CSOR W4231 Analysis of Algorithms I - Spring 2023

Homework 1

Tong Wu, tw2906

Problem 1

(a)

The number of bits of $N!$ can be calculated as $log_2(N!)$

According to the Stirling's approximation formula, $n! \approx (\frac{n}{e})^n$, or

$$log_2(N!) = Nlog_2N - Nlog_2e + \mathrm{O}(log_2N)$$

Based on these results, since $N! < N^N$ for all $N \geq 1$, according to the rule of logarithm can get that

$$\begin{aligned} log(N!) &< log(N^N) \\ &< N \times log(N) \\ log(N!) &= \mathrm{O}(NlogN) \end{aligned}$$

For the lower bound, compare $N!$ and $\frac{N}{2}^{\frac{N}{2}}$:

$$\begin{aligned} N! &= 1 \times 2 \times 3 \times \ldots \times (N-1) \times N \\ &= 1 \times 2 \times 3 \times \ldots \times \frac{N}{2} \times (\frac{N}{2}+1) \times \ldots \times (N-1) \times N \\ &= (\frac{N}{2}-1)! \times \frac{N}{2} \times \underbrace{(\frac{N}{2}+1) \times \ldots \times (\frac{N}{2}+\frac{N}{2})}_{\frac{N}{2}\ terms} \end{aligned}$$

Thus,

$$\begin{aligned} N! &\geq \frac{N}{2}^{\frac{N}{2}} \\ log(N!) &\geq \frac{N}{2} log(\frac{N}{2}) = \frac{1}{2}(Nlog(N)-1) \end{aligned}$$

Hence,

$$log(N!) = \Omega(Nlog(N))$$

According to the transitivity of the asymptotic,

$$log(N!) = \Theta(Nlog(N))$$

So $N!$ has $\Theta(Nlog_2(N))$ bits.

According to the question, N is an n-bit integer, so it can also written as $N!$ has $\Theta(n \times 2^n)$ bits.

(b)

```
problem1(N)
    if (N==0 || N == 1)
        return 1
    else if (N<0)
        return 0
    else
        return N*problem1(N-1)
```

The algorithm calculate the fraction of N recursively by continuously call the problem1 function and push the parameter in order to achieve the goal of factorial calculation.

Let $N \geq 0$, after N-th loop, the output is equal to the factorial calculation of N.

The algorithm will run N times of the loop to complete the calculation. The running time should be $\mathrm{O}(Nn^2)$.

Correctness:

Base case:

When $n = 0$, the value of $N!$ is defined as 1, which is the correct value that returned by the algorithm.

Induction hypothesis:

Assume that for any $N < k$, the algorithm can correctly calculate the factorial number of $N!$.

Induction:

For $N = k$, according to the definition of the factorial, the factorial number of N multiplies all the integers from 1 to k to get $N!$, $N! = N \times (N-1)!$. According to the hypothesis, the algorithm can give a correct result for $(N-1)!$, then multiply with N to get the result of $N!$. The induction hypothesis then be proofed.

Problem 2

(a)

i.

The binary tree is a visual model of the sorting algorithm. Each leaf in the tree representing one possible permutation of integers from the specific ordering of integers, and all nodes and leaves gives every possible permutation that the algorithm recursive given, so in this case every possible permutation of the n numbers must appear as a leaf in the tree. If one of the permutation not appear on the tree, which means the algorithm brings error or incomplete.

ii.

Thus the tree has at least $n!$ leaves.

iii.

At each node - or on the other words, the bifurcation point which will break to result or the next comparison - will do a comparison. In this case, the maximum number of comparisons performed by the algorithm is already visualized by the binary tree, performed as the longest path from the first node to the final result in the tree. So in general, the maximum number of comparisons performed by the algorithm correspond to the depth of the tree

iv.

Set the letter N states the number of leaves on a random binary tree, which has depth stated with letter D.

Claim 1:

The possible case of the maximum number of leaves on a binary tree must smaller or equal to the power of 2 according to the depth, which can be also stated as: $N \leq 2^D$.

Proof of claim 1:

For each binary tree, the root level (with depth 0) must has 1 node, and this node split to 2 nodes which may varied to result or the next comparison. Assume that the every node from depth 0 to depth $D-1$ will split into 2 nodes in the next depth level, which means that from depth 0 to depth $D-1$ will all nodes but not results. In this case, the number of nodes in each depth level can be calculated by $N_{node} = 2^{D_{current}}$, for example the root level has $2^0 = 1$ node, and the depth 1 has $2^1 = 2$ nodes since each father node will split to two nodes in the next depth level. At the last depth level, D, the number of leaves will be 2^D since all nodes in the last depth level must be the leaves and all nodes in other depth level are comparison. However this situation will not occur for the binary tree of sorting algorithm since the results may be given

from the previous depth level, $D-1$, which means that the final number of leaves on the last depth level must smaller than the previous calculated value 2^D since the number of nodes on the previous depth level is lower. Hence, the maximum number of leaves has the upper bound in terms of the depth of the tree, 2^D, and $N = \mathrm{O}(2^D)$

v.

From the previous question (ii) and (iv) can know that, the tree has least $n!$ and most 2^D leaves. In the previous question can known that:

$$n! \leq n^n$$
$$log(n!) = \Omega(nlog(n))$$

In this problem,

$$n^n \leq n! \leq 2^D$$
$$nlog(n) \leq log(n!) \leq Dlog(2)$$

Since,

$$log(n!) = \Omega(nlog(n)),\ n^n \leq n! \leq 2^D$$

Can get that,

$$D = \Omega(nlog(n))$$

(b)

In the question, $D = \max_i x_i - \min_i x_i$ means that a line or several lines of code should execute the number of the difference of the maximum and minimum of the list. One of the easy way to achieve this is to create an array which has $\max_i x_i - \min_i x_i$ terms. Use these information, after browsing the textbook and web, the counting sort can achieve the goal.

The counting sort algorithm convert the input integers to index and store in a wider array space, which is also the core of the counting sort. The counting sort has four steps:

1. Find maximum and minimum element in unsorted array
2. Count each elements in the unsorted array and stored in a new array
3. Summing all counting
4. Re-fill the array to finish sorting

```
CountingSort(n) {
    // Create a counting array, with (max-min) zeros
    for i in range(n)
        c[i] = 0
    // Count each element
    for i=0 to n do
        c[n[i]] += 1
    end for
    // Summing
```

```
    for i=0 to max(n)-min(n)+1 do
        c[i] += c[i-1]
    end for
    // Refill and output the result
    o = array(length(n))
    for i=0 to n do
        o[c[n[i]]] = n[i]
        c[n[i]] -= 1
    end for
    return o
}
```

Time complexity:

In step 1, create array `c = array(range(max(n)-min(n)+1))` need running time `max(n)-min(n)`, which is $\mathrm{O}(D)$.

In step 2, counting each element with total n elements costs $\mathrm{O}(n)$ running time.

In step 3, summing elements costs `max(n)-min(n)+1` running time, which is $\mathrm{O}(D)$.

In step 4, refill and output the result execute total n elements, so the running time is $\mathrm{O}(n)$

In total, the running time cost in counting sort algorithm is:

$$T(n) = \mathrm{O}(D) + \mathrm{O}(n) + \mathrm{O}(D) + \mathrm{O}(n) = \mathrm{O}(n + D)$$

Correctness:

In order to proof the correctness of the counting sort algorithm, should proof that the output array is sorted, which means that later neighbor elements is smaller than the previous, $o[n+1] < o[n]$. However, in the algorithm, if two elements $n[n+1] < n[n]$, the smaller element will put after the bigger. So the correctness is proofed.

(c)

If the D is small, the lower bound of the algorithm proposed in part (b) is $\Omega(n)$ rather than $\Omega(nlog(n))$. So the lower bound of the algorithm proposed in part (b) is not same with the lower bound in part (a).

Problem 3

For the matrix-vector product, the general formula to solve it is:

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ \dots \\ b_m \end{bmatrix} = \begin{bmatrix} a_{11}b_1 + a_{12}b_2 + \dots a_{1n}b_m \\ a_{21}b_1 + a_{22}b_2 + \dots a_{2n}b_m \\ \dots \\ a_{m1}b_1 + a_{m2}b_2 + \dots a_{1n}b_m \end{bmatrix}$$

In this question, each elements are similar, so the formula can be simplified:

$$\begin{aligned} H_k \times v &= \begin{bmatrix} H_{k-1} & H_{k-1} \\ H_{k-1} & -H_{k-1} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} \\ &= \begin{bmatrix} H_{k-1}v_1 + H_{k-1}v_2 \\ H_{k-1}v_1 - H_{k-1}v_2 \end{bmatrix} \\ &= \begin{bmatrix} H_{k-1}(v_1 + v_2) \\ H_{k-1}(v_1 - v_2) \end{bmatrix} \end{aligned}$$

From the matrix vector multiplication, can found that each element H_{k-1} multiplied a set which is formed by the sum or the difference of all elements in column vector v. In this case, one of the solution to boost the efficiency of the matrix-vector multiplication is that, to split the original problem into several sub-problems, then conquer the subproblem and finally combine the sub-problems back to output the result, which is the divide & conquer.

By using the principle of the divide & conquer, first split the vector v into two equal-size vector:

$$v = \underbrace{\begin{bmatrix} v_1 & v_2 & \dots & v_n \end{bmatrix}}_{\text{n terms}}^T$$

$$v_a = \underbrace{\begin{bmatrix} v_1 & v_2 & \dots & v_{\frac{n}{2}} \end{bmatrix}}_{\frac{n}{2}\text{ terms}}^T$$

$$v_b = \underbrace{\begin{bmatrix} v_{\frac{n}{2}+1} & v_{\frac{n}{2}+2} & \dots & v_n \end{bmatrix}}_{\frac{n}{2}\text{ terms}}^T$$

So the matrix-vector multiplication for the Hadamard matrix has four steps:

1. Split column vector v into v_1 and v_2, both has $\frac{n}{2}$ terms.
2. Compute $v_1 + v_2$ and $v_1 - v_2$
3. Iteratively compute $H_{k-1}(v_1 + v_2)$ and $H_{k-1}(v_1 - v_2)$
4. Combine the result from step 3 into one vector

```
MVM(H,v)
    n = length(v)
```

```
3     // split the vector
4     v_a = split(v, n/2)[0]
5     v_b = split(v, n/2)[1]
6     // Recursively compute slices
7     for i in range(length(v_a))
8         slice1[i] = VectorMul(H[i], (v_a+v_b))
9         slice2[i] = VectorMul(H[i+n/2], (v_a-v_b))
10    end for
11    // Combine the result
12    result = [slice1, slice2]
13    return result
14 end
```

Correctness of MVM algorithm

Claim 1: The algorithm MVM can calculate the correct function for the multiplication of a particular vector and a column vector.

Proof of claim 1:

Base condition: where $k = 1$, the dimension of Hardamard matrix is 1×1, where the "matrix-vector multiplication" for H_k and v is $H_k v$.

Induction hypothesis: Assume that algorithm "MPM" can calculate the correct result of the matrix-vector multiplication.

Induction: For $k = 2$, the matrix-vector multiplication can be calculated as followed:

$$\begin{aligned} H_k \times v &= \begin{bmatrix} H_{k-1} & H_{k-1} \\ H_{k-1} & -H_{k-1} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} \\ &= \begin{bmatrix} H_{k-1}v_1 + H_{k-1}v_2 \\ H_{k-1}v_1 - H_{k-1}v_2 \end{bmatrix} \\ &= \begin{bmatrix} H_{k-1}(v_1 + v_2) \\ H_{k-1}(v_1 - v_2) \end{bmatrix} \end{aligned}$$

In the algorithm function, it first split column vector v into two halved vector v_a and v_b, then recursively compute $H_{k-1}(v_a + v_b)$ and $H_{k-1}(v_a - v_b)$ in line 7,8,9. Finally combine the result and output. The output vector is combined from the first row result $H_{k-1}(v_a + v_b)$ and $H_{k-1}(v_a + v_b)$ for the second. So the calculation of the algorithm function is correct.

Running time:

Line 4,5 both cost $\mathrm{O}(n)$ of the running time, and in line 8,9, the running time is $2T(\frac{n}{2})$, where $\frac{n}{2}$ is the number of each recursive calculation times, and there are 2 recursive calculations. In total, the running time is:

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

According to the master theorem in order to solving the recurrences: If $T(n) = aT(\lceil n/b \rceil) + \mathrm{O}(n^k)$ for some constants $a > 0, b > 1, k \geq 0$, then

$$T(n) = \begin{cases} \mathrm{O}(n^{log_b a}), & if\ a > b^k \\ \mathrm{O}(n^k log n), & if\ a = b^k \\ \mathrm{O}(n^k), & if\ a < b^k \end{cases}$$

The running time can be written as:

$$a = 2,\ b = 2,\ k = 1 \rightarrow a = b^k$$
$$T(n) = \mathrm{O}(nlog(n))$$

Where the running time $T(n) = \mathrm{O}(nlog(n))$ is smaller than the running time of straightforward algorithm $\mathrm{O}(n^2)$

Problem 4

(a)

Claim: $F_n \geq 2^{n/2},\ n \geq 6$

Proof of claim:

Base case: When $n = 6$:

$$
\begin{aligned}
F_n &= F_{n-1} + F_{n-2},\ \text{if}\ n \geq 2 \\
F_6 &= F_5 + F_4 \\
&= (F_4 + F_3) + (F_3 + F_2) \\
&= (F_3 + F_2) + 2(F_2 + F_1) + F_2 \\
&= 3(F_1 + F_0 + F_1) + 2(F_1 + F_0) \\
&= 3 \times 2 + 2 \times 1 \\
&= 8
\end{aligned}
$$

Hence, the base case is correct.

Induction hypothesis: For all positive integer $n \geq 6$, the statement $F_n \geq 2^{n/2}$ must correct

Induction:

According to the rule of Fibonacci numbers: $F_n = F_{n-1} + F_{n-2},\ \text{if}\ n \geq 2$

$$
\begin{aligned}
F_n &= F_{n-1} + F_{n-2} \\
&\geq 2^{(n-1)/2} + 2^{(n-2)/2} \quad (\text{hypothesis based}) \\
&= 2^{n/2} \times \frac{1}{\sqrt{2}} + 2^{n/2} \times \frac{1}{2} \\
&= 2^{n/2} \times \underbrace{(\sqrt{2} + \frac{1}{2})}_{\approx 1.9142 > 1}
\end{aligned}
$$

Since based on the induction hypothesis, $F_n \geq 2^{n/2}$. The hypothesis has been proofed as correct by induction.

(b)

i)

```
1  RecursiveFib(n)
2      // Determine the result by three different conditions
3      if (n < 2)
4          return n
5      else
6          F_n = 0
7          // Recursive compute Fibonacci number
8          return RecursiveFib(n-1) + RecursiveFib(n-2)
9      end if
10 end
```

The algorithm is designed following the definition that three conditions to calculate the Fibonacci number.

Running time:

From line 3 to line 6, the running time can be stated as $\mathrm{O}(1)$. For line 8, the recursive computing as $n-1$ times and $n-2$ times, so the running time should be $T(n-1)+T(n-2)$. The total running time is:

$$T(n) = \mathrm{O}(1) + T(n-1) + T(n-2)$$

Lower bound:

$$\begin{aligned} T(n) &= \mathrm{O}(1) + T(n-1) + T(n-2) \\ &\geq 2T(n-2) + \mathrm{O}(1) \\ &\geq 4T(n-4) + \mathrm{O}(1) \\ &\geq \dots \\ &\geq 2^{(n-1)/2} \\ &\geq 2^{n/2} \\ T(n) &= \Omega(2^{n/2}) \end{aligned}$$

The same result can be also found from the result in part (a).

Correctness:

Base case:

When $n \leq 1$, the Fibonacci number $F_n = n$.

Induction hypothesis:

The algorithm can calculate the Fibonacci number F_n correctly for any n.

Induction:

According to the principle of Fibonacci sequence, the nth Fibonacci number F_n is calculated by $F_{n-1} + F_{n-2}$. The algorithm use recursive to call $n-1$ and $n-2$, then calculate F_{n-1} and F_{n-2} separately. Finally add F_{n-1} and F_{n-2} in order to output the Fibonacci number F_n. So in this case, the induction hypothesis can be proofed.

ii)

```
1  NonRecursiveFib(n)
2      // Determine the result by three different conditions
3      if (n < 2)
4          return n
5      else
6          F_n = 0
7          F_n-1 = 0
8          F_n-2 = 1
9          for i=2 to range(n)
10             F_n = F_n-1 + F_n-2
11             F_n-2 = F_n-1
12             F_n-1 = F_n
13         end for
14         return F_n
15     end if
16 end
```

The algorithm use for loop to compute the Fibonacci number, by storing two previous state of number as temp and reuse in the new loop round.

Running time upper bound:

In the code, line 9-13 run $n-1$ times since it start from 2. So the running time should be:

$$T(n) \leq T(n-1) + \mathrm{O}(1) = \mathrm{O}(n)$$

Correctness:

Base case:

When $n \leq 1$, the Fibonacci number $F_n = n$.

Induction hypothesis:

The algorithm can calculate the Fibonacci number F_n correctly for any n, if the previous two Fibonacci numbers F_{n-1} and F_{n-2} can be calculated.

Induction:

According to the hypothesis and principle of Fibonacci sequence, the algorithm can calculate the $(n-1)$th and $(n-2)$th Fibonacci number. Hence, when the algorithm summing two previous Fibonacci number to form the nth Fibonacci number, the algorithm can then, calculated the correct number. Hence the induction has been proofed.

iii)

From problem 3 can found that, the time complexity $\mathrm{O}(nlog(n))$ can achieved by using divide & conquer. Also, the time complexity $\mathrm{O}(log(n))$ can be achieved if $k = 0$ in master theorem, which needs that $\mathrm{O}(1)$ in running time.

By using the principle of the divide & conquer, first split the vector $[F_1, F_0]$ into two equal-size vector:

$$v = \underbrace{[F_1 \quad F_0]}_{2 \text{ terms}}{}^T$$
$$v_a = \underbrace{[F_1]}_{1 \text{ term}}{}^T$$
$$v_b = \underbrace{[F_0]}_{1term}{}^T$$

So the algorithm for Fibonacci number has four steps:

1. Split column vector v into v_1 and v_2
2. Compute $v_1 + v_2$
3. Iteratively compute F_2 and F_1
4. Combine the result from step 3 into one vector

```
1  DCFib(n,F)
2      F_a = F[0]
3      F_b = F[1]
4      // Iteratively compute top and bottom
5      for i in range(n)
6          slice1 = F_a + F_b
7          slice2 = F_a
8      end for
9      result = [slice1, slice2]
10     return result
11 end
```

Running time:

From line 2-3 and line 9, running time is $\mathrm{O}(1)$, from line 5-7, the running time is $T(\frac{n}{2})$. The total running time is:

$$T(n) = T(\frac{n}{2}) + \mathrm{O}(1)$$
$$a = 1,\ b = 2,\ k = 0$$
$$T(n) = \mathrm{O}(log(n))$$

Correctness:

Base case: When $n = 2$, the algorithm runs as it states

$$\begin{aligned}\begin{bmatrix}F_2\\F_1\end{bmatrix} &= \begin{bmatrix}1 & 1\\1 & 0\end{bmatrix}\begin{bmatrix}F_1\\F_0\end{bmatrix}\\ &= \begin{bmatrix}1\times F_1 + 1\times F_0\\1\times F_1 + 0\times F_0\end{bmatrix}\\ &= \begin{bmatrix}1\\1\end{bmatrix}\end{aligned}$$

Hypothesis: For all Fibonacci number which the order is inclusive between 1 and n, the number can be correctly computed.

Induction: Assume the Fibonacci number is F_m where $1 \leq m \leq n$. The equation can be written:

$$\begin{aligned}\begin{bmatrix}F_n\\F_{n-1}\end{bmatrix} &= \begin{bmatrix}1 & 1\\1 & 0\end{bmatrix}\begin{bmatrix}F_{n-1}\\F_{n-2}\end{bmatrix}\\ &= \begin{bmatrix}1\times F_{n-1} + 1\times F_{n-2}\\1\times F_{n-1} + 0\times F_{n-2}\end{bmatrix}\\ &= \begin{bmatrix}F_{n-1} + F_{n-2}\\F_{n-1}\end{bmatrix}\end{aligned}$$

In the equation, the n-th Fibonacci can be correctly computed. So the hypothesis is correct.

(c)

According to the previous part can show that $F_n \geq 2^{n/2}$, which means that any Fibonacci number in this question has at least $\frac{n}{2}$ bits. The term m and n can be seen as equal when calculate running time when the condition is bigger enough.

Part i:

The running time of add or subtract action is changed to $\mathrm{O}(m)$, so the term in original running time $\mathrm{O}(1)$ should change to $\mathrm{O}(m) = \mathrm{O}(n)$.

$$T(n) = \mathrm{O}(n) + T(n-1) + T(n-2)$$
$$T(n) = \Omega(2^{n/2})$$

Since n is smaller than 2^n, it can be found that although the efficiency of the first algorithm is still the same.

Part ii:

The original running time is $\mathrm{O}(n)$. Although the running time of element arithmetic action changes, the running time will not change, which is still efficient.

$$T(n) = \mathrm{O}(n)$$

Part iii:

In this algorithm, calculate Fibonacci number by calculating the matrix uses multiplication and add. The original running time is $T(n) = T(\frac{n}{2}) + \mathrm{O}(1) = \mathrm{O}(log(n))$. The new running time should be:

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

$$a = 1,\ b = 2,\ k = 2\ \rightarrow a < b^k$$

$$T(n) = \mathrm{O}(n^2)$$

Since the multiplication costs running time $\mathrm{O}(m^2)$, which is equivalent to $\mathrm{O}(n^2)$. For this algorithm, the running time becomes less efficient since the $log(n)$ is smaller than n^2.