CSC 236 HW 4

Check in period: 7/20 - 7/25

About

Please see the guide to hw, and guide to check ins for information and tips for the HW and the check ins!

These homeworks are on lectures and tutorials 1-8 inclusive.

1 Task Scheduling Part 2

This question extends some ideas from HW3. It might be a good idea to review that problem! I have made an optional companion notebook for this problem which you can access here. It contains the prerequisite relationships for the CSC courses.

Let $\prec$ be a valid prerequisites relation[1] over a finite non-empty set of tasks Tasks. For any two tasks $a, b \in \mathrm{Tasks}$, $a \prec b$ if and only if a is a prerequisite for b.

In HW3 we showed, by induction, that there exists an ordering of all the tasks in Tasks that respects $\prec$. In this problem we'll prove the same thing **algorithmically**. I.e. to show that the ordering exists, we'll demonstrate an algorithm that finds the ordering!

In this problem, we'll view the prerequisite relation $\prec$ over Tasks as a graph. Specifically, the vertices will be the tasks and there will be a directed edge (a, b) if and only if a is a direct prerequisite of b. I.e. $a \prec b$ and there does not exists a different task c such that $a \prec c$, and $c \prec b$. Formally, define $G_{\mathrm{Tasks},\prec} = (\mathrm{Tasks}, E)$ where $E = \{(a, b) : (a \prec b) \wedge \neg\exists c.(a \prec c \wedge c \prec b)\}$

Consider the following implementations of isMinimal and findMinimal:

Algorithm 1: isMinimal($a, G = (V, E)$)

for $b \in V$ **do**
  if $(b, a) \in E$ **then**
    return False
  end
end
return True

Algorithm 2: findMinimal($G = (V, E)$)

for $a \in V$ **do**
  if isMinimal(a, V, E) **then**
    return a
  end
end

We propose Algorithm 3, FindOrdering1, as an algorithm to find an ordering of Tasks that respects $\prec$.

[1] Defined in HW3

Algorithm 3: FindOrdering1($G = (V, E)$)

ordering = [];
$n = \text{len}(V)$;
while len(ordering) $< n$ **do**
    t = `findMinimal`(V, E);
    ordering.append(t);
    V.remove(t);
end
return ordering

Question 1a. Prove that FindOrdering1 is correct. I.e. Let Tasks be any non-empty finite set of tasks and $\prec$ be any valid prerequisites relation over Tasks. Prove that FindOrdering1 on input $G_{\text{Tasks},\prec}$ returns an ordering of all the tasks in Tasks that respects $\prec$.

You may assume findMinimal and isMinimal are correct.

Define the following loop invariant.

$P(i)$. At the end of the ith iteration, the following hold:

a.) $\texttt{ordering}_i = [o_1, ..., o_i]$ such that for any j, k with $j < k$, $o_k \not\prec o_j$. I.e. the ordering is so far valid and has length i.

b.) For all t in $\texttt{ordering}_i$, and $t' \in V_i$, $t' \not\prec t$. Nothing in V_i is a prerequisite for anything in $\texttt{ordering}_i$

c.) V_i and $\texttt{ordering}_i$ form a partition of the tasks. I.e., every task is in exactly one of V or `ordering`.

Initialization. The loop invariant holds at the start of the for loop since $\texttt{ordering}_0 = []$, and V_0 contains all the tasks.

Maintenance. Let $i \in \mathbb{N}$, and assume $P(i)$, we'll show $P(i+1)$.

By $P(i)$.a, $P(i)$.c, and the fact that the loop condition passed, we have that V_i is non-empty and finite. By HW3, there is a minimal element, and by the assumption that `findMinimal`, is correct, t is a minimal element in V_i. It is appended to `ordering` and removed from V. We'll now show the loop invariant holds.

We have $\texttt{ordering}_{i+1} = \texttt{ordering}_i + [t]$, $V_{i+1} = V \setminus \{t\}$.

a.) $P(i+1)$.a holds because $P(i)$.b implies that t is not a prerequisites of anything in $\texttt{ordering}_i$.

b.) Since t is minimal in V_i, and $P(i)$.b, $P(i+1)$.b.

c.) $P(i+1)$.c holds from the fact that we simply moved one task from V to `ordering`

Thus, the loop invariant holds at the end of iteration i.

Termination. Part a.) of the Loop Invariant implies that the loop will terminate after the nth iteration, at which point, $P(n)$.a, and $P(n)$.c imply that $\texttt{ordering}_n$ contains an ordering of all the tasks in Tasks that respects $\prec$.

Here's the story so far.

- HW3: There **exists** an ordering
- Previous problem: We can **find** an ordering

The next chapter has to do with efficiency - how **efficiently** can we find the ordering?

Let $T_1(n,m)$ be the runtime of FindOrdering1 on a graph $G = (V, E)$ where $|V| = n$, and $|E| = m$. Assume V and E are given as sets, and that set operations are constant time.

Question 1b. Prove that the worst case runtime of FindOrdering1 is $O(n^3)$

`isMinimal` takes time $O(n)$ in the worst case since it iterates over all the vertices. `FindMinimal` takes time $O(n^2)$ in the worst case since there may be just one minimal element, and if we get unlucky with the iteration order and it was the last element, we needed to run `isMinimal` $(n-1)$ times.

Then `FindOrdering1` runs `FindMinimal` n times. Thus, in the worst case `FindOrdering` takes time $O(n^3)$.

Question 1c. Find a non-empty finite set of tasks, Tasks, and a valid prerequisite relation $\prec$ over Tasks. Such that, FindOrdering1 run on $G_{\text{Tasks},\prec}$ takes time $\Omega(n^3)$. You can assume that the order of iteration over the sets is bad (works in your favor to prove the lower bound).

Let $\text{Tasks} = [n]$, and $\prec$ be $<$. Note that there is only ever 1 minimal element. Let $|V_i| = n - i + 1 = k_i$, in the worst case, `FindMinimal` tests the minimal element last, and so we do at least $k_i(k_i - 1) + k_i = \Omega(k_i^2)$ work. In the first $n/2$ iterations, we do $\Omega(n^2) + \Omega((n-1)^2) + ... + \Omega((n/2)^2)$ work, which is at least $n/2 \cdot \Omega((n/2)^2) = \Omega(n^3)$, as required.

Algorithm 4 is another proposed algorithm for the same task.

Note that `numPrereqs` and `adjacent` can be computed by iterating over all edges once and, and `minimal` can be computed by iterating over all the vertices. In the algorithm we use Python dictionary and list comprehension syntax.

Algorithm 4: FindOrdering2($G = (V, E)$)

```
ordering = [];
numPrereqs = {b : |{(a, b) ∈ E}|};
adjacent = {a : [b : (a, b) ∈ E]};
minimal = {a : numPrereqs[a] == 0};
for i=0,...,n-1 do
    a = minimal.pop();          // retrieves and removes some element
    ordering.append(a);
    for b in adjacent[a] do
        numPrereqs[b] -= 1;
        if numPrereqs[b] == 0 then
            minimal.add(b)
        end
    end
end
return ordering
```

> **Question 1d.** Prove that this algorithm is correct. You may assume the inner for loop does what you want it to do - you could prove the inner for loop works as well but that would make the proof extra long.

We'll show the following loop invariant

> $P(i)$: At the end of the ith iteration, Let $G_i = (V_i, E_i)$ be the graph with G with the vertices already in $\texttt{ordering}_i$ removed. I.e. $V_i = V \setminus \texttt{ordering}_i$, and $E_i = \{(u, v) : u, v \in V'\}$
>
> a.) $\texttt{numPrereqs}_i$ is a mapping such that for all $v \in V_i$, $\texttt{numPrereqs}[v]$ is the number of tasks $u \in V_i$ such that $(u, v) \in E_i$.
>
> b.) $\texttt{minimal}_i$ contains all the vertices $v \in V_i$ with $\texttt{numPrereqs}_i[v] = 0$.
>
> c.) $\texttt{ordering}_i$ is a list of i elements that respects $\prec$.
>
> d.) For any $v \in \texttt{ordering}_i, u \in V_i$, $u \not\prec v$. Nothing in V_i is a prereq for anything in $\texttt{ordering}_i$.
>
> e.) Every task v is either in $\texttt{ordering}_i$, $\texttt{minimal}_i$, or has $\texttt{numPrereqs}_i(v) > 0$.

Initialization. $\texttt{numPrereqs}_0, \texttt{minimal}_0$ are assumed to be correct, thus a., b., and e. hold. $\texttt{ordering}_0 = []$ so c. and d. also hold.

Maintenance. Let $i \in \mathbb{N}$, and assume $P(i)$, we'll show $P(i + 1)$.

By HW3 and $P(i)$.b, $\texttt{minimal}_i$ is non-empty so we get a task a.

We have $\texttt{ordering}_{i+1} = \texttt{ordering}_i + [a]$. By $P(i)$.d, a is not a prereq for anything in $\texttt{ordering}_i$ and thus loop invariant c.) holds for the start of iteration $i+1$.

Since a was in `minimal`, $P(i)$.b implies $\texttt{numPrereqs}(a) = 0$, thus, $P(i+1)$.d holds.

Next, we decrement $\texttt{numPrereqs}[b]$ for every $b \in V'$ for which a was a prerequisite, and update `minimal` if any vertex now has 0 prerequisites. This step ensures a.) and b.), and e.) hold.

Thus, $P(k+1)$.

Termination. After iteration n, the loop invariant $P(n)$.c, and $P(n)$.e implies that `ordering`is a list of n elements that respects $\prec$, which is what we wanted.

> **Question 1e.** Prove that the worst case runtime of FindOrdering2 is $O(n+m)$.

`numPrereqs`, and `adjacent` are computed by iterating over the edges once, which take $O(m)$ time, `minimal` is then computed by iterating through all the vertices which takes $O(n)$ time.

Let k be the sum of all the values in `numPrereqs`. Note that before the first for loop, $k = m$ since each edge contributes one prerequisite. Every time we reach line 9, k is decremented by 1. Since, $\texttt{numPrereqs}[v]$ is always at least 0 for each v, $k \geq 0$, so we reach lines 9 through 12 at most m times. Lines 6 and 7 are executed n times in the loop.

In total, the algorithm takes time at most $O(n+m)$.

Since m is always at most n^2, FindOrdering2 is strictly better than FindOrdering1 in the worst case!

2 Majority

Let l be any list of length n. The majority element of a list is an element that occurs $> n/2$ times. Note that not all lists have majority elements.

The following are two algorithms that return the majority element in a list if one exists. If one does not exists, it returns None implicitly.

Recall that it requires $\Theta(\log(n))$ bits of space to represent the natural number n.

checkMajority(l, m) returns `True` iff m occurs in $l > n/2$ times. The implementation keeps a counter and iterates over the list. In the worst case, the runtime is $O(n)$ and the algorithm requires $O(\log(n))$ space (to store the counter, which has value at most $n/2 + 1$.).

Algorithm 5: MajorityNaive

Input: a list l

```
n = len(l);
counts = defaultdict(int);
for x in l do
    counts[x] += 1;
    if counts[x] > n/2 then
        return x;
    end
end
```

Algorithm 6: Majority2

Input: a list l

```
m = None;
i = 0;
for x in l do
    if i = 0 then
        m = x; i = 1;
    else if m = x then
        i += 1;
    else
        i -= 1;
    end
end
if checkMajority(l, m) then
    return m
end
```

Question 2a. Let n be the length of the list.

Show that MajorityNaive requires at least $\Omega(n)$ space in the worst case.

Hint. Create an example list that makes the variable `counts` very big!

Consider the list $l = [1, 2, 3..., n]$. Since the keys to the count dictionary are the unique elements in the list and each of the elements is distinct, by the end of the for loop, count is a dictionary with n keys and thus requires $\Omega(n)$ space.

Question 2b. Trace Majority2 on input $l = [2, 1, 3, 1, 1]$. Report the values of m and i at the state of every iteration.

$$
\begin{aligned}
m_0 &= \texttt{None}, & i_0 &= 0 \\
m_1 &= 2, & i_1 &= 1 \\
m_2 &= 2, & i_2 &= 0 \\
m_3 &= 3, & i_3 &= 1 \\
m_4 &= 3, & i_4 &= 0 \\
m_5 &= 1, & i_5 &= 1
\end{aligned}
$$

The return value is $m_5 = 1$.

Question 2c. Let l be any list with a majority element. Prove that by the end of the for loop, the variable m contains the majority element. You should use a well-selected loop invariant.

Let α be the majority element. Let $b(k)$ be the number non-α elements in $l[k:]$, and $a(k)$ be the number of αs in $l[k:]$.

We'll show the following loop invariant.

$P(j)$: At the end of the jth iteration i is non-negative and

a.) Either $m = \alpha$ and $b(j) - a(j) < i$.

b.) Or $m \neq \alpha$ and $a(j) - b(j) > i$.

Initialization. $m_0 \neq \alpha$ $i_0 = 0$, and we have $a(0) - b(0) > 0$ since α is the majority element.

Maintenance. Let $j \in \mathbb{N}$ and suppose $P(j)$. We'll show that $P(j+1)$ holds.

There are several cases.

Case $m_j = \alpha, x = \alpha, i_j = 0$.

$$\begin{aligned} b(j+1) - a(j+1) &= b(j) - (a(j) - 1) \\ &< i_j + 1 \\ &= i_{j+1} \end{aligned}$$

Case $m_j = \alpha, x = \alpha, i_j > 0$. The calculation is the same as above.

Case $m_j = \alpha, x \neq \alpha, i_j > 0$.

$$\begin{aligned} b(j+1) - a(j+1) &= (b(j) - 1) - a_j \\ &< i_j - 1 \\ &= i_{j+1} \end{aligned}$$

Case $m_j = \alpha, x \neq \alpha, i_j = 0$. Then we claim $P(j+1).b$. We have $m_{j+1} \neq \alpha$, and

$$\begin{aligned} a(j+1) - b(j+1) &= a(j) - (b(j) - 1) \\ &= a(j) - b(j) + 1 \\ &> i_j + 1 && (b(j) - a(j) < i_j) \\ &= i_{j+1} \end{aligned}$$

Case $m_j \neq \alpha, x = \alpha, i_j = 0$. We claim that $P(j+1)$.a

$$\begin{aligned} b(j+1) - a(j+1) &= b(j) - (a_j - 1) \\ &< i_j + 1 \\ &< i_{j+1} \end{aligned}$$

Case $m_j \neq \alpha, x = \alpha, i_j > 0$.

$$\begin{aligned} a(j+1) - b(j+1) &= a(j) - 1 - b(j) \\ &> i_j - 1 \\ &> i_{j+1} \end{aligned}$$

Case $m_j \neq \alpha, x \neq \alpha, i_j = 0$.

$$\begin{aligned} a(j+1) - b(j+1) &= a(j) - (b(j) - 1) \\ &> i_j + 1 \\ &> i_{j+1} \end{aligned}$$

Case $m_j \neq \alpha, x \neq \alpha, i_j > 0$. The calculation is the same as the above.

Thus, in every case, the loop invariant holds.

Termination. The loop invariant holds at the end of the nth iteration. Note that $b.)$ from the loop invariant can not hold because $a(n) - b(n) = 0$ and $i_n \geq 0$. Thus, we $a.)$ holds and we have $m_n = \alpha$ as required.

Question 2d. Show that the worst-case runtime of Majority2 is $O(n)$

The for loop runs n times, and every operation in the for loop is constant time. checkMajority also runs time $O(n)$ time. Thus, the overall runtime of the algorithm is $O(n)$.

Question 2e. Show that the worst-case space usage of Majority2 is $O(\log(n))$

In the for loop, we only ever keep track of a single element and a single counter which has value at most n (hence requiring $O(\log(n))$ space to store). Furthermore, checkMajority requires $O(\log(n))$ space, thus we never use more than $O(\log(n))$ space.

3 Modular Exponentiation

In lecture we studied algorithms for multiplication. In this problem, we'll explore another common mathematical operation - exponentiation! To make things a little more interesting/manageable we'll study **modular exponentiation**.

Given $b, e, p \in \mathbb{N}$, the goal is to compute $b^e \mod p$. Here are several proposed algorithms.

For the each of the algorithms, the precondition is that $b, e, p \in \mathbb{N}$, with $0 \leq b < p$. The postcondition is that the function returns $b^e \mod p$

Algorithm 7: ExpNaive

Input: $b, e, p \in \mathbb{N}$

$r = 1$;
for $i = 0 ... e - 1$ **do**
    $r = r \cdot b$
end
return $r \mod p$

Algorithm 8: ExpModFirst

Input: $b, e, p \in \mathbb{N}$

$r = 1$;
for $i = 0 ... e - 1$ **do**
    $r = (r \cdot b) \mod p$
end
return r

Algorithm 9: ExpRec

Input: $b, e, p \in \mathbb{N}$

if $e = 0$ **then**
    return $1 \mod p$;
else if e is even **then**
    return ExpRec($b^2 \mod p, e/2, p$) $\mod p$;
else e is odd
    return $b \cdot$ ExpRec($b^2 \mod p, (e-1)/2, p$) $\mod p$;
end

The next algorithm (see next page) uses the binary expansion of e, i.e. Write $e = \sum_{j=0}^{k-1} x_j 2^j$ for some $x_0, ..., x_{k-1} \in \{0, 1\}$

Algorithm 10: ExpRepeatedSquaring

Input: $b, e, p \in \mathbb{N}$

$x_0, ..., x_{k-1}$ is the binary expansion of e. ;
$r = 1$;
$a = b$;
for $i = 0...k-1$ **do**
    if $x_i = 1$ **then**
        $r = (r \cdot a) \mod p$;
    end
    $a = (a \cdot a) \mod p$;
end
return r

You may assume the following lemma

Lemma. Let $a, b, p \in \mathbb{N}$. Then

$$(a \mod p) \cdot (b \mod p) \mod p = (a \cdot b) \mod p$$

For any number $a \in \mathbb{N}$, assume the number of digits for a is $\Theta(\log(a))$.

Assume it takes time $\Theta(n^2)$ to multiply/divide two numbers both with at most n digits. In particular, for any two numbers $x, y \in \mathbb{N}$, if $x \leq p$ and $y \leq p$, then it takes time $\Theta(\log^2(p))$ to compute $x \cdot y$, and $x \mod y$. Note $\log^2(p)$ is shorthand for $(\log(p))^2$.

Question 3a. Show that worst case asymptotic runtime of ExpNaive in terms of e and p is at least $\Omega(e^3 \log^2(p))$.

Pick $b = p - 1$. On the $e/2 + 1$th iteration, you're computing $b^{e/2} \cdot b$ which takes time $\Omega(\log(b^{e/2})^2) = \Omega(e^2 \log^2(b))$. Summing over the last $e/2$ iterations (which each take at least the time of iteration $e/2 + 1$), we get that the total runtime of the algorithm is at least $e/2 \cdot \Omega(e^2 \log^2(b)) = \Omega(e^3 \log^2(b)) = \Omega(e^3 \log^2(p))$.

Question 3b. Show that the worst case asymptotic runtime of ExpModFirst in terms of e, and p is at most $O(e \log^2(p))$.

Since $b \leq p$ and $r \leq p$, the multiplication and modding at each iteration requires time at most $O(\log^2(p))$. There are e iterations, so the overall runtime is at most $O(e \log^2(p))$.

Question 3c. Prove ExpRec is correct.

Define $P(e)$: For all $b, p \in \mathbb{N}$ with $0 \leq b < p$, ExpRec returns $b^e \mod p$ on input b, e, p.

By induction on e.

Base Case. Note that when $e = 0$, we return 1, and $b^0 = 1$, so we are good.

Inductive Step. Let $k \in \mathbb{N}$ with $k > 0$, and suppose $P(0), ..., P(k-1)$. We'll show $P(k)$. If k is even, then we return ExpRec called with the e parameter being $k/2$. Since k is even and $k > 0$, $k \geq 2$, and $0 \leq k/2 \leq k-1$. Thus, the IH applies. Therefore, we return

$$(b^2 \mod p)^{k/2} \mod p = b^{2k/2} \mod p = b^k \mod p$$

as expected. Note that we used the Lemma here.

The calculation is similar in the other case. If k is odd, then $0 \leq (k-1)/2 < k$, so again, the IH applies. In this case, we return

$$b \cdot (b^2 \mod p)^{(k-1)/2} \mod p = b \cdot b^{2(k-1)/2} \mod p = b^k \mod p$$

as required.

Question 3d. Informally explain why the running time of ExpRec is $O(\log(e)\log^2(p))$.

The recurrence for the runtime is $T(e) = T(e/2) + \log^2(p)$, since T is being called on $\lfloor e/2 \rfloor$, and we do at most 2 multiplications of numbers at most p (which takes $\log^2(p)$ time. Since the non-recursive work doesn't depend on e, the total work is just $\log^2(p)$ times the height of the recursion tree which is $\log(e)$. Hence we have $T(e) = O(\log(e)\log^2(p))$.

Question 3e. Prove ExpRepeatedSquaring is correct by defining a loop invariant and proving initialization/maintenance/termination.

Hint.

$$b^e = b^{\sum_{j=0}^{k-1} x_j 2^j}$$

It might be useful to define the empty product to be equal to 1 or the empty sum to be equal to 0.

Define the empty sum to be equal to zero. That is if you have $\sum_{i=a}^{b} ...$, and $b < a$, then the sum is equal to 0.

Let $P(i)$ be the following loop invariant.

At the end of the ith iteration,

- $r_i = b^{\sum_{j=0}^{i-1} x_j 2^j} \mod p$
- $a_i = b^{2^i} \mod p$

Initialization. $r_0 = 1 = b^0$, and $a_i = b = b^1$. Thus, the loop invariant holds before the start of the loop.

Maintenance Let $m \in \mathbb{N}$ and suppose $P(m)$. The variables get updated as follows.

If $x_m = 0$, then r is unchanged

$$\begin{aligned} r_{m+1} &= r_m \\ &= b^{\sum_{j=0}^{m-1} x_j 2^j} \mod p \\ &= b^{\sum_{j=0}^{m} x_j 2^j} \mod p \end{aligned}$$

If $x_m = 1$

$$\begin{aligned} r_{m+1} &= r_m \cdot a_m \mod p \\ &= (b^{\sum_{j=0}^{m-1} x_j 2^j} \mod p) \cdot (b^{2^m} \mod p) \mod p \\ &= b^{\sum_{j=0}^{m} x_j 2^j} \mod p \end{aligned}$$

Finally,

$$\begin{aligned} a_{m+1} &= a_m \cdot a_m \mod p \\ &= (b^{2^m} \mod p) \cdot (b^{2^m} \mod p) \mod p \\ &= b^{2^{m+1}} \mod p \end{aligned}$$

as required.

Termination. At the end of the kth iteration the loop invariant implies that the return value r_k is equal to

$$b^{\sum_{j=0}^{k-1} x_j 2^j} \mod p = b^e \mod p$$

as required.

Question 3f. Find and prove the worst case asymptotic runtime of ExpRepeatedSquaring in terms of e and p. Just prove Big-O. Assume everything before the for loop takes constant time.

$k = O(\log(e))$. There are k iterations, and each iteration does two multiplications of numbers at most p. In total, this takes time $O(\log(e)\log^2(p))$.