APP2: Hold'em for n00bs

Pashevich Alexander Alisa Patotskaya Yuliya Patotskaya
Ruben Rey SID-LAKHDAR Riyane

09/10/2015

1 Introduction

A problem arises and that is, we find ourselves playing cards with our little sister in a two (2) people game, and our firm purpose is to beat her, but somehow she manages to win at least half of the games. Because of this situation it was decided to implement an algorithm to improve our chances of success, motivated at this we will be evaluating a series of different algorithms, in those algorithms we will found ourselves evaluating the greedy algorithms, dynamic programming and brutal forced; which they will be detailed later in this report, also we will be explaining which one is better applied for this situation and if we could achieve a solution to the problem encountered.

2 Greedy algorithm

To solve the problem, we have first tried to consider it using a greedy point of view:
Let's consider that solving the problem is equivalent to find, at each step of the game, the best card to pick, regarding only the actual state of the game. This greedy method does not consider the possible evolutions of the game after this step.
In the following greedy algorithm description and analyze, we will first assume that the sister can use any strategy. In a second time, we will determine the complexity and the failure cases of the algorithme when the sister is choosing the biggest card at any step.

2.1 Algorithme descritpion

```
boolean winIsPossible(cards       : array of size N containing cards
                                    guiven by them values
                      sisterStart : boolean)
{
    int min        = 0;   // Lowest index of the deck in the input array
    int max        = N-1; // Highest index of the deck
    int sisterScore = 0;
    int myScore    = 0;
```

```
        // The variables used with "&" may be modified by the function
        if (sisterStarts) playSister(cards, &min, &max, &sisterScore);
        for (i=0; i< N/2; i++)
        {
            d0 = cards[min] - Math.max(cards[min+1], cards[max]);
            d1 = cards[max] - Math.max(cards[min]   , cards[max-1]);
            if (|d0| < |d1|) {myScore += cards[min]; min++;}
            else             {myScore += cards[max]; max--;}
            if (min != max) playSister(cards, &min, &max, &sisterScore);
        }
    }
```

The function p̈laySister ẅill play according to the choosen sister's strategy. It will also update all the game parameters.
Assuming that it is our turn to pick a card, we can pick one card belonging to {cards[0], cards[N-1]}. Given one card of this set, we can compute the points we will earn, and the points the sister can earn. The aim of our algorithm is to chose, at each, step the card that will make the difference between our gain and the sister's gain as big as possible.

This algorithm has many obvious advantages. First of all, it is very easy to implement and to understand. No specific data structure or algorithm are required. Secondly, as we will prove in the next paragraph, this algorithm seems to have an interesting time complexity. However, as we do not consider the whole aspects of the problem, we can wonder if this algorithm will always give an optimal (or at least a working) solution?

2.2 Time and space complexity of this greedy resolution

The algorithm realises in a N/2 length loop, 3 instruction with an O(1) complexity. Our algorithm complexity can be written as:

$$C(N) = \frac{N}{2} * 3 * O(1) = O(N) \tag{1}$$

Thus, the time complexity of our algorithm is linear.
For all the game steps, our algorithm uses the same set of 4 integers (in addition to the N cards array).
Thus, the space complexity of our algorithm is O(N + 4) = O(N)

You may have noticed that in our complexity analyze, we do not consider the complexity of the sister's decision algorithm. This is done on purpose, as the problem we deal with is to find an algorithm which determines our decision. However, the next algorithms we have designed have to compute the sister's decision to evaluate our decisions. Thus, to make the comparison between all our algorithm more faire, we will include the complexity of the sister's algorithm in the computation of the time complexity of our greedy algorithm:
Assuming that at each step the sister pick up the biggest card, we can rewrite our time complexity as follows:

$$C(N) = \frac{N}{2} * (3 * O(1) + O(1)) = O(N) \tag{2}$$

2.3 Failure cases

Despite its easy implementation and its interesting complexity, the greedy algorithm we designed is not an acceptable way to solve the given problem.
An easy way to get convinced of this fact is to consider the following counterexample:

Index	0	1	2	3	4	5
Cards	4	4	6	6	12	9

Running our algorithm on this input deck would make us loose the game by choosing the following cards:

Index	0	1	2	3	4	5
Cards	4	4	6	6	12	9
Player ID	Sister 3	You 3	Sister 2	You 2	Sister 1	You 1

However, a solution which could make us win exists (respecting the same player's order).

Index	0	1	2	3	4	5
Cards	4	4	6	6	12	9
Player ID	You 1	Sister 3	You 3	Sister 2	You 2	Sister 1

This counterexample shows the limits of our method: By considering only the current card we can choose at a given step, we may allow the sister to access to a card which is much butter at the next step.
Thus, at a given step of the game, we should be able to consider the implications of the current choice on all stages of the game until the end. And that is the purpose of the improvement made by the next algorithm.

3 Brute force algorithm

3.1 Algorithme descritpion

We developed an algorithm of brute force which presumes that sister plays optimal algorithm, as if we could win her in her optimal strategy, we could win any of her strategies.

First of all, we would like to state that there is some cardâĂŹs set, on which there is no possibility for us to win. For example, if we will consider cardâĂŹs set [8 4 2 2 4 8] and if our sister plays greedily, optimal play for us will be [8, 4, 2] and optimal play for sister will be [8, 4, 2] no matter who will start the game. That is why in some cases we could not win, it will be draw. But we want to prove that according to our strategy we will not lose, at least it will be a draw.

On every step of the game we have 2 possibilities:

1) It's sister's turn. 2) It's our turn.

On every step we count current player's balance. Current player's balance is equal to sum of player's card minus card's sum of the opponent. We calculate the score for first position as if we start the game first. If player's balance for the first move is greater than zero, than we start the game first. If the balance is less than zero, than we let our sister start and she will lose. If balance equals zero, it means, that we do not have any chance to win, we have only a chance to have a draw. In this situation no matter who will start (in our realization

we will start first). So by this strategy as we choice, who will make the first move, we always have a chance at least nor lose. To start the game, we need to call getBeginStrategy function and to define, who will be the first player. Than after the first sister move we need to call gameOptimal function with left set of cards and so on.

As we said before, we suppose that our sister plays optimal strategy, because if we could win her in her optimal strategy, we could win her in any strategy. We created a gameOptimal function (see the code below), which calculate an optimal player's play. This function returns current balance and the card (left or right), which we should pick.

Algorithm 3.1: OPTIMAL PLAY($cards$)

comment: This function returns pair of values:
comment: first value = 0 if we need to pick left card
comment: first value = 1 if we need to pick right card
comment: second value equals current game balance

procedure GAMEOPTIMAL($cards$)
$N =\leftarrow length(cards)$**if** $N < 2$
then return $(0, sum(cards))$
$balanceLeft \leftarrow cards[0] -$ GAMEOPTIMAL$(cards[1 : N])[1]$
$balanceRight \leftarrow cards[N-1] -$ GAMEOPTIMAL$(cards[0 : N-1])[1]$
if $balanceLeft > balanceRight$
then
return $(0, balanceLeft)$
else
return $(1, balanceRight)$

comment: This function should be called on your very first turn.
comment: Returns the next values:
comment: 0 if you need to pick left card;
comment: 1 if you need to pick right card;
comment: -1 if you need to give sister the first move.

procedure GETBEGINSTRATEGY($cards$)
$card, balance \leftarrow$ GAMEOPTIMAL$(cards)$
comment: balance is positive or equals zero, we should start first
if $balance >= 0$
then
return $(c)ard$
else
return (-1)

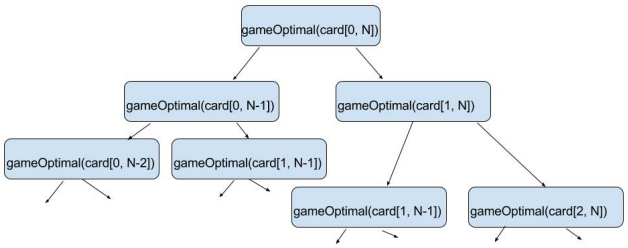

Figure 1: Tree of callings gameOptimal function

3.2 Time complexity

Let's suppose that we have N cards. On every step of the game we call gameOptimal function (on the very first step we call getBeginStrategy function, but it also calls gameOptimal function). Number of calls of gameOptimal function is represented on the figure 3.2. You could notice that the tree of calls is a binary tree, as on every step we perform 2 recursive gameOptimal calls. The height of this tree of calls equals number of elements in the input array of cards (suppose N). Therefore the total time complexity of picking one card will be:

$$C = O(1) * \sum_{r=0}^{n-1} 2^r = O(2^N)$$

=

4 Dynamic programming algorithm

4.1 Algorithm

Algorithms of dynamic programming aim to achieve the optimal result in affordable time. Bearing in mind that an opponent can use any strategy, we developed an algorithm which finds an optimal solution of the problem. Idea of the algorithm is general for dynamic programming: to store and reuse calculated results. The pseudocode is represented in algorithm 4.1.

Calculated results are saved in a table with size $[n, n]$ where n is lengths of the card deck. Left index of the table is position of the first card in remaining deck and right is position of the last. For instance, considering an example with cards $[A, 5, 7, K, 4, 9]$, $l = 2$ and $r = 5$ we assume that current deck is $[7, K, 4, 9]$. Therefore only table cells with $l \leq r$ make sense. The cell with indexes l and

	0 cards[0] = 14	1 cards[1] = 4	2 cards[2] = 7	3 cards[3] = 2	4 cards[4] = 5	5 cards[5] = 12
0 cards[0] = 14	bestTurn = left oppSum = 0 mySum = 14	bestTurn = left ↓ oppSum = 4 mySum = 14	bestTurn = left ↓ oppSum = 7 mySum = 18	bestTurn = left ↓ oppSum = 6 mySum = 21	bestTurn = left ↓ oppSum = 12 mySum = 20	bestTurn = left ↓ oppSum = 18 mySum = 26
1 cards[1] = 4		bestTurn = left oppSum = 0 mySum = 4	bestTurn = right ← oppSum = 4 mySum = 7	bestTurn = left ↓ oppSum = 7 mySum = 6	bestTurn = right ← oppSum = 6 mySum = 12	bestTurn = right ← oppSum = 12 mySum = 18
2 cards[2] = 7			bestTurn = left oppSum = 0 mySum = 7	bestTurn = left ↓ oppSum = 2 mySum = 7	bestTurn = left ↓ oppSum = 5 mySum = 9	bestTurn = right ← oppSum = 9 mySum = 17
3 cards[3] = 2				bestTurn = left oppSum = 0 mySum = 2	bestTurn = right ← oppSum = 2 mySum = 5	bestTurn = right ← oppSum = 5 mySum = 14
4 cards[4] = 5					bestTurn = left oppSum = 0 mySum = 5	bestTurn = right ← oppSum = 5 mySum = 12
5 cards[5] = 12						bestTurn = left oppSum = 0 mySum = 12

Figure 2: Calculated by the algorithm table with input [14, 4, 7, 2, 5, 12]

r stores information about the optimal turn when only cards from l to r are in the deck. Every cell also stores an optimal sum of a player and sum of an opponent which can be achieved after this turn. Example of a filled table is given in Figure 4.1.

To explain how we fill the table, we should start from a trivial case where only one card is left (cells on diagonal with $l = r$). In this case we pick the card which means that we store this action (here "pick up left" is the same as "pick up right") in the corresponding cell as well as a player's sum equals to the value of the card and opponent's sum equals to zero (as there are no cards left afterwards).

Next we fill the n^{th} diagonal of the table. This corresponds to the case when remaining deck consists of exactly n cards. We will do it using the results of the previous diagonal which we assume already calculated. According to the rules of the game from any cell on the n^{th} diagonal (assuming indexes l, r) we can move to two cells on the $n-1^{st}$ diagonal: $l+1, r$ (pick the left card) and $l, r-1$ (pick the right card). Choosing between these two options, we should assess sums that we can achieve in each of them and choose the biggest. However, as we don't know the opponent's strategy, we should do it in general case. The worst case for us is an opponent playing with an optimal strategy (like we do). So if we assume him playing optimally then we will know what to expect. If he plays with any other strategy, we will only benefit. Assuming that in every cell of the previous diagonal we store the optimal sum which can be possibly achieved, we just compare two sums: value of the left card added to the sum which can be achieved after picking it (it should be stored in the table) and the same thing but for the right card. But where should we look for this "future" sum? Remember that in every cell a sum of opponent is stored? Therefore, our future sum is just a field *oppSum* from a cell in the previous diagonal. To sum up, we compare values $cards[l] + table[l+1, r].oppSum$ and $cards[r] + table[l, r-1].oppSum$ and choose the card which gives the biggest result. But what if they are the same? It means that in both cases our final score will be the same (in case of opponent playing optimally, otherwise we can score more) and we can choose any but for simplicity we will choose the biggest of two cards.

However, we didn't answer the question how we fill the fields $oppSum$, $mySum$ and $bestTurn$ in general case (for diagonals with number more than 1). For every cell we choose an optimal turn based on the equation above and store it in $bestTurn$ field. In $mySum$ we write either $cards[l]+table[l+1,r].oppSum$ or $cards[r]+table[l,r-1].oppSum$ depending on which card we pick. Field $oppSum$ is just $table[l+1,r].mySum$ in case we pick the left card or $table[l,r-1].mySum$ otherwise. Therefore in $oppSum$ we store the maximum sum which can be achieved by an opponent taking into account out current optimal turn.

It should be mentioned that storing $oppSum$ is not necessary as we can obtain it by looking at the $mySum$ on next diagonal but for the sake of simplicity we decided to store it in this implementation of the algorithm.

To conclude, we store optimal choices and sums in the table. To fill it, we calculate the very last cell in the beginning ($table[n-1,n-1]$) which will fill the rest of the table recursively. The table is calculated once so in the game we will use the saved results. In case our turn is the first we will move in the table starting from diagonal n (which is only one cell) then switch to diagonal $n-2$, $n-4$ and so on. Otherwise, we will use diagonals $n-1$ (which is two cells), $n-3$, $n-5$ and so on.

4.2 Optimality

To prove that our algorithm is optimal we should use mathematical induction on size of current deck. Base of induction is the situation when only one card is left. In this situation our algorithm will pick the only card which obviously is optimal.

Assuming that condition of optimality is met for step n, we should prove the same for step $n+1$. Here we should refer to the decision policy which is used by the algorithm. It should be mentioned that during filling the table we fill every cell as it would be our turn. However, during the game we use only half of the diagonals of the table to make decisions since the rest is referred to the opponent's turns. After our next turn opponent will need to make a decision. The best turn he can make is stored in n^{th} diagonal which contains optimal turns according to the assumption of the induction. Therefore we can easily get the sum which we can achieve after second player's turn.
TODO: finish writing

4.3 Complexity

Complexity of the algorithm equals to sum of costs of computing all the cells which can be written like $\sum_l \sum_{r \geq l} c(l,r)$ where $c(l,r)$ is cost of computing cell with indexes l and r. It can be see from the algorithm that if we compute cells sequentially, $c(l,r) = O(1)$ as it contains no loop and two calls of $BestTurn$ procedure will return cached values. Therefore:

$$C = O(1) * \sum_{r=0}^{n-1} \sum_{l=0}^{r} 1 = O(1) * \sum_{r=1}^{n} r = O(1) * \frac{n*(n+1)}{2} = O(n^2)$$

Memory needed for the algorithm has the same assessment as every value needs

constant amount of bytes.

4.4 Choice of order of turns

TODO: write here something

4.5 Pseudocode

Algorithm 4.1: DYNAMIC PROGRAMMING($cards$)

$table \leftarrow cell[n, n]$
$n \leftarrow length(cards)$
procedure FILLCELL($l, r, bestTurn, mySum, oppSum$)
 $table[l, r].bestTurn \leftarrow bestTurn$
 $table[l, r].mySum \leftarrow mySum$
 $table[l, r].oppSum \leftarrow oppSum$

procedure BESTTURN(l, r)
 comment: We assume that l and r are correct
 if $table[l, r] \neq 0$ **return** $(table[l, r])$
 if $l = r$
  then FILLCELL($l, r,$ "$left$", $cards[l], 0$)
 if $l \neq r$
  then
   $left \leftarrow$ BESTTURN(L-1,R)(
)$right \leftarrow$ BESTTURN(L,R-1)(
)$leftSum \leftarrow cards[l] + left.oppSum$
   $rightSum \leftarrow cards[r] + right.oppSum$
   if $leftSum > rightSum$
    then FILLCELL($l, r,$ "$left$", $cards[l] + left.oppSum, left.mySum$)
   if $leftSum < rightSum$
    then FILLCELL($l, r,$ "$right$", $cards[r] + right.oppSum, right.mySum$)
   if $leftSum = rightSum$
    then
     if $cards[l] > cards[r]$
      then FILLCELL($l, r,$ "$left$", $cards[l] + left.oppSum, left.mySum$)
      else FILLCELL($l, r,$ "$right$", $cards[r] + right.oppSum, right.mySum$)
 return $(table[l, r])$
main
 return (BESTTURN($n - 1, n - 1$))

5 Conclusion

As can be seen it was able to implement different algorithm in which could be reached and achieve our objective that was increase our chances of victory, and thus solve our problem. However, when analyzing more in depth each of these algorithms was manage to determine that these results may be positive, but they were not necessarily the most optimal. For this reason was able to reach these conclusions:

- The greedy algorithm has many obvious advantages such as being easy to implement and understanding and also does not require to do a data structure, but nevertheless given those advantages is not the most optimal as it could be shown, since it is necessary to consider the implications of every decision in every moment of the game until his completion.
- The brute force algorithm has the advantage of taking into account all possible moves that could be made and can be easily implemented, however this causes us a problem at the same time and that is the cost of these executions is proportional to the possible solutions.
- On the other hand, dynamic programming helps reduce the execution time of the algorithm to find an optimal solution to our problem, dividend our problem in parts and thus using this process it can be achieve know our optimal card in each turn regardless the method that the sister use, and in this way improve our chances of winning.