CSCI 270 Problem Set 2 Solutions

Fall 2025

Problem 1. *Given an unsorted array $A := \{a_1, \ldots, a_n\}$, we may assume that all elements are distinct. The median value of this array is the number a_i such that exactly half of the numbers in A are smaller than a_i and half are larger than a_i.*

Now, assume there is a subroutine ApproxMedian(B) *that returns a value p that is close to the median of B for any array B. Concretely, at least one-third of the numbers in B are smaller than p, and at least one-third of the numbers in B are larger than p. You do not need to worry about the implementation of this subroutine, and each call to* ApproxMedian() *takes constant time.*

Given this ApproxMedian *subroutine as a black box, can you design an efficient algorithm to find the median? Please provide an $O(n)$-time algorithm with pseudocode, a running-time analysis, and a correctness proof.*

Solution. We employ the following algorithm.

```
def select(A, k):
        while True:
            n = len(A)
            if n == 1:
                assert k == 0
                return A[0]

            p = ApproxMedian(A)
            L = [x for x in A if x < p]
            R = [x for x in A if x >= p]

            if k < len(L):
                A = L
                continue
            else:
                k = k - len(L)
                A = R
                continue

def SelectMedian(A):
    n = len(A)
```

```
22      if n % 2 == 1:
23          k = n // 2
24          return select(A, k)
25      else:
26          k1 = n // 2 - 1
27          k2 = k1 + 1
28          ans = 0.5 * (select(A, k1) + select(A, k2))
29          return ans
```

To demonstrate correctness of `SelectMedian`, it suffices to prove that `Select(A, k)` always returns the kth-smallest entry in A when $0 \le k < \text{len}(\texttt{A})$. This is owed to the fact that `SelectMedian(A)` simply calls `select(A, k)` for the appropriate index (or indices) k, on lines 23 – 24 and 26 – 29.

Lemma 0.1. *For any non-empty list* `A` *of length* n *and index* $0 \le k < n$, `select(A, k)` *returns the* k*-smallest entry of* A.

Proof. Fix such a list `A` and index k. First observe that each pass through the `while` loop preserves the value of `sorted(A)[k]` (i.e., the value of the kth smallest entry in `A`), even as `A` and k are modified in-place. This property clearly holds for lines 3 – 6, as `A` and k are not changed at all. It further holds for lines 13 – 14, as `A` is replaced with a prefix `L` of `sorted(A)` (up to reordering), the value of k is preserved, and we are assured that k is less than the length of this prefix, by line 12. Lastly, it holds through lines 16 – 18 because `A` is replaced with a suffix `R` of `sorted(A)`, and `k` is decremented by `len(A) - len(R)` = `len(L)`.

The program terminates because `A` strictly decreases in size after each pass of the `while` loop (by our assumption on `ApproxMedian`). It is correct upon termination due to the previous invariant and the fact that it only returns the unique entry of `A` when `A` has length 1 and `k` = 0. □

Finally, we claim that `SelectMedian` runs in linear time. As `SelectMedian` calls `select` twice and otherwise performs a fixed number of algebraic operations, it suffices to demonstrate that `select` runs in linear time. Note that the runtime $T(n)$ of `select` is governed by the recurrence relation

$$T(n) \le T\left(\frac{2}{3}n\right) + O(n).$$

In particular, a single pass of the `while` loop in `select` performs a constant amount of work on lines 3 – 8, a linear amount of work on lines 9 – 10, and a constant amount of work through the remainder of the loop. By our guarantee on `ApproxMedian`, it reduces the size of `A` by a factor of at least 1/3, before commencing an identical iteration of the `while` loop. This justifies the recurrence, which yields a runtime of $T(n) = O(n)$ by the Master Theorem. □

Problem 2. *Following the problem we discussed in our lecture: Given n points in a two-dimensional space, assume that all these points are located on three parallel horizontal lines. Can you design an $O(n \log n)$-time algorithm to find the closest pair among these points? Please provide pseudocode, a running-time analysis, and a correctness proof.*

Solution. Given an array of points A, we use the following pseudocode.

0. If A has ≤ 2 points, we compute the result directly and return it.
1. Sort A according to its points' x-coordinates, resulting in the array A_X.
2. Using the median of A_X to divide A down the vertical line $x = c$, resulting in the arrays L and R. That is, $L = \{a \in A : a[0] < c\}$ and $R = \{a \in A : a[0] \geq c\}$.
3. Using A_X, compute the x-sorted versions of L and R, i.e., L_X and R_X.
4. Recursively solve the closest pairs problem for L_X and R_X, obtaining solutions δ_X and δ_Y.
5. Set $\delta = \min(\delta_X, \delta_Y)$.
6. Using L_X and R_X, find the rightmost points in L and leftmost points in R on each of the 3 horizontal lines.
7. Compute the minimum distance between all 9 pairs of extremal points. Denote it ϵ.
8. Return $\min(\delta, \epsilon)$.

We first demonstrate that the algorithm is correct, by induction on $n = |A|$. When $n \leq 2$, correctness follows from manual computation of Step 0. Suppose now that that the algorithm is correct for inputs of size $\leq n$ and let $|A| = n + 1 > 2$. By the inductive hypothesis, Step 4 produces correct solutions for the subproblems L and R. Thus δ equals the minimum distance attained by a pair $(u, v) \in L^2 \cup R^2$. It remains to show only that ϵ equals the minimum distance attained by a pair $(u, v) \in L \times R$, as this exhausts all pairs in A. This claim follows immediately from the fact that any pair $(\ell, r) \in L \times R$ can be made only closer by replacing ℓ with a point on the same horizontal line which has a greater x-coordinate, and likewise for r (with lesser x-coordinate).

We now analyze runtime. Discounting the one-time $O(n \log n)$ cost of sorting A, we have the following recurrence

$$T(n) = 2T(n/2) + O(n).$$

In particular, Step 2 requires $O(1)$ time to compute c and $O(n)$ time to create the arrays L and R. Step 3 requires $O(n)$ time, as L_X and R_X will simply be, respectively, a prefix and suffix of A_X. Step 4 accounts for the term $2T(n/2)$, whereas Step 5 is $O(1)$ and Step 6 is $O(n)$ by iterating through the elements in L_X and R_X from reverse or forward direction, respectively. Finally, Step 7 is $O(1)$ owing to the fact that there are only 3 horizontal lines, thus 9 extremal pairs, and Step 8 is a single operation.

This justifies the recurrence relation, which yields a bound of $T(n) = O(n \log n)$ as desired. □

Problem 3. *Given two large n-bit integers a and b, design an algorithm to multiply a and b. It must run in time $O(n^{1.9})$.*

Solution. We use the strategy from polynomial multiplication; the pseudocode is as follows.

```
1  def mult(a, b):
2      n = len(a)
3      if n == 1:
4          return a * b
5      r = n / 2
6
7      f0 = a[:r]
8      f1 = a[r:]
9      g0 = b[:r]
10     g1 = b[r:]
11
12     h0 = mult(f0, g0)
13     h1 = mult(f0 + f1, g0 + g1)
14     h2 = mult(f1, g1)
15
16     ans = 2 ** (2 * (r + 1)) * h2 + 2 ** (r + 1) * (h1 - h2 - h0) + h0
17     return ans
```

Correctness is by induction on n. It follows immediately for $n = 1$. Now fix binary strings a, b of length $n \geq 2$, with $r = n/2$. Recall that $a = a_0 \dots a_{n-1}$ represents the number $\overline{a} = \sum_{i=0}^{n-1} a_i 2^i$, which can equivalently be written

$$\begin{aligned}
\overline{a} &= \sum_{i=0}^{n-1} a_i 2^i \\
&= \sum_{i=0}^{r} a_i 2^i + 2^{r+1} \cdot \sum_{i=r+1}^{n-1} a_i 2^{i-(r+1)} \\
&= \sum_{i=0}^{r} a_i 2^i + 2^{r+1} \cdot \sum_{j=0}^{n-r-2} a_{r+1+j} 2^j \\
&= \overline{a[:r]} + 2^{r+1} \cdot \overline{a[r:]}.
\end{aligned}$$

Using this observation and the definitions of f_0, f_1, g_0, g_1 above, we have

$$\begin{aligned}
\overline{a} \cdot \overline{b} &= (\overline{f_0} + 2^{r+1}\overline{f_1}) \cdot (\overline{g_0} + 2^{r+1}\overline{g_1}) \\
&= 2^{2(r+1)} \cdot (\overline{f_1}\overline{g_1}) + 2^{r+1}(\overline{f_0}\overline{g_1} + \overline{f_1}\overline{g_0}) + \overline{f_0}\overline{g_0},
\end{aligned}$$

which agrees precisely with `ans` on line 16 due to our inductive hypothesis, save for the term multiplying 2^{r+1}. However, note that

$$(f_0 + f_1)(g_0 + g_1) - f_0 g_0 - f_1 g_1 = f_0 g_1 + f_0 g_0.$$

This completes the proof of correctness.

Regarding runtime, we claim that the algorithm obeys the recurrence relation

$$T(n) = 3T(n/2) + O(n).$$

This is due to the fact that the algorithm performs $O(n)$ work on lines $2-10$, invokes 3 recursive calls with inputs of size $n/2$ on lines $12-14$, and again performs linear work on line 16. In particular, addition of n-bit integers is linear time using the grade school algorithm, and products of the form $2^k \cdot c$ for an integer c can be performed in linear time by simply padding c with k many zeroes.

The recurrence then yields a bound of

$$T(n) = O(n^{\log_2 3}) \subseteq O(n^{1.6}),$$

using the Master Theorem. □