Kliment Mamykin, UNI 2770

Algorithms for Data Science, Homework 1

Problem 1(a)

Process: restate the algorithm, establish loop invariant and prove the correctness by showing that the loop invariant holds at the initialization before the loop, after each iteration of the loop, and after the termination.

```
HORNER(A, x)
    n = A.length
    z = A[n]
    for i = n-1 down to 0 do
        z = zx + A[i]
    end
    return z
```

Insight: a polynomial of order n

$$p(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n = \sum_{k=0}^{n} a_k x^k$$

can be re-written as

$$p(x) = a_0 + (a_1 + (a_2 + \ldots (a_{n-1} + (a_n)x) \ldots)x)x$$

Loop invariant: At the start of each i loop, z contains the value of polynomial of order $n - (i + 1)$ with coefficients $A[i + 1..n]$. Mathematically

$$z_i = \sum_{k=0}^{n-(i+1)} a_{k+i+1} x^k$$

Initialization: We need to prove that for $i = n - 1$ the initialization value of $z = A[n]$ satisfies the loop invariant.

z contains the value of the polinomial of order $n - i - 1 = n - (n - 1) - 1 = 0$ with coefficients $A[i + 1..n] = A[(n - 1) + 1..n] = A[n]$. Therefor initialization of $z = A[n]$ satisfies the loop invariant.

$$z_{n-1} = \sum_{k=0}^{n-(n-1+1)} a_{k+(n-1)+1} x^k = a_n x^0 = a_n$$

Maintenance: At each iteration of the loop z is assigned a new value to be used on the next loop iteration z_{i-1} (loop is counting down) based on the value before the loop iteration z_i. At a particular interation i we have

$$z_{i-1} = a_i + z_i x = a_i + (\sum_{k=0}^{n-(i+1)} a_{k+i+1} x^k)x = \sum_{k=0}^{n-i} a_{k+i} x^k = \sum_{k=0}^{n-((i-1)+1)} a_{k+(i-1)+1} x^k$$

After the loop completes, z_{i-1} for the next iteration of the loop satisfies teh invariant.

Termination: Eventually i will be assigned the value -1, the condition of the *for* loop fails and loop terminates. At this point z contain the value of the last loop iteration at $i = 0$. We need to prove that at this point $z = \sum_{k=0}^{n} a_k x^k$ based on the invariant.

$$z = \sum_{k=0}^{n-(-1+1)} a_{k+(-1)+1} x^k = \sum_{k=0}^{n} a_k x^k$$

Problem 1(b)

Horner's rule uses n multiplications and n additions. Each iteration of the loop from $n-1$ to 0 (n iterations performs one addition and one multiplication).

For a polynomial with large n and all lower terms with 0 coefficients $A = [0, 0, \ldots, 0, a_n]$ the Horner's rule will perform many unnecessary calculations, and there may be an algorithm that is more efficient.

Problem 2

```
 1 Hadamand(v)
 2    // input: v is a vector of length n, n=2**k, where k is an int
eger
 3    // output: returns a product of Hadamand matrix of size n and
 vector v
 4    // Use divide and conquer algorithm
 5    if n == 1 return [1]
 6    v_high, v_low = partition(v) // split vector of size 2**k into
 two halfs of size 2**(k-1)
 7    h_high = Hadamand(v_high) // recursive call on vector size n/2
 8    h_low  = Hadamand(v_low)  // recursive call on vector size n/2
 9    result_high = vector_add(h_high, h_low)
10    result_low  = vector_subtract(h_high, h_low)
11    return concatenate(result_high, result_low)

partition(v)
    n = v.length
    h = [], l = []
    for i = 1..n/2
```

```
        h[i] = v[i]
        l[i+n/2] = v[i+n/2]
    return h, l

vector_add(a, b)
    r = []
    for i = 1..a.length
        r[i] = a[i] + b[i]
    return r

vector_subtract(a, b)
    r = []
    for i = 1..a.length
        r[i] = a[i] - b[i]
    return r

concatenate(a, b)
    r = [1..a.length + b.length]
    for i = 1..a.length
        r[i] = a[i]
    for i = 1..b.length
        r[a.length + i] = b[i]
    return r
```

Correctness proof

First use the fact that for any matrix A, its product with a vector v can be expressed through blocks of partitioned matrix A and partitioned vector v. This is a special case of a partitioned (block) matrices product rule.

$$Av = \left[\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right] \left[\begin{array}{c} v_1 \\ \hline v_2 \end{array}\right] = \left[\begin{array}{c} A_{11}v_1 + A_{12}v_2 \\ \hline A_{21}v_1 + A_{22}v_2 \end{array}\right]$$

Now we prove by induction the correctness of the `Hadamand(v)` algorithm.

Let v^h and v^l be the high and the low part of the vector v partitioned into 2 halves.

Proposition $P(k)$ - this is the essence of the proposed algorithm, which recursively calculates lower order Hadamand product of high and low halves of vector v and then computes the result from the sum and difference of those vectors.

$$H_k v = \begin{bmatrix} H_{k-1}v^h + H_{k-1}v^l \\ H_{k-1}v^h - H_{k-1}v^l \end{bmatrix}, \forall k > 0$$

Base Case: for $k = 1$, matrix H_1 is size 2, vector v is size 2. Matrix $H_0 = [1]$ by definition of the problem.

$$H_1 v = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} v_1 + v_2 \\ v_1 - v_2 \end{bmatrix} = \begin{bmatrix} H_0 v_1 + H_0 v_2 \\ H_0 v_1 - H_0 v_2 \end{bmatrix}$$

Inductive step: for $k > 1$ assume $P(k-1)$ is correct. Need to prove that $P(k)$ is also correct.

$$H_k v = \begin{bmatrix} H_{k-1} & H_{k-1} \\ H_{k-1} & -H_{k-1} \end{bmatrix} \begin{bmatrix} v^h \\ v^l \end{bmatrix}, \text{ expanded Hadamand matrix definition and partitioned vector}$$
$$= \begin{bmatrix} H_{k-1}v^h + H_{k-1}v^l \\ H_{k-1}v^h - H_{k-1}v^l \end{bmatrix}, \text{ using partitioned matrix/vector product rule}$$

Proposition holds true for k.

Running time T(n)

line 5: $\Theta(1)$

line 6: $\Theta(n)$ - partitioning a vector involves copy of all elements in vector into 2 subvectors, done in linear time

line 7: $T(n/2)$ - recursive call on a vector with half of the size

line 8: $T(n/2)$ - recursive call on a vector with half of the size

line 9: $cn/2 = \Theta(n)$ - vector addition is done in linear time

line 10: $cn/2 = \Theta(n)$ - vector subtract is done in linear time

line 11: $\Theta(n)$ - copy all elements into resulting vector of size n

Combined running time: $T(n) = 2T(n/2) + \Theta(n)$

Using Master theorem with $a = 2, b = 2, f(n) = \Theta(n)$, we conclude that the running time of our algorithm is $T(n) = O(n \log n)$

Problem 3(a)

Let $[e_1, e_2, \ldots, e_k]$ be the elements seen after k iteration.

Let s_k be a r.v. storing the sampled element after interation k. The index k indicates the state of the stored element after k iteration, we only ever store one element, so in this notation s is not an indexed array.

Need to prove that after iteration k, s_k is sampled uniformly. Equivalently, the probability of each element e_i to be sampled and stored in s_k is $1/k$. Formally, need to prove that

$$P[s_k = e_i] = 1/k, 1 \leq i \leq k$$

We use induction to prove this.

Proposition: $k \geq 2, 1 \leq i \leq k, P[s_k = e_i] = 1/k$. $P[s_1 = e_1] = 1$ by definition of the problem (on the first iteration we pick first element with probability 1).

Base case: $k = 2$. Prove that $P[s_2 = e_1] = P[s_2 = e_2] = 1/2$

At this state, we have seen 2 elements, stored the first element with probability 1 and replaced it with the second element with probability 1/2. Probability that the second element is stored is $P[s_2 = e_2] = 1/2$, and the probability that the first element stays stored is $1 - 1/2$ (second element is not chosen), so $P[s_2 = e_1] = 1/2$.

Inductive step: Assume $P[s_k = e_i] = 1/k$ is true. Prove that $P[s_{k+1} = e_i] = 1/(k+1)$ for $1 \leq i \leq k+1$

So far we have assumed that after seeing k elements the probability distribution of being stored on each element is $1/k$. When we see $k+1$ element, that element is kept with probability $1/(k+1)$ by definition.

$$P[s_{k+1} = e_{k+1}] = 1/(k+1)$$

The probability of the previous k elements of being stored changes with a rate of $1 - 1/(k+1)$, meaning they stay stored only if the $k+1$ element did not replace them.

$$P[s_{k+1} = e_i] = \begin{cases} 1/(k+1), & i = k+1 \\ P[s_k = e_i]\, P[s_{k+1} \neq e_{k+1}], & 1 \leq i \leq k \end{cases}$$

$$P[s_k = e_i]\, P[s_{k+1} \neq e_{k+1}] = P[s_k = e_i]\,(1 - P[s_{k+1} = e_{k+1}])$$

$$= \frac{1}{k}(1 - \frac{1}{k+1}) = \frac{1}{k}\frac{k+1-1}{k+1} = \frac{1}{k+1}$$

That proves that the probability of each element being stored at $k+1$ iteration is $1/(k+1)$, proving the inductive step.

Problem 3(b)

In general, let r be the probability that at iteration k we replace the stored element s_k with the new element e_k (excluding the case for $k = 1$, when the first element is always stored)

Observe that after seeing k elements the probability of each element being stored ($P[s_k = e_i]$) can be reasoned as a sequence of independent events: first event is that the element is stored with probability r and then events that consequent elements do not replace it.

$$P[s_k = e_i] = P[s_i = e_i \cap s_{i+i} \neq e_{i+1} \cap \cdots \cap s_k \neq e_k] = P[s_i = e_i]P[s_{i+i} \neq e_{i+1}] \cdots P[s_k \neq e_k]$$

The probability of i-th element to replace stored element is r except the first element which is picked with probability 1.

$$P[s_i = e_i] = \begin{cases} 1, & i = 1 \\ r, & i \geq 1 \end{cases}$$

$$P[s_{i+1} \neq e_{i+1}] = 1 - P[s_{i+1} = e_{i+1}] = 1 - r$$

Therefor for after seeing k elements, the probability of element e_i being sampled

$$P[s_k = e_i] = \begin{cases} (1-r)^{k-1}, & i = 1 \\ r(1-r)^{k-i}, & 1 < i \leq k \end{cases}$$

For this particular problem with $r = 1/2$ the probabilities become:

$$P[s_k = e_i] = \begin{cases} (1/2)^{k-1}, & i = 1 \\ (1/2)^{k-i+1}, & 1 < i \leq k \end{cases}$$

For example with $k = 3$ and $r = 1/2$ the following probability table applies

$P[s_3 = e_1]$	$P[s_3 = e_2]$	$P[s_3 = e_3]$
$(1/2)^2 = 1/4$	$(1/2)^2 = 1/4$	$(1/2)^1 = 1/2$

Problem 4(a)

```
Naive_Check_Matrix_Product(A, B, C)
    let n = A.rows
    for i in 1..n
        for j in 1..n
            s = 0 // accumulator for the value of i,j cell
            for k in 1..n
                s = s + A[i,k]*B[k,j]
            if s != C[i,j] return false
    return true
```

This is a deterministic algorithm with the worst-case running time (when only C_{nn} is incorrect) as $O(n^3)$.

Problem 4(b)i

Restate the problem:

Let x be an n-dimensional vector with entries randomly and independently chosen to be 0 or 1, each with probability 1/2. Prove that if M is a non-zero $n \times n$ matrix, then $Pr[Mx = 0] \leq 1/2$

$$
\begin{aligned}
Pr[Mx = 0] &= Pr\left(\begin{bmatrix} m_{11}x_1 + \cdots + m_{1n}x_n \\ \vdots \\ m_{n1}x_1 + \cdots + m_{nn}x_n \end{bmatrix} = \begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix}\right), \text{Expanded matrix vector product and 0 v} \\
&= Pr[\text{all row elements in the left vector} = 0] \\
&= \prod_{i=1}^{n} Pr[m_{i1}x_1 + \cdots + m_{in}x_n = 0], \text{by probabilities of independent events} \\
&\leq 1/2
\end{aligned}
$$

To prove a product of probabilities $\leq 1/2$ it is sufficient to prove that there exist at least one factor in the product that $\leq 1/2$. Since all probabilties are in the range $[0, 1]$, multiplying a value by a probability can only make the result same or smaller, not larger.

Since matrix M is non-zero by definition, there exists a row i where an element in column j is non-zero. Let i and j be such indices. We prove that probability of that row = 0 is $\leq 1/2$.

$$
\begin{aligned}
Pr[m_i x = 0] &= Pr[m_{i1}x_1 + \cdots + m_{ij}x_j + \cdots + m_{in}x_n = 0] \\
&= Pr[m_{ij}x_j + m_{i1}x_1 + \cdots + m_{in}x_n = 0] \text{ , re-arranged terms, first non-negative } m_{ij} \\
&\text{(let } m_{rest} = m_{i1}x_1 + \cdots + m_{in}x_n \text{)} \\
&= Pr[m_{ij}x_j + m_{rest} = 0] \\
&= Pr[(m_{ij}x_j + m_{rest} = 0 | m_{rest} = 0) \cup (m_{ij}x_j + m_{rest} = 0 | m_{rest} \neq 0)] \\
&= Pr[(m_{ij}x_j + m_{rest} = 0 | m_{rest} = 0)]Pr[m_{rest} = 0] + Pr[(m_{ij}x_j + m_{rest} = 0 | m_{rest} \neq
\end{aligned}
$$

$$
Pr[(m_{ij}x_j + m_{rest} = 0 | m_{rest} = 0)] = Pr[(x_j = 0)] = 1/2 \text{ , } m_{ij} \text{ is non-zero, only } x_j \text{ affects this}
$$

$$
Pr[(m_{ij}x_j + m_{rest} = 0 | m_{rest} \neq 0)] = Pr[(x_j = 1) \cap (m_{ij} = -m_{rest})] \leq Pr[x_j = 1] = 1/2
$$

$$
\begin{aligned}
Pr[m_i x = 0] &\leq 1/2 Pr[m_{rest} = 0] + 1/2 Pr[m_{rest} \neq 0] \\
&= 1/2 Pr[m_{rest} = 0] + 1/2(1 - Pr[m_{rest} = 0]) \\
&= 1/2
\end{aligned}
$$

Arguing back to the original statement we wanted to prove, there exist a row element in M such that $Pr[m_i x = 0] \leq 1/2$, from which follows that $Pr[Mx = 0] \leq 1/2$

Problem 4(b)ii

Show that $Pr[ABx = Cx] \leq 1/2$ if $AB \neq C$.

$$
AB \neq C \implies AB - C \neq 0
$$

Let $M = AB - C$. M is non-zero matrix. From the previous proof in (i.) follows that:

$$
\begin{aligned}
Pr[ABx = Cx] &= Pr[ABx - Cx = 0] \\
&= Pr[(AB - C)x = 0] \\
&= Pr[Mx = 0] \\
&\leq 1/2
\end{aligned}
$$

Algorithm

```
Randomized_Check_Matrix_Product(A, B, C, k)
    // input: matrices A, B, C of n x n size, k - number of trials,
 k > 0
    // return: Yes if AB=C, No otherwise. False positives may be ret
urned, no false negatives.
    // for k trials repeat
    for trial in 1..k
        // generate random x
        x = random_vector(n, 0.5) // O(n)
        // compute r = A*(B*x)-C*x
        b' = matrix_vector_product(B, x)  // O(n^2)
        a' = matrix_vector_product(A, b') // O(n^2)
        c' = matrix_vector_product(C, x)  // O(n^2)
        r = vector_sub(a', c') // O(n)
        // return No if r is not a 0 vector
```

```
        for i in 1..n // O(n)
            if r[i] != 0 return No // O(1)
    retun Yes
end

matrix_vector_product(A, x)
    r = [1..x.length]
    for i = 1..A.rows
        c = 0
        for j = 1..A.cols
            c = c + A[i,j] * x[j]
        r[i] = c
    return r

random_vector(n, p)
    r = [1..n]
    for i = 1..n
        rn = random_number(0,1)
        if rn > p then r[i] = 0
        else r[i] = 1
    return r
```

The algorithm is better then the naive algorithm because it uses matrix/vector products, which have $O(n^2)$ running time. The asymptotic runtime for the code inside the loop for each trial is dominated by the matrix/vector product and is $O(n^2)$. Because we may use multiple trials, and it is a parameter to the algorithm, the overall runtime $T(n, k) = O(kn^2)$.

The naive algorithm has $O(n^3)$ running time as shown earlier, so the randomized approach is more efficient.

The success rate of each trial: When $AB = C$ the algorithm will deterministically return Yes (since r is guaranteed to be a 0 vector). When $AB \neq C$, the algorithm will return Yes (which is an error, a false positive) with probability $\leq 1/2$ and No with $> 1/2$ probability.

To increate the success rate of the algorith - increase the number of trial. Each additional trial will half the error rate, since to be successful - all trials need to be successful. Overall error rate (false positive) for multiple trials is $\frac{1}{2^k}$, and success rate is $1 - \frac{1}{2^k}$