Elissa Ito October 7, 2022

6.1210 Problem Set 2

Problem 1 *(Collaborators: Katie Sap, Jessie Jin)*

(a) Sort Critters by ID Number, a positive integer less than 894.
Counting Sort, $O(n)$, since all keys are < 894.

(b) Sort Critter by weight with only one comparison at a time.
Merge Sort, $O(nlog(n))$, since we are sorting by comparison.

(c) Sort Critter by weight after one got heavier.
Insertion Sort, $O(n)$, by how insertion sort is performed, if only 1 element is out of place it would take $O(n)$ to put it in the right place.

(d) Sort Critters alphabetically by name, which are strings of length at most $log_2(n) + 1$, lowercase letters of the English alphabet.
Radix Sort, $O(n)$ since there are 26 letters in the alphabet thus the value of names will be at most $26^{log2(n+1)}$, which can be rewritten as n^c.

(e) Sort Critters by number of times they battled; each critter did at least one battle and battled each other at most once.
Counting Sort, $O(n)$, keys are the number of times battled which ranges from 1 to $n - 1$.

(f) Sort critter by win rate.
Radix Sort, O(n), the smallest difference between any ratios is greater than $1/n^2$, thus multiplying the battle ratios by n^2 and rounding to an integer would have distinct integers per battle rate. Keys are at most n^2 which is n^c.

Problem 2 *(Collaborators: None)*

Part 2(a)

1. $r(L_i) = (c_i * 128^{m-1} + c_{i+1} * 128^{m-2} + ... + c_{i+m-1}) \mod p$
   $r(L_{i+1}) = (c_{i+1} * 128^{m-1} + c_{i+2} * 128^{m-2} + ... + c_{i+m}) \mod p$
   We are given $f = 128^m \mod p$

   (a) Multiplying $r(L_i)$ by 128 yields:
   $128 * r(L_i) = (c_i * 128^m + c_{i+1} * 128^{m-1} + ... + c_{i+m} * 128) \mod p$

   (b) Substituting $r(L_{i+1})$ in:
   $128 * r(L_i) = (c_i * 128^m - c_{i+m}) \mod p + r(L_{i+1})$

   (c) Isolating $r(L_{i+1})$:
   $r(L_{i+1}) = 128 * r(L_i) + (c_{i+m} - c_i * 128^m) \mod p$

   (d) Substituting f in:
   $r(L_{i+1}) = 128 * r(L_i) + (c_{i+m} - c_i * f) \mod p$

   Then the algorithm would be, given $r(L_i)$ and f:

   (a) Multiply $r(L_i)$ by 128

   (b) Add c_{i+m} to that value

   (c) Subtract $c_i * f$ from that value

   (d) Take the modulus p of that value

   (e) The resulting value is $r(L_{i+1})$

2. The algorithm is correct since the math shows that the expressions are equal. There is only one possible solution, and it is returned. (It does not matter when or how many times the modulus is calculated, since by nature of modulus calculations repeated modulus calculations yield the same solution i.e. $A \mod b = A \mod b \mod b$)

3. Since each step is $O(1)$, the total algorithm has runtime $O(1)$.

Part 2(b)

(i) Algorithm:
L has length n; Q has length $m << n$
We are given p, and Q mapped to m characters $(c_0, c_1, ...c_{m-1})$

(1) Calculate f by:

i. Start with $f = 1$

ii. Multiply f by 128
iii. Repeat step b) m times
iv. Take modulus p of that value
v. Resulting value is f

(2) Calculate $r(L_0) = (c_{L,0} * 128^{m-1} + c_{L,1} * 128^{m-2} + ... + c_{L,m-1}) \mod p$.

(3) Calculate $r(Q) = (c_{q,0} * 128^{m-1} + c_{q,1} * 128^{m-2} + ... + c_{q,m-1}) \mod p$.

(4) Check if $r(L_0) = r(Q)$: if it is, return True, if not, continue.

(5) For all i, in integers 1 to $n - m$ (increasing):

i. Calculate $r(L_i)$ using algorithm in part a). (For $i = 1$, use $r(L_0)$, for $i = 2$, use $r(L_1)$ calculated in previous iteration, etc.)
ii. Check if $r(L_i) = r(Q)$
iii. If it is, compare each character of Q with $L[i \text{ to } i + m]$ and if all characters are equal, return True. If all characters are not equal, continue to next index.
iv. If it is not, continue to next index.

(6) If for loop is exited and nothing has been returned, return False.

(ii) Proof of Correctness:

(a) Every solution picked is a valid solution:
Step 1 is correct by nature of power operations. Step 2 and 3 are correct because the hash function is given to us. Step 4 is correct by problem definition. Since we iterate though the whole L if we continue to not find a hash value match, the solution is correct if no match is found. If a hash value match is found, each character is checked to match, and if it does the algorithm returns True which is correct, and if it doesn't the algorithm returns False which is also correct.

(b) Every valid solution is picked:
The problem states to check IF a query Q appears inside L, thus returning true if one match is found is correct, and returning false if no match is found is also correct.

(iii) Runtime Analysis:
Step 1 takes $O(1)$ time based on part a). Step 2, 3, and 4 each take $O(1)$ time. Step 5i and 5ii take $O(1)$ time each. Step 5iii, if performed, performs $O(1)$ comparisons m times thus takes $O(m)$. Since the probability of any $r(A) = r(B)$ is $1/m$, and the maximum number of hash values comparisons done possible is n, total runtime for 5iii is $O(n)$. Step 6 takes $O(1)$ time. The total algorithm takes $O(n)$.

Problem 3 *(Collaborators: Sam Bruce)*

Part 3(a)

(i) Algorithm

1) Let A be the array of the n integers in the range $[0, N]$. Create a Direct Access Array (to be called DAA from this point onward) with elements $[0, 1, ...n + 1]$. Hash each element in A into DAA using the hash function $h(k) = (k \bmod (n+1))$. Items with the same hash key will be stored through chaining (hash table). Thus, the number of keys will be $n + 1$, and the maximum chain length possible will be n.
2) Iterate through each key of the hash table to find a key with no elements mapped to it.
3) Return the unmapped key.

(ii) Runtime Analysis
Step 1 will take $O(n)$ time since there are n elements, and insertion costs $O(1)$. (and build() for a hash table takes average $O(n)$ time according to lecture). Step 2 costs $O(n + 1) = O(n)$ since there are $n + 1$ keys to iterate through. (and find() for a hash table takes $O(1)$ according to lecture). Step 3 costs $O(1)$. The sum of all steps costs $O(n)$.

(iii) Proof of Correctness

(a) Every solution picked is a valid solution:
Every element in A is inserted into the hash table. In step 3, the unmapped key is returned. That value, which would be between 0 and $n + 1$, would not have been in A since if it had been it would have been mapped to that key. Such a key will exist since mapping n elements to $n + 1$ keys will ensure one key will remain unmapped.

(b) Every valid solution is picked:
The problem states that we can return any missing number, thus the algorithm is correct since it returns one correct answer.

Part 3(b)

(i) Algorithm: Polish Flag Algorithm where elements $> n + 1$ are "white" and elements $\leq n + 1$ are "red". Swapping uses $O(1)$ extra space and the Polish Flag Algorithm takes $O(n)$ time according to lecture.

(1) Place pointer at the leftmost index. From this point onward, index of pointer will be called i. If at any point after this $i > n$, skip to step 5.

(2) Check if i is equal to the element at that pointer (i.e. $i = A[i]$).

(3) If it is, move pointer to the right by one index ($i+ = 1$).

(4) If it is not, check if the element is smaller than n (i.e. $A[i] < n$).

   i. If it is, swap element $A[i]$ with element at $A[A[i]]$ and repeat from step 2.

   ii. If it is not, move pointer to the right by one index ($i+ = 1$).

(5) Iterate through the list to find the first element which $i! = A[i]$. Return that index.

(ii) Runtime Analysis
Step 1 is $O(1)$, step 2-4 is $O(1)$ repeated n times thus $O(n)$. Step 5 is $O(n)$. The total runtime is therefore $O(n)$.

(iii) Proof of Correctness

(a) Every solution picked is a valid solution:
Every element in in A is iterated through in steps 2-4. Thus, all elements in A which is $< n$ would be swapped such that their index is their value (i.e. $A[i] = i$). Then, any element in indices 0 to n that does not equal its index indicates that there were no elements in A that equaled that number, thus that index is a valid missing number in A that is within the range $[0, N]$ since $n \leq N$.

(b) Every valid solution is picked:
The problem states that we can return any missing number, thus the algorithm is correct since it returns one correct answer.