CSOR W4231 Analysis of Algorithm

Jialin Zhao

February 5, 2018

1 Problem I

f	g	O	o	Ω	ω	Θ
$\log^2 n$	$6 \log n$	No	No	Yes	Yes	No
$\sqrt{\log n}$	$(\log \log n)^3$	No	No	Yes	Yes	No
$4n \log n$	$n \log 4n$	Yes	No	Yes	No	Yes
$n^{3/5}$	$\sqrt{n} \log n$	No	No	Yes	Yes	No
$5\sqrt{n} + \log n$	$2\sqrt{n}$	Yes	No	Yes	No	Yes
$n^5 4^n$	5^n	Yes	Yes	No	No	No
$\sqrt{n} 2^n$	$2^{n/2+\log n}$	No	No	Yes	Yes	No
$n \log 2n$	$\frac{n^2}{\log n}$	Yes	Yes	No	No	No
$n!$	2^n	No	No	Yes	Yes	No
$\log n!$	$\log n^n$	Yes	No	Yes	No	Yes

2 Problem II

2.1 (a)

Correctness:

Base case: i = 0
After the loop, $z_0 = a_0$
The solution of the polynomial is true.
Induction hypothesis: Assume that the statement is true for case $i \geq 0$.
Inductive step: Show it true for case i+1
This case equals to case i, when we set the initial $z = a_{i+1}x + a_i$ instead of $z = a_i$
By the induction hypothesis, case i is correct which means $z = a_0 + a_1 x + ... + a_i x^i$
Then in the case i+1, $z = a_0 + a_1 x + ... + (a_{i+1}x + a_i)x^i = a_0 + a_1 + ... + a_i x^i + a_{i+1}x^{i+1}$
Which means the solution of the polynomial is also true in case i+1.

Conclusion:
It follows that the statement is true for all n since we can apply the inductive step for n = 2; 3; : : :

2.2 (b)

This function uses n multiplications and n additions, where n equals the number of iterations of the for loop.
A special polynomial:
A single non-zero coefficient $a_i x^i$
Better alternative method:
Using the repeated squaring method of evaluating x^i:
Repeated squaring method:

Pseudocode:

Algorithm 1 Function exp-by-squaring(x, i)

```
if i < 0 then
  return exp-by-squaring(1/x, -i)
else if i = 0 then
  return 1
else if i = 1 then
  return x
else if i is even then
  return exp-by-squaring(x * x, i / 2)
else if i is odd then
  return x * exp-by-squaring(x * x, (i - 1) / 2)
end if
```

Algorithm 2 Function A-exp-by-squaring(a, x, i)

```
return a * exp-by-squaring(x,i)
```

Time Complexity:

The problem is converted into a half-scale subproblem and using 1 or 2 multiplation in O(1) time:

$$T(n) = T(n/2) + O(1)$$

According to master thereom, the method Will use O($\log i$) multiplications.

Space Complexity:

The depth of recursion tree is $\log n$, the space used by each recuirsion step is $O(1)$.
So, the space complexity is $O(\log n)$

Correctness:

Base case: n = 0,1

$$x_0 = 1, x_1 = x$$

The statement is true.
Induction hypothesis: Assume that the statement is true for every case n from 0 to i,$i \geq 1$.
Inductive step: Show it true for case i+1
Since we can find a case i/2 or (i+1)/2 in which the statement is true as a base case.

If i+1 is an odd number, $x^{i+1} = x * (x^2)^{(i)/2} = x^{i+1}$
If i+1 is an even number, $x^{i+1} = (x^2)^{(i+1)/2} = x^{i+1}$
Which means the statement is also true in case i+1.
Conclusion:
It follows that the statement is true for all n since we can apply the inductive step for every odd or even number n = 2; 3; : : :

3 Problem III

Let v also be represented by two halfs:

$$v = \begin{bmatrix} v_h \\ v_l \end{bmatrix}$$

Where v_h and v_l are of n/2 scale.
Then:

$$H_k v = \begin{bmatrix} H_{k-1} & H_{k-1} \\ H_{k-1} & -H_{k-1} \end{bmatrix} * \begin{bmatrix} v_h \\ v_l \end{bmatrix} = \begin{bmatrix} H_{k-1}v_h + H_{k-1}v_l \\ H_{k-1}v_h - H_{k-1}v_l \end{bmatrix}$$

Correctness:

Base case: n = 2^k, k = 0
Then $H_0 v = v$
The statement is true.
Induction hypothesis: Assume that the statement is true for case $k \geq 0$.
Inductive step: Show it true for case k+1
Since the $H_k v$ is correct,
Then $H_{k+1}v = \begin{bmatrix} H_k & H_k \\ H_k & -H_k \end{bmatrix} * \begin{bmatrix} v_h \\ v_l \end{bmatrix} = \begin{bmatrix} H_k v_h + H_k v_l \\ H_k v_h - H_k v_l \end{bmatrix}$ is also correct.
Which means the statement is also true in case k+1.
Conclusion:
It follows that the statement is true for all n since we can apply the inductive step for every odd or even number k = 1; 2; 3; : : :

Time Complexity:

So each step we convert the problem of $H_k v(T(n)$ time) into $H_{k-1}v_h(T(n/2)$ time) and $H_{k-1}v_l(T(n/2)$ time) and some addition or substraction($O(n)$ time), since elementwise operations take $O(1)$ time.
Therefore:

$$T(n) = 2T(n/2) + O(n)$$

According to Master Teorem:

$$T(n) = \Theta(n \log n) < \Theta(n^2)$$

where $n = 2^k$.
Faster than the straight forward algorithm.

Space Complexity:

The depth of the recursion tree is $k = (\log_2 n)$, each recursion uses $O(n)$ space, so the space complexity should be computed by adding the space used in the deepest recursion branch: $\sum_1^{\log_2 n} O(n/2^i) = O(n)$

Pseudocode:

Algorithm 3 Function Hadamard(k, v)

if k = 0 **then**
  return v
end if
$v_h \Longleftarrow$ Higher Half Part of v
$v_l \Longleftarrow$ Lower Half Part of v
A $\Longleftarrow$ Hadamard$(k-1, v_h)$
B $\Longleftarrow$ Hadamard$(k-1, v_l)$
Result $\Longleftarrow \begin{bmatrix} A+B \\ A-B \end{bmatrix}$
return Result

4 Problem IV

4.1 (a)

If we use divide and conquer method to get a $O(n \log n)$-time solution.
We can divide the problem into two (n/2) scale subproblem and use $O(n)$ time to combine subproblems.
Here is my sample solution:

Pseudocode:

Algorithm 4 Function Majority(array A)

if $Length(A) = 0$ **then**
  return None
end if
if $Length(A) = 1$ **then**
  return the only element in A
end if
$A1 \Longleftarrow$ Array of first n/2 elements of A
$A2 \Longleftarrow$ Array of the left elements of A
$a1 \Longleftarrow$ Majority$(A1)$
$a2 \Longleftarrow$ Majority$(A2)$
if ($a1$ is None) and ($a2$ is None) **then**
  return None
else if ($a1$ is not None) and ($a2$ is not None) and ($a1 = a2$) **then**
  return a1
else if $a1$ is not None **then**
  num $\Longleftarrow$ the amount of $a1$ in A
  if num> $\lfloor 1/2 Length(A) \rfloor$ **then**
    return $a1$
  end if
else if $a2$ is not None **then**
  num $\Longleftarrow$ the amount of $a2$ in A
  if num> $\lfloor 1/2 Length(A) \rfloor$ **then**
    return $a2$
  end if
else
  return None
end if

Correctness:

As stated, the method is based on divide-conquer principle by induction on the size of the two lists.
If A has a majority element v, v must also be a majority element of A1 or A2 or both.
If both parts have the same majority element, it is automatically the majority element for A.
If one or two of the parts has a different majority element, count the number of that element in both parts (in O(n) time) to check if it is a majority element.
If not the cases above, return None.
Base case: subarray of length k = 1
Will return the only element which is definitely the majority.
The statement is true.
Induction hypothesis: Assume that the statement is true for case of length $k \geq length \geq 1$.
Inductive step: Show it true for case of combined array of length k+1
According to the base case, the two subarray of this case is in length (k+1)/2 and will return their majority number correctly.
After choosing and checking the two potential majority numbers, the array in this case can also return its majority number correctly.
Conclusion:
It follows that the statement is true for all array since we can apply the inductive step for any length k = 1; 2; 3; : : :

Time Complexity:

A recurrence relation is T(n) = 2T(n/2) + O(n)
According to Master Teorem:

$$T(n) = O(n \log n)$$

Space Complexity:

The depth of the recursion tree is $\log n$, each recursion step uses $O(n/2^i)$ space.
So the space complexity should be calculated by $\sum_1^{\log n} O(n/2^i) = O(n)$

4.2 (b)

If we can use a map storing pairs of (element,count), the problem will be simplified a lot because we can easily look through all elements in the array A, add new element into map and set the count as 0, or we let count+=1. Then just find the only element with a count more than half of the length of A.
However, let's suppose we need to use limited techniques(only array):
My solution is based on an intersting fact of majority number in an array: By discarding two different elements of A, the majority number will not change in the array left.
After discarding all different element pairs the remaining element should be the majority, unless A doesn't have a majority element at all.
So we need to check if the left element is the majority, if yes return it, if no return None.

Correctness:

Base case: Array of length k = 0,1
Will return None or the only element which is definitely the majority.
The result is true.
Induction hypothesis: Assume that the statement is true for case of length $k \geq length \geq 0$.
Inductive step: Show it true for case of array of length = k+2
After discarding two different elements in the array, the majority element number is reduced by 0 or 1 but the total length is reduced by 2, so the majority number will still exceed half number of the arrray.
So this case will return its majority if it has.
The result is true.
Conclusion:
It follows that the statement is true for all array since we can apply the inductive step for any length k = 1;

2; 3; : : :

Time Complexity:

A recurrence relation is T(n) = T(n-1) + O(1)
Therefore:

$$T(n) = O(n)$$

Space Complexity:

This is an recurrence method which needs only O(1) extra space.

Pseudocode:

Algorithm 5 Function Majority(array A)

```
N ⟸ Length of A
if N = 0 then
  return None
end if
Element ⟸ A[1]
Count ⟸ 1
i ⟸ 2
for i ≤ N do
  i = i + 1
  if Element = A[i] then
    Count ⟸ Count + 1
  else if Count ≥ 1 then
    Count ⟸ Count - 1
  else if Count = 0 then
    Element ⟸ A[i]
    Count ⟸ 1
  end if
end for
if Look through A and check: Element occurs more than half of the length of A then
  return Element
else
  return None
end if
```

5 Problem V

5.1 (a)

n	0	1	2	3	4	5	6	7
F_n	0	1	1	2	3	5	8	13

Correctness:

Base case: n = 6,7

$F_6 = 8 \geq 2_{n/2} = 8$
$F_7 = 13 \geq 2_{n/2} = 8\sqrt{2}$
The statement is true.
Induction hypothesis: Assume that the statement is true for case $n \geq 6$ and case $n + 1$.
Inductive step: Show it true for case n+2
$F_{n+2} = F_{n+1} + F_n \geq 2^{(n+1)/2} + 2^{(n)/2} \geq 2 * 2^{(n)/2} = 2^{(n+2)/2}$
Which means the statement is also true in case n+2.
Conclusion:
It follows that the statement is true for all $n \geq 6$ since we can apply the inductive step for n = 8; 9; : : :

5.2 (b)

5.2.1

Pseudocode:

Algorithm 6 Function Fibonacci(n)

```
if n ≤ 1 then
  return n
end if
return Fibonacci(n-1)+Fibonacci(n-2)
```

Correctness:

This is the definition, no need to prove it.
Base case: n = 0,1 then $F_0 = 0, F_1 = 1$ is true.
Induction step: if case $n \geq 0$ and $n + 1$ is true then case n+2 : F_{n+2}=F_{n+1}+F_n=true.
So the case n+2 is also true.
Conclusion: The Fabonacci number is calculated correctly for every case n =1;2;3:::

Time Complexity:

The problem of computing F_n is converted to computing F_{n-1} and F_{n-2} and a constant-time addition.
Then we have:

$$T(n) = T(n-1) + T(n-2) + O(1)$$

So in this prograss, we only need to calculate the sum of adding two numbers:
From T(n) to T(n-1): 1 addition;
From T(n-1) to T(n-2): 2 addition;
From T(n-2) to T(n-3): 3 addition;
From T(n-3) to T(n-4): 5 addition;
.....
From T(n-x) to T(n-x-1): F_{x+2} addition;
.....
From T(2) to T(1) and T(0): F_n addition;
Then, $T(n) = \Theta(\sum_2^n F_i)$
According to the expression in question(a):

$$F_i \geq 2^{(n)/2}$$

Therefore:

$$T(n) = \Theta(\sum_2^n F_i) = \Omega(\sum_2^n 2^{(n)/2}) = \Omega(2^{(n)/2})$$

To give a tight bound:
We know from the book that F_n has a closed form expression:

$$\frac{\phi^n - \varphi^n}{\sqrt{5}}$$

where $\phi = \frac{1+\sqrt{5}}{2}$ and $\varphi = \frac{1-\sqrt{5}}{2}$
Therefore:

$$T(n) = \Theta(\sum_{2}^{n} F_i) = \Theta(\sum_{2}^{n} \phi^i) = \Theta(\phi^n)$$

Space Complexity:

The depth of the recursion tree is n, each recursion step uses $O(1)$ extra space.
The space complexity is $O(n)$.

5.2.2

Pseudocode:

Algorithm 7 Function Fibonacci(n)

```
Memory ⟸ 0
Result ⟸ 1
if n ≤ 1 then
  return n
end if
for i = 2 up to n do
  Temp ⟸ Result
  Result ⟸ Result + Memory
  Memory ⟸ Temp
end for
return Result
```

Correctness:

This is the definition, no need to prove it.
Base case: n = 0,1 then $F_0 = 0, F_1 = 1$ is true.
Induction step: if case $n \geq 0$ and $n + 1$ is true then case n+2 : F_{n+2}=F_{n+1}+F_n=true.
So the case n+2 is also true.
Conclusion: The Fabonacci number is calculated correctly for every case n =1;2;3:::

Time Complexity:

$$T(n) = T(n-1) + O(1)$$

To compute F_n, the algorithm needs n-1 loops, each loop needs 1 addition in $O(n)$. So we need $n-1*O(1) = O(n)$ time.

Space Complexity:

This is a recurrence method using $O(1)$ extra space.

5.2.3

To prove

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^N = \begin{bmatrix} F_{N-1} & F_N \\ F_N & F_{N+1} \end{bmatrix}$$

where $N \geq 1$
Proof:
Base case: N = 1

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^1 = \begin{bmatrix} F_0 & F_1 \\ F_1 & F_2 \end{bmatrix}$$

The statement is true in base case.
Induction hypothesis: Assume that the statement is true for case $N \geq 1$.
Inductive step:

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^{N+1} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^{N} * \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} F_{N-1} & F_N \\ F_N & F_{N+1} \end{bmatrix} * \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} F_N & F_{N+1} \\ F_{N+1} & F_{N+2} \end{bmatrix}$$

The statement is also true in case N+1. Conclusion: It follows that the statement is true for all n since we can apply the inductive step for n = 2; 3; : : :

The $O(\log n)$ time method is:

Pseudocode:

Algorithm 8 Function FabonacciMatrix(A, n)

```
if n = 1 then
  return A
else if n = 2 then
  return A*A
else if n is even then
  return FabonacciMatrix(A * A, n/2)
else if n is odd then
  return A*FabonacciMatrix(A * A, (n − 1)/2)
end if
```

Algorithm 9 Function Fabonacci(n)

```
A ⟸ [0 1; 1 1]
if n ≤ 1 then
  return n
else
  F ⟸ FabonacciMatrix(A, n)
  return F[0][1]
end if
```

Time Complexity:

So each step we convert the problem of $A^n(T(n)\text{time})$ into $A^{n/2}(T(n/2)\text{time})$ and one or two 2*2 matrix multiplition($O(1)$time)), since elementwise operations take $O(1)$ time.
Therefore:

$$T(n) = 2T(n/2) + O(1)$$

According to Master Teorem:

$$T(n) = O(\log n)$$

Space Complexity:

The depth of the recursion tree is $\log n$, each recursion step takes $O(1)$ extra space.
The space complexity is $O(\log n)$

5.3 (c)

Because we are multiplying numbers that have value on the order of ϕ^n, hence have order n bits.
(b)1.
Beacause we only use integer addition in this reccursion method, so:

$$T(n) = T(n-1) + T(n-2) + \Theta(n)$$

Since we know the number of addition is not changed, therefore:

$$T(n) = \Theta(n\sum_{2}^{n} F_i) = \Theta(n\sum_{2}^{n} \phi^i) = \Theta(n\phi^n) = O(2^n)$$

(b)2.
Beacause we only use integer addition in this iteration method, so:

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

Therefore,

$$T(n) = (n-1) * \Theta(n) = \Theta(n^2)$$

(b)3.
We are using only multiplation of time $\Theta(n^2)$ in every loop, so:

$$T(n) = T(n/2) + \Theta(n^2)$$

According to master theorem:

$$T(n) = \Theta(n^2)$$