UM-SJTU JOINT INSTITUTE

Introduction to Algorithms
(VE477)

Homework #1

Prof. Manuel

Xinmiao Yu
518021910792

Sept. 18, 2020

Q1.

1. Obviously $1 \leq k \leq n$, otherwise the statement would not be true. To prove, because the hash table has n slots and the probability of n keys to hash to any slot is equal, for each key, it has a probability of $\frac{1}{n}$ to hash to any slot. So the number of keys hash to a same slot follows a binomial distribution with parameter n and p, where $p = \frac{1}{n}$. Then the probability for exactly k keys hash to a same plot is

$$P_k = \left(\frac{1}{n}\right)^k \left(1 - \frac{1}{n}\right)^{n-k} \binom{n}{k}.$$

2. The probability of a slot to have k keys is P_k. More than one slots may have k keys, but no slots would have more than k keys. Then, P'_k =the probability of at least one slot has k keys and other slots have no more than k keys, which is **smaller than or equal to** the probability that at least one slot has k keys. Because we have n slots, the probability of only one slot has k keys is $\binom{n}{1}P_k = nP_k$, and this probability is **larger than or equal to** the probability that at least one slot has k keys. Through this two inequality, $P'_k \leq nP_k$.

3. We have Stirling formula $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$, and $1 \leq k \leq n$

$$\begin{aligned}
P_k &= \left(\frac{1}{n}\right)^k \left(1 - \frac{1}{n}\right)^{n-k} \frac{n!}{k!(n-k)!} \\
&\approx \frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}{\sqrt{2\pi(n-k)}\left(\frac{n-k}{e}\right)^{n-k} k!} \left(\frac{1}{n}\right)^k \left(1 - \frac{1}{n}\right)^{n-k} \\
&= \sqrt{\frac{n}{n-k}} \left(\frac{n}{n-k}\frac{n-1}{n}\right)^n \left(\frac{n-k}{e}\frac{1}{n}\frac{n}{n-1}\right)^k \frac{1}{\sqrt{2\pi k}\left(\frac{k}{e}\right)^k} \\
&< \sqrt{\frac{n}{2\pi k(n-k)}} \frac{e^{-k}}{\left(\frac{k}{e}\right)^k} \\
&< \frac{e^k}{k^k}
\end{aligned}$$

4. From the conclusion of the last problem, we could obtain $logP_k < -klogk$. Because k is a positive integer, $klogk$ would increase when k increased. The lower bound of k could be used and get the inequality

$$\begin{aligned}
logP_k &< -c\frac{logn}{loglogn} \cdot log(\frac{clogn}{loglogn}) \\
&\leq -c\frac{logn}{loglogn}(loglogn - logloglogn) \\
&\leq -\frac{c}{2}logn.
\end{aligned}$$

So $P_k \leq \frac{1}{n^{c/2}}$, $P'_k \leq nP_k < \frac{n}{n^{c/2}}$. With $c = 6$, $P'_k < 1/n^2$.

Problem 1 continued on next page...

5. From the definition of expected value, denote the probability of $M = i$ as $Pr[M = i]$, $i \leq n$

$$\begin{aligned} E[M] &= \sum_i i \cdot Pr[M = i] = (\sum_{i \leq \frac{clogn}{loglogn}} i + \sum_{i > \frac{clogn}{loglogn}} i) \cdot Pr[M = i] \\ &\leq (\sum_{i \leq \frac{clogn}{loglogn}} \frac{clogn}{loglogn} + \sum_{i > \frac{clogn}{loglogn}} n) \cdot Pr[M = i] \\ &\leq \frac{clogn}{loglogn} Pr(M \leq \frac{clogn}{loglogn}) + nPr(M > \frac{clogn}{loglogn}) \\ &\leq \frac{clogn}{loglogn} + n \cdot \frac{1}{n^{c/2}} \end{aligned}$$

Therefore, $E(M) = \mathcal{O}(\frac{logn}{loglogn})$

Q2.

Suppose G is an undirected graph G with weighted edges and the weight of an edge e is decreased where $e \notin T$, $e = (u, v)$. When add e into the original MST, a cycle will be created and the new MST could be found after delete the edge with the largest weight in this cycle.

Algorithm 1: Find MST

Input : $G =< V, E >$, a original MST T for G, weight of edge $e = (u, v) \notin T$ decreased
Output: a new MST T'

```
1  Function DetectCycle(x, S, pred):
2      for vertex i that is adjacent to x do
3          if !A[i] then
4              A[i] ← true;
5              add (x,i) into S;
6              if detectCycle(i, S, x) then
7                  return true
8              end if
9              A[i] ← false;
10             delete (x,i) from S;
11         else if i ≠ pred then
12             return true
13         end if
14     end for
15     return false;
16 end
17 Function DynamicMST(G, e, T):
18     T' ← T ∪ {e};
19     S ← ∅ ;
20     n ← T'.size;
21     A[1..n] ← a new boolean array of false ;
22     A[u] ← true;
23     detectCycle(u, S);
24     maxEdge.weight ← 0 ;
25     for edge in S do
26         if edge.weight > maxEdge.weight then
27             maxEdge ← edge;
28         end if
29     end for
30     T' ← T' - {maxEdge} ;
31     return T'
32 end
```

Q3.

1. The pseudocode of the computation of sum is

Algorithm 2: Compute Sum

Input : two n-bits integer stored in array $num1$, $num1$ separately

Output: array *result* stores the sum of two integers

```
1  Function AlgoHw(num1, num2):
2      i ← 0;
3      carry ← 0;
4      while i < n do
5          x ← num1[i] + num2[i] + carry;
6          if x < 10 then
7              result[i] ← x;
8              carry ← 0;
9          else
10             result[i] ← x % 10;
11             carry ← 1;
12         end if
13         i ← i + 1;
14     end while
15     if carry == 1 then
16         result[i] ← 1;
17     end if
18     return result
19 end
```

2. a) The pseudo-code according to the problem description is

Algorithm 3: Multiplication

Input : two integers x, y

Output: the product of these two integers x, y

```
1  Function mult(x, y):
2      if x == 0 || y == 0 then
3          return 0
4      else
5          return x · (y % 2) + mult (2x, ⌊y/2⌋);
6      end if
7      return result
8  end
```

b) From the pseudo-code, we could write the final result as

$$\begin{aligned} result &= x \cdot (y\%2) + 2x \cdot (\lfloor y/2 \rfloor \%2) + ... + 2^n x \cdot (\lfloor y/2^n \rfloor \%2) + ... + 0 \\ &= x \cdot ((y\%2) + 2 \cdot (\lfloor y/2 \rfloor \%2) + ... + 2^n \cdot (\lfloor y/2^n \rfloor \%2) + ... + 0)\,. \end{aligned}$$

We could find that $(\lfloor y/2^n \rfloor \%2)$ would be same as shift left $(y << n)$ in binary system and it is the digit 0/1 on the n_{th} bit, from the procedure that converting a binary number to a decimal number, we could know that $y = (y\%2) + 2 \cdot (\lfloor y/2 \rfloor \%2) + ... + 2^n \cdot (\lfloor y/2^n \rfloor \%2) + ... + 0$, so $result = x \cdot y$, this algorithm is correct.

Q4.

a. If the time for each horse to complete the race could be recorded, **the minimum number of races is 5**. Randomly divide the 25 horses into 5 groups, through 5 races we could obtain all the time needed for each horse to complete the race.

b. If only the rank of each race could be recorded, **the minimum number will be 7**.

1) Randomly divide 25 horses into 5 groups, label as A, B, C, D, E, let each group have one race. Number every horse in each group as $1 > 2 > 3 > 4 > 5$, where 1 is the fastest horse in this group.

2) $4, 5$ for each group would not become one of the three fastest ones. Make the 6th race among $A1, B1, C1, D1, E1$, assume the result as $A1 > B1 > C1 > D1 > E1$. All the horses in group D, E could be eliminated.

3) $A1$ must be the fastest horse, $A2, B1$ might be the second fastest horse, $A3, B2, C1$ might be the third fastest horse. Make the 7th race among $A2, B1, A3, B2, C1$. Then the three fastest ones are determined.

Q5.

1. These two algorithms both might solve the Knapsack problem. This problem could have **no solution** with specific definition, i.e. $S = 1, 2, n = 4$.

   If these two algorithms are defined as, every time will choose the smallest/largest item from the remaining items and taken items could not be put back, these two algorithms might fail to solve it.

   - Choose the smallest item:
     $S = 3, 5, n = 10$, we would choose two 3 and could not solve it. Correct: choose two 5.
   - Choose the largest item
     $S = 9, 5, n = 10$ we would choose one 9 and fail to solve it. Correct: choose two 5.

If the taken items could be put back, they will have a higher probability to solve this problem. The implementation of fitting the knapsack with the largest item first is shown with pseudo-code. With smallest item first is almost the same, only the first input index is different and could detect no possible solution easier (i.e. when item is larger than the remaining sum, no solution if we keep change the index).

However, if we want to get an optimal algorithm, such that with smaller time complexity and the number of elements in this subset is the smallest, neither of them is optimal. We could choose dynamic programming to find a better solution based on the such requirement.

Algorithm 4: Knapsack Problem with Greedy Algorithm (largest item first)

Input : a set S, a number n

Output: a subset S' of S such that all elements in it add up to n

```
1  Function Knapsack(n, S, i, S'):
2      if n == 0 then
3          return true
4      end if
5      if i == 0 then
6          return false
7      end if
8      if S[i-1] > n then
9          return Knapsack (n, S, i-1, S')
10     end if
11     return Knapsack (n, S, i-1, S') || Knapsack (n - S[i-1], S, i-1, S' ∪ {S[i − 1]})
12 end
13 Function AlgoHw(S,n):
14     S ← S sorted by increasing order;
15     S' ← ∅;
16     if Knapsack (n, S, S.size, S') then
17         return S'
18     else
19         return ∅
20     end if
21 end
```

2. If m is power of 2, such that $m = 2^p, p \geq 0$, then $H(k) = kmodm$ will be the p lowest-order bits of k, part of the data will be cut. Data that different from the higher-order bits will create the same hash.

   If the population of keys shares a same common factor, they will hash to a same slot. Prime number could reduce the collisions.

3. We could choose from three numbers $1, 5, 11$ for any number of times to have a total sum of 15. And the number of numbers should be as small as possible.

Denote the sum of numbers we have chosen as w. Based on greedy algorithm, the locally optimal choice would be choose the largest number that is smaller than $15 - w$. Because in that way, the sum of numbers we need to choose will be the smallest, seems like less number will needed. Then we could have one 11 and four 1, 5 numbers in total. However, the globally optimal solution should be three 5, 3 numbers in total.