Solutions to Problem Set 2

Rashmi Dwaraka

September 29, 2016

Contents

1 Fibonacci numbers

Problem statement:
The Fibonacci numbers are defined by the following recurrence:

$$F_0 = 0$$
$$F_1 = 1, \ and$$
$$F_i = F_{i-1} + F_{i-2} \ for \ i \geq 2$$

(a) Consider the following algorithm for computing the nth Fibonacci number.

Algorithm 1 Fibonacci

```
1: procedure: FIBONACCI(n)
2:     if n = 0 then
3:         return 0
4:     if n = 1 then
5:         return 1
6:     return FIBONACCI(n − 1) + FIBONACCI(n − 2)
```

Derive a recurrence for Fibonacci(n) and determine the time taken by Fibonacci(n) in terms of the input n, expressed in $\Theta()$-notation.

Solution:

Based on the given algorithm, we can define the runtime function in terms of n, where $n \geq 2$ to calculate n^{th} Fibonacci number as below:

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

As we are dividing the problem into two sub problems of size n-1 and n-2 respectively, in each recurrence step.

Generating a Characteristic equation by considering $T(n) = x^n$ in (1), we get

$$x^n = x^{n-1} + x^{n-2} \tag{2}$$

Divide (2) by x^{n-2}

$$x^2 = x + 1$$
$$x^2 - x - 1 = 0$$

Solving the above quadratic equation, we can derive the roots as ϕ and $\hat{\phi}$, where, $\phi = \frac{(1+\sqrt{5})}{2}$ and $\hat{\phi} = \frac{(1-\sqrt{5})}{2}$. Since the roots ϕ and $\hat{\phi}$ are distinct real roots, the general solution for recurrence can be deduced as $x^n = c_1 \cdot \phi^n + c_2 \cdot (\hat{\phi})^n$
Hence,

$$T(n) = c_1 \cdot \phi^n + c_2 \cdot (\hat{\phi})^n \tag{3}$$

We can derive the constants c_1 and c_2 by applying the initial conditions of the fibonacci sequence, i,e (n = 0) and (n = 1)

When $n = 0$,

$$T(n) = c_1 + c_2 = 0$$

$$c_1 = -1 \cdot c_2$$

When $n = 1$,

$$T(n) = c_1 \cdot \phi + c_2 \cdot \hat{\phi}$$

Substituting the value for ϕ and $\hat{\phi}$ int the above formula, we can derive the constants as $c_1 = \frac{1}{\sqrt{5}}$ and $c_2 = \frac{-1}{\sqrt{5}}$
Substituting c_1 and c_2 in equation (3), we get

$$T(n) = \frac{1}{\sqrt{5}} \cdot \{\phi^n - \hat{\phi}^n\}$$

Since $\hat{\phi} < 1$, we ignore the constant and derive the Fibonacci recurrence as

$$T(n) = \Theta\left(\frac{\phi^n}{\sqrt{5}}\right)$$

(b) Prove that for any $n \geq 2$, F_n equals $A_{1,1}^{n-1}$, where A^n is the n^{th} power of the following matrix:

$$A = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

Solution:

Considering the Nth fibonacci sequence formula,

$$F(n) = F(n-1) + F(n-2) \tag{1}$$

We can rewrite the equation (1) in terms of the given matrix A as below

$$\begin{pmatrix} F(n) \\ F(n-1) \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} F(n-1) \\ F(n-2) \end{pmatrix} \tag{2}$$

$$\begin{pmatrix} F(n-1) \\ F(n-2) \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} F(n-2) \\ F(n-3) \end{pmatrix} \tag{3}$$

Substituting (3) in (2), we get

$$\begin{pmatrix} F(n) \\ F(n-1) \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} F(n-2) \\ F(n-3) \end{pmatrix} \tag{4}$$

rewriting (4) in terms of A,

$$\begin{pmatrix} F(n) \\ F(n-1) \end{pmatrix} = A^2 \cdot \begin{pmatrix} F(n-2) \\ F(n-3) \end{pmatrix} \tag{5}$$

Similarily, we can generalize (5) in the below form:

$$\begin{pmatrix} F(n) \\ F(n-1) \end{pmatrix} = A^i \cdot \begin{pmatrix} F(n-i) \\ F(n-i-1) \end{pmatrix} \tag{6}$$

Substituting $i = (n-1)$ in equation (6), we get

$$\begin{pmatrix} F(n) \\ F(n-1) \end{pmatrix} = A^{n-1} \cdot \begin{pmatrix} F(n-(n-1)) \\ F(n-(n-1)-1) \end{pmatrix}$$

$$\begin{pmatrix} F(n) \\ F(n-1) \end{pmatrix} = A^{n-1} \cdot \begin{pmatrix} F(1) \\ F(0) \end{pmatrix}$$

$$\begin{pmatrix} F(n) \\ F(n-1) \end{pmatrix} = A^{n-1} \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

$$F(n) = A_{1,1}^{n-1}$$

Hence, Proved by Substitution technique.

(c) Use part (b) to obtain an algorithm that computes F_n in O(log n) time.
Solution:

Based on the solution to 1(b) we can deduce that $F_n = A_{1,1}^{n-1}$.

Algorithm: n^{th} Fibonacci

```
1: procedure: FIBONACCI(n)
2:     if n = 0 then
3:         return 0
4:     if n = 1 then
5:         return 1
6:     Let A = [1 1; 1 0]
7:     return Matrix-to-power-n(A,n-1)[1,1]
```

Algorithm: Matrix-to-power-n(A,n)

```
1: procedure: Matrix-to-power-n(A,n)
2:     if n = 1 then
3:         return A
4:     if n is even
5:         res = Matrix-to-power-n(A, n/2)
6:         return (res · res)
```

7: else
8: res = Matrix-to-power-n(A, $\frac{(n-1)}{2}$)
9: return (res · res ·A)

The recurrence relation for the above algorithm can be put as, where c is a constant to compute the multiplication of matrix.

$$T(n) = T\left(\frac{n}{2}\right) + c$$

By substitution method we can show that the time complexity is $O(\log n)$

$$\begin{aligned} T(n) &\leq T(\frac{n}{2}) + c \\ &\leq T(\frac{n}{4}) + 2 \cdot c \\ &\leq T(\frac{n}{8}) + 3 \cdot c \\ &\vdots \\ T(n) &\leq T(1) + \log n \cdot c \\ T(n) &\leq c \cdot \log n \end{aligned}$$

2 Selection from two sorted lists

Problem statement:
Design an $O(logn)$ time algorithm to select the median from a set of 2n keys given in the form of two sorted lists, each of length n. For convenience, you may assume that all of the keys are all distinct.
Briefly justify the correctness of the algorithm. Analyze its running time.

Solution:

Algorithm: Median(A_1[m..n],A_2[m..n])

1: **procedure: Median(A_1[m..n],A_2[m..n])**
2: if Length(A_1) = 1 then
3: return $\frac{(A_1[1]+A_2[1])}{2}$
4: if Length(A_1) = 2 then
5: A = Merge(A_1[1,2]+A_2[1,2])
6: return $\frac{(A[2]+A[3])}{2}$
7: M_1 = FindMedian(A1)
8: M_2 = FindMedian(A2)
9: if $M_1 < M_2$ then
10: Median($A_1[\frac{n}{2}, n], A_2[1, \frac{n}{2}]$)
11: else
12: Median($A_1[1, \frac{n}{2}], A_2[\frac{n}{2}, n]$)

Algorithm: FindMedian(A[1..n])

1: **procedure: FindMedian(A[1..n])**
2: if n is even
3: return $\frac{A[\frac{n}{2}]+A[(\frac{n}{2}+1)]}{2}$
6: else
7: return $A[\frac{n}{2}]$

The Algorithm works by finding the median of both the arrays. Based on the medians found, we can say that the median of the two array combined will be found in the second half of the array which has the smaller median and in first half of the array which has the larger median.

We can consider the above as correct since the arrays are sorted, elements are distinct and are of equal length.
We can recursively shorten the input size and finally at the base condition, the 2 arrays with 2 elements each can be combined in a constant time to find the median.

Let T(n) be the time complexity for the algorithm. The lines 2 to 8 involve only constant time operations. The line 10,12 will be computed in T($\frac{n}{2}$) time.
By substitution method we can show that the time complexity is $O(\log n)$

$$\begin{aligned} T(n) &\leq T(\frac{n}{2}) + c \\ &\leq T(\frac{n}{4}) + 2 \cdot c \\ &\leq T(\frac{n}{8}) + 3 \cdot c \\ &\vdots \\ T(n) &\leq T(1) + \log n \cdot c \\ T(n) &\leq c \cdot \log n \end{aligned}$$

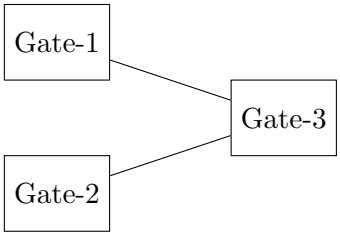
3 A fault-tolerant OR-gate

Problem statement:
Assume we are given an infinite supply of two-input, one-output gates, most of which are OR gates and some of which are AND gates. Unfortunately the OR and AND gates have been mixed together and we cant tell them apart. For a given integer k 0, we would like to construct a two-input, one-output combinational k-OR circuit from our supply of two-input, one output gates such that the following property holds: If at most k of the gates are AND gates then the circuit correctly implements OR. Assume for simplicity that k is a power of two.

For a given integer k 0, we would like to design a k-OR circuit that uses the smallest number of gates.

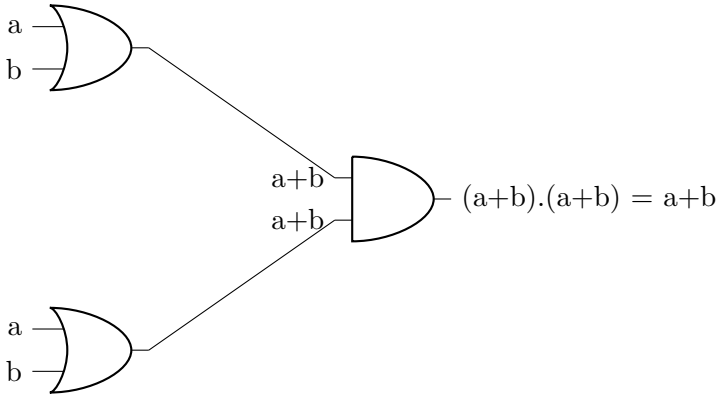
(a) Design a 1-OR circuit with the smallest number of gates. Argue the correctness of your circuit.
Solution:

The 1-OR circuit can be designed with 3 gates as below:

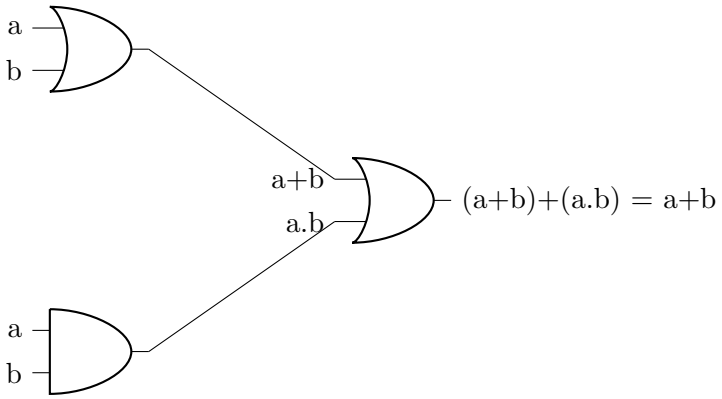


The above circuit can be proved to hold the 1-OR circuit property by considering all the possible cases as below:

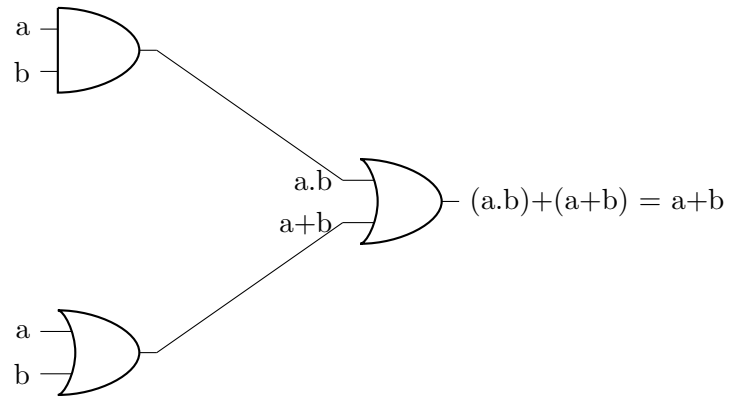
Case(1): Let Gate-1 and Gate-2 be not faulty and consider Gate-3 is faulty, then circuit formed will be as below:



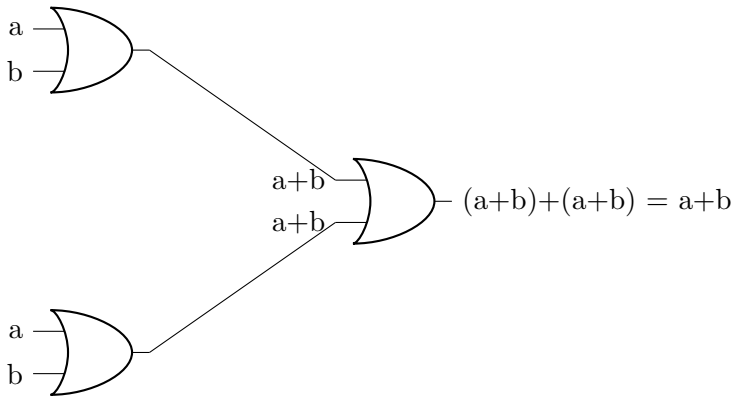
Case(2): Let Gate-1 and Gate-3 be not faulty and consider Gate-2 is faulty, then circuit formed will be as below:



Case(3): Let Gate-2 and Gate-3 be not faulty and consider Gate-1 is faulty, then circuit formed will be as below:



Case(4): Let us consider all gates Gate-1, Gate-2 and Gate-3 are not faulty, then circuit formed will be as below:

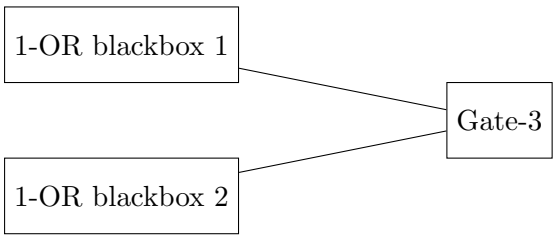


Based on the above mentioned 4 cases, we can conclude that the 1-OR circuit can be formed using 3 gates where the 1-OR circuit will behave as an OR gate when atmost 1 gate is faulty.

(b) Using a 1-OR circuit as a black box, design a 2-OR circuit. Argue the correctness of your circuit.

Solution:

Using 1-OR circuit as blackbox, we can design the circuit to form 2-OR circuit as below:



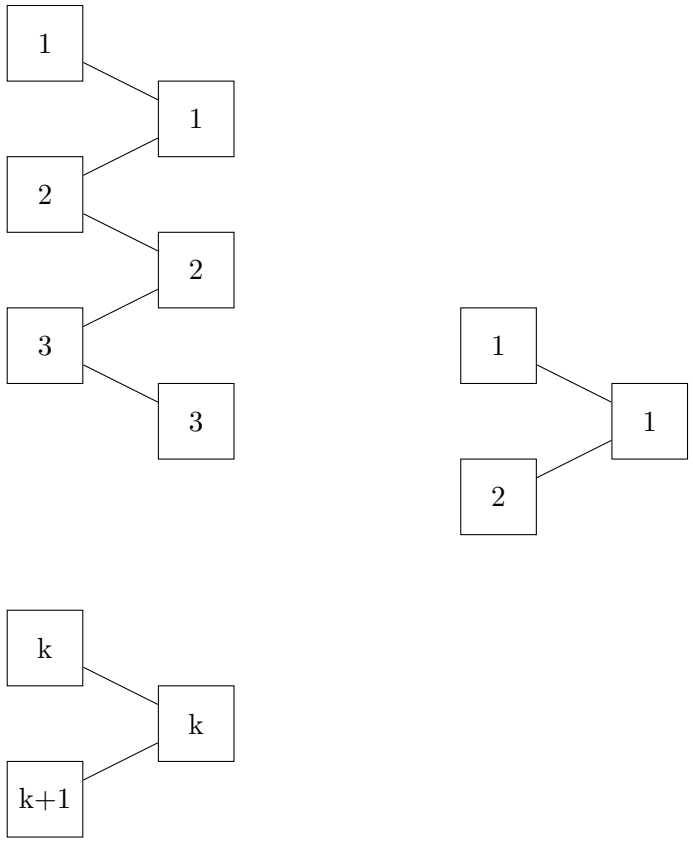
In order to prove the correctness of the circuit, we can consider the following cases:

- **case(1):** Consider one of the 1-OR blackbox is faulty, that means there are more than 1 gate in 1-OR blackbox which are faulty. By this we can consider that the other 2 components are not faulty and has to behave as an OR gate and hence the overall circuit works irrespective of their position as proved in (3a solution).

- **case(2):** Considering all 1-OR blackbox behave as OR and is not faulty, and the 3^{rd} gate is faulty, then the circuit behaves similar to (3a case(1)) and hence behave as OR-gate
- **case(3):** If all the gates inside the 2 blackbox are not faulty and the 3^{rd} gate is not faulty, then the overall circuit behaves as OR circuit.

(c) Generalizing the above approach, or using a different approach, design the best possible k-OR circuit you can and derive a Θ-bound (in terms of the parameter k) for the number of gates in your k-OR circuit. Argue the correctness of your circuit.
Solution:

Generalizing the approach provided in 3a solution, we can say that the number gates at each level for k-OR circuit should be k+1 gates and each next level we have 1 less than the previous till 1 gate as shown below:



By the above arrangement of circuits, we can calculate the total number of gates required for k-OR circuit as:

$$Number\ of\ Gates = (k+1) + (k) + ... + 2 + 1$$

$$Number\ of\ Gates = \frac{(k+1)\cdot(k+2)}{2} \tag{1}$$

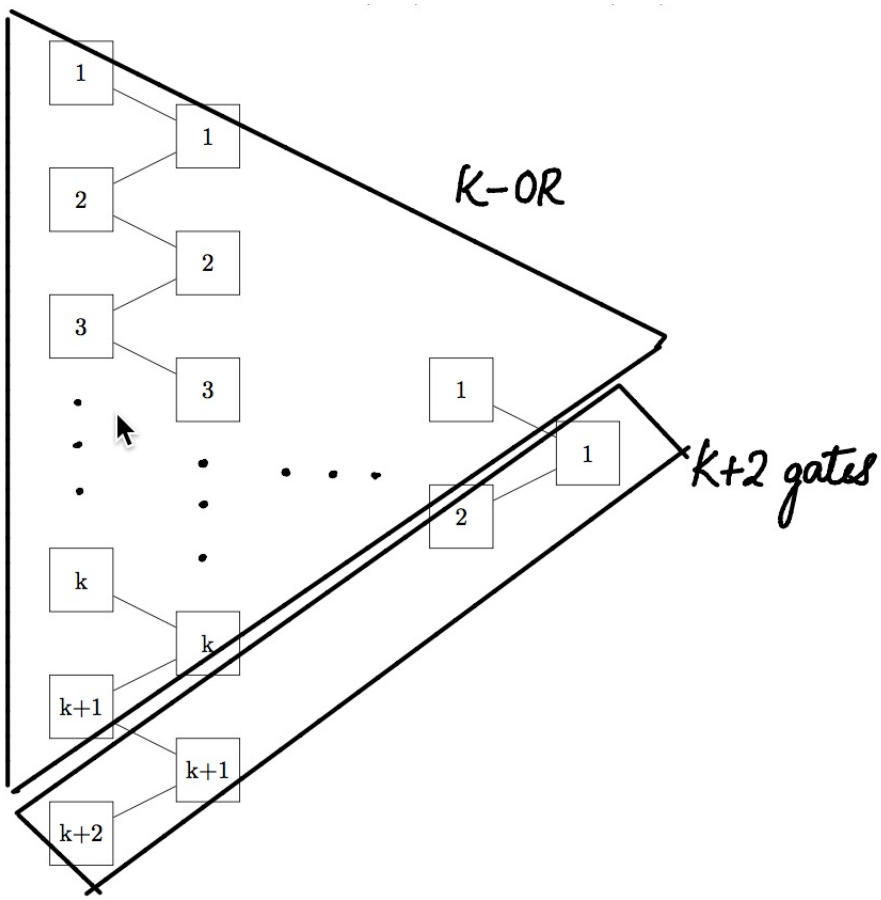
Therefore, from above we can conclude that it requires $\Theta\frac{(k^2+3\cdot k+2)}{2}$ which can be approximated to $\Theta(k^2)$ gates from above calculation.

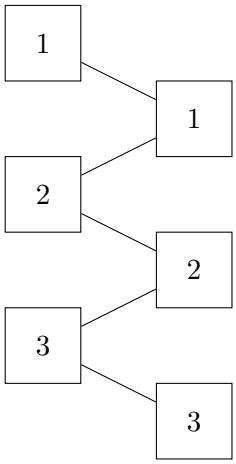
The correctness of the above circuit generalization can be proved using Mathematical induction.

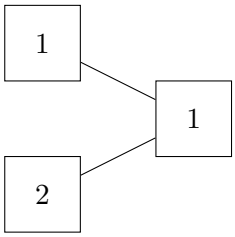
Base Case: Let k = 1, for 1-OR circuit, we can have 2 gates at the first level and 1 gate at the last level. By solution 3a, we can prove the generalization is true for k=1

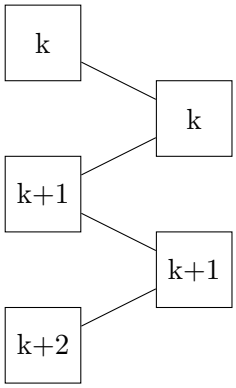
Induction Hypothesis: Let us assume that the generalization is true for k-OR circuit.

Induction Step: We now have to prove that it is true for (k+1)-OR circuit. Using the k-OR circuit above we can add a gate at each level i,e (k+2) gates and make (k+1)-OR circuit.









- **case(1):** Considering k-OR circuit has k faulty gates, then if one of the (k+2) gates being added to build $(k+1)$-OR circuit can be faulty to satisfy the property of $(k+1)$-OR circuit.
- **case(2):** Considering k-OR circuit has $(k-m)$ faulty gates, then if $(m+1)$ gates of the $(k+2)$ gates being added to build (k+1)-OR circuit can be faulty to satisfy the property of $(k+1)$-OR circuit.
- **case(3):** Considering k-OR circuit has no faulty gates, then if $(k+1)$ gates of the $(k+2)$ gates being added to build $(k+1)$-OR circuit can be faulty to satisfy the property of $(k+1)$-OR circuit.
- **case(4):** Considering k-OR circuit has no faulty gates, and if no gates of the $(k+2)$ gates being added to build $(k+1)$-OR circuit is faulty, still this satisfies the property of $(k+1)$-OR circuit.

Substituting k with (k+1) in equation (1):

$$Number\ of\ Gates\ of\ (k+1)-OR\ circuit = \frac{(k+2)\cdot(k+3)}{2}$$

4 Reconstructing a total order

Problem statement:
A group of n runners finished a close race. Unfortunately, the officials at the finish line were unable to note down the order in which the racers finished. Each runner, however, noted the jersey number of the runner finishing immediately ahead of her or him. (There were no ties.) The race officials ask each runner to give an ordered pair, containing two pieces of information: (i) first, his or her own jersey number and (ii) second, the jersey number of the runner who finished immediately ahead of him or her. The winner of the race, who did not see anybody finish ahead of her, did not turn any information in.
You have been asked to design an algorithm that takes as input the n 1 pairs and returns the order in which the runners finished the race. Assume each runner is honest.

(a) Give a deterministic $O(n \cdot \log n)$ time algorithm.

(b) Give a randomized algorithm with expected running time $O(n)$.

You need not prove the correctness of your algorithms. In each case, analyze the running time of your algorithm.

Solution:

Algorithm: FindRanking(A[1..(n-1)][1..(n-1)])

```
1: procedure: FindRanking(A[1..(n-1)][1..(n-1)]
2:    Let S = empty Set and H = empty HashMap
3:    for i ← 1 to n − 1
4:       S = A[i][2]
5:       H[A[i][1]] = A[i][2]
6:    lastPlayer = H.keySet - S
7:    return PlayerOrder(lastPlayer,H)
```

Algorithm: PlayerOrder(player,H)

```
1: procedure: PlayerOrder(player,H)
2:    if player not in H
3:       Let A = empty array of players
4:       return A[1] = player
5:    return Append (PlayerOrder(H(player),H)) with player
```

(a) Give a deterministic $O(n \cdot \log n)$ time algorithm.

The Algorithm FindRanking has the time complexity of $O(n \cdot \log n)$ when we use Red Black tree implementation of HashMaps. Lines 3-5 in FindRanking(A) algorithm runs for $(n-1)$ units and line 6 runs in a constant unit of time. The Red Black tree implementation of Hashmap takes $O(\log n)$ time for each lookup in PlayerOrder Algorithm. Therefore in total, the algorithm requires $(n-1) + c + (n \cdot \log n)$ times. We can approximate the total runtime for the algorithm as $O(n \cdot \log n)$.

(b) Give a randomized algorithm with expected running time $O(n)$.

The Algorithm FindRanking has the time complexity of $O(n)$ when we use the implementation of HashMaps as hash functions. Lines 3-5 in FindRanking(A) algorithm runs for $(n-1)$ units and line 6 runs in a constant unit of time. In PlayerOrder Algorithm, the HashTable implementation provides a randomized hash function whose average runtime for lookup is $O(1)$ time. Therefore in total, the algorithm requires $(n-1)+c+(n\cdot 1)$ times. The expected runtime for the algorithm is $O(n)$.