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.

UM-SJTU JOINT INSTITUTE

Introduction to Algorithms
(VE477)

Homework #2

Prof. Manuel

Xinmiao Yu
518021910792

Oct. 2, 2020

Q1.

1.a Prove that there exist three constants c_1, c_2, n_0 such that for $n > n_0$, $c_2 n^3 \leq n^3 - 3n^2 - n + 1 \leq c_1 n^3$. Obviously choose $c_1 = 2, n_0 = 1$, $n^3 - 3n^2 - n + 1 \leq n^3 \leq c_1 n^3$. So $T(n) = \mathcal{O}(n^3)$.

Choose $n_0 = 5$, then $3n^2 \leq \frac{3}{5}n^3$, $n \leq \frac{1}{25}n^3$. So

$$n^3 - 3n^2 - n + 1 \geq n^3 - \frac{3}{5}n^3 - \frac{1}{25}n^3 + 1 \geq \frac{9}{25}n^3 + 1 \geq \frac{1}{5}n^3.$$

Choose $c_2 = \frac{1}{5}$, then $T(n) = \Omega(n^3)$. So $n^3 - 3n^2 - n + 1 = \Theta(n^3)$

1.b Prove that there exist two constants c, n_0 such that for all $n > n_0$, $n^2 < c2^n$, which is same as $2ln(n) < cnln2$. Take the derivative, $\frac{2}{n} < cln2$, so when $n > n_0 = 2$, choose $c = 1$. As $n_0^2 = 2^{n_0}$ and 2^n will grow faster, $n^2 < c2^n$ for specific n_0 and c have been proved, so $n^2 = \mathcal{O}(2^n)$.

2.a $f(n) = \mathcal{O}g(n)$. For $n > n_0 = 2$

$$n\sqrt{n} \leq n * n - 1$$

Choose $c = 1$, it satisfies $f(n) = \mathcal{O}g(n)$.

3.a Such two functions do not exist. When $f(n) = \Theta(g(n))$, existn_0, c_1, c_2 such that for $n > n_0$, $c_2 g(n) \leq f(n) \leq c_1 g(n)$, so the condition of $f(n) = \mathcal{O}(g(n))$ must meet.

3.b f(n) = n^3, g(n) = n^2.

4 $f_2(n) < f_3(n) < f_1(n) < f_4(n)$

As shown in lecture $n > \sqrt{n} > logn$,

$$\lim_{n\to\infty} \frac{\sqrt{n}logn}{n\sqrt{logn}} = \lim_{n\to\infty} \frac{logn}{n} = 1 < \infty$$

so $f_2(n) = \mathcal{O}(f_3(n))$.

As $(\sqrt{x} + \sqrt{y})^2 = x + y + 2\sqrt{xy} > (\sqrt{x+y})^2$,

$$\sum_{i=1}^{n} \sqrt{i} > \sqrt{\sum_{i=1}^{n} i} > \sqrt{\frac{n^2+n}{2}} > \sqrt{n^2} > \sqrt{n}logn$$

So $f_2(n) = \mathcal{O}(f_1(n))$

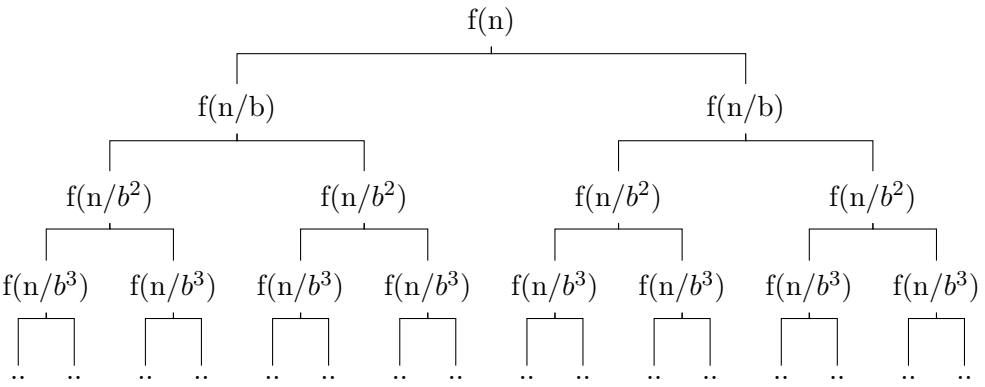
$$\sum_{i=1}^{n} \sqrt{i} < n\sqrt{n} < 24n^{3/2} + 4n + 6$$

$f_1(n) = \mathcal{O}(f_4(n))$.

Then $f_2(n) < f_3(n) < f_1(n) < f_4(n)$ has been proved.

Q2.

1.a Begin with function $f(n)$ and assume $a = 2$



1.b.1 As for each node, the input size is n/b^j. When $n/b^j = 1$, it is the depth of the tree. So depth: $log_b n$

1.b.2 From the graph, the number of leaves at each depth j is a^j.

1.b.3 at each depth j, total cost: $a^j f(n/b^j)$

1.b.4

$$T(n) = \sum_{j=0}^{log_b n} a^j f(n/b^j) = \sum_{j=0}^{log_b n-1} a^j f(n/b^j) + a^{log_b n} f(1)$$
$$= \sum_{j=0}^{log_b n-1} a^j f(n/b^j) + n^{log_b a} f(1) = \sum_{j=0}^{log_b n-1} a^j f(n/b^j) + \Theta(n^{log_b a})$$

2.a.1 From $f(n) = \Theta(n^{log_b a})$, obtain that there exists three constants n_0, c_1, c_2 such that for all $n > n_0$

$$c_1 \cdot n^{log_b a} \leq f(n) \leq c_2 \cdot n^{log_b a}$$

Then if replace n with n/b^j and multiply with a_j then calculate the sum,

$$c_1 \cdot \sum_{j=0}^{\log_b n-1} a^j (\frac{n}{b^j})^{\log_b a} \leq g(n) \leq c_2 \cdot \sum_{j=0}^{\log_b n-1} a^j (\frac{n}{b^j})^{\log_b a}$$

So

$$g(n) = \Theta\left(\sum_{j=0}^{\log_b n-1} a^j (\frac{n}{b^j})^{\log_b a}\right)$$

2.a.3 From the conclusion obtained in 2.a.2,

$$g(n) = \Theta\left(\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a}\right) = \Theta(n^{\log_b a} \log_b n)$$

2.b.2

$$\begin{aligned}\sum_{j=0}^{\log_b n-1} a^j \left(n/b^j\right)^{\log_b a-\varepsilon} &= n^{\log_b a-\varepsilon}\sum_{j=0}^{\log_b n-1} a^j b^{-j\log_b a} b^{j\varepsilon} = n^{\log_b a-\varepsilon}\sum_{j=0}^{\log_b n-1} a^j a^{-j} b^{j\varepsilon}\\ &= n^{\log_b a-\varepsilon}\sum_{j=0}^{\log_b n-1} b^{\varepsilon j} = n^{\log_b a-\varepsilon}\frac{b^{\varepsilon(\log_b n)}-1}{1-b^{\varepsilon}}\\ &= n^{\log_b a-\varepsilon}\frac{n^{\varepsilon}-1}{b^{\varepsilon}-1}\end{aligned}$$

2.b.3 From 2.b.1 and 2.b.2,

$$g(n) = \mathcal{O}\left(\sum_{j=0}^{\log_b n-1} a^j (\frac{n}{b^j})^{\log_b a-\varepsilon}\right) = \mathcal{O}(n^{\log_b a-\varepsilon}\frac{n^{\varepsilon}-1}{b^{\varepsilon}-1})$$

As

$$n^{\log_b a-\varepsilon}\frac{n^{\varepsilon}-1}{b^{\varepsilon}-1} \le n^{\log_b a-\varepsilon}\frac{n^{\varepsilon}}{b^{\varepsilon}-1} = n^{\log_b a}\frac{1}{b^{\varepsilon}-1},$$

where $\frac{1}{b^{\varepsilon}-1}$ is a constant. So,

$$g(n) = \mathcal{O}(n^{\log_b a})$$

2.c.1 From $g(n) = \sum_{j=0}^{log_b n-1} a^j f(n/b^j) = \sum_{j=1}^{log_b n-1} a^j f(n/b^j) + f(n) > f(n)$, we could conclude that $g(n) = \Omega(f(n))$.

2.c.2 From $af(n/b) \le cf(n)$, we could set $x = n/b^{j-1}$, so that

$$a^j(n/b^j) = a^{j-1}\cdot af(x/b) \le a^{j-1}\cdot cf(x) = ca^{j-1}f(n/b^{j-1})$$

So that we could repeat the steps above, until we get $f(n)$.

$$a^j f\left(n/b^j\right) \le c\cdot a^{j-1}f(n/b^{j-1}) \le c^2\cdot a^{j-2}f(n/b^{j-2}) \le \cdots \le c^j f(n)$$

2.c.3 $g(n) = \sum_{j=0}^{log_b n-1} a^j f(n/b^j) \le \sum_{j=0}^{log_b n-1} c^j f(n) \le \frac{1}{1-c} f(n)$, so $g(n) = \mathcal{O}(f(n))$

2.c.4 So to prove that there exist three constants c_1, c_2, n_0 such that for $n > n_0$, $c_2 f(n) \le g(n) \le c_1 f(n)$. As proved in 2.c.3 with $c_1 = \frac{1}{1-c}$, only need to find $c_2 f(n) \le g(n)$

$$g(n) = \sum_{j=0}^{\log n-1} a^j f\left(n/b^j\right) = af\left(n\right) + \sum_{j=1}^{\log n-1} a^j f\left(n/b^j\right) > f(n)$$

so $g(n) = \Omega(f(n))$. Therefore, $g(n) = \Theta(f(n))$.

3 Let $a \ge 1$ and $b > 1$ be constants, and n is an exact power of b. Then T(n) could be defined as

$$T(n) = \begin{cases}\Theta(1) & \text{if} \quad \text{n} = 1\\ aT(n/b) + f(n) & \text{if} \quad \text{n} = b^i\end{cases}$$

where i is a positive integer. Then from the conclusion we obtained,

- $f(n) = \Theta(n^{log_b a})$, $T(n) = g(n) + \mathcal{O}(n^{log_b n}) = \mathcal{O}(n^{log_b n} log_b n) + \mathcal{O}(n^{log_b n}) = \mathcal{O}(n^{log_b n} log_b n)$
- when $f(n) = O(n^{log_b a - \varepsilon})$, same procedure as above.
- $f(n) \geq \frac{a}{c} f(n/b)$, because $f(n) = \Omega(n^{log_b a + \varepsilon})$, take $g(n) = \Theta(f(n))$ only.

$$T(n) = \begin{cases} \Theta(n^{log_b a} log_b n) & f(n) = \Theta(n^{log_b a}) \\ \Theta(n^{log_b a}) & f(n) = O(n^{log_b a - \varepsilon}) \\ \Theta(f(n)) & f(n) \geq \dfrac{a}{c} f(n/b) \end{cases}$$

Q3.

Algorithm 1: Ramanujam numbers

Input : positive integer n

Output: all Ramanujam numbers smaller or equal to n

```
1  Function FindRamanujam(n):
2      i ← 1 ;
3      cubeList ← [ ];
4      while i < ∛n do
5          push i^3 into cubeList;
6      end while
7      sumList ← [ ];
8      for j ← 1 to ⌈∛n⌉ do
9          for k ← 1 to ⌈∛n⌉ do
10             push j + k into sumList;
11         end for
12     end for
13     for all elements in the sumList do
14         count the time they occur;
15     end for
16     result ← elements that occur twice;
17     return result
18 end
```

- Find cube for all numbers that smaller than $\sqrt[3]{n} = \mathcal{O}(n^{1/3})$
- Sum all pairs in the cubeList = $\mathcal{O}(n^{2/3})$
- Find all numbers that occur twice = $\mathcal{O}(n)$

So the total time complexity should be $\mathcal{O}(n)$.

Q4.

The six pirates is P1, P2, P3, P4, P5, P6 and P1 is the senior-most pirate.
We consider from the situation such that only one pirate survive. Because for those pirates with proper thinking abilities, they would consider all the possible distribution methods to decide whether they should vote.

1. Only P6 alive, P6 will have 300 coins.

2. P5 and P6 alive. P5 will vote YES for any distribution he proposed (1 out of 2), so any distribution method could be accepted. Then distribution should be (P5, P6) have (300, 0) respectively,

3. P4, P5 and P6 alive. As P4 knows P6 will have no coin if P4 dies, P4 will give P6 one coin. Then 2 out of 3 will vote YES, P5 will not vote YES for P4 as he always could have better income afterwards. Then distribution should be (P4, P5, P6) have (299, 0, 1) respectively.

4. P3, P4, P5 and P6 alive. P3 need to get support from P5, as P5 obtain nothing in P4's distribution. Then with 2 out of 4 YES, P3's distribution could be accepted. Then distribution should be (P3, P4, P5, P6) have (299, 0, 1, 0) respectively.

5. P2, P3, P4, P5 and P6 alive. P2 need to get support from P4 and P6, because they have nothing in P3's distribution. then 3 out of 5 YES will make it accepted. Then distribution should be (P2, P3, P4, P5, P6) have (298, 0, 1, 0, 1) respectively.

6. All alive. P1 need to get support from P3 and P5. 3 out of 6 YES will make it accepted. Then distribution should be (P1, P2, P3, P4, P5, P6) have (298, 0, 1, 0, 1, 0) respectively.

Number of alive pirates / coins for each pirates	1	2	3	4	5	6
1	\	\	\	\	\	298
2	\	\	\	\	298	0
3	\	\	\	299	0	1
4	\	\	299	0	1	0
5	\	300	0	1	0	1
6	300	0	1	0	1	0

UM-SJTU JOINT INSTITUTE

Introduction to Algorithms
(VE477)

Homework #3

Prof. Manuel

Xinmiao Yu
518021910792

Oct. 16, 2020

Q1.

1. Depth-First Search, is used to search or traverse tree or graph. It will explore edges out of the most recently discovered vertex v. Once all the edges of v has been discover, it will backtrack to the vertex that v is discovered. When all the vertex has been explored, the process will end. When use DFS for a graph, it will create a depth-first forest, composing several depth-first trees. Define each vertex initially white, when it is first discovered, grayed. When it is finished, such that all the adjacent list has been explore, blacked. Also use variable *time* to timestamp for the discovering time and finishing time.

Algorithm 1: DFS

Input : graph $G = (V, E)$
Output: depth-first forest

```
1  Function DFS(G):
2      for each vertex u ∈ G.V do
3          u.color ← WHITE
4          u.π ← NULL
5      end for
6      time ← 0
7      for each vertex u ∈ G.V do
8          if u.color == WHITE then
9              DFS-VISIT(G, u)
10         end if
11     end for
12 end
13 Function DFS-VISIT(G,u):
14     time ← time + 1
15     u.d ← time
16     u.color ← GRAY
17     for each v ∈ G.adj[u] do
18         if v.color == WHITE then
19             v.π = u
20             DFS-VISIT(G, v)
21         end if
22     end for
23     u.color ← BLACK
24     time ← time + 1
25     u.f ← time
26 end
```

2. Topological sorting is for directed acyclic graph. A linear ordering of vertices such that if a directed edge $u \leftarrow v$, then u should be before v in the ordering. It will print a vertex before its adjacent vertices while DFS will print a vertex then call DFS for its adjacency vertices. It could be easily realized through DFS. Because a vertex with smaller finish time should be

deeper in the graph, so put each finished vertex at the begin of the result sorting list then could get a topological sorting list.

Algorithm 2: Topological Sorting

Input : graph $G = (V, E)$
Output: topological sorted list

```
1 DFS(G) to compute each vertex's finish time
2 as each vertex is finished, insert it into the front of the result list
3 return list
```

3. Given a DAG, because Hamiltonian path is a path that includes every vertex, perform a topological sorting and check whether successive vertices are connect in the graph. These edges form a directed Hamiltonian path. The DFS algorithm is modified a bit, only result

that would be used for finding Hamiltonian path is kept.

Algorithm 3: Hamiltonian Path in DAG

Input : A DAG graph $G = (V, E)$

Output: whether G contains a Hamiltonian path

```
1  Function DFS(G):
2      for each vertex u ∈ G.V do
3          u.color ← WHITE
4          u.π ← NULL
5      end for
6      list ← []
7      for each vertex u ∈ G.V do
8          if u.color == WHITE then
9              DFS-VISIT(G, u, list)
10         end if
11     end for
12     return list
13 end
14 Function DFS-VISIT(G, u, list):
15     u.color ← GRAY
16     for each v ∈ G.adj[u] do
17         if v.color == WHITE then
18             v.π = u
19             DFS-VISIT(G, v, list)
20         end if
21     end for
22     u.color ← BLACK
23     insert u at the front of list
24 end
25 result ← DFS(G)
26 for i ← 0 to result.size − 1 do
27     if result[i] do not connect with result[i+1] then
28         return False
29     end if
30 end for
31 return True
```

4. Assume the graph $G = (V, E)$. Because in function `DFS`, first initialize all the vertex (line 2-5), which is in $\Theta(V)$, then the loop on line 7-11 will be performed in $\Theta(V)$ if exclude the time for `DFS-VISIT`. Using aggregate analysis, because each vertex will be coloured after discovered, so each vertex is discovered for only one time. `DFS-VISIT` is called one time for each $v \in V$. For each v, its adjacent vertex u will be invoked into this turn if it is white. So during each

`DFS-VISIT(`G, u`)`, the loop on line 16-22 is executed $|Adj[v]|$ times. As we have

$$\sum_{v \in V} |Adj[v]| = \Theta(E),$$

the total time for line 16-22 is $\Theta(E)$. Then the running time for `DFS` is $\Theta(V + E)$.

As we have V elements in $result$, the loop in line 26-30 will be executed in $\mathcal{O}(V)$. So its complexity is $\mathcal{O}(V + E)$.

5. It belongs to NP-class.

Q2.

1. No. To prove a function $f(n)$ is polynomial bounded, it is same to prove $log(f(n))$ is bounded by $\mathcal{O}(log(n^k))$, such that there exist constant c, n_0, k such that for any $n > n_0$, $log(f(n)) \leq cklogn$.

   First prove $log(n!) = \Theta(n \cdot logn)$

   $$\begin{aligned} log(n!) &= log(1) + log(2) + ...log(n) \\ &\leq log(n) + log(n) + ...log(n) = nlog(n) \\ log(n!) &\geq log(n/2) + log(n/2 + 1) + \ldots + log(n-1) + log(n) \\ &\geq log(n/2) + log(n/2) + \ldots + log(n/2) = (n/2)log(n/2) \\ &= (n/2)(logn + log(1/2)) \\ &= (n/2)logn - n/2 \geq 1/4(nlogn). \end{aligned}$$

   Then

   $$log(\lceil logn \rceil !) = \Theta(\lceil logn \rceil \cdot log(\lceil logn \rceil)).$$

   Also it is easy to get $\lceil logn \rceil = \Theta(logn)$. So we have

   $$log(\lceil logn \rceil !) = \Theta(logn \cdot log(logn)).$$

   Then prove $\lceil logn \rceil !$ is not bounded by a polynomial by contradiction. Assume it is bounded by a polynomial, then $log(\lceil logn \rceil !) = \mathcal{O}(logn)$, such that $logn \cdot log(logn) = \mathcal{O}(logn)$, which is obviously wrong. So $\lceil logn \rceil !$ is not bounded by a polynomial.

2. Yes. When $n > 1$, $log^*n = 1 + log^*logn$. log^*n is the number of log need to be applied to n to reach the fixed value 1.
   When $n > 1$, it is easy to obtain $logn < n - 1$ as $\frac{d(logn)}{dn} = \frac{1}{n} < 1 = \frac{d(n-1)}{dn}$. So

   $$loglog^*n = log(1 + log^*logn) < log^*logn.$$

3. The minimized number of weighing could be 2.

Algorithm 4: Find a lighter ball

Input : eight balls with similar size $a1, a2, ..., a8$, one of which is lighter

Output: the lighter ball $a0$

Divide the balls into three groups $(a1, a2, a3), (a4, a5, a6), (a7, a8)$
First Weighing: compare the weight of $(a1, a2, a3)$ with $(a4, a5, a6)$
if $(a1, a2, a3).weight == (a4, a5, a6).weight$ **then**
    Second Weighing: compare the weight of $a7$ and $a8$
    $a0 \leftarrow min(a7.weight, a8.weight)$
`/* assume` $(a1, a2, a3)$ `is lighter without losing generality */`
else
    Second Weighing: compare the weight of $a1$ and $a2$
    if $a1.weight == a2.weight$ **then**
        $a0 \leftarrow a3$
    else
        $a0 \leftarrow min(a1.weight, a2.weight)$
    end if
end if

Q3.

Rubik's Cube is made up of 26 mini cubes and each of them is called cubie. Turn the cube to make each face (a side of the cube) to have the same color. The center, edge and corner cubie refers to a cubie with one sticker, two stickers and three stickers respectively. Twist if a 90 or 180 degree turn of one of the six face, use F, B, U, D, L, and R designate 90-degree turns of the front, back, up, down, left, and right faces, respectively. Add ' (such that F') is a 90-degree counterclockwise "prime" twist, and add 2 (such that F2) is a 180-degree twist.

1. the beginner's method[1]. The whole step is shown in the Figure 1. It divides the cube into layers and solve each layer without messing up the pieces that already in place.

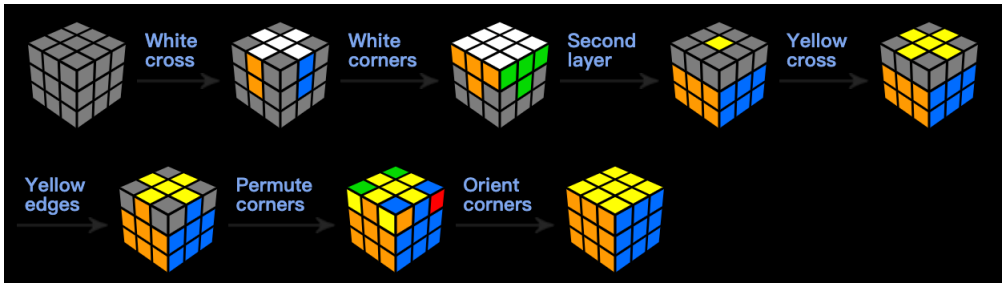


Figure 1: Step by step solution for beginner's method

   a. First layer: choose white (or any other color) to get white cross, then get white corners

to finish the first layer. An algorithm that would always be valid to bring a white corner piece to the top layer without break the solved pieces is R'D'RD.

b. Second layer: put the top layer face down now. Two algorithms to make the front-up edge to the right-front or left-front positions, called right algorithm (URU'R'U'F'UF) and left algorithm (U'L'ULUFU'F').

c. Last layer: make a yellow cross first (FRUR'U'F') then for yellow edges (RUR'URU2R'U to switch the front and left yellow edges). Premutation for the last layer (URU'L'UR'U'L) then do the orientation.

2. CFOP method [2]. The advanced Fridrich (CFOP) method still divides the problem into different layers, and apply different algorithm for each step shown in Figure 2.

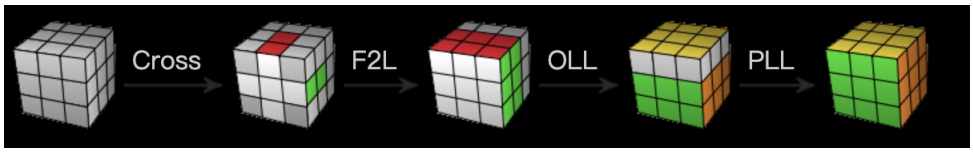


Figure 2: Steps of CFOP method

a. Complete the white cross (or choose any other color), but need to foresee 7 steps to succeed,

b. Solve the first two layers (F2L) through pairing white corner and second layer edge pieces. It could be done intuitively or memorize different algorithm for different situations.

c. Orienting the last layer (OLL). Solve the yellow face without matching the side colors.

d. Permutate the last layer (PLL).

References

[1] "How to solve the Rubik's Cube?" *Ruwix Twisty Puzzle Wiki.* `https://ruwix.com/the-rubiks-cube/how-to-solve-the-rubiks-cube-beginners-method/`

[2] "Rubik's Cube solution with advanced Fridrich (CFOP) method" *Ruwix Twisty Puzzle Wiki.* `https://ruwix.com/the-rubiks-cube/advanced-cfop-fridrich/`

Q4.

To prove a problem is in $\mathcal{NP}$, show that a certificate could be used with a verifier running in deterministic polynomial time to recognize the given problem.

1. Certificate: a simple path y. Verifier: traverse the vertices to see whether y is a simple path, running in $\mathcal{O}(|V|)$.

2. Certificate: a factor y of the given integer n. Verifier: do $n\%y$, easily implement in polynomial time.

3. Certificate: a vertex cover of size k. Verifier: For each edge $e = (u, v) \in E$, check whether either u or v is in the vertex cover, running in $\mathcal{O}(|E| \cdot k)$.

Q5.

It is **not sufficient** to prove that PRIMES is in $\mathcal{P}$. It is obvious that by dividing n using incremented integers, the process could be stopped when the integer is larger than $\sqrt{n}$ and the algorithm trivial division could be written in pseudo-code as

Algorithm 5: trivial division for PRIMES

Input : integer n

Output: whether n is a prime number

```
1 i ← 2
2 while i ≤ √n do
3 |   if n % i == 0 then
4 |   |   return True
5 |   end if
6 |   i ← i + 1
7 end while
8 return False
```

The loop will be executed for at most $\sqrt{n}$ steps, however, it does not mean PRIMES is in $\mathcal{P}$.

Prime Number Theorem: $\pi(N) \sim \frac{N}{log(N)}$, where $\pi(N)$ is the function counting the prime number smaller or less than N and $log(N)$ is the natural logarithm of N.

Then, consider a base two number a with $n - digits$. In the worst case, it will start from two and works up to the square root of a. The algorithm requires

$$\pi(2^{n/2}) \sim \frac{2^{n/2}}{\frac{n}{2} ln(2)}$$

trivial divisions.

So an algorithm with up to $\sqrt{N}$ steps respect to input N is not sufficient. An algorithm with $\lceil logN \rceil$ number of steps could prove PRIMES is in $\mathcal{P}$.

UM-SJTU JOINT INSTITUTE

Introduction to Algorithms
(VE477)

Homework #4

Prof. Manuel

Xinmiao Yu
518021910792

Oct. 23, 2020

Q1.

1. For 2^{64} operations, time is $\frac{2^{64}}{33.86 \times 10^{15}} = 544.8[s]$

   For 2^{80} operations, time is $\frac{2^{80}}{33.86 \times 10^{15}} = 3.6 \times 10^{7}[s]$

2. As in each day, we have 86400 seconds

   For 2^{64} operations, number is $\frac{2^{64}}{3.8 \times 10^{9} \times 86400} = 56186$

   For 2^{80} operations, number is $\frac{2^{80}}{3.8 \times 10^{9} \times 31 \times 86400} = 1.19 \times 10^{8}$

3. For 2^{64} bits, number is $\frac{2^{64}}{16 \times 8 \times 10^{12}} = 1.4 \times 10^{5}$

   For 2^{80} bits, number is $\frac{2^{80}}{16 \times 8 \times 10^{12}} = 9.4 \times 10^{9}$

Q2.

Algorithm 1: Algorithm R

Input : set S of size n

Output: subset S' of size k

```
1  for i ← 1 to k do
2  |  S'[k] ← S.pop()
3  end for
4  for i ← k+1 to n do
5  |  randomly choose an integer j in range [1, i]
6  |  if j ≤ k then
7  |  |  S'[j] ← S.pop()
8  |  else
9  |  |  S.pop()
10 |  end if
11 end for
```

Prove the probability of selecting each element is k/n by induction.

a. **Base Case:** the first k elements are included as the first $n = k$ members of the result set S', so the $n - seen$ result set includes each element with $\mathbf{Pr}[k/n = 1]$.

b. Assume the current k elements have already been chosen with $\mathbf{Pr}[k/n]$.

c. For the (n+1) element, it was chosen with $\mathbf{Pr}[k/(n+1)]$. Each element is in the result set S' has $\mathbf{Pr}[1/k]$ to be replaced. Then the probability that an element from the $n - seen$ result set

is replaced by the $n+1-seen$ result set is $\mathbf{Pr}[\frac{1}{k} \cdot \frac{k}{n+1} = 1/(n+1)]$. Similarly, an element is not replaced is $\mathbf{Pr}[1 - 1/(n+1) = n/(n+1)]$.

Then, the $n+1-seen$ result contains (1) either an element in the $n-seen$ result and not be replaced ($\mathbf{Pr}[k/n \cdot \frac{n}{n+1} = k/(n+1)]$), (2) or the element is chosen ($\mathbf{Pr}[k/(n+1)]$).

So the proof is complete.

Q3.

1. The algorithm is

Algorithm 2: Algorithm

Input : i-th line

Output: sum on all elements in the i-th line

```
return 3^(i-1)
```

2. Time complexity with input i: $\mathcal{O}(3^i)$.

   Prove the correctness by induction.

   a. **Base:** $i = 1$, $3^{i-1} = 1$, which is true.

   b. **Assume when $i = k-1$ it is true:** then we have $sum(k-1) = 3^{k-2}$

   c. **When $i = k$:** denote the $n-th$ number in the k row as $N_n(k)$. Then by definition,

$$N_n(k) = \sum_{x=n-1}^{n+1} N_x(k-1)$$

   Each number in the $k-1$ row will appear three times, so the sum of k row is

$$\sum_{n=1}^{2k-1} N_n(k) = \sum_{n=1}^{2k-1} \sum_{x=n-1}^{n+1} N_x(k-1) = 3 \sum_{x=0}^{2k} N_x(k-1) = 3 \cdot 3^{k-2} = 3^{k-1}$$

   Then the result consistent with the assumption, so the correctness is proved.

Q4.

Q5.

1. Given an undirected graph G and positive integer k, determine whether a set S with size k of vertices form a **clique**, such that each vertex in S is adjacent to each other in S.

2. Certificate: a set S consisting of vertices in the clique and S is a subgraph of G.
   Verifier: To check whether exists a clique of size k.

   a. $\mathcal{O}(1)$ to check the item number of S is k.

Problem 5 continued on next page...

b. Each vertex should connect to every other vertices, then edge number should be $\binom{k}{2} = \frac{k(k-1)}{2}$. So it takes $\mathcal{O}(k^2) = \mathcal{O}(n^2)$ to check.

Then we could prove the correctness in polynomial time.

3. Assume for each clause $C_1, C_2, ..., C_k$ in F, it has literals x_{j1}, x_{j2}, x_{j3}.

   To construct the graph, for each literal x_{jn}, create a distinct vertex in G. Then let G contains all the edges except

   a. joining two vertices in same clause

   b. joining two vertices whose negation are each other

   To prove F is satisfiable if and only if G has a $k-$clique.
   If F is satisfiable, at least one literal in each clause is True. Pick one such true literal from each clause. Then there will be subgraph constructed by those literals. As each pair of them could not belong to any same clause and could not be negation of each other (because they both are true), there must be edge between them. So this subgraph is a clique of size k.
   If G has a clique with size k, then this clique must have exactly one literal from each clause. Assign True to all the literals corresponding to the vertices in the clique, then F is satisfiable.

4. $\mathcal{NP}-$complete

   Because from 3., 3-SAT could be reduced to Clique in polynomial time.

Q6.

1.

2. Given an undirected graph and a number k, determine whether the graph contains an independent set of size k. A set of vertices inside a graph G is an independent set if there are no edges between any two of these vertices.

3. Certificate: A set of vertices with size k.
   Verifier: Check all vertices in the certificate are in the graph, and no edge between any two of them, run in $\mathcal{O}(V + E)$.

   So the correctness could be verified in polynomial time, it is in $\mathcal{NP}$.

4. Given a graph $G = (V, E)$ has a k-clique, set $G' = (V', E')$ as the complement of G, such that $V' = V$ but each $e \in E \rightarrow e \notin E'$.

   Then from each vertex in $S(clique)$ is adjacent to each other, when exclude all the edges in E, no vertex in S' would have any edge to connect any of other vertices in S'. So "G has a k-clique" is equivalent to "G' has an independent set of size k".

5. $\mathcal{NP}-$complete.

   Because from 3., Clique could be reduced to IND SET in polynomial time, and Clique is $\mathcal{NP}-$complete.

UM-SJTU JOINT INSTITUTE

Introduction to Algorithms
(VE477)

Homework #5

Prof. Manuel

Xinmiao Yu
518021910792

Nov. 1, 2020

Q1.

1. *Linear partition problem:* A given arrangement S consisting n nonnegative integers $s_1, ..., s_n$ and an integer k, partition S into k ranges so as to minimize the maximum sum over all the ranges.

   Often arises in parallel processing, because of the demand to balance the work done across processors so to minimize the total elapsed run time.

2. No, not a good solution. It will not systematically evaluate all the possibilities. Consider $S = \{3, 3, 3, 2, 5, 5\}$ and $k = 3$. With the this approach we have the average size of partition as $21/3 = 7$, then the set will be divided as $3, 3 \mid\mid 3, 2 \mid\mid 5, 5$ with maximum sum as 10. However, the solution should be $3, 3, 3 \mid\mid 2, 5 \mid\mid 5$ with maximum sum as 9.

3. The problem could be transformed into finding the minimum value of the lager one between 1) cost of the last partition $\sum_{j=i+1}^{n} s_j$ and 2) the cost of the largest partition cost formed to the left of i.

$$M(n,k) = min_{i=1}^{n}\ max(M[i,k-1], \sum_{j=i+1}^{n} s_j)$$

$$\text{With Basis M[1, k]} = s_1, \forall k > 0$$

$$M[n,1] = \sum_{i=1}^{n} s_i$$

4. If keep $M[i][j]\ \forall i \le n, j \le k$, total cell will be $k \cdot n$ in this table. For any $M[n'][k']$, need to find the minimum among n' quantities, each of which is the maximum through table lookup and a sum of at most n' elements. Then fill each cell need $\mathcal{O}(n^2)$. So total need $\mathcal{O}(kn^3) = \mathcal{O}(n^3)$

5. When update each cell, instead of selecting the best of up to n possible points to place the divider, each of which need to sum up to n possible terms, we could store the set of n prefixes sum

$$p[i] = \sum_{k=1}^{i} s_k, \text{ since p[i] = p[i} - 1] + \text{s}_\text{i}$$

Problem 1 continued on next page...

6. Dynamic programming approach

Algorithm 1: Linear Partition Problem

Input : arrangement S consisting n nonnegative integers $s_1, ..., s_n$ and an integer k

Output: the cost of the largest range when partition S into k ranges so as to minimize the maximum sum over all the ranges

```
   /* compute prefix sum                                                    */
 1 p[0] ← 0
 2 for i ← 1 to n do
 3 |   p[i] ← p[i − 1] + s_i
 4 end for
   /* boundary condition                                                    */
 5 for i ← 1 to n do
 6 |   M[i, 1] ← p[i]
 7 end for
 8 for j ← 1 to k do
 9 |   M[1, j] ← s_1
10 end for
   /* evaluate main recurrence                                              */
11 for i ← 2 to n do
12 |   for j ← 2 to k do
13 |   |   M[i, j] ← ∞
14 |   |   for x ← 1 to i - 1 do
15 |   |   |   s ← max(M[x, j − 1], p[i] − p[x])
16 |   |   |   if M[i,j] > s then
17 |   |   |   |   M[i, j] ← s
18 |   |   |   |   D[i, j] ← x ;                  /* used to reconstruct */
19 |   |   |   end if
20 |   |   end for
21 |   end for
22 end for
23 return M[n,k]
```

7. First the prefix sum and boundary condition is obviously true. It settle the smallest possible values for each of the arguments of the recurrence. With the evaluation order such that computes the smaller values before the bigger values, it will obtain the right result as long the previous results are true, which must be true as the boundary conditions are true.

8. When update each cell, we do not need to select the best among n possible points to place the divider because of the prefix sum we stored. So for each call, only need linear time. Then the total time complexity would be $\mathcal{O}(kn^2)$.

9. This could be achieved by D, as it record which divider position required to achieve such cost. So to reconstruct the path used to get to the optimal solution, we work backward from

$D[n,k]$ and add a divider at each specified position.

Algorithm 2: Reconstruct

Input : S, D, n, k

Output: S with divider

```
1 Function Reconstruct(S, D, n, k):
2     if k == 1 then
3         print the first partition (s_1, s_2, ..., s_n)
4     else
5         Reconstruct(S, D, D[n,k], k-1)
6         print the k − th partition (s_{D[n,k]+1}, ..., s_n)
7     end if
8 end
```

Q2.

As B would produce number $0, 1, 2, 3, 4$ with equal probability $1/5$, we could get number in range $[0, 24]$ by $B * 5 + B$ with equal probability $1/25$. And if drop the number that larger than 23, the probability for generating number in $[0, 23]$ is $1/24$. Then the number in $[0, 7]$ with equal probability could be obtained by $[0, 23]/3 = 1/8$, which returns the integer part of the result.
To extend the generation procedure, the critical part is to generate numbers larger than expected n ($[0, 24]$ previously) with equal probability.

- Denote the original B that produce $[0, 4]$ as B_0
- Denote that produce $[0, 24]$ as B_1 such that $B_1 = 5 * B_0 + B_0 = (5 + 1)$
- $B_2 = 25 * B_0 + B_1$ produce $[0, 124]$ with equal probability as $1/125$
- $B_3 = 125 * B_0 + B_2$ produce $[0, 624]$ with equal probability $1/625$

Then summarize as

$$Range[B_n] = [0, 5^{n+1} - 1], \text{ with P} = \frac{1}{5^{\text{n+1}}}$$

Therefore, we could use the original B, to have any generator B_i we need to produce number in range $[0, 5^{i+1} - 1]$. Restriction on n will be $n \geq 0$.
As in the previous example, the random number in $[0, 7]$ is obtained through $(B_1.output < 24)/3$. Because the range is $[0, 5^2 - 1] = [0, 24]$, which is too large if we just simply keep the number that $B_1.output \leq 7$. So we could apply the same method that find an integer a such that

$$a * n < 5^{i+1} - 1, \quad where\ 5^i - 1 < n \leq 5^{i+1} - 1$$

The the random number is $B_i.output/a$

Algorithm 3: Random Number Generator

Input : nonnegative integer n

Output: a random number in range $[0, n]$

Find i that $5^i - 1 < n \leq 5^{i+1} - 1$
$a \leftarrow 1$
while *(a+1)*(n+1)* $\leq 5^{i+1} - 1$ **do**
  $a \leftarrow a + 1$
end while
Get the random number generator B_i
num $\leftarrow B_i$*.output*
while *num > (n+1)*a* **do**
  num $\leftarrow B_i$*.output*
end while
return *num/a*

Q3.

Bellman-ford algorithm

Algorithm 4: Detect negative cycle

Input : weighted graph $G = (V, E)$

Output: whether the graph has negative cycle

Chosen a vertex s randomly
`/* Initialization */`
for *each vertex* $v \in G.V$ **do**
  $v.d \leftarrow \infty$
end for
$s.d \leftarrow 0$
`/* Relax */`
for $i \leftarrow 1$ **to** $|G.V| - 1$ **do**
  for *each edge* $(u, v) \in G.E$ **do**
    if *v.d > u.d + w(u,v)* **then**
      $v.d \leftarrow u.d + w(u, v)$
    end if
  end for
end for
for *each edge* $(u, v) \in G.E$ **do**
  if *v.d > u.d + w(u,v)* **then**
    return *True* ; `/* only possible when negative cycle exists */`
  end if
end for
return *False*

Q4.

Q5.

Denote the position of k internet hostspots as $loc_p = loc_1, loc_2, ..., loc_k$ and the position of n clients as $pos_c = pos_1, pos_2, ..., pos_n$.

Construct a graph with all the clients and hostspots as vertices, all the clients could be viewed as a sink nodes while hostspots as source nodes. If one of clients (t_i) could connect to one of the hostpots (s_j) (whether user in the range parameter r), add an edge with capacity $c(s_j, t_i) = 1$

Then we add a *supersource*(s) such that add edge with capacity is the load of this hostspot such that $c(s, s_j) = l_j,\ \forall 1 \leq j \leq k$. Add a *supersink*($t$) such that add edge with capacity 1 such that $c(t_i, t) = 1,\ \forall 1 \leq i \leq n$.

Then apply the **Edmonds-Karp** algorithm to this graph, with source node s and sink node t. The maximum flow (n) happens only when all the users connects to the network. Because the supersource node connect each hostpot with *capacity* == *load* and ssuperink node connect each client with capacity 1.

Algorithm 5: Wifi network

Input : r, l, loc_p (location of k hostspots), pos_c (position of n clients)

Output: whether all user could connect to the network

```
/* initialize the graph                                          */
Empty graph G ← (V, E)
for i ← 1 to n do
    for j ← 1 to k do
        dis ← (loc_j.x − pos_i.x)^2 + (loc_j.y − pos_i.y)^2
        if √dis < r_j then
            Add edge (s_j, t_i) with capacity 1 to E
        end if
        Add vertex s_j, t_i into V
    end for
end for
for j ← 1 to k do
    add edge ((s, s_j)) with capacity l_j
end for
for i ← 1 to n do
    add edge ((t_i, t)) with capacity 1
end for
f ← Edmonds − Karp(G)
return f==n
```

There are $k + n + 2$ vertices in the graph as we add all the hostspots, clients and one supersource, supersink. The maximum edge will be that all the clients could connect to all the hostpots, which is kn. Then the total number of edges are $kn + k + n$.

The three for loops took $\mathcal{O}(kn) + \mathcal{O}(k) + \mathcal{O}(n)$. Time complexity for Edmonds-Karp(G) will be same as discussed in lecture $\mathcal{O}(|V||E|^2) = \mathcal{O}((k + n + 2)(kn + k + n)^2)$

Problem 5 continued on next page...

Total time complexity: $\mathcal{O}(kn)+\mathcal{O}(k)+\mathcal{O}(n)+\mathcal{O}((k+n+2)(kn+k+n)^2) = \mathcal{O}((k+n+2)(kn+k+n)^2)$

UM-SJTU JOINT INSTITUTE

Introduction to Algorithms
(VE477)

Homework #6

Prof. Manuel

Xinmiao Yu
518021910792

Nov. 6, 2020

Q1.

1. Denote that $L = l_1, l_2, ..., l_n$ and $R = r_1, r_2, ..., r_n$, $i, j \leq n$. As the matrix A is defined as

$$a_{i,j} = \begin{cases} X_{i,j}, & (l_i, r_j) \in E, \\ 0, & (l_i, r_j) \notin E \end{cases}$$

and the expression of determinant is

$$Det(A) = \sum_{\pi \in S_n} (-1)^{sgn(\pi)} \Pi_{i=1}^{n} a_{i,\pi(i)},$$

where S_n is the set of all permutations on $[n]$ and $sgn(\pi)$ is the sign of the permutation π.

There is a one to one correspondence between a permutation $\pi \in S$ and a possibly exists perfect matching $\{(l_1, r_{\pi(1)}), (l_2, r_{\pi(2)}), ..., (l_n, r_{\pi(n)})\}$. As for not perfect matching case , $\Pi_{i=1}^{n} a_{i,\pi(i)}$ will be zero. Then we could denote the set of perfect matchings in G as P, and the determinant could be rewritten as

$$Det(A) = \sum_{\pi \in P} (-1)^{sgn(\pi)} \Pi_{i=1}^{n} X_{i,\pi(i)}$$

If G has no perfect matching, then the set $P = \emptyset$. Therefore, the determinant is identically zero.

If the determinant is identically zero, this only happens when the summation of all the term is zero, which is same as $P = \emptyset$ and no perfect matching exist.

So $Det(A)$ is identically zero if and only if G as no perfect matching.

2. Then to detect whether a perfect matching exist is same as if a multivariate polynomial of degree at most n is equivalent to 0. At most $n!$ terms will be in the $Det(A)$.

Assign random weight for each edge $\in E$ such that$w_{ij} \in \{1, 2, ..., 2m\}$, where $m = |E|$, by the isolating lemma, the minimum weight perfect matching in G will be unique with probability at least 1/2. Set each $X_{i,j} = 2^{w_{ij}}$. Let A_{ij} be the matrix obtained from A by removing the i^{th} row and j^{th} column.

Suppose there is a perfect matching P has a unique minimum weight W. Then with two lemma

- $Det(A) \neq 0$ ans the highest power of 2 that divides $Det(A)$ is 2^W.
- Edge (i, j) belongs to P if and only if $Det(A_{ij}/2^{W - w_{ij}})$ is odd.

Problem 1 continued on next page...

Algorithm 1: randomized algorithm to find a perfect matching

Input : graph $G = (V, E)$, $V = L \cup R, L \cap R = \emptyset$

Output: whether exists a perfect matching

```
Choose n^2 integers from w_ij ∈ {1, 2, ..., 2m} randomly
Compute Det(A) by Gaussian elimination
Compute adj(A) and recover each Det(B_ij)
for each edge ∈ E do
    if Det(A_ij)/2^{W−w_ij} is odd then
        add the edge to M
    end if
end for
```

3. Time complexity $\mathcal{O}(nlogn)$. If we run the algorithm for k times and output $noperfectmatching$ if and only if all says no, then the error probability is 2^{-k}.

4. *The deterministic polynomial time algorithm* is to reduce this problem to the max-flow problem as discussed in the lecture. It is not useful for parallel algorithms.

   Then this algorithm still viewed as useful, as the error probability could be reduced to a rather low level.

Reference : *web.stanford.edu/class/msande*319/*MatchingSpring*19/*lecture*01.*pdf*

Q2.

The basic idea is to hold two pointers, one $fast$ and one *slow*.

1. Find the middle one, when even number n of nodes, will return $(n/2 + 1)$-th node (assume the first one as index 1).

Algorithm 2: Find the middle node

Input : the *head* of a singly linked list

Output: the middle node of the list

```
slow ← head
fast ← head
while fast is not None do
    fast ← fast.next
    if fast is None then
        return slow
    end if
    fast ← fast.next
    slow ← slow.next
end while
return slow
```

Problem 2 continued on next page...

2. Detect whether a list contains a cycle.

Algorithm 3: Detect cycle

Input : the *head* of a singly linked list

Output: whether exists a cycle

```
slow ← head
fast ← head
while fast and slow not None do
    if fast.next is None then
        return False
    end if
    fast ← fast.next.next
    slow ← slow.next
    if fast == slow then
        return True
    end if
end while
return False
```

With the linked list of n nodes, the time complexity is $\mathcal{O}(n)$.

Q3.

1. At least n boxes, for each time obtain a coupon not appeared before.

2. From the definition of X and X_j, $X = X_1 + ... + X_j + ... + X_n$. The probability of collecting coupon j given that already obtain $j-1$ coupon is $P_j = \dfrac{n-(j-1)}{n}$. Therefore, X_j is a geometric distribution with expectation

$$E[X_j] = \frac{1}{P_j} = \frac{n}{n-j+1}$$

3. To prove $E[X] = \Theta(nlogn)$

$$\begin{aligned} E[X] &= E[X_1] + E[X_1] + ... + E[X_j] + ... + E[X_n] \\ &= \frac{n}{n} + \frac{n}{n-1} + ... + \frac{n}{n-j+1} + ... + \frac{n}{1} \\ &= n\sum_{i=1}^{n} \frac{1}{i} \approx n\int_1^n \frac{1}{x}dx \\ &= nlogn \end{aligned}$$

4. Find that for any $1 \le i < j \le n$, $E[X_j] > E[X_i]$. So more coupons have already get, more coupons need to buy for collecting a new kind of coupon.