Homework 3

Kyle Krause

January 29, 2025

1. **Self assessment.**
   My answer for 4. b. on Homework 1 is as follows:

> $T(n) = 2T(\frac{2n}{3}) + T(\frac{n}{3})$; If $n > 2$, the array is split into its first two-thirds, and its last third. The sort function is called on the first two-thirds, then the last third, then the last two-thirds again, every call.
>
> The master theorem does not apply to this recurrence relation, as it is not in the form $T(n) = kT(\frac{n}{b}) + O(n^d)$, where $k \geq 1, b > 1, d \geq 0$.

   This solution is incorrect. I did not properly identify the form of the recurrence relation, and thought that the second split of the array in the function call was of size $\frac{n}{3}$, when it is actually of size $\frac{2n}{3}$. If I correctly identified this, the recurrence relation would be $T(n) = 3T(\frac{2n}{3}) + O(1)$, and I would have been able to apply the master theorem.

2. **The power of divide and conquer.**

   (a) NAÏVE is not efficient with respect to input size. If b is a 4-bit number, the algorithm will run in worst-case $O(2^4) = O(16)$ time. If b is an 8-bit number, the algorithm will run in worst-case $O(2^8) = O(256)$ time. Even though it looks like it is running in linear time, the algorithm is inefficient with respect to input size in bits because an added bit of complexity to b means exponentially more work.

(b)

- In $pq \bmod m$, p and q can be rewritten as a sum of components of m and a remainder, i.e., $p = km + r$ and $q = sm + t$. So:

$$\begin{aligned} pq \bmod m &= (km + r)(sm + t) \bmod m \\ &= ksm^2 + kst + rsm + rt \bmod m \\ &= ksm^2 + rsm + kst + rt \bmod m \\ &= m(ks + rs + kt) + rt \bmod m \end{aligned}$$

We can remove all of the m terms from the equation from the mod$m$ operation, so the final equation is:

$$rt \bmod m$$

By definition, r is the remainder of p divided by m, and t is the remainder of q divided by m. So, $r = p \bmod m$ and $t = q \bmod m$. Therefore, the final equation is:

$$pq \bmod m = ((p \bmod m)(q \bmod m)) \bmod m$$

- The blank can be filled in as follows:

$$\text{if } b \text{ is even, then } (a^{\frac{b}{2}} \bmod m)^2 \bmod m = (a^b) \bmod m$$

The blank must be filled by $(a^{\frac{b}{2}})$, because if the equation $((p \bmod m)(q \bmod m)) \bmod m = pq \bmod m$ is true, we can plug in $p = a^{\frac{b}{2}}$ and $q = a^{\frac{b}{2}}$ to get the equation:

$$(a^{\frac{b}{2}} \bmod m)^2 \bmod m = (a^b) \bmod m$$

(c)
```
Require: Integers a, b, m
Ensure: Computes a^b mod m
1: function ExpModulus(a, b, m)
2:     result = 1
3:     base = a mod m
4:     while b > 0 do
5:         if b mod 2 == 1 then
6:             result = (result × base) mod m
7:         base = (base × base) mod m
8:         b = ⌊b/2⌋
9:     return result
```

Correctness:
Base case: If $b = 0$, then the function will return 1, which is the correct answer.
Assume that the function works correctly for $b = k$. This results in two cases: if k is even, then the function will return $(a^{\frac{k}{2}})^2 \bmod m = a^k \bmod m$, and if k is odd, then the function will return $(a^{\frac{k-1}{2}})^2 \times a \bmod m = a^k \bmod m$.
Thus, the function works correctly for all b.
Runtime:
Each iteration halves the value of b, so the function will run in $O(\log b)$ time. There are no other loops or recursive calls, so the function will run in $O(\log b)$ time.

(d) With an A^n matrix, the algorithm would be modified by starting with a result of the identity matrix, and the base would be the matrix A. The base would be squared each iteration, and the result would be multiplied by the base if the power is odd. The algorithm would run in $O(n^3 \log n)$ time, as each iteration halves the value of n. This still means $O(\log n)$ iterations, and thus numerical operations, but each iteration would take $O(n^3)$ time.

(e) Using this 2x2 matrix, the algorithm from part (d) without any modulus operations would run in $O(\log n)$ time and compute the kth Fibonacci number. As you multiply the matrix by itself k times, the resultant matrix has the kth Fibonacci number in the first row, second column, or the first column, second row.

3. **The peak of divide-and-conquer.**

(a)

Require: Array $C[1 \dots n]$ (a rotation of a strictly increasing array)
Ensure: Returns the peak in C

```
1:  function FindPeak(C)
2:      ℓ = 1
3:      r = n
4:      while ℓ < r do
5:          m = ⌊(ℓ+r)/2⌋
6:          if C[m] > C[r] then
7:              ℓ = m + 1
8:          else
9:              r = m
        ▷ Now ℓ (which equals r) is the pivot index, or A's minimum. The peak is right before it.
10:     if ℓ = 1 then
11:         return C[n]
12:     else
13:         return C[ℓ − 1]
```

Correctness:

i. Since the original array is strictly increasing, the only place it "breaks" in the rotation is exactly where max element wraps around to the beginning. This is the pivot (the minimum in the rotated array).

ii. The loop maintains the invariant that the pivot index lies in $[\ell, r]$.
- If $C[m] > C[r]$, the pivot must be to the right of m (since in a strictly increasing array, $C[m] > C[r]$ indicates that the rotation break has not yet been passed).
- Otherwise, the pivot is between ℓ and m.

iii. Once $\ell = r$, we have identified the pivot exactly. The peak (largest element) is thus immediately before it (wrapping to the end if $\ell = 1$).

Runtime:

- Each iteration of the while loop performs a single comparison and halves the search interval, reducing $r - \ell$ by at least a factor of 2. This is $O(\log n)$.
- Finding the maximum after identifying the pivot is $O(1)$, so the overall time complexity is $O(\log n)$.

(b)

Require: Array $A[1 \dots n]$ of integers (not necessarily sorted), with $A[0] = A[n+1] = -\infty$

Ensure: Returns an index i such that $A[i]$ is a local peak; $A[i] \geq A[i-1]$ and $A[i] \geq A[i+1]$

```
1:  function FindLocalPeak(A, 1, n)
2:      ℓ = 1
3:      r = n
4:      while ℓ ≤ r do
5:          m = ⌊(ℓ+r)/2⌋
6:          if (A[m] ≥ A[m−1]) and (A[m] ≥ A[m+1]) then
7:              return m
8:          else if A[m−1] > A[m] then      ▷ A local peak must lie to the left.
9:              r = m − 1
10:         else                            ▷ A local peak must lie to the right.
11:             ℓ = m + 1
```

Correctness:

i. At any point, our search interval $[\ell, r]$ contains at least one local peak. When we pick $m = \lfloor (\ell + r)/2 \rfloor$:
   - If $A[m] \geq A[m-1]$ and $A[m] \geq A[m+1]$, then m is immediately returned as a local peak.
   - If $A[m-1] > A[m]$, we move the search to $[\ell, m-1]$; there must be a peak there because we have discovered a "rising slope" to the left.
   - Otherwise, we move the search to $[m+1, r]$, because $A[m+1] \geq A[m]$ shows a rising slope to the right, guaranteeing a local peak to the right.

ii. Eventually the interval $[\ell, r]$ collapses to a single position, which must satisfy the local-peak condition (or we found a local peak earlier).

Runtime: Each iteration of the while loop performs a single comparison and halves the search interval, reducing $r - \ell$ by at least a factor of 2. This is $O(\log n)$.

4. **Warming up with recurrence relations.**

   (a) Let $W(k)$ be the number of ways to reach the k-th step. Then:

   $$W(k) = W(k-a) + W(k-b),$$

   with base cases $W(0) = 1$ (one way to do nothing) and $W(k) = 0$ for $k < 0$.

   (b) Let $R(n)$ be the maximum profit obtainable from a rod of length n. Then:

   $$R(n) = \max_{1 \le i \le n} P[i] + R(n-i),$$

   with the base case $R(0) = 0$.

   (c) Suppose $S = \{s_1, s_2, \ldots, s_m\}$ and T is the target. Define $F(i,t)$ to be true if-and-only-if there is a subset of $\{s_1, \ldots, s_i\}$ summing to t. Then:

   $$F(i,t) = F(i-1,t) \vee (t \ge s_i) \wedge F(i-1, t-s_i).$$

   With base cases:

   $$F(0,0) = \text{true}, \quad F(0,t) = \text{false}(\text{for } t > 0).$$

5. **Hiking in Michigan!**

   (a) Define $p(i)$ to be the minimum possible total penalty when starting from hotel 0 (Ann Arbor) and ending at hotel i. To reach hotel i, one must come from some hotel j with $0 \le j < i$. The "daily penalty" for traveling directly from j to i is $200 - (d_i - d_j)^2$. Therefore, to find $p(i)$, we consider all possible previous hotels j and choose the one that minimizes the total penalty.

   $$p(i) = \min_{0 \le j < i} p(j) + 200 - (d_i - d_j)^2 \quad \text{for } i = 1, 2, \ldots, n,$$

   With base case

   $$p(0) = 0.$$

   Here, $d_0 = 0$ (Ann Arbor) and $d_1 < d_2 < \cdots < d_n$ are the distances of the n hotels from Ann Arbor. The minimum possible total penalty is $p(n)$.

(b)

Require: An array of hotel distances $d[0..n]$, where $d_0 = 0 < d_1 < \cdots < d_n$; $p[i]$ will be the minimum penalty to end at hotel i.
Ensure: Returns the minimum total penalty for traveling from hotel 0 (Ann Arbor) to hotel n (Hancock).

```
Initialize an array p[0..n] for storing minimum penalties.
Initialize an array choice[0..n] for back-pointers (used to reconstruct which hotels to stop at).
p[0] ← 0   ▷ Base case: no penalty before any travel.
for i ← 1 to n do
    p[i] ← ∞
    for j ← 0 to i − 1 do
        cost ← p[j] + 200 − (d_i − d_j)^2
        if cost < p[i] then
            p[i] ← cost
            choice[i] ← j
return p[n]
```

Correctness:

i. We compute $p(i)$ in increasing order of i. By the time we compute $p(i)$, we already know all $p(j)$ with $j < i$, ensuring correctness of each step.

ii. Since $p(i)$ is correctly computed for all i in ascending order, $p(n)$ necessarily holds the minimum total penalty for reaching hotel n. Storing $choice[i]$ lets us trace back which j provided the minimum penalty at each step, thus recovering an optimal sequence of stops.

Runtime: There are $n + 1$ entries $p[0], \ldots, p[n]$. For each i, we perform an inner loop from 0 to $i - 1$, taking $O(i)$ time. A loop from 0 to i, i times, gives an $O(n^2)$ overall time complexity.