Careful about how many digits the sum is.)

(c) Deduce that you can compute $A \cdot B$ in time $O(n^2)$.

(d) Let p, q, r, s be numbers and let X be a "symbol", so

$$(p + qX) \cdot (r + sX) = pr + (ps + qr)X + qsX^2.$$

Forget about computer algorithms and digits for a moment; just think about basic math operations. Show how to compute the three "coefficients" pr, $ps + qr$, and qs using only three multiplications (and any number of additions and subtractions). If, for example, $X = \sqrt{-1}$, this lets you multiply two complex numbers with three real multiplications rather than four. If, on the other hand, $X = 10^{n/2} \ldots$

(e) For A and B being n-digit numbers, use divide-and-conquer to describe and analyze a way to compute $A \cdot B$ in time $O(n^{\lg 3}) = O(n^{1.6})$.

(f) Specifically using python,[4] write a computer program that computes $A_n \cdot B_n$ where A_n is $333 \cdots 3$ (n digits) and B_n is $777 \cdots 7$ (n digits) for various values of n. First, you should create integer variables holding A_n, B_n. (My favorite way to do this would be to make them as strings first, using python's wacky ability to multiply a string by a number... Then I'd use `int()` to convert strings to integers.) Once A_n and B_n are created, simply execute python's built-in multiplication, as in `An * Bn`. (You don't even have to assign the result to a variable, you can just let it be thrown away.) At a minimum, do this for all n's of the form $\lfloor 1.1^k \rfloor$ for $100 \leq k \leq 150$. For each multiplication, time

[3] Of course, on a computer it's more natural to work in base 2. And, if you want to be hyper-efficient, it's in fact better to use base-W "digits" where W is the largest integer that fits in one word of memory. But let's stick with base 10, for fun.

[4] If you don't know python... well, learn a little bit. It's easy. :-)

how long python takes. (Do this by starting your code with `import time`, and then subtracting the values of `time.time()` just before and after each multiplication. Don't time the creation of A_n, B_n.) Make a log-log plot of the running time versus n, and assuming your log-log plot looks roughly linear, eyeball the slope and report the value you get. (For this problem, submit your code as text, as well as the plots.)

4. **Array Allocation**

Recall that we work in the "Word RAM" model, where you assume that memory is divided into "words" of length w bits. (Think $w = 32$ or $w = 64$.) These words can hold things like integers and pointers to memory locations, and "basic operations" can be done on one or two words in $O(1)$ time.

In this model, it is assumed that in $O(1)$ time you can ask the computer to allocate for you an "array" $A[\cdot]$ of n contiguous words of memory (with the assumption $n \leq 2^w$, so that an index into the array can be stored in one word). You can then do reads/writes to this array in $O(1)$ time.

It is also typical to assume that when you allocate an array like this, each entry is automatically initialized to 0. (That is, the word 0^w consisting of w bits of 0.) You might argue that this is an unfair assumption since, on real computers, uninitialized memory may contain some worst-case, arbitrary, unknown "junk" w-bit numbers, and it might take $\Theta(n)$ time to go through and set them all to 0^w. Nevertheless, show that there is a data structure that makes the assumption warranted.

More precisely, design a data structure that implements "array of n words, each initialized to 0^w ". Your data structure should work in the model described above, but with the assumption that words read for the first time might have any unknown junk w-bit number in them. Your data structure should support "Initialization", "Reading A Word", and "Setting a Word" all in $O(1)$ time. You may use just $O(n)$ total memory cells. You can assume w is large enough that one word can store an index into any of the $O(n)$ total words you're using.

(Hint: this problem is pretty tricky! The way I know for solving it involves allocating three actual arrays of length n to implement the one virtual length-n array. And sometimes you may have to bravely look into an array at a potentially-junk index. . . !)