CSOR W4231: Analysis of Algorithms (sec.001) - Problem Set #1

Jiawen Huang (jh4179). Collaborators: A. Turing. - jh4179@columbia.edu

February 9, 2020

Problem 1

(a)

i. For all the comparison of elements a_i and a_j, it must satisfy either $a_i < a_j$ or $ai \geq a_j$, which create the corresponding two branches. Since the binary tree makes comparison for each elements, all the permutations must appear as a leaf in the tree.
ii. n!
iii. nodes (the circles in the figure)
iv. ***Proof*** Since each node in the tree can lead to at most two branches and there are d nodes in depth d, a binary tree of depth d has no more than 2^d leaves.
v. Suppose we have x reachable leaves. According to ii. and iv., we have $n! \leq x \leq 2^d$, which implies $d \geq \log(n!) = \Omega(n \log n)$.

(b)

We assume input array A has n elements with range D. We need array B to store the sorted output and array C to store the working area. First, we find the minimum value in array A, then we construct array C to contain the numbers of element in array A. We then change the element in Array C to be cumulative, which adds the original value and the value before it. Lastly, we put the value back to the sorted array B.
Correctness:
This is a modified counting sort, which is stable. No need to prove the correctness because the algorithm just counts the number of element in array A and puts them in order based on the value of the element.
Running Time:
Finding the minimum value of input takes time $O(n)$. The for loop to initialize array C and the for loop to make C[i] contain the number of elements no more than i+min both take time $O(D)$. The for loop to make C[i] contain the number of elements equal to i+min and the for loop to put values back into the ordered array B both take time

$O(n)$. Thus, the overall time is $O(n + D)$.
Space:
This is not a in-place algorithm since we need extra space to store array B and C. The space complexity is also $O(n + D)$.
Pseudocode:

sort(A, D)

```
//Find the minimum value min, initialized to A[1]
for $m = 2$ to n do
  if $A[m] < min$ then
    $min = A[m]$
  end if
end for
Let C[0,1,...,D] be a new array, elements initialized to be 0
for $j = 1$ to n do
  $C[A[j] - min] = C[A[j] - min] + 1$
end for
//C[i] now contains the number of elements equal to i+min
for $i = 1$ to k do
  $C[i] = C[i] + C[i - 1]$
end for
//C[i] now contains the number of elements less than or equal to i+min
for $j = n$ to 1 do
  $B[C[A[j] - min]] = A[j]$
  $C[A[j] - min] = C[A[j] - min] - 1$
end for
return B
```

(c)

No, because actually the modified counting sort if not a comparison-based sorting algorithm. It does not implement any comparison when sorting.

Problem 2

We can use fast Walsh-Hadamard transform to design an algorithm, which is to use deivide and conquer method.
Pseudocode:

```
FWHT($k, v$)
  if k = 0 then
    return v
  end if
  $v_1 \leftarrow$ top half part of v
  $v_2 \leftarrow$ bottom Half part of v
  $A \leftarrow FWHT(k-1, v_1)$
  $B \leftarrow FWHT(k-1, v_2)$
  $result \leftarrow \begin{pmatrix} A+B \\ A-B \end{pmatrix}$
  return result
```

Correctness:
Base case: n = 2^k, k=0 then $H_0 v = v$ and the statement is true.
Induction hypothesis: assume that the statement is true for $k \geq 0$.
Inductive step: show it true for case k+1
Since the $H_k v$ is computed correctly, $H_{k+1}v = \begin{pmatrix} H_k & H_k \\ H_k & -H_k \end{pmatrix} * \begin{pmatrix} v_1 \\ v_2 \end{pmatrix} = \begin{pmatrix} H_k v_1 + H_k v_2 \\ H_k v_1 - H_k v_2 \end{pmatrix}$ is also computed correctly, which means the statement is also true in case k+1.
Conclusion: it follows that the statement is true for $k \geq 0$.
Running Time:
For each recursion, we convert the problem of $H_k v$ into $H_{k-1}v_1$ and $H_{k-1}v_2$ with additional additions or substractions, which take $O(1)$ time.
Therefore:

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

According to Master Theorem:

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

where $n = 2^k$, which is faster than the straightforward algorithm.
Space:
The recursive algorithm needs to store extra variable of size 2^k for each recursion and the maximum depth of the recursive tree is k, so the space complexity is $O(k2^k)$, which is also equal to $O(n \log n)$.

Problem 3

(a)

We can use divide and conquer method and divide the problem into two subproblems and then combine them.
Pseudocode:

```
majority(A)
  if $Length(A) = 0$ then
    return None
  end if
  if $Length(A) = 1$ then
    return A
  end if
  $A_1 \leftarrow$ Array of the first $\lfloor n/2 \rfloor$ elements of A
  $A_2 \leftarrow$ Array of the rest elements in A after extracting $A_1$
  $a_1 \leftarrow majority(A_1)$
  $a_2 \leftarrow majority(A_2)$
  if $a_1 = a_2 = None$ then
    return None
  else if $a_1 = a_2 \neq None$ then
    return $a_1$
  else if $(a_1 \neq None, a_2 = None)$ or $(a_2 \neq None, a1 = None)$ then
    $value \leftarrow$ the not-None value in $a_1, a_2$
    $num \leftarrow$ the amount of $a_1$ or $a_2$ in A
    if $num > \lfloor 1/2 * Length(A) \rfloor$ then
      return value
    end if
    return None
  else
    //else condition deals with $a_1 \neq None$, $a_2 \neq None$, but $a_1 \neq a_2$
    return None
  end if
```

Correctness:
As stated, the method is based on divide-conquer principle and find out the majority element by raising candidate value. 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 the majority element for A; if one of the parts has a majority element and the other part not, count the number of that element in A (taking O(n) time) to check if it is a majority element; if not the cases above, which means the two parts have different majority elements or both don't have

majority elements, then A has no majority element.
The correctness of this algorithm simply follows from the fact that if a majority exists in the array, it must exist in either the left or right halves as well. We then collect these majorities and check the entire array to verify that they are in fact majorities.
Running Time:
A recurrence relation is T(n) = 2T(n/2) + O(n)
According to Master Teorem:

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

Space:
The depth of the recursion tree is $\lceil \log n \rceil$ and each recursion step uses $O(n)$ space, so the space complexity should be $O(n \log n)$.

(b)

I want to mimic the algorithm in (a) which is to get a candidate value first, but use another way of raising one instead of divide-and-conquer. We know that the occurrence of the majority element is more than the sum of the occurrence of other elements, so if we can pair every two different elements and discard the pair, what's left must be the majority element as long as there is one.
Correctness:
The correctness of the algorithm depends on the fact that if there is a majority element, the algorithm will always find it and let it to be the candidate value. We can prove it by proving the statement that if there is a majority element, we may miss that and choose another value as the candidate wrong.
Supposing that the majority element is m, m will appear more than half of the length(A) times. If the counter statement is true, we can assume we miss m and choose another one as the candidate.
Under this situation, since m is different from all the others values, they must be represented by different notation (one is -1 and the other is +1), so when we compare current candidate and the current value in working area, they will be offset.
However, if m is the majority element, there must be extra m values left in the array which are represented by -1 since the assumption is that we choose another value in the end. This makes no sense because when we terminate the for loop and return the candidate, the count is supposed to be positive.
Now we know if there is a majority element, the algorithm will always find it as the candidate.
Running Time:
The running time of the for loop to find candidate and the for loop to check if the candidate is a majority element are O(n). The running complexity is O(n).
Space:
The algorithm needs only O(1) extra space.
Pseudocode:

```
majority(A)
  $n \leftarrow Length(A)$
  if n = 0 then
    return None
  end if
  $cand = A[1]$
  $count = 1$
  for $i = 2$ to n do
    if $cand = A[i]$ then
      //same numbers can't be paired, need to be stored to make a pair in the future
      $count = count + 1$
    else
      //offset the previously stored number and discard a pair of different numbers
      $count = count - 1$
    end if
    if $count = 0$ then
      //pairs all discarded and start to store a new number for pair use
      $cand = A[i]$
      $count = 1$
    end if
  end for
  //check if the candidate value is the majority element
  if $check(cand) > \lfloor n/2 \rfloor$ then
    return cand
  else
    return None
  end if
```

```
check(cand)
  $count = 0$
  for $i = 1$ to n do
    if $cand = A[i]$ then
      $count = count + 1$
    end if
  end for
  return count
```

Problem 4

f	g	O	o	Ω	ω	Θ
$n \log^2 n$	$6n^2 \log n$	yes	yes	no	no	no
$\sqrt{\log n}$	$(\log \log n)^3$	no	no	yes	yes	no
$4logn$	$nlog4n$	yes	yes	no	no	no
$n^{3/5}$	$\sqrt{n}logn$	no	no	yes	yes	no
$5\sqrt{n} + \log n$	$2\sqrt{n}$	yes	no	yes	no	yes
$\frac{5^n}{n^8}$	$n^5 4^n$	no	no	yes	yes	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

Problem 5

(a)

Correctness:
Base case: $F_6 = 8 \geq 2^{6/2}$ and $F_7 = 13 \geq 2^{7/2}$, then the statement is true for n=6 and n=7.
Induction hypothesis: assume that the statement is true for n and $n+1$, $n \geq 6$.
Inductive step: show it true for case n+2
Since $F_n \geq 2^{n/2}$ and $F_{n+1} \geq 2^{n+1/2}$,
$F_{n+2} = F_n + F_{n+1} \geq 2^{n/2} + 2^{n+1/2} \geq 2^{n/2} + 2^{n/2} = 2^{n+2/2}$, which means the statement is also true in case n+2.
Conclusion: it follows that the statement is true for $n \geq 6$.

(b)

part 1

Pseudocode:

```
Fib(n)
  if $n \leq 1$ then
    return n
  end if
  return $Fib(n-1) + Fib(n-2)$
```

Correctness: This is the definition and no need to prove it.
Running Time:

$$\begin{aligned}
T(n) &= T(n-1) + T(n-2) + O(1) \\
&= 2T(n-2) + T(n-3) + O(1) \\
&= 3T(n-3) + 2T(n-4) + O(1) \\
&= 5T(n-3) + 3T(n-4) + O(1) \\
&= ... \\
&= F_{n-1} * T(1) + F_{n-2} * T(0) + O(1)
\end{aligned}$$

According to textbook, F_n is equal to $\lfloor \phi^n/\sqrt{5} + 1/2 \rfloor$, where $\phi = (1+\sqrt{5})/2$, so the running time is $\Theta(\phi^n)$, where $\phi = (1+\sqrt{5})/2$. Since $\phi < 2$, the upper bound of running time is $O(2^n)$.

According to (a), we know $F_{n-1} \geq 2^{n-1/2}$ and $F_{n-2} \geq 2^{n-2/2}$, so the lower bound of running time is $\Omega(2^{n/2})$.

Space:
Since the Fib algorithm is a recursion tree, whose space depends on the depth of the recursion and each recursion needs O(1) space, the space complexity is O(n).

part 2

Pseudocode:

```
Fib($n$)
  $pre = 0$
  $cur = 1$
  if $n \leq 1$ then
    return n
  end if
  for $i = 2$ to n do
    $result = pre + cur$
    $pre = cur$
    $cur = result$
  end for
  return result
```

Correctness: This is the definition and no need to prove it.
Running Time:
Since the for loop takes time O(n) and the extra addition and data movement operations take time O(1),

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

which means the time complexity is O(n).
Space:
The algorithm needs constant variable, so the space complexity is O(1).

part 3

Correctness: We need to prove $\begin{bmatrix} F_n & F_{n-1} \\ F_{N-1} & F_{n-2} \end{bmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1}$ is true for $n \geq 2$
Base case: the statement is true for n=2
Induction hypothesis: assume the statement is true for $n \geq 2$
Inductive step: show it true for n+1
Since $\begin{bmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{bmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1}$,
$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n} = \begin{bmatrix} F_n + F_{n-1} & F_n \\ F_{n-1} + F_{n-2} & F_{n-1} \end{bmatrix} = \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix}$, which means the statement is also true in case n+1.
Conclusion: it follows that the statement is true for $n \geq 2$.

Pseudocode:

```
Fib($n$)
  $F = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$
  if $n \leq 1$ then
    return n
  end if
  $power(F, n-1)$ //calculate $F^{n-1}$
  //return the first element in the top-left corner of the matrix
  return $F[1][1]$
```

```
power($F, n$)
  if n=1 then
    return
  end if
  $M = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$
  $power(F, \lfloor n/2 \rfloor)$
  $multiply(F, F)$ //now we get $F^n$ if n is even and $F^{n-1}$ if n is odd
  if i is odd then
    $multiply(F, M)$
  end if
```

```
multiply($F, M$)
  $f1 = F[1][1] * M[1][1] + F[1][2] * M[2][1]$
  $f2 = F[1][1] * M[1][2] + F[1][2] * M[2][2]$
  $f3 = F[2][1] * M[1][1] + F[2][2] * M[2][1]$
  $f4 = F[2][1] * M[1][2] + F[2][2] * M[2][2]$
  $F = \begin{bmatrix} f1 & f2 \\ f3 & f4 \end{bmatrix}$
```

Running Time:
For each step we convert the problem of $T(n)$ time into $T(n/2)$ time plus constant extra integer additions and multiplications taking $O(1)$ time, so we have

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

According to Master Teorem:

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

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

(c)

part 1

We know that $F_n \geq 2^{n/2}$, so F_n has $\lceil log_2 F_n + 1 \rceil$ bits. The addition of F_{n-1} and F_{n-2} requires $\Theta(\lceil log_2 F_{n-1} + 1 \rceil)$ time.

$$
\begin{aligned}
T(n) &= T(n-1) + T(n-2) + \Theta(\lceil log_2 F_{n-1} + 1 \rceil) \\
&= 2T(n-2) + \Theta(\lceil log_2 F_{n-2} + 1 \rceil) + T(n-3) + \Theta(\lceil log_2 F_{n-1} + 1 \rceil) \\
&= 3T(n-3) + 2\Theta(\lceil log_2 F_{n-3} + 1 \rceil) + 2T(n-4) + \Theta(\lceil log_2 F_{n-2} + 1 \rceil) + \Theta(\lceil log_2 F_{n-1} + 1 \rceil) \\
&= ... \\
&= F_{n-1} * T(1) + F_{n-2} * T(0) + \sum_{i=1}^{n-1} F_i * \Theta(\lceil log_2 F_{n-i} + 1 \rceil)
\end{aligned}
$$

The upper bound of running time is $O(n\phi^n)$, where $\phi = (1+\sqrt{5})/2$.

part 2

The running time will be

$$
\begin{aligned}
T(n) &= O(n) + \sum_{i=1}^{n-1} \Theta(\lceil log_2 F_i + 1 \rceil) \\
&\leq O(n) + \sum_{i=1}^{n-1} c * i * log_2 \phi \\
&= O(n) + O(n^2) = O(n^2)
\end{aligned}
$$

.

part 3

The running time will be

$$
\begin{aligned}
T(n) &= T(n/2) + \Theta(\lceil log_2 F_{\lfloor n-1/2 \rfloor} + 1 \rceil^2) \\
&\leq T(n/2) + c * log_2 \phi * (n/2)^2 \\
&= T(n/2) + O(n^2)
\end{aligned}
$$

According to master theorem,

$$T(n) = O(n^2)$$

.