Homework 2

Kyle Krause

January 22, 2025

1. **Self assessment.**
   My answer for 6. b. on Homework 1 is as follows:

> DeMorgan's Law states: If $x \in A \cap \overline{B}$, then $x \in \overline{A} \cup \overline{B}$.
> Contrapositive: If $x \notin \overline{A} \cup \overline{B}$, then $x \notin A \cap \overline{B}$.
> **Proof:**
>
> (a) If $x \notin \overline{A} \cup \overline{B}$, then $x \notin \overline{A}$ and $x \notin \overline{B}$.
>
> (b) From $x \notin \overline{A}$, it follows that $x \in A$. From $x \notin \overline{B}$, it follows that $x \in B$.
>
> (c) For $x \in A \cap \overline{B}$, x must belong to A and $x \notin B$. However, since $x \in B$, $x \notin A \cap \overline{B}$.
>
> (d) Thus, if $x \notin \overline{A} \cup \overline{B}$, then $x \notin A \cap \overline{B}$.

   I think this solution does not show what is required as it does not show the correct contrapositive, that is: if $x \notin \overline{A} \cup \overline{B}$, then $x \notin \overline{A \cap B}$. I did have correct reasoning, because I showed $x \notin \overline{A}$ and $x \notin \overline{B}$, but added extra unnecessary reasoning for my incorrect contrapositive.

2. **(Potential) infinite games**.

   (a) This potential function does not satisfy both properties. It is not known whether or not the potential decreases by some fixed amount every unit (round of the game), because on any given round of the game, the potential could increase by at most 2 (maize chip picked, three blue chips added) or decrease by at most 1 (blue chip picked).

   A potential function that satisfies both properties is as follows:

   Define a time unit to be a round of the game, and the potential n_i after i rounds to be $n_i = M + \frac{B}{4}$, where M is the number of maize chips and B is the number of blue chips. This satisfies both properties, as in either case (maize or blue chip picked up), the potential function decreases by at least $\frac{1}{4}$:

Case 1: Maize chip picked up $n_{i+1} = M - 1 + \frac{B+3}{4} = M + \frac{B}{4} - 1 = n_i - 1$

Case 2: Blue chip picked up $n_{i+1} = M + \frac{B-1}{4} = M + \frac{B}{4} - \frac{1}{4} = n_i - \frac{1}{4}$

The game will end once the potential function reaches 0, i.e. $n_i = M + \frac{B}{4} = 0$, but is guaranteed to end in B rounds once $n_i = \frac{B}{4}$, as the potential function decreases by $\frac{1}{4}$ every round.

(b) This process will never halt. There is no possible valid potential function for this process, as the distance between Eric and his duck can be modeled by the function $d_i = \frac{d_{i-1}+1}{2}$, where d_i is the distance between Eric and his duck after i rounds. This distance will never reach 0, as there is no fixed positive amount that the distance decreases by every round, the distance decreases by a decreasing amount every round. $d_{i+1} = \frac{d_i+1}{2} < d_i$ for all i, so the distance will never reach 0.

3. **Potential functions of functions.**

(a) The input $(4, 3)$ for the function FUNC(a, b) loops. Tracing FUNC(4, 3) gives:

i	function	calls
0	FUNC(4, 3)	FUNC(2, 6)
1	FUNC(2, 6)	FUNC(4, 3)
2	FUNC(4, 3)	FUNC(2, 6)
3	FUNC(2, 6)	FUNC(4, 3)
⋮	⋮	⋮

An infinite loop.

(b) Define a time unit to be two calls of this function, and let the potential function n_i after i time units be $n_i = a - b$. The function will halt when $n_i <= 0$, i.e. $a - b <= 0$. Assuming $a > b$, after ALG(a, b) is called, there are two cases to consider: b is even, and b is odd.

Case 1: b is even
If b is even, then the function calls ALG$(a - 1, b + 1)$ and then ALG$((a - 1), (b + 1) + 1)$. Thus,
$n_{i+1} = (a - 1) - (b + 2) = a - b - 3 = n_i - 3$

Case 2: b is odd
If b is odd, then the function calls ALG$(a, b + 1)$ and then ALG$((a) -$

$1, (b+1)+1)$. Thus,
$n_{i+1} = (a-1) - (b+2) = a - b - 3 = n_i - 3$

Thus, $n_i = a - b$ is a valid potential function for this function, as it decreases by 3 every two time units, and is guaranteed to halt once $n_i <= 0$.

(c) Define a time unit to be two calls of this function, and let the potential function n_i after i time units to be $n_i = x$. This function will halt when $n_i <= 10$. Assuming $x > 10$ after FOO(x) is called, there are two cases to consider: x is even, and x is odd.

If x is odd, then $x = x + 3$. If x is odd, then $x + 3$ is even, and the result will be $x = x + 3 = (x + 3)/2$
Because x is at least 10, the potential function will decrease by at least 5 every two time units.

If x is even, then $x = x/2$. The potential function will decrease by at least 5 every two time units.

Thus, $n_i = x$ is a valid potential function for this function, as it decreases by at least 5 every two time units, and is guaranteed to halt once $n_i <= 10$.

4. **Can you *master* the sorting algorithms?**

(a) $T(n) = 2T(\frac{n}{2}) + O(1) + T(n-1)$; Array is split into two and both sides sorted, swap is performed in constant time, and then the sort is called again on $[0, ..., n-1]$ items.

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 $k \geq 1, b > 1, d \geq 0$.

(b) $T(n) = 2T(\frac{2n}{3}) + T(\frac{n}{3})$; 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 $k \geq 1, b > 1, d \geq 0$.

5. **Karatsuba for polynomials.**

(a)

$$\begin{aligned}
&A(x) \cdot B(x) \\
&= (p(x) \cdot x^{\frac{n}{2}} + q(x))(r(x) \cdot x^{\frac{n}{2}} + s(x)) \\
&= p(x)r(x) \cdot x^{\frac{n}{2}} x^{\frac{n}{2}} + p(x)s(x) \cdot x^{\frac{n}{2}} + q(x)r(x) \cdot x^{\frac{n}{2}} + q(x)s(x) \\
&= (p(x)r(x)) \cdot x^n + (p(x)s(x) + q(x)r(x)) \cdot x^{\frac{n}{2}} + (q(x)s(x))
\end{aligned}$$

(b) Note that in the above multiplcation, without the $x^{\frac{n}{2}}$ factor, the $(p(x)s(x)+q(x)r(x))$ exists from one polynomial multiplcation, $A(x) \cdot B(x)$, with a $(p(x)r(x))$ term and a $(q(x)s(x))$ term.

$$A(x) \cdot B(x) = (p(x)r(x)) + (p(x)s(x) + q(x)r(x)) + (q(x)s(x))$$

Because we multiplied $A(x)$ and $B(x)$, we have already calculated the $(p(x)r(x))$ term from $(p(x)r(x)) \cdot x^n$, and the $(q(x)s(x))$ term from $(q(x)s(x))$. We can calculate the $(p(x)s(x) + q(x)r(x))$ term by calculating $(p(x) + q(x)) \cdot (s(x) + r(x))$ and subtracting the $(p(x)r(x))$ and $(q(x)s(x))$ terms.

(c)

Require: Arrays A and B of size n, storing the coefficients of the polynomials, of degree at most $n-1$
Ensure: Array C of size $2n-1$, storing the coefficients of the product of the two polynomials

```
1:  function KaratsubaCoeff(A, B, n)
2:      if n <= 1 then
3:          return A[0] · B[0]
4:      mid = n / 2
5:      p = A[0..mid-1]
6:      q = A[mid..n-1]
7:      r = B[0..mid-1]
8:      s = B[mid..n-1]
9:      p1 = KaratsubaCoeff(p, r, mid)
10:     p2 = KaratsubaCoeff(q, s, mid)
11:     p_plus_q = add(p, q)
12:     r_plus_s = add(r, s)
13:     p3 = KaratsubaCoeff(p_plus_q, r_plus_s, mid)
14:     # (p(x)s(x) + q(x)r(x)) = (p + q)(r + s) − pr − qs
15:     p3 = subtract(p3, p1)
16:     p3 = subtract(p3, p2)
17:     return p1, p2, p3
```

6. **Sorting out errors.**

   (a) A brute force algorithm to solve this problem would be to start at the end of the load order, and check downwards whether or not there is a load that is heavier than the current load. If there is, then print the error, and move to the next load from the end. If there isn't, then there is no error, and move to the next load from the end. This would be bound by $O(n^2)$ time complexity, as there is a loop for every loop through the array.

   (b) An $O(n)$ algorithm that returns the number of printed error messages in this specific scenario would be as follows:

   Start from the start of both arrays, with a pointer to the start of each. Until we are out of load values in either array, compare the current load values at the start of each array. If the pointer value of array 1 has a lighter load than the pointer value of array 2, nothing will be printed, and increment the pointer for array 1. If the pointer value of array 1 has a heavier load than the pointer value of array 2, then every remaining load in array 1 is heavier than the current load in array 2, and thus increment the error count by the number of remaining loads in array 1. Then, move to the next value in array 2. Once we are out of load values for either array, we have the total number of error messages printed.

   (c) An $O(nlogn)$ algorithm that returns the number of printed error messages would be as follows:

Require: Array $A[1..n]$ of distinct priorities, and indices *left* and *right* with $1 \leq left \leq right \leq n$.
Ensure: The total number of inversions (error messages) in the subarray $A[left..right]$.

```
1: function CountInversions(A, left, right)
2:     if left ≥ right then
3:         return 0
4:     mid = ⌊(left + right)/2⌋
5:     invLeft = CountInversions(A, left, mid)
6:     invRight = CountInversions(A, mid + 1, right)
7:     invCross = MergeAndCount(A, left, mid, right) # Modified version of algorithm from Part B
8:     return invLeft + invRight + invCross
9: function MergeAndCount(A, left, mid, right)
10:    L = A[left..mid]
11:    R = A[mid + 1..right]
12:    count = 0
13:    i = 1, j = 1, k = left
14:    while i ≤ length(L) and j ≤ length(R) do
15:        if L[i] ≤ R[j] then
16:            A[k] = L[i]
17:            i = i + 1
18:        else
19:            A[k] = R[j]
20:            count = count + (length(L) − i + 1)
21:            j = j + 1
22:        k = k + 1
23: # Leftover elements of L and R are copied to A
24:    while i ≤ length(L) do
25:        A[k] = L[i]
26:        i = i + 1
27:        k = k + 1
28:    while j ≤ length(R) do
29:        A[k] = R[j]
30:        j = j + 1
31:        k = k + 1
32:    return count
```