CS 1510 Homework 6

Brian Falkenstein, Brian Knotten, Brett Schreiber

September 9, 2018

8

Define algorithm A :

1. Read the next page access in the input sequence.
2. If fast memory is not full, add the page into fast memory.
3. If fast memory is full, evict the page that will be accessed furthest into the future by scanning through the input sequence. If a page in fast memory does not get accessed again, evict that page.
4. Repeat for all pages in the input sequence.

This algorithm is correct for all inputs. Assume to reach a contradiction that A is not correct, and there exists some input I that causes A's output to be suboptimal. Define $Opt(I)$ to be the optimal output on I that agrees with $A(I)$ for the most steps (where a step is a one iteration in the algorithm, either evicting or not evicting a page). Let t be the first time that $A(I)$ and $Opt(I)$ disagree. At t, the two algorithms disagree because $A(I)$ chose to evict a page x and $Opt(I)$ evicted page w, to swap in page z. It couldn't be the case that $A(I)$ chose to evict a page and $Opt(I)$ was able to add the page to fast memory without evicting, because this would contradict our definition of A. It also couldn't be the case that $Opt(I)$ evicted a page when $A(I)$ didn't have to, because the algorithms agreed up until this point, and there is no way this could be an optimal solution, as you could decrease the amount of evictions necessary by simply adding z into fast memory without any evictions.
We can construct $Opt'(I)$ by instead of evicting w at time t, evict x, to agree with A at t. This clearly agrees with $A(I)$ for one more step. This solution is also still optimal. If neither w or x occurs again in the input sequence, then this decision makes no difference. If its the case that w occurs again, and x does not, then $A(I)$ is more optimal than $Opt(I)$, as Opt will need to make another eviction to add w into fast memory, and A will not. If its the case that both w and x occur again, then x will occur after w, because A evicted x, and by definition A will evict the page that will need to be accessed furthest into the future. Then, when w occurs again, it will still be in $A(I)'s$ fast memory and A will not require an eviction, whereas Opt's solution did. Thus, we have constructed an output that agrees with A for one more step while still being an optimal solution, a contradiction.

13

a

Proof: Assume to reach a contradiction that our algorithm, hereafter referred to as R, is incorrect. Then there exists an input I composed of $r = \{r_1, ..., r_n\}$, $c = \{c_1, ..., c_n\}$ on which R does not produce a satisfying matrix M, which must exist by the definition of the problem (if M does not exist, then R not producing a satisfying matrix on I is correct). Let A be non-satisfactory matrix produced by $R(I)$. Let O be a matrix satisfying the problem on I that agrees with $R(I)$ for the most number of steps i.e. up to index i, j O and A have been constructed with the exact same cells.
Because each step is filling a cell with either a 1 or a 0, the disagreement can be one of two cases:

1. $A_{i,j} = 1$ and $O_{i,j} = 0$:
   Because the sum of row i must $= r_i$, it must be the case that elsewhere in i A has a cell set to 0 and O has a cell set to 1 - let this column be k. Creating a new matrix O' by modifying $O_{i,j}$'s value from 0 to 1 would cause column j to have too many 1s and would cause column k to have too few 1s since c_j must be at least as large as c_k by R's definition. j must therefore have $c_j + 1$ 1s and k must have $c_j - 1$ 1s. Because of this, it

must be the case that there is another row, m, such that $A_{m,j} = 0$, $A_{m,k} = 1$ and $O'_{m,j} = 1$, $O'_{m,k} = 0$. We can further adjust O' by setting $O'_{m,j} = A_{m,j}$ and $O'_{m,k} = A_{m,k}$. With this second adjustment, O' is now closer to A than O is and remains a satisfying matrix as columns j and k both have the correct amount of 1s.

2. $A_{i,j} = 0$ and $O_{i,j} = 1$:
   This case is very similar to the first case; creating a new matrix O' by setting $O_{i,j} = 0$ would cause column j to have too few 1s and column k to have too many 1s following the same logic as in case 1. Setting $O'_{m,j} = A_{m,j}$ and $O'_{m,k} = A_{m,k}$ remedies this, as it must be the case that $A_{m,j} = 1$ and $A_{m,k} = 0$, meaning j and k now have the correct amount of 1s after this adjustment. Therefore O' is now a satisfying matrix that agrees with A more closely than O.

Therefore O can be modified into O' where O' agrees with A for more steps than O, contradicting the statement that O is the satisfying matrix that agrees with A for the most number of steps. Therefore, by contradiction, R is correct.

b

The presented algorithm (hereafter referred to as B) is incorrect on the following input I:
$r = \{2, 1\}$
$c = \{1, 2\}$
Following the steps of the algorithm, B would first set $M_{1,2} = 1$
B would then arbitrarily choose from $M_{1,1}$, $M_{2,1}$, and $M_{2,2}$ to set to 1.
If B chooses $M_{2,1}$ to set to 1, then the next loop would end with a failure declaration as neither of the two remaining choices ($M_{1,1}$ and $M_{2,2}$) have both $\hat{r}_i$ and $\hat{c}_i$ greater than 0.
However, a satisfactory matrix for I does exist: setting $M_{1,1} = 1$, $M_{1,2} = 1$, and $M_{2,2} = 1$ will construct a satisfactory matrix.
Therefore $B(I)$ is not optimal and B is incorrect.

1

a

Let $Time$ be a function denoting the number of steps required to compute a T on an input.
Computing $T(n)$ recursively requires at least the number of steps to calculate $T(n-1)$ and $T(n-2)$, as well as a multiplication step.
$Time(n) \geq Time(n-1) + Time(n-2) + 1$
So computing $T(n)$ takes more steps than computing $T(n-2)$ twice.
$Time(n) \geq 2 * Time(n-2)$
$Time(n) \geq 4 * Time(n-4) \geq 8 * Time(n-6) \geq 16 * Time(n-8)...$
$Time(n) \geq 2^{n/2}$
So computing $T(n)$ requires exponentially many operations.

b

Assuming $T(0), T(1), T(2)...T(n-2), T(n-1)$ are already computed, it takes n multiplication steps, $T(0)*T(1), T(1)*T(2)...T(n-2)*T(n-1)$ and n addition steps to compute $T(n)$, so it takes $2n = O(n)$ steps to do the multiplication and addition.
From the bottom up, $T(0), T(1), T(2)...T(n-1)$ need to be computed, so that is n computations, each of which take $O(n)$ steps. Therefore by starting from the bottom up and caching results, computing $T(n)$ takes $O(n^2)$ steps.

c

```
int T(int n) {
    int T[n + 1];
    T[0] = 2;
    T[1] = 2;
```

```
    int sum = 0;
    for(int i = 2; i <= n; i++) {
        sum += T[i - 1] * T[i - 2];
        T[i] = sum;
    }

    return T[n];
}
```