Sample Solution for Problem Set 1

Data Structures and Algorithms, Fall 2019

September 26, 2020

Contents

1 Problem 1

This algorithm is selection sort.

SelectionSort(A)

```
1: for i = 1 to n − 1 do
2:     s ← i
3:     for j = i + 1 to n do
4:         if A[j] < A[s] then
5:             s ← j
6:     swap A[i] and A[s]
```

(a)

statement	cost	times
1	c_1	n
2	c_2	$n-1$
3	c_3	$\sum_{i=1}^{n-1} t_i$
4	c_4	$\sum_{i=1}^{n-1}(t_i-1)$
5	c_5	$0 \le t \le \sum_{i=1}^{n-1}(t_i-1)$
6	c_6	$n-1$

(b)

- $t_i = n - i$, t is depended on the elements of A.
- $\sum_{i=1}^{n-1} t_i = \sum_{i=1}^{n-1}(n-i) = \frac{n^2-n}{2}$
- $\sum_{i=1}^{n-1}(t_i - 1) = \sum_{i=1}^{n-1} t_i - (n-1) = \frac{n^2-3n+2}{2}$

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_3\left(\frac{n^2-n}{2}\right) + c_4\left(\frac{n^2-3n+2}{2}\right) + c_5 t + c_6(n-1) \\
&= \left(\frac{c_3}{2} + \frac{c_4}{2}\right) n^2 + \left(c_1 + c_2 - \frac{c_3}{2} - \frac{3c_4}{2} + c_6\right) n + (-c_2 + c_4 - c_6) + c_5 t
\end{aligned}
$$

- best-case: $t = 0$, worst-case: $t = \frac{n^2-3n+2}{2}$
- At best-case and worst-case, $T(n) = an^2 + bn + c = \Theta(n^2)$.

(c)

- Loop invariant: After the i-th loop, subarray $A[1...i]$ is sorted and $A[i]$ is the smallest element of A$[i...n]$.
- Proof:
  - Initialization: There is only one element, A[1], when $i = 1$.
  - Maintain: $A[i-1]$ is the smallest element of $A[i-1...n]$. After exchange, $A[i]$ is the smallest one of $A[i...n]$ and $A[1...i]$ is still sorted, when $i \leftarrow i + 1$.
  - Termination: $A[1...n]$ is sorted, when $i = n$.
- Correctness: Elements are exchanged only and $A[1...n]$ is sorted.

2 Problem 2

(a) $T(n) = c_1 + c_2(n+2) + c_3(n+1) = (c_2 + c_3)n + (c_1 + 2c_2 + c_3) = \Theta(n)$.

(b)

- Loop Invariant: After i-th loop, $y = \sum_{j=i}^{n} c_j x^{j-i}$.
- Proof:
  - Initialization:

$$\begin{aligned} y &= c_n + x * 0 \\ &= c_n = c_n x^0 \\ &= \sum_{j=i}^{n} c_j x^{j-i} \end{aligned}$$

    , when $i = n$.
  - Maintain:

$$\begin{aligned} y_{new} &= c_i + x * y_{old} \\ &= c_i + x * \sum_{j=i+1}^{n} c_j x^{j-(i+1)} \\ &= c_i + \sum_{j=i+1}^{n} c_j x^{j-i} \\ &= \sum_{j=i}^{n} c_j x^{j-i} \end{aligned}$$

    , when $i \leftarrow i - 1$.
  - Termination: $y = \sum_{j=0}^{n} c_j x^j$, when $i = 0$.
- Correctness: $y = \sum_{j=0}^{n} c_j x^j$ is equal to P(x).

3 Problem 3

(a) We know

$$\forall n > 0, 0 < \frac{f(n) + g(n)}{2} \leq \max(f(n), g(n)) < f(n) + g(n).$$

Therefore, by definition $\max(f(n), g(n)) \in \Theta(f(n) + g(n))$.

(b)

$$\lim \frac{(n+a)^b}{n^b} = 1.$$

Therefore, by definition $(n + a)^b \in \Theta(n^b)$.

(c) None.

4 Problem 4

$$
\begin{aligned}
1 = n^{1/\lg n} &\ll \\
\lg(\lg^* n) &\ll \\
\lg^* n = \lg^*(\lg n) &\ll \\
2^{\lg^* n} &\ll \\
\ln\ln n &\ll \\
\sqrt{\lg n} &\ll \\
\ln n &\ll \\
\lg^2 n &\ll \\
2^{\sqrt{2\lg n}} &\ll \\
(\sqrt{2})^{\lg n} &\ll \\
n = 2^{\lg n} &\ll \\
n\lg n = \lg(n!) &\ll \\
n^2 = 4^{\lg n} &\ll \\
n^3 &\ll \\
(\lg n)! &\ll \\
(\lg n)^{\lg n} = n^{\lg\lg n} &\ll \\
(3/2)^n &\ll \\
2^n &\ll \\
n\cdot 2^n &\ll \\
e^n &\ll \\
n! &\ll \\
(n+1)! &\ll \\
2^{2^n} &\ll \\
2^{2^{n+1}}
\end{aligned}
\tag{1}
$$

5 Problem 5

We may assume that queries are valid. Let S, T be two stacks.

Algorithm

- Enqueue(x): Push x to stack S.
- Dequeue(x): If T is empty, pop element from stack S and push it to stack T repeatedly, until stack S is empty. After above operations, T is non-empty, and we will pop element from T, and return it.

Correctness

We will show that following condition holds after each type of query.

- Element x arrives earlier than element y iff x is in stack T and y is in stack S, or x, y are both in stack S but y is on the top of x, or x, y are both in stack T but x is on the top of y.

Note that stack T is nonempty before we pop element from T in each dequeue operation, we can imediately see that top element in stack T arrives earliest among all remaining elements, which shows the correctness of our algorithm. Therefore, the only thing we need to prove is above property holds after query.

Base case is obvious and let's move on to the induction step. If we enqueue element x, above condition still holds as x is on the top of stack S, and all other elements remain the same. If T is empty before we do the dequeue operation, condition still holds after the first half of that operation as what we do here is actually reversing the stack and cut it to T. The second half of dequeue operation maintains above property with almost the same reason in the first case.

Therefore, we prove the correctness of our algorithm.

Time Complexity

- Worst case: $\Theta(n)$ per query, where n is the remaining elements in the "queue".
- Amortized time complexity: $\Theta(1)$ per query. Note that each element will be pushed and popped constant times.

Remark

- Algorithm will be scored according to correctness proof mainly. You don't have to achieve $\Theta(1)$ amortized complexity to get full points.
- **Do NOT simulate stack via array!** Use $S.pop(), S.push(x), S.top(), S.empty, etc.$ instead.
- Please be cautious about variants in condition statement of for-loop, especially when you have something to do with it in this loop implicitly.
- If you have implemented something you cannot see correctness from pseudocode directly, please at least leave some comments on your implementation.

6 Problem 6

We may assume that queries are valid. Let S be the original stack(in order to distinguish operations on S from operations on min-stack, we will use $S.push$ and $S.pop$ in the following analysis).

Algorithm

- push(x): If S is nonempty, let $y = S.pop()$, then push y to the stack S, and lastly push $\{x, min(y.second, x)\}$ to the stack S; otherwise, push $\{x, x\}$ to the stack S.
- pop(): Let $y = S.pop()$, return $y.first$.
- min(): Let $y = S.pop()$, push y to the stack S, and lastly return $y.second$.

Correctness

All we have to verify is that $S.top().second = \min_{x \in S} x.first$. Let's prove it by induction.

- If we do the push operation, it can be seen that $\min x, y.second$ (or x if stack is empty before push operation) is exactly the minimum value among all remaining elements after that step by induction hypothesis.
- If we do the pop operation, correctness can be seen directly by induction hypothesis.

Time Complexity

Obviously, in the worst case, time complexity of above algorithm is $\Theta(1)$ per query, as we only do constant times of push or pop operation in each type of query.

Remark

- Algorithm will be scored according to correctness proof and time complexity mainly.
- and again, **Do NOT simulate stack via array!** Use $S.pop(), S.push(x), S.top(), S.empty, etc.$ instead.
- Please check corner cases like pushing duplicate elements in a row before you hand in your homework.
- You should not assume elements are integers. To be more specific, you should not assume that operation like addition exists.

7 Problem 7

Algorithm

The data structure contains an array $A[]$ and an integer $size$. $A[]$ is initialized to empty and $size$ is initialized to 0.

Algorithm 1 add(x)

$A[size] \leftarrow x$
$size \leftarrow size + 1$

Algorithm 2 remove()

$i \leftarrow random(x) - 1$
$size \leftarrow size - 1$
swap($A[i]$,$A[size]$)
return $A[size]$

Correctness

We claim that the data structure maintains the following invariance:

invariance:$\{A[0], A[1], ..., A[size - 1]\}$ are all the elements in the queue.

initialization: $size = 0$ and $queue$ is empty, which is true.

maintaining: After function add, we plus $size$ by 1. Since $\{A[0], ..., A[size - 2]\}$ are elements before we add x to queue, and we set $A[size - 1] = x$, $\{A[0], ..., A[size - 1]\}$ are elements in queue. After function $remove$, the $\{A[0], ..., A[size - 1]\}$ are exactly all the elements in queue except $A[i]$.

To prove that we always return a uniform random element in $remove$ procedure, notice that $A[size]$ is equal to $A[i]$ in the queue, and each element in the queue has the same probability to be $A[i]$ according to the property of $random$.

Complexity

It is trivially $O(1)$ for both add and $remove$.

8 Problem 8

We first give an algorithm to solve the general problem of converting an infix expression to postfix expression. Denote the string of the infix expression as $A[]$ and n is the size of the string.

Algorithm

Create stack S initialized to empty. Set $pri[!] > pri[x] > pri[+]$.

1: **for** $i = 0$ **to** n **do**
2: **if** $A[i]$ is digit **then** Print(A[i])
3: **else**
4: **while** S is not empty and $pri\,[S.top()] \geq pri\,[A[i]]$ **do** $Print\,(S.pop())$
$S.push(A[i])$
5: **while** S is not empty **do** $Print\,(S.pop())$

Correctness

We prove it by induction. Induction hypothesis: the algorithm will turn infix expression to sufix expression correctly when the length of input is at most n.

When $n = 1$, the input is a single digit, the output is correct.

When $n > 1$, denote the last operator with the smallest priority as $A[x]$. (When there are three operator $!x+$, that means $A[x]$ is the last $+$ in the expression). Then $A[x]$ should be in the last place in postfix expression. That is true in our algorithm, since when we add $A[x]$ to the stack, it will pop all elements in stack out (because it has the smallest priority), so it will fall into the bottom of the stack. And nothing will pop it out until the fifth line of our algorithm, which will add $A[x]$ to the last place in the postfix expression.

Now consider $A[0...x-1]$ and $A[x+1...n]$ (might be empty string). They both have length smaller than n. Before $A[x]$ is push into the stack, the procedure can be seen as running our algorithm in $A[0...x-1]$, since $A[x]$ will pop all elements out, which is just like what we do in line 5. Thus, according to induction hypothesis, $A[0...x-1]$ will be turned to postfix expression correctly. $A[x+1...n]$ can also be seen as running our algorithm independently, since all elements in $A[0....x-1]$ is not in the stack, and $A[x]$ is in the bottom of the stack while no one popping it out. Thus, the final string is $postfix(A[0...x-1]) + postfix(A[x+1...n]) + A[x]$, which is correct.

Complexity

Each operator will be popped and pushed at most twice, and each digit will be looped at most once. The complexity is $O(n)$.

Remark

For this special problem (with constant type of operators), there is another algorithm to solve the problem. Simply shift the first $+$ to the last place, and shift the first x between each two $+$ to the position before the $+$. That algorithm runs in $O(cn)$ while there are c types of operators. It is not efficient if c is $\omega(1)$. Many students use this algorithm which is fine for this problem, but the algorithm describe above is more universal.

Sample Solution for Problem Set 2

Data Structures and Algorithms, Fall 2020

October 29, 2020

Contents

1 Problem 1

This upper bound can be verified using the substitution method.
Assuming that $T(n) \leq c(n-2)\lg(n-2) - d(n-2)$ for some $c, d > 0$,

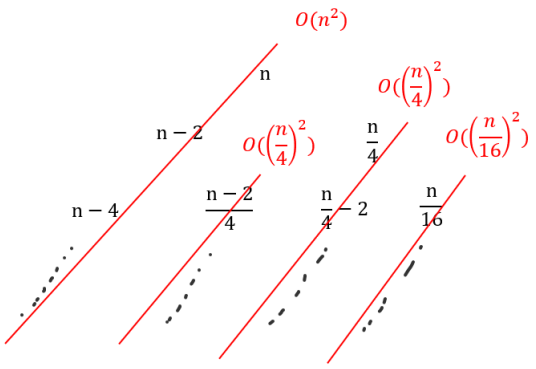
$$\begin{aligned} T(n) &= 2T(\lfloor n/2 \rfloor + 1) + n \\ &\leq 2c\left(\lfloor n/2 \rfloor - 1\right)\lg\left(\lfloor n/2 \rfloor - 1\right) - 2d(\lfloor n/2 \rfloor - 1) + n \\ &\leq c\left(n-2\right)\lg\left(n-2\right) - (d + c - 1)n - 2(d+c) \\ &\leq c(n-2)\lg\left(n-2\right) \end{aligned}$$

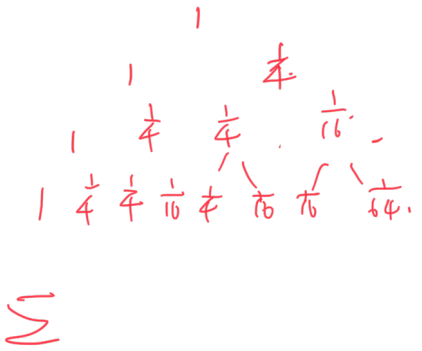
when $d > 1$. Therefore, $T(n) \in O(n \lg n)$.

2 Problem 2

(a)

Substitution method is a way to prove the recursion solution strictly. there are several ways to guess a good upper bound. Use recursion tree and sum all the layers is not a good idea for this problem since it will give you an exponential function. You can see that the recursion tree is quite unbalanced, which inspired use to sum the tree diagonally as the following figure shows.





For any i, $\left(\frac{n}{4^i}\right)^2$ will apear at most n^i times. (Each $\left(\frac{n}{4^i}\right)^2$ will generate at most n times of $\left(\frac{n}{4^{i+1}}\right)^2$). Now we get a guessed upper bound

$$\sum_{i=1}^{\log n} \left(\frac{n}{4^i}\right)^2 \cdot n^i = n^{O(\log n)}$$

The recursion tree method is equivalent to the following expansion

$$T(n) = T(n-2) + T(\frac{n}{4}) + n \leq nT(\frac{n}{4}) + n^2 = n^{O(\log n)}$$

We prove it by substition method.

Substition method: Suppose $T(k) < k^{c \log k}$ for $k < n$, then

$$T(n) = T(n-2) + T(\frac{n}{4}) + n \leq (n-2)^{c \log(n-2)} + \left(\frac{n}{4}\right)^{c \log n - 2c} + n$$

Consider $f(x) = x^{c \log(n-2)}$ which is a convex function and $f'(n-2) \geq (n-2)^{c \log(n-2)-1}$, which lead to

$$(n-2)^{c \log(n-2)} \leq n^{c \log(n-2)} - (n-2)^{c \log(n-2)-1}$$

For sufficiently large n and c, we have

$$(n-2)^{c \log(n-2)-1} \geq \left(\frac{n}{4}\right)^{c \log n - 2c} + n$$

Thus

$$T(n) \leq n^{c \log(n-2)} - (n-2)^{c \log(n-2)-1} + \left(\frac{n}{4}\right)^{c \log n - 2c} + n \leq n^{c \log n}$$

Remark 2.1. *$n^{O(\log n)}$ is not equivalent to $O(n^{\log n})$.*

(b)

$T(n) = \Theta(n \lg n)$.

3 Problem 3

(a) $(1, 5)$, $(2, 5)$, $(3, 4)$, $(3, 5)$, $(4, 5)$.

(b) The running time of insertion sort is a constant times the number of inversions. Let $\text{Inv}(i)$ denote the number of $j < i$ such that $A[j] > A[i]$. Then $\sum_{i=1}^{n} \text{Inv}(i)$ equals the number of inversions in A. Consider the while loop of the insertion sort algorithm. This loop will execute once for each element of A which has index less than j is larger than $A[j]$. Thus, it will execute $\text{Inv}(j)$ times. We reach this while loop once for each iteration of the for loop, so the number of constant time steps of insertion sort is $\sum_{j=1}^{n} \text{Inv}(j)$ which is exactly the inversion number of A.

(c) We modify the MERGE procedure of the merge-sort algorithm on page 31 of CLRS 3rd Edition.
Insert **inv** $= 0$ before the for loop at line 12.
Insert **inv** $=$ **inv** $+ n_1 - i + 1$ before line 17.
Insert **return inv** at the end of the MERGE procedure.
Then we modify the main procedure of the merge-sort algorithm on page 34. Replace line 3,4,5 with
$\text{inv}_l =$ MERGE-SORT(A, p, q)
$\text{inv}_r =$ MERGE-SORT$(A, q + 1, r)$
return MERGE$(A, p, q, r) + \text{inv}_l + \text{inv}_r$.

4 Problem 4

1. $T(n) = 2T(n/2) + cn = \Theta(n \lg n)$.

2. $\Theta(n^2)$.

$$\begin{aligned}
T(n) &= 2T(n/2) + cn + 2N = 4N + cn + 2c(n/2) + 4T(n/4) \\
&= 8N + 2cn + 4c(n/4) + 8T(n/8) \\
&= \sum_{i=0}^{\lg n - 1} (cn + 2^i N) \\
&= \sum_{i=0}^{\lg n - 1} cn + N \sum_{i=0}^{\lg n - 1} 2^i \\
&= cn \lg n + N \frac{2^{\lg n} - 1}{2 - 1} \\
&= cn \lg n + nN - N = \Theta(nN) \\
&= \Theta(n^2).
\end{aligned}$$

3. $\Theta(n \lg n)$.

$$\begin{aligned}
T(n) &= 2T(n/2) + cn + 2n/2 \\
&= 2T(n/2) + (c+1)n \\
&= \Theta(n \lg n).
\end{aligned}$$

5 Problem 5

Suppose a and b are two n-digit number. Now we have an algorithm to square a n-digit number in $T_1(n)$ time.

We can calculate $a \cdot b$ by

$$a \cdot b = \frac{a^2 - b^2 - (a-b)^2}{2}$$

$a - b$ is an $n - digit$ number. Plus and minusbetween two n-digit number cost $O(n)$ time, dividing the number by 2 cost $O(n)$ time, thus we can calculate $a \cdot b$ in

$$3 \cdot T_1(n) + O(n)$$

We claim that $T_1(n) = \Omega(n)$, since any algorithm calculating the square of a number need to use $O(n)$ to read the number. Thus,

$$3 \cdot T_1(n) + O(n) = O\left(T_1(n)\right)$$

Since $T_2(n)$ is the running time of the fastest algorithm to multiply two $n - digit$ number, we get

$$T_2(n) = O\left(T_1(n)\right)$$

Which contradict the claim $T_1(n) = o\left(T_2(n)\right)$.

Remark 5.1. *If you use $a \cdot b = (a+b)^2 - a^2 - b^2$ to calculate multiplication, the running time is acctually $T_1(n+1) + 2T_1(n) + O(n)$, since $a + b$ might be a $n + 1$ digit number. However, it is hard to show that $T_1(n+1) = O\left(T_1(n)\right)$. Acctually, the conclusion is not right, we have a counter example*

$$T_1(n) = \begin{cases} n^2 & n \text{ is even} \\ n & n \text{ is odd} \end{cases}$$

One can varify that $T_1(n) = O\left(T_1(n)\right)$ is not right. However, you might come up with some techniques to make it right given some prerequisite. That is not recommended since use $(a-b)^2$ is already an perfect way to prove it.

6 Problem 6

Key idea for this task is to maintain $\max_{i=l}^{r} \sum_{j=l}^{i} a_j$ and $\max_{i=l}^{r} \sum_{j=i}^{r} a_j$ for each segment. To simplify our solution, the algorithm we provide will NOT output index (u, v) such that $\sum_{i=u}^{v} a_i$ reaches the maximum. However, it can be done by adding some extra variables to track the corresponding segments recursively.

6.1 Algorithm

Algorithm 1 FIND-MAXIMUM-SUBARRAY(FMS)

Require: array a, left boundary l, right boundary r.
Ensure: following values in order:

- $\sum_{i=l}^{r} a_i$
- $\max_{i=l}^{r} \sum_{j=l}^{i} a_j$;
- $\max_{i=l}^{r} \sum_{j=i}^{r} a_j$;
- $\max(l, r) = \max_{l \le i <= j \le r} \sum_{k=i}^{j} a_k$;

1: **if** $l == r$ **then**
2: return $(a[l], a[l], a[l], a[l])$
3: $mid \leftarrow (l + r) >> 1$;
4: $(leftsum, leftlmx, leftrmx, leftmx) \leftarrow FMS(a, l, mid)$
5: $(rightsum, rightlmx, rightrmx, rightmx) \leftarrow FMS(a, mid + 1, r)$
6: $crossmx \leftarrow leftrmx + rightlmx$
7: $lmx \leftarrow \max(leftlmx, leftsum + rightlmx)$
8: $rmx \leftarrow \max(rightrmx, rightsum + leftrmx)$
9: return $(leftsum + rightsum, lmx, rmx, \max(leftmx, rightmx, crossmx))$

6.2 Correctness

We will prove the output values are correct during the process. Note that $crossmx$ is exactly $\max_{\substack{l \le u \le mid \\ mid < v \le r}} \sum_{i=u}^{v} a_i$ as $leftrmx$ is maximum value over all segments ending at mid and $rightrmx$ starting from $mid + 1$. And correctness of other values can be verified similarly.

6.3 Time Complexity

Note that We only use $\Theta(1)$ time during the combine step, we may write down following recurrence relation $T(n) = 2T(n/2) + \Theta(1)$, which is said to be $T(n) = \Theta(n)$ according to master theorem.

6.4 Remark

If we divide segment into two subsegments with length $n - 1$ and 1 rather than $n/2$ and $n/2$ in the divide step, above algorithm is exactly the same with Excercise 4.1-5 in CLRS. (but that can NOT be your reason to argue if your algorithm is almost the same with CLRS').

7 Problem 7

Some definitions: Suppose n friends are numbered from 1 to n, and we have an operation $query(a, b)$ for $1 \leq a \neq b \leq n$, which returns a pair (I_1, I_2) where $I_1, I_2 \in \{C, W\}$ (C means $citizen$ and W means $werewolf$), I_1 is the identity of a told by b and I_2 is the identity of b told by a.

Define function $Judge(a, A)$ where A is an array of citizens as follows:

- $cnt \leftarrow 0$
- For $i = 1$ to $A.length$ where $A[i] \neq a$ do
  - $(I_1, I_2) \leftarrow query(a, A[i])$, if $I_1 = C$ then $cnt \leftarrow cnt + 1$.
- If $cnt \geq \frac{A.length-1}{2}$ then return C. Otherwise return W.

Lemma 1. *If the number of citizens in $A[1]$ to $A[A.length]$ is larger than the number of werewolves, then $Judge(a, A)$ will correctly return the identity of a.*

Proof. If a is a citizen, then $A[1]$ to $A[A.length]$ minus a at least half of citizens, since the number of citizens is larger than the number of werewolf. These citizens will say a is citizen, which will make our algorithm output the right answer.

If a is a werewolf, then the number of citizens in $A[1]$ to $A[A.length]$ minus a is larger than the number of werewolf. Thus, less than $\frac{A.length}{2}$ friends will say a is a citizen, which will make our algorithm output the right answer. □

(a) Suppose a is the given input. Call $Judge(a, A)$. According to Lemma 1, the correctness is proved. The complexity is obviously $O(n)$.

(b) Define function $FindCitizen(l, r)$ as following:

1 If $l == r$, return $A[l]$.

2 Set $m = \lfloor \frac{l+r}{2} \rfloor$. Let $a = FindCitizen(l, m)$ and $b = FindCitizen(m + 1, r)$

3 If $Judge(a, A[l...r]) = C$, retunr a. If $Judge(b, A[l..r]) = C$, return b.

Lemma 2. *If $l \leq r$ and there are more $citizen$ than $werewolf$ in $A[l...r]$, then $FindCitizen(l, r)$ will return a real citizen.*

Proof. We prove it by induction on the length $r - l + 1$.

Base case: When $l == r$, $A[l]$ must be a citizen given that there are more $citizen$ than $werewolf$.

Induction: Suppose $l < r$ and there are more $citizen$ than $werewolf$ in $A[l...r]$. One of $A[l...m]$ and $A[m+1, r]$ must have more $citizen$ than $werewolf$ (Otherwise the sum of them will contradict the prerequisite). Since $m = \lfloor \frac{l+r}{2} \rfloor$, obviously $l \leq m < r$. According to induction hypothesis, one of a and b is $citizen$. According to Lemma 1, we can return a real citizen in step 3.

Thus, the induction hypothesis holds. □

Now by calling $FindCitizen(1, n)$, we can get a citizen.

Complexity: Suppose $FindCitizen(l, r)$ use time $T(n)$ where $n = r - l + 1$, then

$$T(n) \leq 2T(\lceil \frac{n}{2} \rceil) + O(n)$$

According to mater method, $T(n) = O(n \log n)$.

(c) I will give two method based on different ideas.

(1) The algorithm use recursion. Define function $FastFind(A)$ where A is a friends array as follows.

1 Create an empty array B index begin with 1.

2 **For**($i = 1; i \leq A.length; i+ = 2$) **do**

3 If $query(A[i], A[i+1]) = (C, C)$, add $A[i]$ to B.

4 If $i + 1 \neq A.length$, $I = Judge(A[A.length], A)$. If $I = C$, return $A[A.length]$.

5 Return $FastFind(B)$.

Lemma 3. *If there are more citizens than werewolves in A, then $FastFind(A)$ will return a real citizen.*

Proof. We prove it by induction on the length of A.

Write $n = A.length$. When $n = 1$, there are only on citizen, which will be return.

When $n > 1$, consider the case when n is odd, we varify the last friends by $Judge$. According to Lemma 1, this friends will return iff. it is citizen. Otherwise it is not considered, and the case come to when n is even. Consider the third line. It is easy to see that $query(A[i], A[i+1]) = (C, C)$ iff. $A[i]$ and $A[i+1]$ are both citizens or werewolves. Thus, if citizens are not more than werewolves in B, that must hold in A (since the number is two times the number in B), which is impossible. So B also has more citizens than werewolves. According to induction hypothesis, B will return a real citizen. □

complexity The size of B is at most a half of the size of A, thus, the complexity is

$$T(n) = T(\frac{n}{2}) + O(n)$$

which is $T(n) = O(n)$ according to master mathod.

Remark 7.1. *In the third line of the algorithm, we throw away any pair with $(C, W), (W, C), (W, W)$. However, one can pick the one that are identified as $citizen$ to put into B, which can also be proved to be correct.*

(b) The second solution is inspired by one of the student's solution (though he did it wrong). Consider the following question:

- Give you n number $A[1...n]$, the only operation you can use is $equal(i, j)$ which will tell you whether $A[i]$ and $A[j]$ are equal. Now suppose there is a number in $A[1...n]$ that appear more than $\frac{n}{2}$ times. Can you find the number in $O(n)$ time?

This is a well-known question called finding the dominant number. In this question $citizen$ is the dominant number, and you have an operation to judge whether two element is the same (just use $query$ and if (C, C) is returned, they are the same.) The only difference is that if the two numbers are both werewolves, they might not be considered the same since they might lie. However, it does not influence our solution. Now the following is the algorithm to solve finding the dominant number using stack.

1 Create a stack S. Initially, $S.push(A[1])$.

2 For $i = 2$ to n, do

– If $equal\ (A[i], S.top())$ is true, $S.push(A[i])$. Otherwise $S.pop()$.

3 Return $S.top()$.

The following loop invariance is hold

- At any time, all elements in S is equal.

We omit the proof here since it is quite simple.

Now define an element as $killed$ iff. it is popped from S or not equal to the top of S in step 2. In each case the element is $killed$ with another elements that is not equal to it. All elements that is not killed will be left in S at last. Consequently, there must be an element left in S, since there are more than half elements that is the same.

This is not a "carafully" proof! But it shows you why the algorithm runs correctly, both for the finding dominant element problem and the citizens and werewolves problem.

Sample Solution for Problem Set 3

Data Structures and Algorithms, Fall 2020

October 15, 2020

Contents

1 Problem 1

(a)

Lemma: Nodes indexed $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$ are nodes with $h = 0$.
Proof: $(\lfloor n/2 \rfloor + 1)$'s left-child should be indexed $2(\lfloor n/2 \rfloor + 1) > n$.

Mathematical Induction:

- I.B: For $h = 0$, from lemma we know there are $n - \lfloor n/2 \rfloor = \lceil n/2 \rceil$ nodes.
- I.H: There are at most $\lceil n/2^{i+1} \rceil$ nodes of height i, $i \geq 0$.
- I.S: For $h = i + 1$, there are $\lceil \frac{\lceil n/2^{i+1} \rceil}{2} \rceil = \lceil n/2^{i+2} \rceil$ nodes.

Note: The definition of **height** of a node in a heap is the number of edges on the longest simple downward path from the node to a leaf.

(b)

HeapDel(A, i)

```
1: if A[i] ≤ A[heap_size] then
2:     A[i] ← A[heap_size]
3:     MaxHeapify(A, i)
4: else
5:     HeapIncreaseKey(A, i, A[heap_size])
6: heap_size ← heap_size − 1
```

Note: If node heap_size is not a descendant of node i. A[heap_size] is probably greater than $A[i]$, in this case we need to adjust the heap upwards.

2 Problem 2

We consider the procedure of extracting all the elements from a max-heap, and prove the best running time is $\Omega(n \log n)$. Suppose the max-heap is H. For convenience we suppose all the number in H are $1, 2, ..., n$ and $n = 2^k - 1$ for some $k \geq 1$, i.e., it is a perfect tree.

Lemma 1. *Among all the number with height* 0 *in* H*, there are at least a half of the number is in* $L = \{1, 2, ..., \lceil \frac{n}{2} \rceil\}$.

Proof. Write $R = \{1, 2, ..., n\} \backslash L$. If a number with height 0 is in R, then its father is in R since the heap is a max-heap. Write the nodes at height 0 in R as R', since a number has at most 2 sons, then we have

$$|R'| + |R'|\frac{1}{2} + |R'|\frac{1}{4} + ... + 1 \leq |R|$$

For sufficiently large n, we have $|R'| \leq \frac{|R|}{2}$. Thus, the lemma is proved. □

Write L' as the nodes with height 0 in L. With the above lemma we know that L' is at least around $\frac{n}{4}$. Consider the first half popping operation, those nodes in L' will not be popping since they are the half smallest number. But they will be exchange to the top of the heap and go down during the procedure. Suppose after the first $\frac{n}{2}$ popping operation, and number i in L' is at position with distance $D(i)$ to the root. Obviously the complexity is at least

$$\sum_{i \in L'} D(i)$$

Since there are at most 2^i nodes with distance i to the root, and $|L'|$ is at least $\frac{n}{4}$, the amount is at least

$$\sum_{i \leq \log \frac{n}{8}} i \cdot 2^i = \Omega(n \log n)$$

Remark 1. *When* n *is not the power of* 2*, we can only consider the running time after the first layer is popping out (and the tree become a perfect tree again).* n *will be at least half of the origin* n*, but the constant is omit in* Ω*.*

Similarly, during the procedure we do not care much about constant, since they are all contained in Ω*.*

3 Problem 3

(a)

$$\begin{bmatrix} 2 & 3 & 4 & \infty \\ 5 & 8 & 9 & \infty \\ 12 & 14 & 16 & \infty \\ \infty & \infty & \infty & \infty \end{bmatrix}$$

(b)

- ExtractMin()

  (1) Record $A[1,1]$ then replace it with ∞. Let $i = 1$ and $j = 1$.

  (2) If $i + 1 > n$ or $A[i, j + 1] < A[i + 1, j]$ goto (2.1), else goto (2.2).

  (2.1) Swap $A[i, j]$ with $A[i, j + 1]$, $i \leftarrow i + 1$. Goto (3).

  (2.2) Swap $A[i, j]$ with $A[i + 1, j]$, $j \leftarrow j + 1$. Goto (3).

  (3) If $i = n$ and $j = m$ then **return** $A[1, 1]_{old}$, else goto (2).

- Prove:

  – Loop invariant: Before step (2), $A[1 : i, 1 : j]$ is a magic matrix.

  – Initialization: There are only one element, $A[1, 1]$.

  – Maintain:

  ∗ If (2.1) run, j-th column is sorted;

  ∗ else (2.2) run, i-th row is sorted.

  – Termination: When $i = n$ and $j = m$, all rows and column are sorted. Therefore $A[1 : n, 1 : m]$ is a magic matrix.

(c)

- InsertMatrix(k)

  (1) $A[n, m] \leftarrow k$. Let $i = n$ and $j = m$.

  (2) If $i = 1$ or $A[i, j - 1] > A[i - 1, j]$ goto (2.1), else goto (2.2).

  (2.1) Swap $A[i, j]$ with $A[i, j - 1]$, $i \leftarrow i - 1$. Goto (3).

  (2.2) Swap $A[i, j]$ with $A[i - 1, j]$, $j \leftarrow j - 1$. Goto (3).

  (3) If $i = 1$ and $j = 1$ then **return**, else goto (2).

- Prove:

  – Loop invariant: Before step (2), $A[i : n, j : m]$ is a magic matrix.

  – Initialization: There are only one element, $A[n, m]$.

  – Maintain:

  ∗ If (2.1) run, j-th column is sorted;

  ∗ else (2.2) run, i-th row is sorted.

  – Termination: When $i = 1$ and $j = 1$, all rows and column are sorted. Therefore $A[1 : n, 1 : m]$ is a magic matrix.

(d)

- Run InsertMatrix n^2 times to build a magical matrix, then run ExtractMin n^2 times. The results are arranged in a new array, which is sorted.
- Prove: Each time we run ExtractMin, we got the minimum of magical matrix. The i-th value we got is the i-th smallest number
- Time Complexity: $\Theta(n^2) \times (O(n+n) + O(n+n)) = O(n^3)$.

(e)

- FindMatrix(k)
  (1) Let $i = 1$ and $j = m$.
  (2) If $A[i, j] = k$, **return** $true$.
  (3) If $i = n$ and $j = 1$, **return** $false$.
  (4) If $i = n$ or $A[i, j] < k$, $j \leftarrow j - 1$
  (5) If $j = m$ or $A[i, j] > k$, $i \leftarrow i + 1$.
  (6) Goto (2).
- Prove:
  - Loop invariant: After step (4) and (5), k cannot be in $A[1 : i, j : m]$.
  - Initialization: From step (2), $A[1, m] \neq k$.
  - Maintain:
    * If $A[i, j] < k$, $A[1, j-1] \leq A[i, j-1] \leq A[i, j] < k$, k cannot be in $A[1 : i, j - 1]$;
    * If $A[i, j] > k$, $A[i+1, m] \geq A[i+1, j] \leq A[i, j] > k$, k cannot be in $A[i + 1, j]$.
  - Termination: When $i = n$ and $j = 1$, k cannot be in $A[1 : n, 1 : m]$.
- i only goes up, j only goes down. That is $O(n + m)$.

4 Problem 4

(a) We'll use the substitution method to show that the best-case running time is $\Omega(n \lg n)$. Let $T(n)$ be the best-case time for the procedure QUICKSORT on an input of size n. We have

$$T(n) = \min_{1 \le q \le n-1} (T(q) + T(n-q-1)) + \Theta(n).$$

Suppose that $T(n) \ge c(n \lg n + 2n)$ for some constant c. Substituting this guess into the recurrence gives

$$\begin{aligned}
T(n) &\ge \min_{1 \le q \le n-1} (cq \lg q + 2cq + c(n-q-1) \lg(n-q-1) + 2c(n-q-1)) + \Theta(n) \\
&= (cn/2) \lg(n/2) + cn + c(n/2-1) \lg(n/2-1) + cn - 2c + \Theta(n) \\
&\ge (cn/2) \lg n - cn/2 + c(n/2-1)(\lg n - 2) + 2cn - 2c\Theta(n) \\
&= (cn/2) \lg n - cn/2 + (cn/2) \lg n - cn - \lg n + 2 + 2cn - 2c\Theta(n) \\
&= cn \lg n + cn/2 - \lg n + 2 - 2c + \Theta(n).
\end{aligned}$$

Taking a derivative with respect to q shows that the minimum is obtained when $q = n/2$. Taking c large enough to dominate the $-\lg(n) + 2 - 2c + \Theta(n)$ term makes this greater than $cn \lg n$, proving the bound.

(b) In the worst case, the number of calls to RANDOM is

$$T(n) = T(n-1) + 1 = n = \Theta(n).$$

As for the best case,

$$T(n) = 2T(n/2) + 1 = \Theta(n).$$

5 Problem 5

a. Since all elements are equal, RANDOMIZED-QUICKSORT always returns $q = r$. We have recurrence $T(n) = T(n-1) + \Theta(n) = \Theta(n^2)$.

b. **PARTITION'**(A, p, r)

```
1  x = A[p]
2  l = h = p
3  FOR j = p + 1 → r
4      IF A[j] < x
5          y = A[j]
6          A[j] = A[h + 1]
7          A[h + 1] = A[l]
8          A[l] = y
9          l = l + 1
10         h = h + 1
11     ELSE IF A[j] = x
12         exchangeA[h + 1] with A[j]
13         h = h + 1
14 RETURN (l, h)
```

Unstable. Since there is only one for-loop in this procedure, it's easy to prove that it takes $\Theta(r - p)$ time, and is in-place.

c. $\Theta(n)$

6 Problem 6

(a)

$p_i = \frac{(i-1)(n-i)}{\binom{n}{3}} = \frac{6(i-1)(n-i)}{n(n-1)(n-2)}$

(b)

$\frac{p_m}{p'_m} = \frac{6(m-1)(n-m)/n(n-1)(n-2)}{1/n} = \frac{6(m-1)(n-m)}{(n-1)(n-2)} \to \frac{3}{2}, \quad n \to \infty \ (m = \lfloor \frac{n+1}{2} \rfloor).$

(c)

$$\begin{aligned} P(\text{good pivot is NOT chosen}) &= 2\sum_{i=1}^{m} p_i \\ &= \frac{12}{n(n-1)(n-2)}\left(-\frac{m(m+1)(2m+1)}{6} + \frac{(n+1)m(m+1)}{2} - nm\right) \\ &= \frac{12m}{n(n-1)(n-2)}\left(-\frac{2m^2+3m+1}{6} + \frac{(3m+1)(m+1)}{2} - 3m\right) \\ &= \frac{2(7m^2-9m+3)}{3(3m-1)(3m-2)} \end{aligned}$$

where $m = \frac{n}{3}$. Let n tends to infinity, $Pr(\text{good pivot is NOT chosen}) \to \frac{14}{27}$, which is greater than unmodified quicksort with failure probability $\frac{2}{3}$.

(d)

No. Reasonable answers will be accepted [1].

7 Problem 7

(a)

The recursion function for $T(n)$ can be seen directly.

$$T(n) = 3T\left(\frac{2}{3}n\right) + \Theta(1)$$

Reserve $\alpha = \frac{1}{1-\log_3 2}$. Let's prove $T(n) = \Theta(n^\alpha)$. Without loss of generality, we will only prove the asymptotic upper bound via substitution method.

Suppose $T(n) \leq 3T\left(\frac{2}{3}n\right) + C$ for some constant C. We will now claim that $T(n) \leq C'n^\alpha - Cn$ holds for some sufficient large constant C'.

Base case is obvious, and note that

$$T(n) \leq 3T\left(\frac{2}{3}n\right) + C \leq 3C'\left(\frac{2}{3}\right)^\alpha n^\alpha - 2Cn + C \leq C'n^\alpha - Cn$$

which ends the proof.

[1] for example, running time of modified quicksort is more stable than original one.

(b)

We will prove the statement by induction on n, the size of array A.

Base case ($n = 2$) is obvious. Let's assume that statement is correct for all $n \leq k$, and our goal is to prove statement holds for $n = k + 1$. By induction hypothesis, array $A[0, \ldots, m-1]$ is sorted after execution of line 5, where $m = \lceil \frac{2}{3} n \rceil$, indicating that $A[n-m, n-1]$ contain i-th largest (if there exist elements with equal value, we will say the element with larger index is larger) element after executing line 5 for all $1 \leq i \leq n - m$ [2]. Hence, the i-th largest element will locate in $A[n-i]$ for $1 \leq i \leq n-m$ after executing line 6. The execution of line 7 simply sort the first m elements, which completes our proof.

(c)

No. Consider $A = [1, 4, 2, 3]$. It can be seen directly that modified algorithm fails to sort this given array.

(d)

Note that each swap will lower the number of inversions by exactly 1. Therefore, the number of swaps is equal to number of inversions, which is bounded by $\binom{n}{2}$.

[2] To be more specific, if i-th largest element is located in $A[0, n-m-1]$ after execution of line 5, then $i > m - (n-m) = 2m - n \geq n - m$ due to the fact $A[0, \ldots, m-1]$ is sorted

8 Problem 8

(a) n.

Proof. There are at most n loops so the worst case is at most n.
Consider the case $n, n-1, ..., 1$, it is exactly n. □

(b) Line 4 is executed in loop i is the same event as $A[i]$ is the smallest in $A[1, ..., i]$. The probability is $\frac{1}{i}$. For the n-th loop, it is $\frac{1}{n}$.

(c) Write I_i as the **Indicator random variable** that takes 1 if line 4 is executed in loop i, otherwise take 0. Now we have $E[I_i] = \Pr[I_i = 1] = \frac{1}{i}$, and the total time of execution is $\sum_{i\in[n]} I_i$, according to linear of expectation, $E[\sum_{i\in[n]} I_i] = \sum_{i\in[n]} E[I_i] = \sum_{i\in[n]} \frac{1}{i}$.

(a) n-1.

Proof. There are at most n loops and the first loop will not call line 7 since $A[1]$ can not be larger than infinity. So the worst cast is at most $n-1$.
Consider the case $n, n-1, ..., 1$, it is exactly $n-1$. □

(b) Line 7 is executed in loop i is the same event as $A[i]$ is the second smallest in $A[1, ..., i]$. The probability is $\frac{1}{i}$ for $i \geq 2$. For the n-th loop, it is $\frac{1}{n}$ if $n \geq 2$, it is 0 if $n = 1$.

(c) Write I_i as the **Indicator random variable** that takes 1 if line 7 is executed in loop i, otherwise take 0. Now we have $E[I_i] = \Pr[I_i = 1] = \frac{1}{i}$ for $i \geq 2$, and the total time of execution is $\sum_{i\in[n]} I_i$, according to linear of expectation, $E[\sum_{i\in[n]} I_i] = \sum_{i\in[n]} E[I_i] = \sum_{2\leq i\leq n} \frac{1}{i}$.

9 Problem 9

Algorithm: Suppose the input is an array P with n points. All array in this problem is indexed begin with 1. For a point $p \in P$, write $p.x, p.y$ as the coordinate of p. We first use $O(n \log n)$ to sort P by the value of x.

We describe a function $FindClosest(l, r, Q)$ where Q is a array of points, and is exactly the points in $P[l...r]$ sorted by y.

1 If $l == r$, return (nul, nul, ∞).

2 Let $m = \lfloor \frac{l+r}{2} \rfloor$. Create two array Q_L and Q_R. For $i = 1$ to $r - l$ do

- If $Q[i].x \leq P[m].x$, then add $Q[i]$ to Q_L. Otherwise add it to Q_R.

3 Let $(p_L, q_L, d_L) \leftarrow FindClosest(l, m, Q_L)$ and $(p_R, q_R, d_R) \leftarrow FindClosest(m + 1, r, Q_R)$. Compare d_L and d_R and take the tuple that contain the smaller one to be (p, q, d)

4 Create a new array Q', For $i = 1$ to $r - l$ do

- If $P[m].x - d \leq Q[i].x \leq P[m].x + d$, then add $Q[i]$ to Q'.

5 For $i = 1$ to $Q'.size$ do

- Let the smallest distance between $Q'[i]$ and $Q'[i + 1], ..., Q'[\min(Q'.size, i + 7)]$ be d', if $d' < d$ then update $(p, q, d) \leftarrow (Q'[i], q', d')$ where q' is the point that we picked with the smallest distance to $Q'[i]$ among $Q'[i + 1], ..., Q'[i + 7]$.

6 Return (p, q, d).

The algorithm simply sort P by y to create Q initially, and call $FindClosest(1, n, Q)$ to get the answer.

Correctness:

Lemma 2. *If P is sorted by x and Q is exactly the points in $P[l..r]$ sorted by y, then $FindClosest(l, r, Q)$ will return the closest tuple bewteen points in $P[l..r]$.*

Proof. We prove it by induction on $r - l + 1$. When $r == l$, the lemma is true since there are no points pair.

Now suppose $r > l$. According to step 2 and Q is sorted by y, we know that Q_L and Q_R are both sorted by y. Thus, the recursion in step 3 will return correct answers. i.e., after step 3, (p, q, d) is the closest tuple between nodes both in Q_L or both in Q_R.

The following claim will help us prove the correctness.

Claim 2.1. *If there are two node in Q_L and Q_R seperately that has distance smaller than d, the pair will be find in step 5*

Proof. Suppose the point pair is (p, q) where $p \in Q_L$ and $q \in Q_R$. Then we have $|p.x - q.x| < d$ and $|p.y - q.y| < d$. Thus, $p, q \in Q'$. We now prove that there are at most 6 points between p and q in Q' as Q' is sorted by y: considet the rectangle between $P[m].x - d$,$P[m].x + d$ and between $p.y, q.y$. We can split the rectangle into 8 rectangle with length of side at most $\frac{d}{2}$, each is contained in space $x \leq P[m].x$ or in space $x > P[m.x]$(since the rectangle has length $2n$ and width n). Each rectanle will contain at most 1 points, otherwise the two points will form a pair that create a distance smaller than d that voilate the recursion result. Thus, there are at most 6 points between p and q (there are at most 8 points in the rectangle). So the claim is proved. □

With this claim, $FindClosest(l, r, Q)$ will return the closest tuple between points in $P[l..r]$. □

Complexity: The complexty for n points has recursion

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

According to master method, $T(n) = O(n \log n)$.

Remark 2. *When $T(n) = 2T\left(\frac{n}{2}\right) + O(n \log n)$, be careful that master method do not cover this case. You can use recursion tree method and substitution method to prove it is $O(n \log^2 n)$.*

Sample Solution for Problem Set 4

Data Structures and Algorithms, Fall 2020

October 21, 2020

Contents

1 Problem 1

(a) This statement ignores all other possible algorithms to solve this problem, so it is not the lower bound of the problem.

(b) Since length of each subsequece is k, there are $(k!)^{n/k}$ possible output permutations. To compute the height h of the decision tree, we must have $(k!)^{n/k} \leq 2^h$. Taking logs on both sides, we know that

$$\begin{aligned} h &\geq \frac{n}{k} \times \lg(k!) \\ &\geq \frac{n}{k} \times \left(\frac{k \ln k - k}{\ln 2} \right) \\ &= \frac{n \ln k - n}{\ln 2} \\ &= \Omega(n \lg k). \end{aligned}$$

2 Problem 2

(a) The algorithm will begin by preprocessing exactly as COUNTING-SORT does in lines 1 through 9, so that $C[i]$ contains the number of elements less than or equal to i in the array. When queried about how many integers fall into a range $[a..b]$, simply compute $C[\min(b,k)] - C[a-1]$ (when $a = 0$, simply use $C[\min(b,k)]$). This takes $O(1)$ times and yields the desired output.

b. Group the integers by the number of digits ($O(n)$), and use radix sorting algorithm to sort in each group ($O(n)$).

3 Problem 3

(a)

$$\begin{aligned}
& \frac{\sum_{j=i}^{i+k-1} A[j]}{k} \leq \frac{\sum_{j=i+1}^{i+k} A[j]}{k} \\
\Leftrightarrow \quad & \sum_{j=i}^{i+k-1} A[j] \leq \sum_{j=i+1}^{i+k} A[j] \\
\Leftrightarrow \quad & \sum_{j=i+1}^{i+k-1} A[j] + A[i] \leq A[i+k] + \sum_{j=i+1}^{i+k-1} A[j] \\
\Leftrightarrow \quad & A[i] \leq A[i+k]
\end{aligned}$$

(b)

- Algorithm: Split the array into k sub-arrays of size n/k. ($A_i = a_i, a_{i+k}, a_{i+2k}, \ldots; i = 1, 2, \ldots, k$. Then use MergeSort to each sub-arrays.
- Complexity: $O(k\left(\frac{n}{k}\log\frac{n}{k}\right)) = O(n\log\frac{n}{k})$.

(c)

- Algorithm: Split the array into k sorted lists. Construct a min-heap from the heads of each of the k list. Pop a list each time, extract its first element, then push the list to the heap. Each operation extract the current smallest element.
- Complexity:

	Split:	$O(n)$
	Build-heap:	$O(k)$
For each operation,		
	Pop:	$O(\lg k)$
	Extract:	$O(1)$
	Push:	$O(\lg k)$

  In total $O(n + k + n\,(\lg k + 1 + \lg k)) = O(n \lg k)$.

(d)

- Assume we can k-sort a array in $T(k, n)$. According to (c), we can sort it in $O(n \lg k)$. Therefore we can sort an arbitrary array in $T(k, n) + O(n \lg k)$, which needs to satisfy the lower bound of the comparison sort.
- Because k is a constant, $T(k, n) + O(n \lg k) \geq \Omega(n \lg n)$ implys $T(k, n) \geq \Omega(n \lg n)$.

4 Problem 4

Suppose the array is $A[1...n]$. For each element $A[i]$ create an empty stack $S[i]$ initialy. Then do the following procedure to find the smallest element in A. Create array $B[1...n]$ to store the index of elements in $A[i]$. Initially set $B[1...n] = \{1, 2, ..., n\}$:

- For $k = 0$ to $(\log n) - 1$ do
  - $N \leftarrow \frac{n}{2^k}$
  - For $i = 1$ to $\frac{N}{2}$ do
    - $S[B[2i-1]].push(A[2i])$, $S[B[2i]].push(A[2i-1])$.
    - If $A[2i-1] < A[2i]$, then $A[i] = A[2i-1]$, $B[i] = 2i-1$. Else, $A[i] = A[2i]$, $B[i] = 2i$.
- Return the smallest element in $S[B[1]]$ (In $S[B[1]].size - 1$ comparisons).

Intuitively, our algorithm first use $n-1$ comparisons to find the smallest element by pairing up, while storing all the elements that have been compared to $A[i]$ in $S[i]$. Then the smallest element that have been compared to the smallest element is the second smallest element.

Correctness and complexity: The following claim is trivial and helpful to prove the correctness.

Claim 0.1. *$S[B[1]]$ has stored all the element that compared to the smallest element during the algorithm.*

Claim 0.2. *The size of $S[B[1]]$ is $\log n$.*

To prove the second smallest element is in $S[B[1]]$, notice that any other element must be thrown out by another element that smaller than it. But the "another element" can not be the smallest element, which means the element is not the second smallest element in $A[1...n]$. Thus, by finding the smallest element that have compared to the smallest element, we get the second smallest element.

5 Problem 5

5.1 (a)

- Divided into groups of 7:
  - At least $\frac{2n}{7}$ numbers are less than *median of medians*.
  - Recursion: $T(n) \leq T(\frac{5n}{7}) + T(\frac{n}{7}) + O(n)$.
  - Assume that $T(n) \leq cn$, $O(n) \leq c'n$,
  - $T(n) \leq c\frac{5n}{7} + c\frac{n}{7} + c'n = c\frac{6n}{7} + c'n$. Just need $c \geq 7c'$.
  - The method is linear.
- Divided into groups of 3:
  - At least $\frac{n}{3}$ numbers are less than *median of medians*.
  - Recursion: $T(n) \leq T(\frac{2n}{3}) + T(\frac{n}{3}) + O(n)$.
  - Assume that $T(n) \leq cn$, $O(n) \leq c'n$,
  - $T(n) \leq c\frac{2n}{3} + c\frac{n}{3} + c'n = cn + c'n$. No solution.
  - The method is not linear.

5.2 (b)

- Algorithm: Divide and conquer method. Find the midpoint recursively. Call Quantiles($A, 1, n, k$).

Algorithm 1: Quantiles(A, l, r, k)

```
1 mid=(l+r-1)/2;
2 if k > 1 then
3 |   QuickSelection(A, l, r, (r − l + 1)/2);
4 |   Quantiles(A, l, mid, k/2);
5 |   Quantiles(A, mid + 1, r, k/2);
6 end
```

- Correctness:
  - I.H: When $k = 2^m$, $m \geq 1$, after the algorithm, i-th quantile of $A[l : r]$ is at $A[(l-1) + \frac{i(r-l+1)}{k}]$, $1 \leq i \leq k-1$.
  - I.B: When $k = 2^1$, this algorithm is the same as QuickSelection. The only quantile is at $A[r-l+1] = A[(l-1) + \frac{r-l+1}{2}]$.
  - I.S: When $k = 2^{m+1}$,
    - After QuickSelection($A, l, r, (r-l+1)/2$), the only quantile is at $A[\frac{r-l+1}{2}] = A[(l-1) + (r-l+1)\frac{1}{2}]$.
    - After Quantiles($A, l, mid, k/2$), i-th quantile of $A[l : mid]$ is at $A[(l-1)+\frac{i(mid-l+1)}{k/2}] = A[(l-1) + (r-l+1)\frac{i}{k}]$, $1 \leq i \leq \frac{k}{2} - 1$.
    - After Quantiles($A, mid+1, r, k/2$), i-th quantile of $A[mid+1 : r]$ is at $A[mid + \frac{i(r-mid)}{k/2}] = A[(l-1) + (r-l+1)\frac{\frac{k}{2}+i}{k}]$, $1 \leq i \leq \frac{k}{2} - 1$.
    - Combine the three cases, i-th quantile of $A[l : r]$ is at $A[(l-1) + (r-l+1)\frac{i}{k}]$, $1 \leq i \leq k-1$.

- Complexity:
  - Each layer is $O(n)$ for all $(\lg k)$ layers. $T(n) = O(n \lg k)$.

6 Problem 6

TL;DR: Use divide and conquer technique with SELECT oracle to solve a generalized version of this task.

6.1 Generalization

In the following, we will provide an algorithm which receives array a, w and a threshold t as input, and outputs element a_k such that $\sum_{a_i<a_k} w_i < t$ and $\sum_{a_i>a_k} w_i \leq \sum_i w_i - t$.

6.2 Algorithm

Algorithm is attached in the Appendix page.

6.3 Correctness

We would like to show that our algorithm runs correctly for input with threshold $t > 0$ and $t \leq \sum_i w_i$. And without any doubt, we will prove it by induction.

Base case holds trivially. If element array a consists of n elements, the algorithm will use SELECT oracle to select the $(1 + \lfloor \frac{n}{2} \rfloor)$-th smallest element, and divide the whole array into two parts, smaller part (whose element are smaller than this selected element) and larger part as algorithm suggested. If the total weight of the smaller part is greater or equal than t, It can be seen that our weighted median should be less than this selected element. Another part can be seen similarly, and that concludes our proof.

6.4 Time Complexity

$T(n) = T(\frac{n}{2}) + O(n)$ implies $T(n) = O(n)$ in the worst case.

7 Problem 7

7.1 Uniqueness

We may prove it by induction on the number of vertices in the given tree. Indeed, it almost tells the algorithm which may constructs the tree via pre-order and in-order numbers.

Base case trivially holds. Given a tree with n vertices, we may need the following easy but powerful observations to conclude our proof and provide our algorithm.

- pre-order(resp. in-order) numbers in a given subtree form a sequence of continous integers.
- Given pre-order numbers of a subtree which is exactly $L \leq i \leq R$, the pre-order number of the root of this subtree is L.
- Given in-order numbers of a subtree, the root's in-order number x is exactly the size of root's left subtree plus the number of vertices which are not in the subtree but in the left side[1] of the subtree.

We may derive our proof using those observations. Due to last two observations, we may know that the left-child of root has pre-order number 1, and right-child has $x + 1$. And our proof concludes immediatly after applying induction hypothesis and the first observation.

7.2 Algorithm

We may assume that rev_pre and rev_in array are given in the input section where $rev\_pre[pre[i]] = i$ for all $0 \leq i < n$ (resp. in). In fact, this can be precomputed in $O(n)$ easily. To present this given tree, we will provide a $parent$ array, representing the parent of vertice i. Specifically, $parent[i] = -1$ if vertice i is root. Initially, $parent[i] = -1$ for all $0 \leq i < n$. Algorithm is attached in the Appendix section.

7.3 Correctness

Proof of correctness is almost the same with previous proof. We will omit the detailed proof here.

7.4 Time Complexity

$T(n) = T(m) + T(n - m - 1) + O(1)$ implies $T(n) = O(n)$.

[1] To be more precise, u is on the left side of this given subtree if and only if there exists an ancestor v of this given subtree so that u is on the left subtree of v and this given tree lies on the opposite side.

8 Problem 8

(a) Suppose the pre-, post-, and in-order of u is $u.pre$, $u.post$, $u.in$ separately. The following algorithm will return the size of subtree rooted at u:

- If $u.pre == 1$, return $u.post$.
- If $u == (u.parent).leftchild$, return $((u.parent).rightchild).pre - u.pre$.
- Else, return $u.post - ((u.parent).leftchild).post$.

(b) Suppose the BFS order of a tree T with n nodes is $(v_1, v_2, ..., v_n)$. We encode the tree T into a $\{0, 1\}$ string S_T (indexed begin with 1) with length $2n$ by the following procedure:

- If v_i has left-child, then let $S_T[2i - 1] = 1$, otherwise let $S_T[2i - 1] = 0$.
- If v_i has right-child, then let $S_T[2i] = 1$, otherwise let $S_T[2i] = 0$.

To decode a string into a tree, simply construct the tree layer by layer, i.e., see the code for the first i-th layer and construct the (i+1)-th layer, using the information of left and right children in the i-th layer.

Appendix

Algorithm 2: Weighted Median Search

Input: element array a of length n, weight array w of length n, and threshold t
Output: element a_k such that above requirement satisfied

assert $t > 0$ and $t \leq \sum_i w_i$;
if $n == 1$ **then**
  return a_1;
end
$median = SELECT(a, 1 + n/2)$;
$small\_element = [], large\_element = []$;
$small\_weight = [], large\_weight = []$;
$sum\_small\_weight = 0$;
for $i \leftarrow 1$ ***to*** n **do**
  if $a_i < median$ **then**
    $small\_element.append(a_i)$;
    $small\_weight.append(w_i)$;
    $sum\_small\_weight+ = w_i$
  end
  else
    $large\_element.append(a_i)$;
    $large\_weight.append(w_i)$;
  end
end
if $sum\_small\_weight < t$ **then**
  return $WMS(large\_element, large\_weight, t - sum\_small\_weight)$;
end
else
  return $WMS(small\_element, small\_weight, t)$
end

Algorithm 3: Tree Reconstruction(TR)

Input: array pre, in, rev_pre, rev_in;
integers $left = 0$, $right = n - 1$, $shift = 0$(default value)
Output: array $parent$

```
if l == r then
    return;
end
root = rev_pre[l];
sz = in[root] − shift;
if sz > 0 then
    left_child = rev_pre[l + 1];
    parent[left_child] = root;
    TR(pre, in, rev_pre, rev_in, l + 1, l + sz + 1, shift);
end
if l + sz < r then
    right_child = rev_pre[l + sz + 1];
    parent[right_child] = root;
    TR(pre, in, rev_pre, rev_in, l + sz + 1, r);
end
```

Sample Solution for Problem Set 5

Data Structures and Algorithms, Fall 2020

October 29, 2020

Contents

1 Problem 1

(a) The third and the fifth.

(b) No. There is a counterexample. A binary search tree with 3 nodes, root (1), root's right child (3), right child of root's right child (2). Then the search path for value 3 makes $A = \{2\}, B = \{1, 3\}, C = \emptyset$, then $2 \in A, 1 \in B$ but $2 > 1$.

2 Problem 2

Algorithm 1: PARENT(T,x)

```
1  if x = T.root then
2  |   return NIL;
3  end
4  y=TREE-MAXIMUM(x).succ;
5  if y = NIL then
6  |   y=T.root
7  end
8  else
9  |   if y.left = x then
10 |   |   return y;
11 |   end
12 |   y=y.left;
13 end
14 while y.right != x do
15 |   y=y.right
16 end
17 return y
```

SEARCH No changes.

Algorithm 2: INSERT(T,z)

```
1  y = NIL ;
2  x = T.root ;
3  pred = NIL ;
4  while x != NIL do
5  |   y = x ;
6  |   if z.key < x.key then
7  |   |   x = x.left ;
8  |   end
9  |   else
10 |   |   pred = x ;
11 |   |   x = x.right ;
12 |   end
13 end
14 if y == NIL then
15 |   T.root = z ;
16 |   z.succ = NIL ;
17 end
18 else if z.key < y.key then
19 |   y.left = z ;
20 |   z.succ = y ;
21 |   if pred != NIL then
22 |   |   pred.succ = z ;
23 |   end
24 end
25 else
26 |   y.right = z ;
27 |   z.succ = y.succ ;
28 |   y.succ = z ;
29 end
```

Algorithm 3: TRANSPLANT(T,u,v)

```
1  p = PARENT(T, u) if p == NIL then
2  |   T.root = v
3  end
4  else if u == p.left then
5  |   p.left = v
6  end
7  else
8  |   p.right = v
9  end
```

Algorithm 4: TREE-PREDECESSOR(T, x)

```
1  if x.left != NIL then
2  |  return TREE-MAXIMUM(x.left)
3  end
4  y = T.root;
5  pred = NIL;
6  while y != NIL do
7  |  if y.key == x.key then
8  |  |  break
9  |  end
10 |  if y.key < x.key then
11 |  |  pred = y;
12 |  |  y = y.right;
13 |  end
14 |  else
15 |  |  y = y.left
16 |  end
17 end
18 return pred
```

Algorithm 5: DELETE(T, z)

```
1  pred = TREE-PREDECESSOR(T, z);
2  pred.succ = z.succ;
3  if z.left == NIL then
4  |  TRANSPLANT(T, z, z.right)
5  end
6  else if z.right == NIL then
7  |  TRANSPLANT(T, z, z.left)
8  end
9  else
10 |  y = TREE-MIMIMUM(z.right);
11 |  if PARENT(T, y) != z then
12 |  |  TRANSPLANT(T, y, y.right);
13 |  |  y.right = z.right;
14 |  end
15 |  TRANSPLANT(T, z, y);
16 |  y.left = z.left;
17 end
```

3 Problem 3

(3.a)

If all nodes in the binary search tree have identical keys, the TREEINSERT procedure always inserts the new node to the right end, which implys $h = O(n)$. $T(n) = T(n-1) + O(h) = T(n-1) + O(n) = O(n^2)$.

(3.b)

Since new nodes always enter the left and right subtrees of x **in turn**, we can find out that the differ from the size of left-subtree to right-subtree is no more than 1, which implys $h = O(\lg n)$. $T(n) = T(n-1) + O(h) = T(n-1) + O(\lg n) = O(n \lg n)$.

(3.c)

There is only one node in the binary search tree, $h = O(1)$. And we need $O(1)$ time to insert z into the list. $T(n) = T(n-1) + O(h) + O(1) = O(n)$.

4 Problem 4

(4.a)

- largest: 2 (a black root node with two red child).
  Proof: Only black nodes' children can be red.
- smallest: 0 (a black root node).
  Proof: Zero is the smallest non-negative number.

(4.b)

- Lemma: At most $n-1$ right rotations suffice to transform the tree into a right-going chain.
- Proof of lemma:
  - I.B: For $n = 1$, no rotation is required.
  - I.H: For $1 \leq n \leq k$, at most $n-1$ right rotations suffice to transform the tree into a right-going chain.
  - I.S: For $n = k+1$, assume the size of the left subtree is m ($1 \leq m \leq k$), then the size of the right subtree is $k-m$. From **I.H**, we can take at most

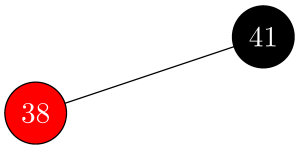
    $$t = (m-1) + \max\{0, (k-m-1)\} \leq k-1$$

    right rotations to transform them into right-going chains. Then take 1 right rotation at root, the whole tree transform into a right-going chain, also. $t+1 \leq n-1$.
- Proof of the original proposition:
  - We can use a stack to record the sequence of nodes that have been taken right rotations, take left rotations in reverse order can restore the original binary search tree.
  - Assume that we need to transform T_1 into T_2. Transform T_1 and T_2 to the right chain, record the right rotation order of T_2. Take these left rotations to T_1, which is a right-going chain in this moment.

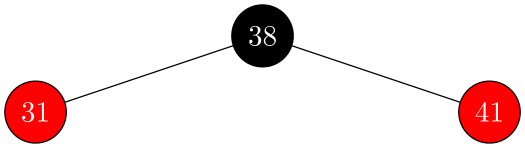
5 Problem 5

- insert 41:

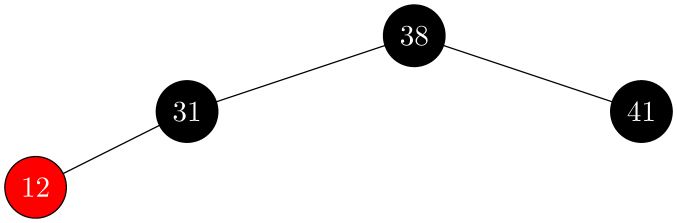
- insert 38:



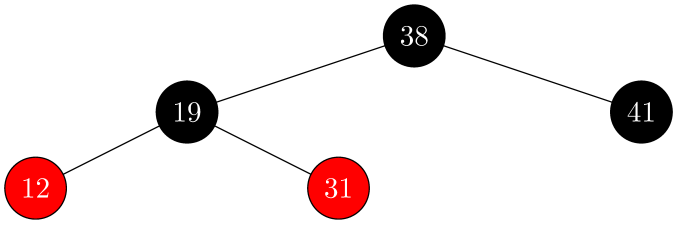
- insert 31:



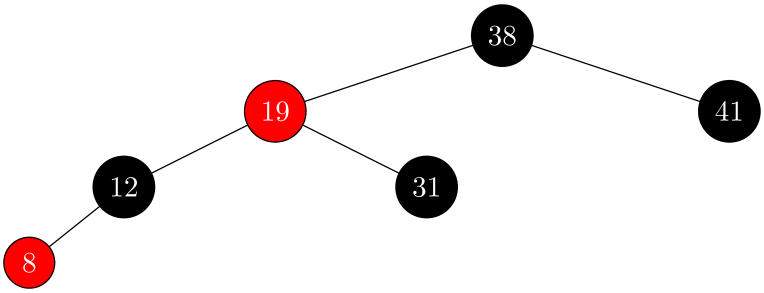
- insert 12:



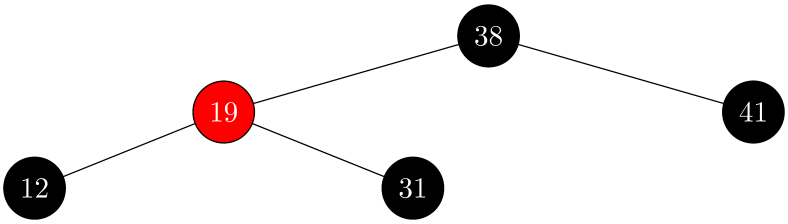
- insert 19:



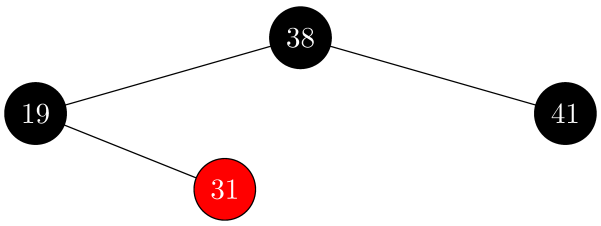
- insert 8:

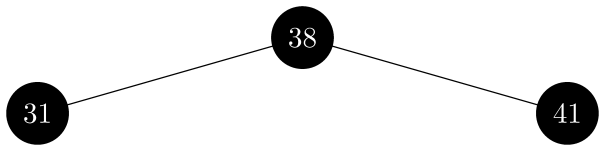


- delete 8:

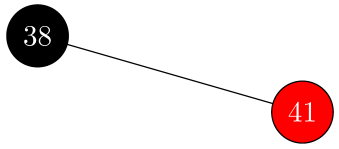


- delete 12:





- delete 19:



- delete 31:
- delete 41: (38)(black)

6 Problem 6

(a)

Let T_h be the minimum number of vertices that an AVL tree with height h has. It can be seen that $T_n \geq T_{n-1} + T_{n-2} + 1$, which shows that there exists some constant $C > 1$, so that $T_n \geq \exp(Cn)$ when n is large enough.

(b)

Algorithm

Algorithm 6: Balance(x)

Input: node x

if $x.left.h > x.right.h + 1$ **then**
  if $x.left.left.h < x.left.right.h$ **then**
    $Left - Rotate(x.left)$;
  end
  $Right - Rotate(x)$;
end
if $x.right.h > x.left.h + 1$ **then**
  if $x.right.right.h < x.right.left.h$ **then**
    $Right - Rotate(x.right)$;
  end
  $Left - Rotate(x)$;
end

Correctness

WLOG, we will only prove correctness when $x.left.h > x.right.h + 1$.

- If $x.left.left.h < x.left.right.h$, it can be seen that $x.left.right.h = x.left.left.h + 1$ from our basic assumption. Hence, We may perform $Left - Rotate$ operation on $x.left$, which maintains balance property for all vertices in the left subtree of x, while negating the fact that $x.left.left.h < x.left.right.h$.
- After first operation, it is guaranteed that $x.left.left.h \geq x.left.right.h$. We may perform $Right - Rotate$ operation on vertice x, achieving the goal.

(c)

(d)

We only needs to check it performs $O(1)$ rotations. It follows from the fact that Balance operation will reduce the height of unbalanced subtree by exactly 1.

Remark

We assume that Rotation operation will maintain information such as the height of subtree and so on.

Algorithm 7: Insert(x, z)

Input: node x, z

```
if x == NIL then
    z.h = 0;
    return;
end
if x.key < z.key then
    Insert(x.right, z);
    if x.right == NIL then
        x.right = z;
    end
end
else
    Insert(x.left, z);
    if x.left == NIL then
        x.left = z;
    end
end
left_h = −1, right_h = −1;
if x.left ≠ NIL then
    left_h = x.left.h;
end
if x.right ≠ NIL then
    right_h = x.right.h;
end
x.h = max(left_h, right_h) + 1;
Balance(x);
```

7 Problem 7

(a) Create n nodes $P[1...n]$ such that $P[i].key = A[i]$, and assign each $P[i]$ a random priority $P[i].pri$ (For example, let $P[i].pri$ be a random real number in $[0, 1]$, we assume this can be done in $O(1)$ time). Recall that each node in a Treap contain two attributes key and $priority$. Now we give a $O(n)$ algorithm to build a Treap T containing nodes $P[1...n]$.

Remark 1. *The structure of a $Treap$ is uniquely fixed when key and $priority$ are fixed for each node, by the following way: pick the node with the largest $priority$ as the root, split the array into its left subtree and right subtree, and do this recursively to build the subtrees. However, the trivial implementation of the idea will cost $O(n^2)$ time in worst case. We will show how to solve this in $O(n)$ time. The first method use rotations to maintain the treap property. The second method use **Monotonous stack** which is much more intricate but useful, it is highly recommanded to understand the second method .*

Method 1: Let $T.root = P[1]$. Build a right chain by the following loop:

- For $i = 1$ to $n - 1$, let $P[i].rightchild = P[i + 1]$ and $P[i + 1].p = P[i]$.

Now the BST property is satisfied, we use the following procedure to adjust the tree to maintain the heap property (suppose we want a **Max-heap**):

1 For $i = 2$ to n
   - Let $u \leftarrow P[i]$.
   - **While** u is not root and $u.pri > (u.p)\ .pri$, **do** $left - rotate(u.p)$ and let $u \leftarrow u.p$.

The correctness of the algorithm is guaranteed by the following loop invariance:

Invariance: After each loop in line 1, the Treap composed by $P[1...i]$ satisfies the heap property.

Initially the invariance is true since $P[1]$ is automatically a one node Treap. The third line of the algorithm try to add $P[i]$ into Treap $P[1...i - 1]$ while maintain the heap property. The proof is left to the reader.

Since rotations will not destroy the BST property, and according to the invariance, the heap property is satisfied, the algorithm correctly build a Treap.

Complexity: Consider the length of the right most chain of the tree T. Each rotation will eliminate one one from the right most chain. Since there are n nodes in the right most chain initially, there can be at most n times of rotations. The running time of the algorithm is dominated by the number of rotations, thus, the complexity is $O(n)$.

Method 2: Build an empty stack S and do the following procedure:

- For $i = 1$ to n, do
   - **While** S is not empty and $S.top().pri < P[i].pri$, **do** $u \leftarrow S.pop()$.
   - If S is empty, then let $T.root \leftarrow P[i]$.
   - Otherwise let $S.top().rightchild = P[i]$ and $P[i].p \leftarrow S.top()$. Let $u.p \leftarrow P[i]$ and $P[i].leftchild \leftarrow u$.
   - $S.push(P[i])$.

The stack S is call **Monotonous stack**, due to the fact that the priority of the nodes in S is always decreasing from bottom to the top. The correctness of the algorithm depend on the same loop invariance as method 1:

Invariance: After each loop, the Treap composed by $P[1...i]$ satisfies the heap property.

The proof is left to the reader.

The complexity is $O(n)$ since each element can be push and pop at most once.

(b) Suppose the skip list contains element $A[1...n]$, and it will raise an element with probability p, where p is a constant.

1. Let F_i be the event that $A[i]$ has been raised at least $2\log_{\frac{1}{p}} n$ times, then the probability for F_i to happen is $p^{2\log_{\frac{1}{p}} n} = \frac{1}{n^2}$. And according to **union bound** (An important method that you should learn carefully)

$$\Pr[\vee_{1\le i\le n} F_i] \le \sum_{1\le i\le n} \Pr[F_i] = n \cdot \frac{1}{n^2} = \frac{1}{n}$$

Thus, the probability for none of F_i happened is at leat $1 - \Pr[\vee_{1\le i\le n} F_i] \ge 1 - \frac{1}{n}$, which means with probability at least $1 - \frac{1}{n}$ all elements are raised at most $2\log_{\frac{1}{p}} n = O(\log n)$ times.

2. Let X_i be the random variable of the number of nodes generated by $A[i]$. Obviously $E[X_i] = \frac{1}{1-p}$. Thus, we have

$$E[\text{number of nodes}] = E\left[\sum_{1\le i\le n} X_i\right] = \sum_{1\le i\le n} E[X_i] = \frac{n}{1-p} = O(n)$$

3. Chernoff bound has several useful forms. In this problem we need the multiplicative chernoff bound.

Lemma 2 (Multiplicative Chernoff bound)**.** *Suppose $X_1, X_2, ..., X_n$ are n independent random variables such that $a \le X_i \le b$ for all $1 \le i \le n$, let $X = \sum_{1\le i\le n} X_i$ and set $\mu = \mathbb{E}[X]$, then for all $\delta > 0$, we have*

$$\Pr\left[X > (1+\delta)\mu\right] \le e^{-\frac{2\delta^2\mu^2}{n(b-a)^2}}$$

The chernoff bound need X_i to be bounded ($a \le X_i \le b$).But recall that we define X_i as the random variable of the number of nodes generated by $A[i]$, which means X_i **is not bounded above**. But according to (a) we know they are bounded by $C \triangleq 4\log_{\frac{1}{p}} n$ with probability at lest $1 - \frac{1}{n^2}$. That inspires us to define

$$X_i' = \begin{cases} 0 & X_i > C \\ X_i & X_i \le C \end{cases} \qquad X' = \sum_{1\le i\le n} X_i' \tag{1}$$

Now X_i is bounded by C. And according to (a), the probability for $X' = X$ is at least $1 - \frac{1}{n^2}$.

Now we have $\mathbb{E}[X_i'] = c_0$ for some constant $1 < c_0 < 2$. Thus, we get $\mu = \mathbb{E}[X'] = c_0 n$. Apply Multiplicative Chernoff bound to $X_1', ..., X_n'$, we get

$$\Pr[X' > 2\mu] \le e^{-\frac{8c_0^2 n^2}{nC^2}}$$

Note that $e^{-\frac{8c_0^2 n^2}{nC^2}} = e^{-\Omega\left(\frac{n}{\log^2 n}\right)}$. For sufficiently large n we have $\Pr[X' > 2\mu] \le \frac{1}{n^2}$.

Let $\mathcal{A}$ be the event that $X' \ne X$, and let $\mathcal{B}$ be the event that $X' > 2\mu$. According to **union bound**, the probability for at least one of $\mathcal{A}$ and $\mathcal{B}$ to happen is at most $\frac{2}{n^2}$, which means with high probability, both $\mathcal{A}$ and $\mathcal{B}$ do not happen. Thus, with high probability, X is bounded by $2\mu = O(n)$.

Sample Solution for Problem Set 6

Data Structures and Algorithms, Fall 2020

November 5, 2020

Contents

1 Problem 1

Choose one of the m spots in the hash table at random. Let n_k denote the number of elements stored at $T[k]$. Next pick a number x from 1 to L uniformly at random. If $x < n_j$, then return the xth element on the list. Otherwise, repeat this process. Any element in the hash table will be selected with probability $1/mL$, so we return any key with equal probability. Let X be the random variable which counts the number of times we must repeat this process before we stop and p be the probability that we return on a given attempt. Then $E[X] = p(1 + \alpha) + (1 - p)(1 + E[X])$ since we'd expect to take $1 + \alpha$ steps to reach an element on the list, and since we know how many elements are on each list, if the element doesn't exist we'll know right away. Then we have $E[X] = \alpha + 1/p$. The probability of picking a particular element is $n/mL = \alpha/L$. Therefore, we have

$$\begin{aligned} E[X] &= \alpha + L/\alpha \\ &= L(\alpha/L + 1/\alpha) \\ &= O(L(1 + 1/\alpha)) \end{aligned}$$

since $\alpha \leq L$.

2 Problem 2

(a) We will show that each string hashes to the sum of it's digits $\mod 2^p - 1$. We will do this by induction on the length of the string. As a base case, suppose the string is a single character, then the value of that character is the value of k which is then taken $\mod m$. For an inductive step, let $w = w_1 w_2$ where $|w_1| \geq 1$ and $|w_2| = 1$. Suppose $h(w_1) = k_1$. Then, $h(w) = h(w_1)2^p + h(w_2) \mod 2^p - 1 = h(w_1) + h(w_2) \mod 2^p - 1$. So, since $h(w_1)$ was the sum of all but the last digit $\mod m$, and we are adding the last digit $\mod m$, we have the desired conclusion.

(b) 0 for empty slots.

- Linear probing: [22, 88, 0, 0, 4, 15, 28, 17, 59, 31, 10]
- Quadratic probing: [22, 0, 88, 17, 4, 0, 28, 59, 15, 31, 10]
- Double hashing: [22, 0, 59, 17, 4, 15, 28, 88, 0, 31, 10]

3 Problem 3

Proof by contradiction:

(1)

Suppose that

$$\epsilon < \frac{1}{|B|} - \frac{1}{|U|}.$$

This means that for all pairs k, l in U, we have that the number $n_{k,l}$ of hash functions in $\mathcal{H}$ that have a collision on those two elements satisfies

$$n_{k,l} \leq \epsilon|\mathcal{H}| < \frac{|\mathcal{H}|}{|B|} - \frac{|\mathcal{H}|}{|U|}.$$

So, summing over all pairs of elements in U, we have that the total number N satisfies

$$N = \sum_{l,l\neq k}^{U}\sum_{k}^{U} n_{k,l} < \frac{|\mathcal{H}||U|^2}{2|B|} - \frac{|\mathcal{H}||U|}{2}.$$

(2)

For any hash function h, the number of conflicting pairs, n_h, is minimized if and only if each slot contains an equal number of keys. (Proved in the appendix) In other words, n_h satisfies

$$n_h \geq |B|\binom{|U|/|B|}{2} = |B|\frac{|U|^2 - |U||B|}{2|B|^2} = \frac{|U|^2}{2|B|} - \frac{|U|}{2}$$

Summing over all hash functions, we get that the total number N satisfies

$$N = \sum_{h}^{\mathcal{H}} n_h \geq |\mathcal{H}|\left(\frac{|U|^2}{2|B|} - \frac{|U|}{2}\right)$$

which is contrary to the conclusion of **(1)**. Therefore $\epsilon \geq \frac{1}{|B|} - \frac{1}{|U|}$.

(appendix)

Let $b \equiv |B|$, suppose that the number of keys contained in each slot is $n_1, n_2, \ldots, n_b$. There is $\sum_{i=1}^{b} n_i = |U|$. The total number of conficting pairs is

$$\begin{aligned}
\binom{n_1}{2} + \cdots + \binom{n_b}{2} &= \frac{1}{2}\left(n_1^2 - n_1 + \cdots + n_b^2 - n_b\right) \\
&= \frac{1}{2}\left[\left(n_1^2 + \cdots + n_b^2\right) - (n_1 + \cdots + n_b)\right] \\
&= \frac{1}{2}\left[\left(n_1^2 + \cdots + n_b^2\right)\left(\frac{1}{\sqrt{b}}^2 + \ldots \frac{1}{\sqrt{b}}^2\right) - (n_1 + \cdots + n_b)\right] \\
&\geq \frac{1}{2}\left[\left(\frac{1}{\sqrt{b}}n_1 + \cdots + \frac{1}{\sqrt{b}}n_b\right)^2 - (n_1 + \cdots + n_b)\right] \\
&= \frac{1}{2}\left(\frac{1}{|B|}|U|^2 - |U|\right) = |B|\binom{|U|/|B|}{2}
\end{aligned}$$

By cauchy's inequality, the equal sign is true if and only if $n_1 = n_2 = \cdots = n_b$.

4 Problem 4

(implement)

S is an array. Initially S is empty and $S.size = 0$.

Algorithm 1: INSERT(S, x)

```
1 S[S.size] ← x;
2 S.size ← S.size + 1;
```

Algorithm 2: DELLARGEHALF(S)

```
1 QUICKSELECT(S, ⌊S.size/2⌋);
2 S.size ← ⌊S.size/2⌋;
```

(complexity)

- Charge 2 dollar for INSERT(S, x)
- Cost $S.size$ dollar for DELLARGEHALF(S)
- There are at least $2 \times S.size$ dollar deposits. **(*)**
  - (*) is true when $S.size = 0$.
  - After INSERT(S, x), $S.size$ increase 1, deposits increase 2. (*) is true.
  - After DELLARGEHALF(S),
    deposits: at least $2 \times S.size_{old} - S.size_{old} = S.size_{old}$;
    $S.size_{new}$: less than $\frac{1}{2} \times S.size_{old}$.
    (*) is true.

5 Problem 5

Suppose the binary string is B, with the lowest digit $B[0]$. We use the following algorithm to perform a single INC operation:

1 Initialize $i = 0$.

2 **While** $B[i] == 1$ **do**

- $B[i] \leftarrow 0$.
- $i \leftarrow i + 1$.

3 $B[i] \leftarrow 1$

Define the potential function Φ_i as the number of 1 in B before the i-th INC operation. Suppose in the i-th INC operation, the **While** loop in step 2 is executed for n_i times, obviously we have $\Phi_i - \Phi_{i+1} = n_i - 1$. The running time for the i-th INC operation is $c_0 n_i + c_1$ for some constant c_0, c_1. Thus, the running time for n INC operations is

$$\sum_{1 \le i \le n} (c_0 n_i + c_1) = c_0 \left(\sum_{1 \le i \le n} (\Phi_i - \Phi_{i+1}) + n \right) + c_1 n \le c_0(\Phi_1 + n) + c_1 n$$

Recall that $\Phi_1 = b$ and $n = \Omega(b)$, we have $c_0(\Phi_1 + n) + c_1 n = O(n)$.

6 Problem 6

Suppose the stack is empty and has size 1 initially. Strictly speaking, we will analysis the running time for any execution combinition of n POP and PUSH operations, and justify that the cost is $O(n)$.

(a) No. For any i and let $n = 2^i$, Consider the following execution: n times of PUSH, followed by n times of PUSH, POP pairs. One can varify that after n times of PUSH the size of the stack is exactly n, and each PUSH, POP pair will cost $\Omega(n)$ time since the stack will double for each PUSH and shrink for each POP. Thus, $3n$ times of operations cost $\Omega(n^2)$ times.

(b) Yes. Suppose store 3 coins for each POP and $PUSH$ operation, we will prove we always own non-negetive number of coins at any time if we use one coin for each time cost. We prove it by induction. Initially we have zero coin. If before the i-th operation, we always have non-negetive number, now consider the i-th operation: if it do not cause an expansion or shrinking, the number of coins will increase by 2. Otherwise:

It is a $PUSH$ operation and cause a expansion from n to $2n$. If the last change of the stack is from $\frac{n}{2}$ to n, then $\frac{n}{2} * 2$ coins must be saved, thus still non-negetive. If the last change of the stack is shrink from $4n$ to n, consider the former change of the stack, if it is from $16n$ to $4n$, then there should be $3n * 2$ coins saved, which means it is non-negetive. Otherwise it is from $2n$ to $4n$, then there must be $n * 2$ coins save, which means it is still non-negetive.

It is a POP operation and cause a shrinking from $4n$ to n, consider the last change of the stack, if it is from $2n$ to $4n$, then there are at least $2n$ elements and $n * 2$ coins must be saved. Otherwise it is from $\frac{n}{2}$ to n, in which case there are $\frac{n}{2} * 2$ coins save. In both case the coins are non-negetive.

(c) Yes. Similar to (b).

7 Problem 7

(a)

We will prove that the amortized time complexity for this case is $O(1)$.

Note that $c_i = k+4$, where $i = 2^k * m$ for some odd integer m. Let $\Phi(x) =$ number of 1s in the array after x-th operat

Therefore, $\hat{c_i} = c_i + \Phi(i) - \Phi(i-1) = 4 + 2 *$ number of bits changes from 0 to $1 \leq 6$.

(b)

We will prove that the amortized time complexity for this case is $O(\log n)$.

Note that $c_i = k + 2^k$, where $i = 2^k * m$ for some odd integer m. WLOG, we may assume that $c_i = 2^k$ to simplify our analysis. Let $\Phi(x) = x\lceil \log n \rceil - \sum_{i=1}^{x} c_i$, and the following observations concludes our proof.

- $\Phi(x) \geq \Phi(0) = 0$. It follows from the fact that

$$\sum_{i=1}^{x} c_i \leq \sum_{k=0}^{+\infty} 2^k \left\lfloor \frac{x}{2^k} \right\rfloor \leq x\lceil \log n \rceil$$

- $\hat{c_i} = c_i + \Phi(i) - \Phi(i-1) = \lceil \log n \rceil$

(c)

We will prove that the amortized time complexity for this case is $O(n)$.

WLOG, we will assume that $c_i = 4^k$, where $i = 4^k * m$ for some odd integer m. Let $\Phi(x) = 4xn - \sum_{i=1}^{x} c_i$, let's prove the following

- $\Phi(x) \geq \Phi(0) = 0$. It follows from the fact that

$$\sum_{i=1}^{x} c_i \leq \sum_{k=0}^{+\infty} 4^k \left\lfloor \frac{x}{2^k} \right\rfloor \leq x \sum_{k=0}^{\lceil \log n \rceil} 2^k \leq 4xn$$

- $\hat{c_i} = c_i + \Phi(i) - \Phi(i-1) = 4n$

Sample Solution for Problem Set 7

Data Structures and Algorithms, Fall 2020

January 4, 2021

Contents

1 Problem 1

(a) We call MAKE-SET n times, which contributes $\Theta(n)$. In each union, the smaller set is of size 1, so each of these takes $\Theta(1)$ time. Since we union $n-1$ times, the runtime is $\Theta(n)$.

(b) Since the loop and the assignment runs in $O(n)$ time, we only need to prove that the operation sequence $Find(node[1]), \ldots, Find(node[n])$ runs in $O(n)$ time in worst case.

Let T be the original forest. Let T' be the forest that results from the operation sequence $Find(v)$ which v is a node. In T' all nodes on the path from v to the root of the tree are now children of the root. Suppose that $Find(w)$ is done later. If the path in T from w to the root meets the path in T from v to the root, say at x, then $Find(w)$ follows only one link for the overlapping path segment since in T' x is a child of the root. Thus the total amount of work done by the $Find$s is bounded by a multiple of the number of distinct branches in T. Since there are n nodes, the total amount of work done by the $Find$s is in $O(n)$.

2 Problem 2

WLOG, we may assume that we will only paint white cells into black.

Given array $X = [x_1, x_2, \ldots, x_n]$, consider $S_1, S_2, \ldots, S_k$ as a partition of $[n]$, satisfying that for each $i \in [k]$, S_i forms a continous segment, and $x_j = 0$ if and only if there exists an $i \in [k]$, such that $j = \max S_i$. We may directly see from the definition that there's a one to one corresponding between X and partition of $[n]$ satisfying above constraints. The crucial observation is the following lemma.

Lemma 1. *For a given binary array X, there exists unique (up to permutation) partition $S_1, S_2, \ldots, S_k$ satisfying above constraints, and*

- $Blacken(i)$ *operator for array X is exactly $Union(i, i+1)$ operator for corresponding partition $S_1, S_2, \ldots, S_k$.*
- $NextWhite(i)$ *query for array X is exactly finding the maximum value of S_j where $i \in S_j$.*

Therefore, we may solve this task via DSU.

(a)

Create two auxiliary array Y, Z, representing size of its component, and leader element(in this case, $\max S_j$). We may merge the smaller set into the larger one in merge process, and output Y_i in query process. Note that for every position i, its information will only be updated $O(\log n)$ times.

(b)

We may use the classical DSU data structure with path compression and union by rank technique. However, we should notice that the leader element in this scenario may not be the largest index in this components. This can be fixed by tracking the largest index in each component.

(c, bonus)

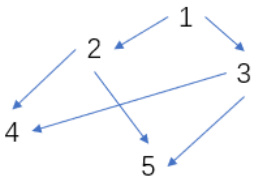
Note that union operation will be operated only on ends of two segments, and concatenate them. Therefore, we may store index of leader and range of segment for both ends of segments. It can be seen that information can be updated efficiently(in $O(1)$) during merge process.

3 Problem 3

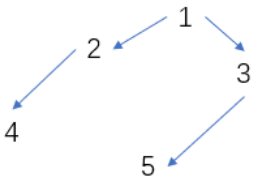
Start by examining position $(1, 1)$ in the adjacency matrix. When examining position (i, j), if a 1 is encountered, examine position $(i + 1, j)$. If a 0 is encountered, examine position $(i, j + 1)$. Once either i or j is equal to $|V|$, terminate.

We claim that if the graph contains a universal sink, then it must be at vertex i. To see this, suppose that vertex k is a universal sink. Since k is a universal sink, row k in the adjacency matrix is all 0's, and column k is all 1's except for position (k, k) which is a 0. Thus, once row k is hit, the algorithm will continue to increment j until $j = |V|$. To be sure that row k is eventually hit, note that once column k is reached, the algorithm will continue to increment i until it reaches k. This algorithm runs in $O(V)$ and checking whether or not i in fact corresponds to a sink is done in $O(V)$. Therefore the entire process takes $O(V)$.

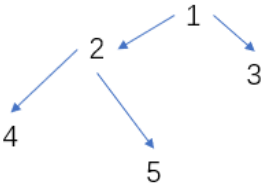
4 Problem 4



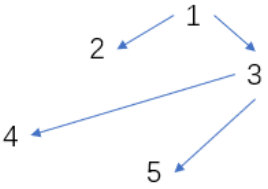
G=<V,E>



G_π=<V,E_π>



Breadth-First-Tree 1



Breadth-First-Tree 2

(a)

- $u.d$ represents the shortest distance from the root node to u, which must be unique.
- As $G =< V, E >$ in the figure above, different breadth-first trees (1 and 2) will be generated based on the visited order of nodes 2 and 3.

(b)

- As $G_\pi =< V, E_\pi >$ in the figure above, No matter which node you visit first in node 2 and 3, you can't generate a G_π.

vertex	d	f
q	1	16
r	17	20
s	2	7
t	8	15
u	18	19
v	3	6
w	4	5
x	9	12
y	13	14
z	10	11

5 Problem 5

(a)

- tree edge: (q,s), (s,v), (v,w), (q,t), (t,x), (x,z), (t,y), (r,u)
- back edge: (w,s), (z,x), (y,q)
- forward edge: (q,w)
- cross edge: (r,y), (u,y)

(b)

Counterexample:

- $G =< V, E >, V = \{x, y, z\}, E = \{(x, y), (y, x), (x, z)\}$
- $x.d < y.d < z.d$
- There is a path from y to z, but z is not a descendant of y.

6 Problem 6

```
DFS-STACK(G)
    for each vertex u ∈ G.V
        u.color = WHITE
        u.π = NIL
    time = 0
    S = Ø
    for each vertex u ∈ G.V
        if u.color == WHITE
            time = time + 1
            u.d = time
            u.color = GRAY
            PUSH(S, v)
            while !STACK-EMPTY(S)
                v = TOP(S)
                isNeighborhoodsAllDiscovered = true
                if v.color == GRAY
                    for each vertex w ∈ G.Adj[v]
                        if w.color == WHITE
                            time = time + 1
                            w.d = time
                            w.color = GRAY
                            PUSH(S, w)
                            isNeighborhoodsAllDiscovered = false
                            break
                if isNeighborhoodsAllDiscovered
                    time = time + 1
                    v.f = time
                    v.color = BLACK
                    POP(S)
```

7 Problem 7

Suppose $G = (V, E)$ and $V = \{v_1 = s, ..., v_n = t\}$.

(a) Create a new graph $G' = (V', E')$, where $V = \{v_{i,j}\}_{1 \le i \le n, 0 \le j \le 2}$, and $E' = \{(v_{i,k}, v_{j,(k+1) \mod 3} \mid (v_i, v_j) \in E\}$. The algorithm is: do BFS or DFS on G' to find whether there is a path from $v_{1,0}$ to $v_{n,0}$. If there is a path, the output yes, otherwise output no. The complexity is $O(|V| + |E|)$ since the number of vertice and edges are multiplied by a constant.

To prove the correctness, suppose there is a path $\{v_{a_1,b_1}, v_{a_2,b_2}, ...\}$ in G' from $v_{1,0}$ to $v_{n,0}$, then the path $\{v_{a_1}, v_{a_2}, ...\}$ is the path with the same length (divided by 3 since and edge will increase b_i by 1, i.e., $b_{i+1} = b_i + 1 \mod 3$) G' from v_1 to v_n. Conversely, if there is a path in G: $\{v_{a_1}, v_{a_2}, ...\}$ such that the length is divided by 3, then the path $\{v_{a_1,0}, v_{a_2,1}, v_{a_3,2}, v_{a_4,0}...\}$ is the path from $v_{1,0}$ to $v_{n,0}$ with the same length.

(b) Similar to **(a)**, we copy the graph for 5 times, creating graph $G' = (V', E')$ where $V = \{v_{i,j}\}_{1 \le i \le n, 0 \le j \le 4}$. E' is composed by 8 part:

- $(v_{i,0}, v_{j,1})$ where (v_i, v_j) is a blue edge.
- $(v_{i,1}, v_{j,2})$ where (v_i, v_j) is a blue edge.
- $(v_{i,0}, v_{j,3})$ where (v_i, v_j) is a red edge.
- $(v_{i,3}, v_{j,4})$ where (v_i, v_j) is a red edge.
- $(v_{i,1}, v_{j,3})$ where (v_i, v_j) is a red edge.
- $(v_{i,2}, v_{j,3})$ where (v_i, v_j) is a red edge.
- $(v_{i,3}, v_{j,1})$ where (v_i, v_j) is a blue edge.
- $(v_{i,4}, v_{j,1})$ where (v_i, v_j) is a blue edge.

Intuitively, the graph is composed by 5 layers, the layer 0 are the initial layer; layer 1 and 2 are the blue layer such that the node has already pass 1 or 2 consecutive blue edges; the layer $3, 4$ are the red layer where the node has already pass 1 or 2 consecutive red edges. Each layer can be seen as a "state" of a node, and the edges in E' is all the possible change between states.

Now run BFS to find the shortest path from $v_{1,0}$ to $v_{n,i}$ where i is arbitrary. The length of the shortest path in G' is the lenght of the required path in G. The proof is similar to **(a)**: paths between two graph can map from each other.

Sample Solution for Problem Set 8

Data Structures and Algorithms, Fall 2020

January 4, 2021

Contents

1 Problem 1

Consider having a list for each potential in degree that may occur. We will also make a pointer from each vertex to the list that contains it. The initial construction of this can be done in time $O(|V|+|E|)$ because it only requires computing the in degree of each vertex, which can be done in time $O(|V| + |E|)$. Once we have constructed this sequence of lists, we repeatedly extract any element from the list corresponding to having in degree zero. We spit this out as the next element in the topological sort. Then, for each of the children c of this extracted vertex, we remove it from the list that contains it and insert it into the list of in degree one less. Since a deletion and an insertion in a doubly linked list can be done in constant time, and we only have to do this for each child of each vertex, it only has to be done $|E|$ many times. Since at each step, we are outputting some element of in degree zero with respect to all the vertices that hadn't yet been output, we have successfully output a topological sort, and the total runtime is just $O(|E| + |V|)$. We also know that we can always have that there is some element to extract from the list of in degree 0, because otherwise we would have a cycle somewhere in the graph. To see this, just pick any vertex and traverse edges backwards. You can keep doing this indefinitely because no vertex has in degree zero. However, there are only finitely many vertices, so at some point you would need to find a repeat, which would mean that you have a cycle.

If the graph was not acyclic to begin with, then we will have the problem of having an empty list of vertices of in degree zero at some point. That is, if the vertices left lie on a cycle, then none of them will have in degree zero.

2 Problem 2

(a) Given the procedure given in the section, we can compute the set of vertices in each of the strongly connected components. For each vertex, we will give it an entry SCC, so that v.SCC denotes the strongly connected component (vertex in the component graph) that v belongs to. Then, for each edge (u, v) in the original graph, we add an edge from u.SCC to v.SCC if one does not already exist. This whole process only takes a time of $O(|V| + |E|)$. This is because the procedure from this section only takes that much time. Then, from that point, we just need a constant amount of work checking the existence of an edge in the component graph, and adding one if need be.

(b) Professor Bacon's suggestion doesn't work out. As an example, suppose that our graph is on the three vertices $\{1, 2, 3\}$ and consists of the edges $(2, 1), (2, 3), (3, 2)$. Then, we should end up with $\{2, 3\}$ and $\{1\}$ as our SCC's. However, a possible DFS starting at 2 could explore 3 before 1, this would mean that the finish time of 3 is lower than of 1 and 2. This means that when we first perform the DFS starting at 3. However, a DFS starting at 3 will be able to reach all other vertices. This means that the algorithm would return that the entire graph is a single SCC, even though this is clearly not the case since there is neither a path from 1 to 2 of from 1 to 3.

3 Problem 3

(a) Use the algorithm in Problem 1 to perform a topological sorting on graph G and create $P[1...n]$ where $n = |V|$ and $P[1...n]$ are n nodes in V in the topololical order ($P[1]$ has in-degree 0 and $P[n]$ has out-degree 0). Initialize $cost[u]$ as p_u for any $u \in V$, then do the following loop:

- For $i = n$ to $i = 1$ do
  - For each node v where $(P[i], v) \in E$, let $cost[P[i]] = \min(cost[P[i]], cost[v])$.

(b) First create the component graph of G as $G' = (V', E')$. For each point u in G', suppose u corresponds to the nodes set $S_u \subseteq G$ as a components. Initialize $p_u = \min_{v \in S_u} p_v$. Then we get an instance $G', \{p_u\}_{u \in V'}$ which can be the input for **(a)**. Use algorithm in **(a)** on $G', \{p_u\}_{u \in V'}$ to get an array $cost[u]$ for $u \in V'$. For any $v \in V$, let $cost[v] = cost[u]$ where $v \in S_u$.

4 Problem 4

Suppose G is not an empty graph. First judge whether G is connected, if not, G cannot be "sort-of-connected". create the component graph of G as $G' = (V', E')$. For each point u in G', suppose u corresponds to the nodes set $S_u \subseteq G$ as a components. Calculate the topological order of points in G' at get $P[1...n]$. If there exists an edge from $P[i]$ to $P[i+1]$ for any $1 \leq i \leq n-1$, then G is sort-of-connected. Otherwise not. To prove the correctness of our algorithm, we prove the following lemma.

Lemma 1. *G is "sort-of-connected" iff. G' is "sort-of-connected".*

Proof. If G is "sort-of-connected", for every two points $u, v \in V'$, take arbitrary $u' \in S_u, v' \in S_v$. There exists a path from u' to v' or v' to u' in V, in which case there is a path from u to v or v to u in G'. Thus, G' is "sort-of-connected".

If G' is "sort-of-connected", for every two points $u, v \in V$, if $u, v \in S_w$ for some $w \in V'$, then there must be a path from u to v since they are in the same components. Otherwise, suppose $u \in S_{u'}, v \in S_{v'}$ for $u', v' \in V'$, there must be a path from u' to v' or v' to u', in which case there is a path from u to v or v to u. Thus, G is "sort-of-connected". □

Lemma 2. *P is defined above. If $P[i]$ has an edge to $P[i+1]$ for any $1 \leq i \leq n-1$, then G' is "sort-of-connected", otherwise not.*

Proof. If $P[i]$ has an edge to $P[i+1]$ for any $1 \leq i \leq n-1$, then any two nodes $P[i]$ and $P[j]$ with $i < j$ has a path from i to j: $P[i], P[i+1], ..., P[j]$. Thus, G' is "sort-of-connected".

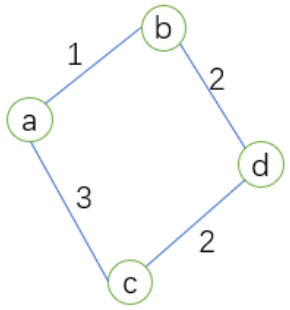
If G' is "sort-of-connected". Define an order on V' as: $u > v$ iff. u has a path to v. Since the graph is acyclic and "sort-of-connected", one can verify that the order is a total order. The topology order of a total order is $P[1...n]$. If $P[i]$ to $P[i+1]$ do not have edge, then there exists $P[j]$ where $P[j] < P[i]$ and $P[i+1] < P[j]$, which is impossible in any case $j < i$ or $j > i+1$. □

5 Problem 5

(a)

Disagree: counterexample is as follow.

- $S = a, b, V - S = c, d$, $A = \{(a, b), (c, d)\}$.
- $e(c, d)$ is a safe edge while it is not a light edge.



(b)

Disagree: counterexample is the same as (a).

- If partition V into $\{a, c\}$ and $\{b, d\}$, then (a, c) will be in the spanning tree.

6 Problem 6

(a)

If we use **Kruscal** to find the M.S.T. of the graph, each selected edge is unique, then the M.S.T. is unique.

(b)

Assume T' is not the M.S.T. of G'. There exists another spanning-tree T'_2,

$$weight(T'_2) < weight(T')$$

. Then $(T - T') \cup T'_2$ is a spanning-tree of G and

$$weight\left((T - T') \cup T'_2\right) < weight(T)$$

, which means T is not a M.S.T. of G.

7 Problem 7

(a)

If not, we assume that T' is the second-best minimum spanning tree which minimize $|E(T') - E(T)|$. By assumption, $|E(T') - E(T)| \geq 2$. Let e be the edge with minimum weight among $T - T'$, we have the following fact:

- There exists exactly one cycle C in $T' \cup e$ which contains e.
- There exists $e \neq e' \in C$ with $w(e') \geq w(e)$.

That leads to contradiction once one spots the fact that $T'' = T' - e' + e$ has smaller weight than T' and $|E(T'') - E(T)| = |E(T') - E(T)| - 1 \geq 1$.

(b)

That can be done via simple dfs. To be more precise, given vertex u, we may calculate $dist(u, v)$ for all $v \in V$ in $O(n)$ via dfs.

(c)

Run Kruskal algorithm, dfs described in (b), and enumerate all possible pair $(u, v) \notin T$.

Sample Solution for Problem Set 9

Data Structures and Algorithms, Fall 2020

January 4, 2021

Contents

1 Problem 1

Note $n \equiv |V|$, $m \equiv |E|$.

- Range from 1 to n:
  - Use counting sort to sort edges, $O(n + m)$.
  - $\Theta(n)$ times Make-Set, $O(n)$.
  - $\Theta(2m)$ times Find-Set, $O(m\alpha(n))$.
  - $O(m)$ times Union, $O(m\alpha(n))$.
  - In total, $O(m\alpha(n))$.
- Range from 1 to W:
  - If $W = O(m \lg m)$, the answer is $O(m\alpha(n) + W)$ (using counting sort).
  - Otherwise, the answer is $O(m \lg m)$ (using merge sort).

2 Problem 2

(a) There is nothing needed to be updated.

(b) Suppose that the edge e is from u to v, and the graph T' is T with the extra edge e. Since T is a tree, T' has a cycle which contains e. We use tree traversal to find this cycle and remove the edge e' with the maximum weight in the cycle. Then we get a new MST.

(c) There is nothing needed to be updated.

(d) Just run Prim algorithm to calculate the new MST. It costs $O((|V| + |E|) \lg |V|)$ time.

Bonus.

Remove e from T, we will get two tree $T_1(V_1, E_1), T_2(V_2, E_2)$.

Find the minimum weight edge e' across T_1, T_2 which means one vertex is in V_1 and the other is in V_2, If $\hat{w}(e') < \hat{w}(e)$, the new MST is $(V, E_1 \cup E_1 \cup \{e'\})$, otherwise the MST is not changed.

3 Problem 3

- Least duration:
  - Counterexample: {[0,3), [2,4), [3,6)}.
  - Greedy: {[2,4)}
  - Truth: {[0,3), [3,6)}
- Fewest overlaps:

  - Counterexample:
  - Greedy: blue and green
  - Truth: blue and yellow
- Earliest start time:
  - Counterexample: {[0,6), [1,2), [3,4)}.
  - Greedy: {[0,6)}
  - Truth: {[1,2), [3,4)}

4 Problem 4

4.1 Algorithm

Choose the most valuable item in the remaining ones repeatedly until knapsack is full.

4.2 Correctness

Correctness holds once we spot the following fact.

- Given items I with least weight and most value, there exists one optimal solution containing this item.

5 Problem 5

Actually, our Huffman code for all character contains 8 bits. To see this, we should notice that sum of frequencies of K elements will never exceed that of $2K$ elements.

6 Problem 6

Algorthm: For convenience assume endpoints are distinct.

Create a new array $K[1...2n]$ where $K = L+R$, i.e., $K[i] = L[i]$ and $K[n+i] = R[i]$ for $1 \leq i \leq n$. Each element $K[i]$ maintain an attribute $K[i].key$ where $K[i].key = L$ if $i \leq n$ and $K[i].key = R$ if $i > n$. Sort the elements in K increasingly in $O(n \log n)$ time (swap two elements will also swap their attribute). Then do the following loop. cnt is an interger initialied to 0, M is an integer initialized to 0.

- For $i = 1$ to $2n$ do
  - If $K[i].key = L$, $cnt \leftarrow cnt + 1$. Else, $cnt \leftarrow cnt - 1$.
  - $M \leftarrow \max(M, cnt)$.

The answer is M.

Complexity: Each loop cost $O(1)$ time and the total time complexity is $O(n \log n)$.

Correctness: Intuitively, cnt is always the number of intervals that cover the point $K[i]$. M will be the max number of cnt. Suppose the optimal coloring uses opt colors. Obviously, we have $opt \geq M$, since we cannot use less than M colors to color the intervals where M of them interates at the same point.

Now we prove $opt \leq M$, i.e., there is a legal coloring way using M colors. We create M colors $C[1...M]$. Create a color set $colors$ intialized to $\emptyset$. In each loop while $K[i].key = L$, we color the interval with left point $K[i]$ with a new color in $C[1...M] \backslash colors$ and add the new color to $colors$. We maintain the following loop invariance:

loop invariance: After the i-the loop, intervals with endpoints in $K[1...i]$ are colored different colors if they interated with each other, and the intervals that cover point $K[i]$ only use colors in set $colors$.

The proof of the loop invariance can be trivially varified.

7 Problem 7

Algorithm: ans is an integer initialied to 0. Suppose the nodes in tree is represented in BFS order and stored in $B[1...n]$ (Which can be done in $O(n)$ time). For each node in the tree, initialize an attribute vis as 0. We do the following loop to find paths and update ans:

- For $i = n$ to 1 do
  - If $B[i].vis = 1$, skip the loop. Otherwise do the following procedure.
  - Let p be the k-th parent of $B[i]$ ($B[i].p$ is the first parent, i.e., $B[i]$ perfom k-moves to the loop and get the k-th parent of $B[i]$). If p do not exists, end the procedure.
  - $ans \leftarrow ans + 1$, mark the attribute vis of all the nodes in the subtree rooted at p as 1 (The procedure can be achieved by running DFS on p).

We define our $DFS(p)$ in the third step of the loop as follows:

- For each child c of p, if $c.vis = 0$ then $c.vis \leftarrow 1$ and $DFS(c)$.

Intuitively, the algorithm find a path in each loop. The path goes from the deepest possible nodes of the tree.

Complexity: The cost for loop is $O(n)$. We need to analysis the cost for the third step(DFS) of the loop. Since each node will be access at most once (change its vis from 0 to 1), the total complexity for DFS is $O(n)$. The total complexity is $O(n)$ where n is the number of nodes.

Correctness: We claim that the path goes from the deepest nodes in a tree must exists in an optimal solution. Denote the deepest node as c and its k-th parent as p. Suppose there is a optimal solution that do not contain the path from c to p. Say a node is occupied by node a if it is on the path started from a in the optimal solution. If p is not occupied, then any node in the subtree of p can not be occupied (Otherwise p must be occupied since c is the deepest node in the tree), in which case we can add a path from c to p, contradicting the fact that it is a optimal solution. If p is occupied by node a, a must exist in the subtree rooted at p. Due to the same reason (c is the deepest node in the tree), other nodes in the subtree of p must not be occupied. We delete the path started at a, and add the path from c to p, then we get the optimal solution with the same number of path containint the path from c to p.

By marking the vis to 1 for each node in the subtree of p, we delete the subtree from the tree. We claim that the optimal solution on the remaining tree adding the path from c to p is the optimal solution. Otherwise, if there exists a better solution, since we have proved that the path from c to p must exists in the solution, we can delete the subtree of p and also get a better solution in the deleted tree.

Sample Solution for Problem Set 10

Data Structures and Algorithms, Fall 2020

January 4, 2021

Contents

1 Problem 1

Algorithm: Let $P'[1...n]$ be n pairs $(P'[i].key, P'[i].ID)$ where $P'[i].key = P[i]$ and $P'[i].ID = i$. Similary, let $S'[1..n]$ be n pairs $(S'[i].key, S'[i].ID)$ where $S'[i].key = S[i]$ and $S'[i].ID = i$.

Sort the array P' by increasing order on $P'[i].key$, and sort the array S' by increasing order on $S'[i].key$. Let $\pi(P'[i].ID) = S'[i].ID$.

In other words, we sort P and S by increasing order and assign the shoes $S[i]$ to $P[i]$.

Complexity: Sorting procedure cost $O(n \log n)$ if we use merge sort. Other operation cost $O(n)$. The total time complexity is $O(n \log n)$.

Correctness: Suppose P and S are the array of sizes of feet and shoes in increasing order. Suppose there is an optimal solution π_{opt} such that there exists $1 \leq i < j \leq n$ and π_{opt} assign shoes $S[j']$ to $P[i]$ and $S[i']$ to $P[j]$ where $i' < j'$. We consider another optimal solution π'_{opt} that assign shoes $S[i']$ to $P[i]$ and $S[j']$ to $P[j]$, while other assignment is the same as π'_{opt}. Let $cost[\pi_{opt}]$ be the average difference between the size of a person's feet and the size of his/her assigned pair of shoes. Then $cost[\pi_{opt}] - cost[\pi'_{opt}]$ is

$$\Delta \triangleq \left|P[i] - S[j']\right| + \left|P[j] - S[i']\right| - \left|P[i] - S[i']\right| - \left|P[j] - S[j']\right|$$

With out loss of generality, we assume $P[i] \geq S[i']$ (Otherwise we can swap P and S).

If $P[i] \geq S[j']$, then $\Delta = 0$.

If $P[i] \leq S[j'] \leq P[j]$, then $\Delta = 2\left|S[j'] - P[i]\right| \geq 0$.

If $S[j'] \geq P[j]$, then $\Delta = 2\left|P[i] - P[j]\right| \geq 0$.

In all cases, π'_{opt} is still an optimal solution. Since any permutation can get to the increasing permutation by finite steps of swapping two inverse element, the increasing permutation in our algorithm is an optimal solution.

2 Bonus

Algorithm: Initially let $U' = U$, $S' = S$ and $cost = 0$. Do the following loop:

- While $U' \neq \emptyset$, do
  - For $s \in S'$, define the weight of s as $w(s) = \frac{|s \cap U'|}{c(s)}$. Let $s_m \in S'$ be the element in S' with the biggest weight.
  - $cost \leftarrow cost + c(s_m)$, $S' \leftarrow S' \backslash \{s_m\}$, $U' \leftarrow U' \backslash s$.
- Return $cost$.

Complexity: Each loop cost $O(n)$ to find the element with the biggest weight. There are at most n loop (We assume each element can be covered by at least one set). The complexity is $O(n^2)$.

Correctness: Suppose in the i-th loop, we choose s_i as the element with the biggest weight (i.e., denote s_m as s_i in the second line of the algorithm for the i-th loop). Denote U' before the i-th loop as U_i, denote S' before the i-th loop as S_i. Obviously we have $U_1 = U$.

Suppose $S_{OPT} \subseteq S$ is one of the optimal solution and $OPT = \sum_{s \in S_{OPT}} c(s)$. We claim that $w(s_i) \geq \frac{|U_i|}{OPT}$. Otherwise, since $w(s_i)$ is the biggest among all the weights of elements in U_i, we have $c(s) > |s \cap U_i| \cdot \frac{OPT}{|U_i|}$ for all $s \in U_i$. Since $\cup_{s \in S_{OPT}} s = U$, and for all $s \in S \backslash S_i$ we have $s \in U \backslash U'$, we get $\cap_{s \in S_{OPT} \cap S_i} = U_i$, which means $\sum_{s \in S_{OPT} \cap S_i} \geq |U_i|$. Thus, $\sum_{s \in S_{OPT} \cap S_i} c(s) > OPT$, which is impossible.

Now $c(s_i) \leq |s_i \cap U_i| \cdot \frac{OPT}{|U_i|}$. Write $k_i = |U_i|$. Obviously $|s_i \cap U_i| = k_i - k_{i+1}$ and we get

$$c(s_i) \leq \frac{k_i - k_{i+1}}{k_i} \cdot OPT \leq \sum_{k_{i+1} < j \leq k_i} \frac{1}{j} \cdot OPT$$

Thus the total cost of our approximation algorithm is

$$\sum_i c(s_i) \leq \sum_i \sum_{k_{i+1} < j \leq k_i} \frac{1}{j} \cdot OPT$$

Since we have $k_1 = n$ and $k_1 > k_2 > ....$, while finally get 1, the value is

$$\sum_{1 \leq j \leq n} \frac{1}{j} \cdot OPT = O\left(\ln n \cdot OPT\right)$$

iteration	1	2	3	4	5	6	7	8	final
A	0								0
B	1								1
C		3							3
D			4						4
E	4								4
F	8	7				6			6
G		7	5						5
H			6						6

3 Problem 2

3.1 (a)

Use the graph in (c) as example:
Minimum spanning tree: $\{$(A,B),(B,C),(C,D),(A,E),(D,G),(G,F),(G,H)$\}$
Shortest path tree: $\{$(A,B),(B,C),(C,D),(A,E),(C,G),(G,F),(G,H)$\}$
They are not identical.

3.2 (b)

Use $flag$ to record whether the distance of nodes changed during the m-th iteration. If not, break.

3.3 (c)

The final shortest-path tree: $\{$(A,B),(B,C),(C,D),(A,E),(C,G),(G,F),(G,H)$\}$.

4 Problem 3

4.1 (a)

Use DFS, only consider the edge that $l_e \leq L$.

4.2 (b)

Use Kruscal, judge whether $Find(s) == Find(t)$ after each $Union(u, , v)$. If so, let $L = l_e(u, v)$ and break. The fuel tank has a capacity of at least L.

Proof: Call the edge that merges s and t into the same set (u_0, v_0). If $L < l_e(u_0, v, 0)$, we can't have enough oil to go through (u_0, v_0). In that case, s and t are not connected. If $L \geq l_e(u_0, v_0)$, we can go from s to t along the minimum spanning tree.

5 Problem 4

Initially, we will make each vertex have a D value of 0, which corresponds to taking a path of length zero starting at that vertex. Then, we relax along each edge exactly $V - 1$ times. Then, we do one final round of relaxation, which if any thing changes, indicated the existence of a negative weight cycle. The code for this algorithm is identical to that for Bellman ford, instead of initializing the values to be infinity except at the source which is zero, we initialize every d value to be 0.

Another solution. Use $\min(w_{u,v}, d_u + w_{u,v})$ to relax d_v. Then d_v will be $\min_{u \in V \wedge u \neq v}\{\text{dist}(u, v)\}$.

6 Problem 5

Give weight 0 to each edge in original graph. Split a vertex v into two vertexes v_0, v_1, and add a new edge (v_0, v_1) with weight v_w. Then, every path through this dual graph corresponds to a path through the original graph.

A job can start if all its previous jobs has finished.

(a) Set d_{s_0} to 0, and then use PERT algorithm in the textbook to calculate the "shortest" paths to each vertex.

Then d_{v_0} is the earliest start time of the job represented by v.

(b) Set d'_{t_1} to 0, and reverse every edge. Then use PERT algorithm in the textbook to calculate the "shortest" paths to each vertex.

Then $d_{t_1} - d'_{v_0}$ is the latest start time of the job represented by v, without affecting the project's duration.

(c) Use d to find the "shortest" paths from s to t. Then v is in some critical path if and only if $v_0(v_1)$ is in some "shortest" paths.

7 Problem 6

Consider the following graph $G = (V, E, w)$.

- $V = \{v_1, v_2, \ldots, v_n\}$.
- for $1 \leq i, j \leq n$ such that $i \neq j$, $v_i v_j \in E$, and $w(v_i, v_j) = -\log r_{ij}$.

We may run Bellman-ford algorithm on it for detecting the presence of anomaly and calculate the most advantageous sequence of currency exchanges for converting currency s into currency t. Time complexity for this algorithm is $O(n^3)$.

Sample Solution for Problem Set 11

Data Structures and Algorithms, Fall 2020

January 4, 2021

Contents

1 Problem 1

(a) $k = 1$

$$\begin{pmatrix} 0 & \infty & \infty & \infty & -1 & \infty \\ 1 & 0 & \infty & 2 & 0 & \infty \\ \infty & 2 & 0 & \infty & \infty & -8 \\ -4 & \infty & \infty & 0 & -5 & \infty \\ \infty & 7 & \infty & \infty & 0 & \infty \\ \infty & 5 & 10 & \infty & \infty & 0 \end{pmatrix}$$

$k = 2$

$$\begin{pmatrix} 0 & \infty & \infty & \infty & -1 & \infty \\ 1 & 0 & \infty & 2 & 0 & \infty \\ 3 & 2 & 0 & 4 & 2 & -8 \\ -4 & \infty & \infty & 0 & -5 & \infty \\ 8 & 7 & \infty & 9 & 0 & \infty \\ 6 & 5 & 10 & 7 & 5 & 0 \end{pmatrix}$$

$k = 3$

$$\begin{pmatrix} 0 & \infty & \infty & \infty & -1 & \infty \\ 1 & 0 & \infty & 2 & 0 & \infty \\ 3 & 2 & 0 & 4 & 2 & -8 \\ -4 & \infty & \infty & 0 & -5 & \infty \\ 8 & 7 & \infty & 9 & 0 & \infty \\ 6 & 5 & 10 & 7 & 5 & 0 \end{pmatrix}$$

$k = 4$

$$\begin{pmatrix} 0 & \infty & \infty & \infty & -1 & \infty \\ -2 & 0 & \infty & 2 & -3 & \infty \\ 0 & 2 & 0 & 4 & -1 & -8 \\ -4 & \infty & \infty & 0 & -5 & \infty \\ 5 & 7 & \infty & 9 & 0 & \infty \\ 3 & 5 & 10 & 7 & 2 & 0 \end{pmatrix}$$

$k = 5$

$$\begin{pmatrix} 0 & 6 & \infty & 8 & -1 & \infty \\ -2 & 0 & \infty & 2 & -3 & \infty \\ 0 & 2 & 0 & 4 & -1 & -8 \\ -4 & 2 & \infty & 0 & -5 & \infty \\ 5 & 7 & \infty & 9 & 0 & \infty \\ 3 & 5 & 10 & 7 & 2 & 0 \end{pmatrix}$$

$k = 6$

$$\begin{pmatrix} 0 & 6 & \infty & 8 & -1 & \infty \\ -2 & 0 & \infty & 2 & -3 & \infty \\ -5 & -3 & 0 & -1 & -6 & -8 \\ -4 & 2 & \infty & 0 & -5 & \infty \\ 5 & 7 & \infty & 9 & 0 & \infty \\ 3 & 5 & 10 & 7 & 2 & 0 \end{pmatrix}$$

(b) If there was a negative-weight cycle, there would be a negative number occurring on the diagonal upon termination of the Floyd-Warshall algorithm.

2 Problem 2

(a) Here is the values of h and $\hat{w}$ computed by the algorithm to simplify the solution.

v	$h(v)$
1	-5
2	-3
3	0
4	-1
5	-6
6	-8

u	v	$\hat{w}(u,v)$	u	v	$\hat{w}(u,v)$
1	2	NIL	4	1	0
1	3	NIL	4	2	NIL
1	4	NIL	4	3	NIL
1	5	0	4	5	8
1	6	NIL	4	6	NIL
2	1	3	5	1	NIL
2	3	NIL	5	2	4
2	4	0	5	3	NIL
2	5	NIL	5	4	NIL
2	6	NIL	5	6	NIL
3	1	NIL	6	1	NIL
3	2	5	6	2	0
3	4	NIL	6	3	2
3	5	NIL	6	4	NIL
3	6	0	6	5	NIL

So, the d_{ij} values that we get are

$$\begin{pmatrix} 0 & 6 & \infty & 8 & -1 & \infty \\ -2 & 0 & \infty & 2 & -3 & \infty \\ -5 & -3 & 0 & -1 & -6 & -8 \\ -4 & 2 & \infty & 0 & -5 & \infty \\ 5 & 7 & \infty & 9 & 0 & \infty \\ 3 & 5 & 10 & 7 & 2 & 0 \end{pmatrix}$$

.

(b) By lemma 25.1, we have that the total weight of any cycles is unchanged as a result of the reweighting procedure. This can be seen in a way similar to how the last claim of lemma 25.1 was proven. Namely, we consider the cycle c as a path that has the same starting and ending vertices, so, by the first half od lemma 25.1, we have that

$$\hat{w}(c) = w(c) + h(v_0) - h(v_k) = w(c) = 0$$

This means that in the reweighted graph, we still have that the same cycle as before had a total weight of zero. Since there are no longer any negative weight edges after we reweight, this is precicely the second property of the reweighting procedure shown in the section. Since we have that the sum of all the edge weights in c is still equal to zero, but each of them individually has a nonnegative weight, it must be the case that each of them individually is equal to 0.

3 Problem 3

3.1 (a)

Update(u,v): For u and its ancestors, they can reach v and its descendants.

Algorithm 1: UPDATE(u, v)

```
1 for i ∈ V do
2     if T[i][u] or i == u then
3         for j ∈ V do
4             if T[v][j] or j == v then
5                 T[i][j] = true;
6             end
7         end
8     end
9 end
```

3.2 (b)

Assume $G =< V, E >$, $V = \{v_1, v_2, \ldots, v_n\}$, $E = \{(v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n)\}$. Now add $e = (v_n, v_1)$. We need update $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$ terms. Therefore, $\Omega(|V|^2)$ time is required for any algorithm.

3.3 (c)

Update2(u,v): Modify the algorithm in (a) by adding a condition.

Correctness:
If $T[i][v]$ is true, no transitive closure need to update.
If $T[i][v]$ is false, the same as the algorithm in (a).

Time complexity:
Use aggregate analysis. If $T[i][v]$ is **false**, the inner loop can make it **true**. Since T has $\Theta(n^2)$ elements, the inner loop (line 3-5) executes $O(n^2)$ times and costs $O(n^3)$. Since there are $x \in O(n^2)$ insertions, the outer loop (line 1-2) executes x times and cost $\Theta(x * n) = O(n^3)$. Therefore, this algorithm is $O(n^3)$.

Algorithm 2: UPDATE2(u, v)

```
1 for i ∈ V do
2     if (T[i][u] or i == u) and ¬T[i][v] then
3         for j ∈ V do
4             if T[v][j] or j == v then
5                 T[i][j] = true;
6             end
7         end
8     end
9 end
```

4 Problem 4

4.1 (a)

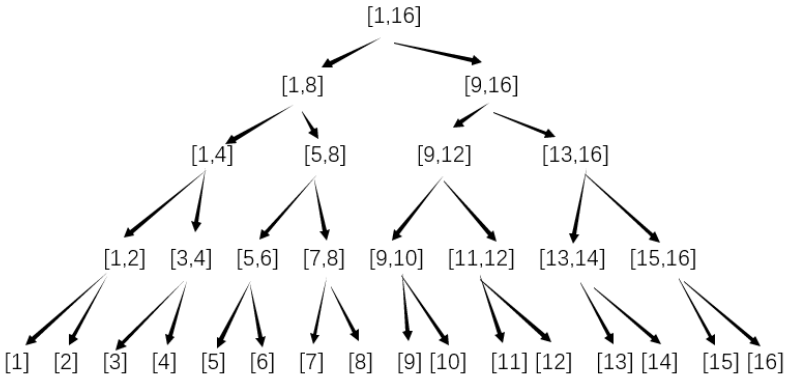
Denote the length of the subproblem $[l, r]$ as $k \equiv (r - l + 1)$. There are $(n + 1 - k)$ subproblems of length k:

$$|V| = \sum_{k=1}^{n}(n + 1 - k) = \frac{n(n+1)}{2}$$

For each problem of length k, there are $2(k - 1)$ subproblems:

$$|E| = \sum_{k=1}^{n}(n + 1 - k) * 2(k - 1) = \frac{n(n+1)(n-1)}{3}$$

4.2 (b)



There are no overlapping subproblems.

4.3 (c)

Counterexample: $n = 4, d = 10, r_{12} = 2, r_{13} = 3, r_{14} = 5, r_{23} = 2, r_{24} = 1, r_{34} = 2, c_1 = 0, c_2 = 0, c_3 = 1000$.
The optimal solution is $1 \to 3 \to 4$, which is equal to $dr_{13}r_{34} - c_1 - c_2 = 60$. However, if you only need to exchange 1 to 3, the optimal solution is $1 \to 2 \to 3$. Therefore, when c_k is arbitrary, there is no optimal substructure for this problem.

5 Problem 5

(a)

Consider $S = \{1, 9, 10\}$ and $target = 18$.

(b)

Let x_i the number of coins with value c_i. The statement follows from the following fact.

- $0 \leq x_i < p$ for all $1 \leq i < k$ if $\mathbf{x} = (x_1, x_2, \ldots, x_k)$ is optimal. (Note that $\mathbf{x}$ is determined with that constraint, and is exactly the same with the one produced in greedy algorithm)

(c)

Let dp_u the number of coins for value u, then $dp_u = 1 + \min_{\substack{1 \leq i \leq k, \\ u \geq c_k}} dp_{u-c_k}$

6 Problem 6

6.1 Dynamic Programming

(a) Suppose the given string is $A[1...n]$.

Overview: We use $dp[i][j]$ to denote the longest subsequence of $A[i...j]$ that is a palindrome. We have the following transition function (omit the base case)

$$dp[i][j] = \begin{cases} dp[i+1][j-1]+2 & A[i] = A[j] \\ \max(dp[i+1][j], dp[i, j-1]) & A[i] \neq A[j] \end{cases} \tag{1}$$

Top-down implementation: Initialize $dp[i][j]$ to -1 for all $0 \leq i, j \leq 1$. Then call $FindAns(1, n)$. $FindAns(l, r)$ is defined as following:

- **(Base case)**: If $l == r$, return 1. If $l > r$, return 0.
- **(Memorize)**: If $dp[i][j] \geq 0$, return $dp[i][j]$.
- If $A[l] == A[r]$, let $dp[l][r] \leftarrow FindAns(l+1, r-1) + 2$.
- Else, let $dp[l][r] \leftarrow \max(FindAns(i+1, j), FindAns(i, j-1))$.
- Return $dp[l][r]$.

Down-top implementation: Initialize $dp[i][i] = 1$ $dp[i][i-1] = 0$ for all $0 \leq i \leq n+1$, then do the following loop.

- For k from 1 to $n-1$
  - For i from 1 to $n-k$
    - Let $j \leftarrow i + k$.
    - If $A[l] == A[r]$, let $dp[l][r] \leftarrow dp[l+1, r-1] + 2$.
    - Else, let $dp[l][r] \leftarrow \max(dp[i+1][j], dp[i, j-1])$.

Then the ans is $dp[1][n]$.

Complexity: Remember that the complexity for dynamic programming is (number of states)$\times$(complexity for transition). The number of states is $O(n^2)$ and the complexity for transition is $O(1)$, the total time complexity is $O(n^2)$.

(b) Suppose the given is string is $A[1...n]$.

Overview: We use $dp[i][j]$ to denote the shortest palindrome supersequence of $A[i][j]$, then we have the following transition function:

$$dp[i][j] = \begin{cases} dp[i+1][j-1]+2 & A[i] = A[j] \\ \max(dp[i+1][j], dp[i, j-1]) + 2 & A[i] \neq A[j] \end{cases} \tag{2}$$

The base case is $dp[i][i] = 1$ and $dp[i][i-1] = 0$ for all $0 \leq i \leq n+1$. The implementation of dynamic programming is described in (a).

6.2 Other algorithms

(a) Let $A[1...n]$ be the given string and $A'[1...n] = A[n...1]$, i.e., $A'[i] = A[n+1-i]$ for $1 \leq i \leq n$. Let the LCS (longest common subsequence) of A, A' be $LCS(A, A')$. We claim that the ans is $LCS(A, A')$.

Remark: The conclusion is **non-trivial**. Suppose the longest palindrome subsequence of $A[i...j]$ is sol. It is easy to prove $LCS(A, A') \geq sol$, since any palindrome subsequence form an equal length LCS between A and A'. But is **HARD** to prove $sol \geq LCS(A, A')$, i.e., any LCS of A, A' can form a palindrome subsequence with good length. it is highly recommended to consider proving it. If you want to get the proof, feel free to talk to me.

(b) Let $A[1...n]$ be the given string. Let sol be the longest palindrome subsequence of $A[i...j]$. The answer is $2n - sol$.

Similarly, to prove the correctness, you need to prove both sides: $ans \geq 2n-sol$ and $ans \leq 2n-sol$.

7 Problem 7

First choose an arbitrary vertex $r \in T$ as the root of T and use DFS to find the parent of u $u.p$ and the children of u for each $u \in T$. Suppose the parent if the root $r.p = -1$. This problem uses recursion on tree. Let the weight of edge (u, v) be $w(u, v)$. Let the weight of vertex u be $w(u)$.

(a) Algorithm overview: we let $dp[u]$ for each u in T be the path with the max weight from u down to the leaf (not required to get to one of the leaves). The algorithm first complute $dp[u]$ us recursion on tree and find $\max(dp[v_1] + w(u, v_1), 0) + \max(0, dp[v_2] + w(u, v_2))$ for each $u \in T$ and v_1, v_2 are two of u's children with the max value of $dp[v_2] + w(u, v_2)$.

Algorithm: Run $DFS(r)$ to compute dp. $DFS(u)$ for $u \in T$ is defined as following:

- If u has no children (u is a leaf), then let $dp[u] = 0$.
- Else, for each child v of u, call $DFS(v)$. Let v_m be the child of u with the max $dp[v_m]+w(u, v_m)$. Let $dp[u] = \max(0, dp[v_m] + w(u, v_m))$.

For each $u \in T$, compute $dp'[u]$ as following:

- $max_1 = -\infty, max_2 = -\infty$.
- For each child v of u,
  - If $dp[v] + w(u, v) > max_1$, then $max_2 \leftarrow max_1$ and $max_1 \leftarrow dp[v] + w(u, v)$.
  - Else, if $dp[v] + w(u, v) > max_2$, then $max_2 \leftarrow dp[v] + w(u, v)$.
- Let $dp'[u]$ be $\max(max_1, 0) + \max(max_2, 0)$.

The final answer is the following value

$$\max_{u \in T} dp'[u]$$

Complexity: DFS takes $O(n)$ time, compute $dp'[u]$ use $O(n)$ time, find answer use $O(n)$ time. The total complexity is $O(n)$.

(b) Algorithm overview: we let $dp[u]$ for each u in T be the weight of the max weight subtree of T that rooted at u. Then $dp[u] = w(u) + \sum_{v \in C(u), dp[v] > 0} dp[v]$ where $C(u)$ is the set of chilren of u.

Algorithm: Run $DFS(r)$ to compute dp. $DFS(u)$ for $u \in T$ is defined as following:

- Let $dp[u] \leftarrow w(u)$.
- For each child v of u, call $DFS(v)$. If $dp[v] > 0$, let $dp[u] \leftarrow dp[u] + dp[v]$.

The final answer is the following value

$$\max_{u \in T} dp[u]$$

Complexity: Trivially $O(n)$.

Sample Solution for Problem Set 12

Data Structures and Algorithms, Fall 2020

January 4, 2021

Contents

1 Problem 1

Algorithm Overview: Use dynamic programming. Let $dp[i][j] = True$ means wee can parenthesize $S[i....j]$ to make the value be $True$. Then we have the transition function:

$$dp[i][j] = \bigvee_{i \le k \le j, S[k] \in \{\wedge, \vee, \oplus\}} dp[i][k-1] S[k] dp[k+1][j] \tag{1}$$

Implementation: We use a simple down-top dynamic programming. Initialize $dp[i][i] = S[i]$ for all even i. Then do the following loop

- For $k = 2, 4, 6, 8, ...$ until $k = 2n$, do
  - For $i = 0, 2, 4, 6, ...$ until $i + k = 2n$, let $j = i + k$, do
    - $dp[i][j] = \bigvee_{i \le k \le j, S[k] \in \{\wedge, \vee, \oplus\}} dp[i][k-1] S[k] dp[k+1][j]$

Complexity: $O(n^3)$. **Recall that the complexity of dynamic programming is the number of states ($O(n^2)$) times the complexity for transition function ($O(n)$).**

2 Problem 2

Let $f_{max}(i)$ represent the largest product in the subarrays ending with i, and $f_{min}(i)$ represent the smallest product in the subarrays ending with i.

Algorithm 1: LARGESTPRODUCT(A, i, j)

$f_{max} = f_{min} = ans = 1$;
for $i = 1$ *to* n **do**
    $temp = f_{max}$;
    $f_{max} = \max(A[i], f_{max} * A[i], f_{min} * A[i])$;
    $f_{min} = \min(A[i], temp * A[i], f_{min} * A[i])$;
    $ans = \max(ans, f_{max})$;
end
return ans;

3 Problem 3

First sort all the points based on their x coordinate. To index our subproblem, we will give the rightmost point for both the path going to the left and the path going to the right. Then, we have that the desired result will be the subproblem indexed by v, where v is the rightmost point. Suppose by symmetry that we are further along on the left-going path, that the leftmost path is going to the ith one and the right going path is going until the jth one. Then, if we have that $i > j + 1$, then we have that the cost must be the distance from the $i - 1$st point to the ith plus the solution to the subproblem obtained where we replace i with $i - 1$. There can be at most $O(n^2)$ of these subproblem, but solving them only requires considering a constant number of cases. The other possibility for a subproblem is that $j \leq i \leq j + 1$. In this case, we consider for every k from 1 to j the subproblem where we replace i with k plus the cost from kth point to the ith point and take the minimum over all of them. This case requires considering $O(n)$ things, but there are only $O(n)$ such cases. So, the final runtime is $O(n^2)$.

4 Problem 4

(a) To index the subproblem, we will give the first k items from the item list and the maximum weight of current knapsack. There can be at most $O(nW)$ of these subproblem. For each item, we can decide to put it in or not. If we put the ist item in, we need to prepare w_i weight for it and earn v_i value. The value is v_i plus the solution to the subproblem with the first $i-1$ items, and $W-w_i$ capacity. Otherwise, the value is as same as the solution to the subproblem with the first $i-1$ items, and W capacity. We take the maximum value from the two choices for each item.

(b) No since W can increase exponential as input length.

5 Problem 5

From the definition, there is an algorithm A_1 solves L_1 in $O(n^{k_1})$, and A_2 solves L_2 in $O(n^{k_2})$.

5.1 $L_1 \cup L_2$

Algorithm 2: $A_3(x)$

```
if A_1(x) then
    return true;
end
if A_2(x) then
    return true;
end
return false;
```

A_3 solves $L_1 \cup L_2$ in $O(n^{\max(k_1,k_2)})$, so $L_1 \cup L_2 \in P$.

5.2 $L_1 \cap L_2$

Algorithm 3: $A_3(x)$

```
if A_1(x) then
    if A_2(x) then
        return true;
    end
end
return false;
```

A_4 solves $L_1 \cap L_2$ in $O(n^{\max(k_1,k_2)})$, so $L_1 \cap L_2 \in P$.

5.3 $L_1 L_2$

Algorithm 4: $A_5(x)$

```
for i = 1 to (n − 1) do
    if A_1(x_1 ... x_i) then
        if A_2(x_{i+1} ... x_n) then
            return true;
        end
    end
end
return false;
```

A_5 solves $L_1 L_2$ in $O(n^{\max(k_1,k_2)+1})$, so $L_1 L_2 \in P$.

5.4 $\overline{L}_1$

A_6 solves $\overline{L}_1$ in $O(n^{k_1})$, so $\overline{L}_1 \in P$.

Algorithm 5: $A_6(x)$

```
if A_1(x) then
    return false;
end
return true;
```

5.5 L_1^*

Let $f[i][j]$ indicate whether $x_i \dots x_j$ is belong to L_1^*.

$\forall x \in L_1^*, x \in L_1$ or $\exists k, x_1 \dots x_k \in L_1^* \wedge x_{k+1} \dots x_n \in L_1^*$.

Therefore, we can construct a dynamic programming algorithm A_7. A_7 solves L_1^* in $O(n^{k_1+2})$, so $L_1^* \in P$.

Algorithm 6: $A_7(x)$

```
for i = 1 to n do
    for j = i + 1 to n do
        f[i][j] = false;
    end
end
for i = 1 to n do
    for j = i to n do
        if A_1(x_i ... x_j) then
            f[i][j] = true;
        end
    end
end
for i = 1 to n do
    for j = i + 1 to n do
        for k = i to (j − 1) do
            if f[i][k] and f[k + 1][j] then
                f[i][j] = true;
            end
        end
    end
end
return f[1][n];
```

6 Problem 6

(a)

Consider (deterministic polynomial time) Turing Machine M defined as follows: Let $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$, Turing Machine M accept input (G_1, G_2, π) if and only if π is a bijection between G_1 and G_2, and $\pi(u)\pi(v) \in E_2$ if and only if $uv \in E_1$.

(b)

Note that $\overline{TAUTOLOGY}$ is the language of boolean formulas which can be false for some input.

7 Problem 7

(a)

for any language $L \in P$, there exists a (deterministic polynomial time) Turing Machine M defined as follows: $M(x) = 1$ if and only if $x \in L$. Therefore, we may define a (deterministic polynomial time) Turing Machine M' in the following way.

- for any x and proof w, $M'(x, w) = 1 - M(x)$.

It can be seen that $\bar{L} \in NP$. Hence, $L \in coNP$.

(b)

If not, $L \in NP \Rightarrow L \in P \Rightarrow \bar{L} \in P \Rightarrow \bar{L} \in NP \Rightarrow L \in coNP$. Similarly, we have $L \in coNP \Rightarrow L \in NP$, leading to contradiction.

Sample Solution for Problem Set 13

Data Structures and Algorithms, Fall 2020

January 4, 2021

Contents

1 Problem 1

(a) Suppose that $\emptyset$ is complete for P. Let $L = \{0,1\}*$. Then L is clearly in P, and there exists a polynomial time reduction function f such that $x \in \emptyset$ if and only if $f(x) \in L$. However, it's never true that $x \in \emptyset$, so this means it's never true that $f(x) \in L$, a contradiction since every input is in L.

Now suppose $\{0,1\}*$ is complete for P, Let $L' = \emptyset$. Then L' is in P and there exists a polynomial time reduction function f'. Then $x \in \{0,1\}*$ if and only if $f'(x) \in L'$. However x is always in $\{0,1\}*$, so this implies $f'(x) \in L'$ is always true, a contradiction because no binary input is in L'.

Finally, let L be some language in P which is not $\emptyset$ or $\{0,1\}*$, and let L' be any other language in P. Let $y_1 \notin L$ and $y_2 \in L$. Since $L' \in P$, there exists a polynomial time algorithm A which returns 0 if $x \notin L'$ and 1 if $x \in L'$. Define $f(x) = y_1$ if $A(x)$ returns 0 and $f(x) = y_2$ if $A(x)$ returns 1. Then f is computable in polynomial time and $x \in L'$ if and only if $f(x) \in L$. Thus, $L' \leq_P L$.

(b) Since L is in NP, we have that $\overline{L} \in coNP$. Since every $coNP$ language has its complement in NP, suppose that we let S be any language in $coNP$ and let $\overline{S}$ be its complement, Suppose that we have some polynomial time reduction f from $\overline{S} \in NP$ to L. Then consider using the same reduction function. We will have that $x \in S \iff x \notin \overline{S} \iff f(x) \notin L \iff f(x) \in \overline{L}$. This shows that this choice of reduction function does work. So we have shown that the complement of any NP complete problem is also $coNP$ complete. To see the other direction, we just negate everything and the proof goes through identically.

2 Problem 2

(a) Let A denote the polynomial time algorithm which returns 1 if input x is a satisfiable formula, and 0 otherwise. We'll define an algorihm A' to give a satisfying assignment. Let $x_1, x_2, \ldots, x_m$ be the input variables. In polynomial time, A' computes the boolean formula x' ehich results from replacing x_1 with true. Then A' runs A on this formula. If it is satisfiable, then we have reduced the problem to finding a satisfying assignment for x' with input variables $x_2, \ldots, x_m$, so A' recursively calls it self. If x' is not satisfiable, then we set x_1 to false, and need to find a satisfying assignment for the formula x' obtained by replacing x_1 in x by false. Again, A' recursively calls it self to do this. If $m = 1$, A' takes a polynomial-time brute force approach by evaluating the formula when x_m is true, and when x_m is false. Let $T(n)$ denote the runtime of A' on a formula whose encoding is of length n. Note that we must have $m \leq n$. Then we have $T(n) = O(n^k) + T(n')$ for some k and $n' = |x'|$ and $T(1) = O(1)$. Since we make a recursive call once for each input variable there are m recursive calls, and the input size strictly decreases each time, so the overall runtime is still polynomial.

(b) Suppose that the original formula was $\wedge_i(x_i \vee y_i)$, and the set of variables were $\{a_i\}$. Then consider the directed graph which has a vertex corresponding both to wach variable, and each negation of a variable. Then for each of the clauses $x \vee y$, we will place an edge going from $\neg x$ to y and an edge from $\neg y$ to x. Then anytime that there an edge in the directed graph, that means if the vertex the edge is coming from is true, the vertex the edge is going to has to be true. Then, what we would need to see in order to say that the formula is satisfiable is a path from a vertex to the negation of that vertex, or vice versa. The naive way of doing this would be to run all pairs shortest path, and see if there is a path from a vertex to its negation. This takes time $O(n^3)$.

3 Problem 3

For any 3-SAT with clauses $C_1, C_2, \ldots, C_m$ where $C_i = l_{i_1} \vee l_{i_2} \vee l_{i_3}$, consider the following zero-one integer programming problem

- The i-th constraint is $y_{i_1} + y_{i_2} + y_{i_3} \geq 1$, where $y_{i_j} = z_{i_j}$ if $l_{i_j} = x_{i_j}$, and $y_{i_j} = 1 - z_{i_j}$ otherwise.

It can be seen that given 3-SAT is satisfiable if and only if corresponding zero-one integer programming problem can be satisfied.

4 Problem 4

(a) For each vertex $v \in V$, create two vertices $v_1, v_2 \in V'$, and an edge $(v_1, v_2) \in E'$ such that the capacity of (v_1, v_2) is $l(v)$. For each edge $(u, v) \in E$, create an edge $(u_2, v_1) \in E'$. Then we have $|V'| = 2|V|$ and $|E'| = |E| + |V|$.

(b) According to flow decomposition theorem, each flow can be decomposed to several $s - t$ path and circles. Thus, after decomposition, there must be a circle with flow 1 that contain (v, s). By deleting the circle we can get a flow f' such that $f'(v, s) = 0$. Algorithm: Find a path from s to v in the flow graph to find a circle containning (v, s). Decrease the flow in each edge in the circle by 1 to create f'.

(c) $2\min(|L|, |R|) - 1$. Example: consider $L = \{l_1, l_2, ..., l_n\}$ and $R = \{r_1, r_2, ..., r_n\}$ and edges $(l_1, r_1), (r_1, l_2), (l_2, r_2), (r_2, l_3), ..., (l_n, r_n)$ with capacity 1. An augmenting path $(l_1, r_1, l_2, r_2, ..., l_n, r_n)$ with flow 1.

5 Problem 5

Overview:

Consider the undirected edges to be two directed edges of capacity 1, run the following algorithm.

Algorithm 1: EDGECONNECTIVITY(G)

```
s ← any vertex in G.V;
ans ← +∞;
for t ∈ (G.V − {s}) do
    maxflow = MAXIMUMFLOW(G, s, t);
    ans = min(ans, maxflow) ;
end
return ans;
```

Correctness:

According to the MAXIMUM FLOW MINIMUM CUT THEOREM, MAXIMUMFLOW(G, s, t) is equal to the minimum number of edges that s and t are disconnected.

6 Problem 6

Algorithm 2: BUILDGRAPH(M, D, S, c)

Note: M is the set of doctors.
// vertices
$V \leftarrow \{s, t\}$;
for $M_i \in M$ **do**
  $V \leftarrow V \cup \{v_i\}$;
end
for $D_j \in D$ **do**
  $V \leftarrow V \cup \{v_j\}$;
end
for $day_k \in \cup_j D_j$ **do**
  $V \leftarrow V \cup \{v_k\}$;
end
// Edges
$E \leftarrow \emptyset$;
for $M_i \in M$ **do**
  $E \leftarrow E \cup \{(s, v_i)\}$;
  $capacity((s, v_i)) \leftarrow c$;
end
for $S_i \in S$ **do**
  for $D_j \in S_i$ **do**
    $E \leftarrow E \cup \{(v_i, v_j)\}$;
    $capacity((v_i, v_j)) \leftarrow 1$;
  end
end
for $D_j \in D$ **do**
  for $day_k \in D_j$ **do**
    $E \leftarrow E \cup \{(v_j, v_k)\}$;
    $capacity((v_j, v_k)) \leftarrow 1$;
  end
end
for $day_k \in \cup_j D_j$ **do**
  $E \leftarrow E \cup \{(v_k, t)\}$;
  $capacity((v_k, t)) \leftarrow 1$;
end
return $\{\langle V, E\rangle, s, t\}$;

Algorithm 3: SOLVE(M, D, S, c)

$\{G, s, t\} \leftarrow$ BUILDGRAPH(M, D, S, c);
$maxflow \leftarrow$ MAXFLOW(G, s, t);
if $maxflow = |\cup_j D_j|$ **then**
  return true;
end
else
  return false;
end

7 Problem 7

(a)

run dfs/bfs on modified residual network to determine if there exists a path with positive capacity from s to t. It is worth mentioning that modified residual network can be calculated in $O(V + E)$.

(b)

If $f(u, v) < c(u, v)$, decrease $c(u, v)$ by 1; Otherwise, run bfs/dfs to find a positive flow path P containing (u, v) from s to t, decrease $f(x, y)$ by 1 for all $(x, y) \in P$, decrease $c(u, v)$ by 1, and lastly run bfs/dfs on residual network like (a).