# CS 344 Homework Four

**Matthew McCaughan**

Due April 18th, 2025

**Problem 1: Academic Integrity**

I affirm that I have not given or received any unauthorized help on this assignment, and that this work is my own.
- Matthew McCaughan

**Problem 2: A prim and proper MST**

**(a):** For a proof of correctness, we will have algorithm $M$ performing Prim's, while some other algorithm $ALG$ will perform some arbitrary algorithm to attempt to find a MST. $M$ and $ALG$ select identical edges up until a certain point, where $M$ will choose edge $e$, and $ALG$ will choose some other edge $o$. Because Prim will select the cheapest edge, we know that $e$ is the minimum selection among the available edges to select (assuming no two edges have the same cost), and $o > e$, and the addition of the respective edges means $\sum weights(ALG) > \sum weights(M)$. At every point where $M$ and $ALG$ differ, we can replace edge $o$ in $ALG$ with the edge $e$ from $M$, to form a new modified algorithm $ALG'$ changing the new weight of $ALG$ to $(weights(ALG) - o + e)$ , and since we know that $o > e$, $(weights(ALG) - o + e) < weights(ALG')$. This shows an improvement over $ALG$, and every we can perform this improvement at every instance where the edges of $M$ and $ALG$ differ. We can perform this until both have performed a spanning tree, ending with a minimum spanning tree.
**(b):** Between any two vertices, there exists a unique simple path between them if no two edges in the graph have the same cost. Therefore to connect all vertices to form a minimum spanning tree, there exists a unique solution to form this minimum spanning tree. Since both Kruskal's algorithm and now Prim's algorithm have been shown to find the minimum spanning tree of a graph, and there exists a unique minimum spanning tree, then both algorithms must find this unique minimum spanning tree. Any deviation from this tree would not include minimal weights and would not be a minimum spanning tree, as both algorithms greedily select their minimum edges for their algorithm.

**Problem 3: Extra-Dynamic Arrays**

**(a):** If we want to pay $\omega(1)$ per operation, that is, strictly WORSE than a constant time per operation, we have to leverage the conditions where these constant time operations can incur a greater cost than just the operation itself. We have to look at the doubling and halving operations to have operations that are more expensive than just the operations themselves. If we can perform operations that double or half the array every time, then we can pay $\omega(1)$. Here is a sequence of these operations to get to this cost:
append()
append()
append()
append()

pop()
append()
pop()
append()
...
We perform 4 appends to get the list to a length of a power of 2 (length of 4 for this example). This gives us a now doubled size of 8. When we pop an element, we dip below half of 8, triggering a halving of the size to 4. With 3 elements in a size 4 array, appending 1 element will trigger a doubling back to 8. This appending and popping can repeat back and forth, and each event will trigger a doubling and halving respectively. Assuming that both of these operations creates a new array and copies all of the elements over, these constant-seeming operations can cost along the order of $O(n)$. This satisfies our requirement of $\omega(1)$
**(b):**

$$\phi(t) = \begin{cases} 2n - c, & \text{if } n \geq c/2 \\ c/2 - n, & \text{if } n < c/2 \end{cases}$$

To show that any sequence of T operations is $O(T)$, we should show that each (amortized) operation will run in constant time. With the potential function and the cost function $A_t = C_i + \phi(t) - \phi(t-1)$, we can go through the cases of appending and popping where there is/is not a growth/shrink
**Appending with growth:**
before: n = c and c is c, $c_i$ is c — after: n = (c+1) and c is 2c
$A_t = C_i + \phi(t) - \phi(t-1)$
$A_t = c + (2(c+1) - 2c) - (2(c) - c)$
$A_t$ = c + 2 - c = **2**
**Appending without growth:**
$A_t = C_i + \phi(t) - \phi(t-1)$ if $n < c/2$:
$A_t = 1 + (c/2 - n) - ((c/2) - (n-1)) =$ **0**
$A_t = C_i + \phi(t) - \phi(t-1)$ if $n \geq c/2$:
$A_t = 1 + (2n - c) - (2(n-1) - c) =$ **3**
**Popping with shrink:**
c goes to c/2
n (c/4 -1) goes to c/4 - 1 - 1
$A_t = C_i + \phi(t) - \phi(t-1)$
$A_t = n + (c/2/2 - (c/4 - 2)) - (c/2 - (c/4 - 1)) =$ **(n - some constant(?))**
**Popping without shrink:**
$A_t = C_i + \phi(t) - \phi(t-1)$ if $n < c/2$:
$A_t = 1 + (c/2 - n - ((c/2) - (n+1)) =$ **2**
$A_t = C_i + \phi(t) - \phi(t-1)$ if $n \geq c/2$
$A_t = 1 + (2n - c) - (2(n+1) - c) =$ **-1**
Each scenario of popping or appending, even if the operation requires a doubling or halving of the array at an operation, gives a constant amortized cost, which with any sequence of T operations should have a runtime of about $O(T)$.

**Problem 4: Rando-bubble sort**

**(a):** In the absolute worst case, our array will be randomly shuffled in the exact reverse order. In this worst case for a list of length $n$, $\sum_{i=1}^{n} i$ (sum of numbers 1 to n) swaps are required to sort the list to the correct order. This is also equal to $n(n+1)/2$. For every pair of numbers, they either are in the correct order or they are not, giving a 50/50 on expectation. Half of the pairs will be in the correct positions on expectation, so the expected number of swaps for a random bubble sort will be half of this worst case, or $(n(n+1)/2)/2$, which is also $n(n+1)/4$.
**(b):** With probability $p$, we will need to perform 0 swaps. With probability $1-p$, we will instead need to perform $n(n+1)/2$ swaps. To perform $n * log(n)$ swaps in expectation, the sum of the number of swaps from both scenarios must be in line with this expectation.

$$p * 0 + (1-p) * n(n+1)/2 = n * log(n)$$

We can disregard the fully sorted version, as it is only the reversed order which we will have to perform any work on.

$$(1-p) * n(n+1)/2 = n * log(n)$$
$$1-p = (n * log(n))/n(n+1)) * 2$$

the common term $n$ cancels out:

$$1-p = (log(n))/(n+1)) * 2$$
$$-p = -1 + (log(n))/(n+1)) * 2$$
$$p = 1 - (log(n))/(n+1)) * 2$$

For bubble sort to perform $n * log(n)$ swaps in expectation, p must follow the formula $p = 1 - (log(n))/(n+1)) * 2$.
For example, an array of length 8 should have a probability of p = $\frac{3}{9}$ to perform to this expectation.

**Problem 5: Hash-o-rama**

**(a):** Defining the hash function as the $n$-bit string that represents the element from $U$ will require creating $2^n$ different cells if they are indexed as n bit strings, and every used cell will contain the n-bit element from $U$. The size of the table alone is huge and a waste of space. Our table $M$ is not small in this case and is violation of our conditions. This hashing function is also not random, so it isn't guaranteed to be universal in practice, though not necessarily barred from being universal.
**(b):** This change will drastically reduce the size of the hash table, but the modulo could lead to a potentially increased chance of collisions, as different keys could output the same hashes. This proposal is also not random, so universality is not guaranteed with this proposal either.