Master M1 MOSIG UGA & Grenoble INP

Algorithmic Problem Solving

APP2: Holdem for n00bs

Authors :
Eslam MOHAMMED
Habib SLIM
Albert STRÜMPLER
Sofiane TANJI
Manuel TREUTLEIN
Archit YADAV

Teacher :
Ms. Malin RAU

Last Version
October 14, 2019

Contents

1 Introduction

In this APP we have to deal with a card game played by two persons. A sequence of n cards lying on a table face up in a line. The two players take turns choosing a card from the leftmost or rightmost end. It is not allowed to take a card which is not at one of the two ends. After one player takes the last card, the game is over. The player with the higher card sum in the end wins. The figure 1 shows an example of a possible card game.

$\boxed{4}$ 3 5 2 2 14 13 5 7 $\boxed{9}$

Figure 1: Example of a card game. Card 4 and 9 can be chosen from the current player.

In this APP we decided to change to python code for representing algorithms. The reason for this is that python code has a simple structure and resembles to pseudo-code in a way. Additionally, it allows us to represent the developed algorithms in a more detailed way.

One player, named the sister, will always play the so called greedy strategy. Therefore we first consider an algorithm applying this method in chapter 3. Because of some limitations of the greedy strategy we will then consider a complete solution space exploration in chapter 4. This gives us an optimal solution, but with an unacceptable runtime. For this reason we will introduce a dynamic algorithm in chapter 5, resulting in an optimal solution with acceptable runtime. Before we begin with the algorithms, we will introduce in chapter 2 some notations.

2 Notations

In the following, one player is referred to as the sister, the other player is referred to as the strategist.

For all of what follows, we will be using the same data structure to represent the playing cards : a simple array of fixed size containing all card costs that have been drawn, unordered, named "`CARDS`". The elements of `CARDS` are *integers* in the range of $[2, 14]$, whereas the value 2 represents the card two and the value 14 represents the value of an ace. All values in between are assigned appropriately, this means in particular for the face cards that the jack is modeled by 11, the queen by 12 and the king by 13.

The input of all algorithms is a list of cards represented as array `CARDS` and the choice of the sister about who makes the first move, represented as a *boolean*.

In order to visually describe a configuration of cards in a game, we will be using the following diagram notations :

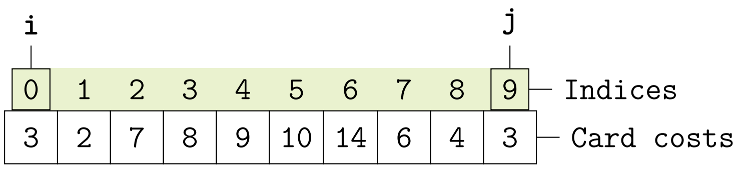

Figure 2: Initial configuration

More generally, a configuration (i, j) describing a situation produced from a set of n cards is such that:

$$(i, j) \in [\![0,\ n-1]\!]^2,\ i \leq j$$

If i is equal to j, we are describing a configuration in which a single card of index i $(= j)$ and of cost `CARDS[i]` (= `CARDS[j]`) is remaining.

Referring to the subconfiguration $(i+1,\ j)$ thus describes a subsequent configuration in which the first player picked the left-most card (here, of cost 3), like shown in figure 3.

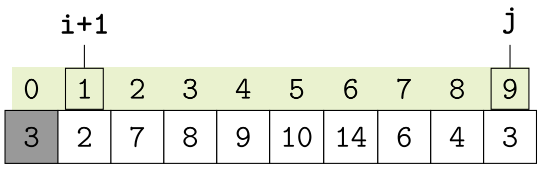

Figure 3: Following subconfiguration

The card of index 0 and of cost 3 has been greyed out, which means that it was selected and can no longer be picked and added to a player's score.

In general, referring to the subconfiguration $(i+n,\ j-k)$ describes a configuration in which n cards have been picked on the left-side of the stack, and k cards have been picked on the right side. Obviously, the constraint $i + n \leq j - k$ must be correct for the configuration to be valid.

3 The greedy algorithm

As presented in the introduction, the first algorithm we studied was the greedy algorithm, it is the method that the sister always uses when she plays the game. In the first part, the strategist player also plays greedy to understand the concept and also to see the algorithm evolution through this APP development.

The greedy algorithm consists in choosing the local optimal choice at each step. It is a naive approach that is preferring what looks like the better choice but may not be globally because of the missing information that isn't considered. In our case the algorithm always picks the biggest card from the two possibilities without considering the other cards.

The tree in figure 4 shows the possibilities for a greedy player trying to pick the best score out of the card deck against another greedy player notated $'' > ''$. As we can see the greedy player always picks the biggest card, but didn't get the best possible result at the end and didn't win.

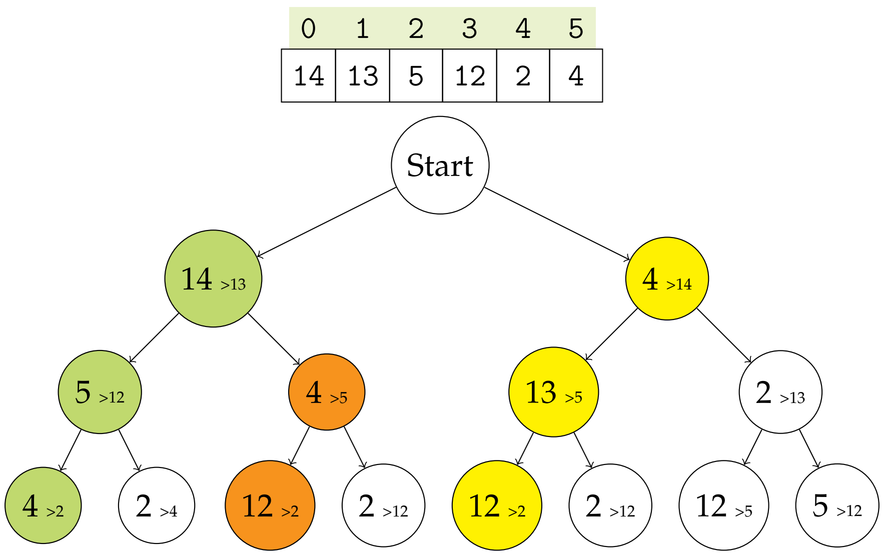

Figure 4: Greedy Choices (green), Best Solution (orange), Better than greedy (yellow)

3.1 Pseudocode

Depending on whether the sister starts or not (meaning if the boolean `sisterFirst` is equal to `True` or `False`), `getScores` computes the resulting scores of the player and the sister when both are playing greedy.

```
def getScores(sisterFirst):
    sum_sister, my_sum = 0, 0
    i, j = 0, len(CARDS) - 1
    if sisterFirst:
        while i <= j:
            # Sister's turn
            if CARDS[i] > CARDS[j]:
                sum_sister += CARDS[i]
                i += 1
            else:
                sum_sister += CARDS[j]
                j -= 1
            # Brother's turn
            if CARDS[i] > CARDS[j]:
                my_sum += CARDS[i]
                i += 1
            else:
                my_sum += CARDS[j]
                j -= 1
    else:
        while i <= j:
            # Brother's turn
            if CARDS[i] > CARDS[j]:
                my_sum += CARDS[i]
                i += 1
            else:
                my_sum += CARDS[j]
                j -= 1

            # Sister's turn
            if CARDS[i] > CARDS[j]:
                sum_sister += CARDS[i]
                i += 1
            else:
                sum_sister += CARDS[j]
                j -= 1
    return my_sum, sum_sister
```

3.2 Complexity, limitations and advantages

Most greedy algorithms have several characteristics in common. Our greedy algorithm for solving the considered problem is no exception. As a quick overview, consider the table 1.

Advantages	**Limitations**
good runtime	not optimal
no extra space	
quick to implement	
easy to understand	

Table 1: Advantages and limitations of the greedy algorithm

As shown in table 1, the list of advantages of a greedy approach is quite appealing. The most important reason for using greedy algorithms is the good run time. Because a greedy algorithm decides locally and without adaptation later on, decisions are made fast. In our case this means a greater or smaller comparison between two integer values for one decision step. The decision is made for all n cards. Because the comparison is a constant operation, we can conclude that the greedy algorithm is in $O(n)$. The worst and best case do not differ and are therefore in $O(n)$ as well.

Another advantage is that we do not have to store anything, except of the array `CARDS` itself. The implementation is done very quickly, in particular we do not need any special data structures or complex algorithms. The idea of the algorithm is intuitively and therefore easy to understand.
So why isn't it possible to stick to this smart solution? The answer is the non optimal solution finding in the global context. Optimal local decisions don't necessarily lead to optimal global solutions. Furthermore, we can think of examples where the greedy algorithm leads into something we could consider a dead end.

For this reason we have to look at different approaches in the following chapters.

4 Complete solution space exploration

The aim of the complete solution space exploration is to find every possible solution for our problem. Therefore we ensure that we find an optimal solution for our problem.

You can compare our solution in the following pseudocode with the figure 5. We have to find every possible branch for the solution. Don't forget that this may not be the solution of the greedy algorithm as seen in figure 4.

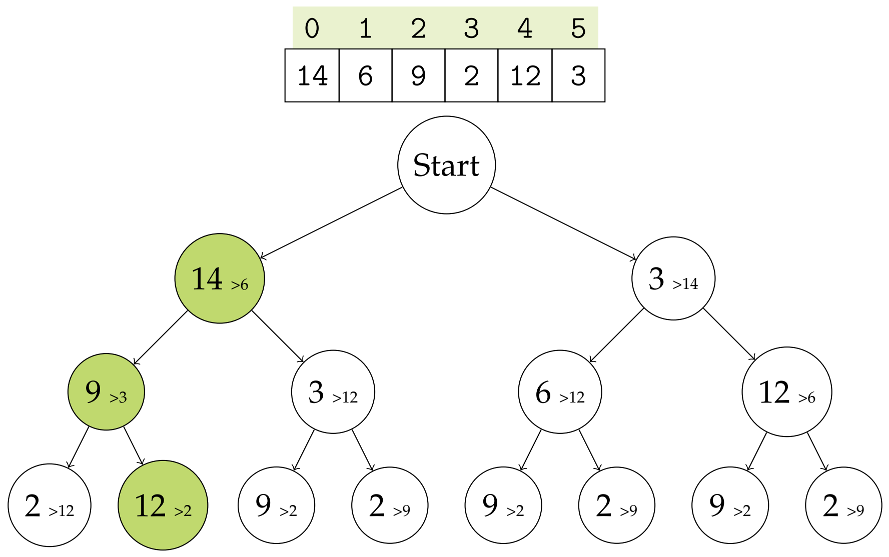

Figure 5: Strategist's best solution (Index values are the sister's picks)

4.1 Pseudocode

We explained that for doing the complete solution space exploration we have to find all possible branches in the corresponding tree. This leads to an algorithm draft where we construct this tree and compute the sum of all branches. Logically, the branch with the highest sum represents the best solution. Nonetheless we decided to use another version in which we don't have to store the whole tree, but compare the best solution 'on the fly'. For this purpose we define a binary number which represents a possible solution strategy for the cards game.

The strategist player has to do $LEN\_CARDS/2$ number of choices. Therefore the number for the solution strategy consists of $LEN\_CARDS/2$ bits. A bit 0 indicates to take the leftmost card, a 1 indicates to take the rightmost card. The strategy is 'incremented' with 1 for every while loop, which means that all the strategies are tested one after another. We stop if we reached the strategy only containing ones. The pseudocode is separated in two functions. The application of the current solution strategy is outsourced.

The `applySolution` function returns the sum for the strategist with regards to the `solutionStrategy`. We update the best value respectively the best strategy if the return of a new `solutionStrategy` is better than our current best solution. After we finished going through all possible solutions we can return the best solution.

```
def completeSolutionSpaceExploration(CARDS, sisterFirst):
    # Define the solutionStrategy in a binary format.
    # If the value is 0, the player should choose the leftmost
    # card, 1 for the rightmost card.
    solutionStrategy = 0b0
    # We reach the last solutionStrategy if the solutionStrategy
    # only consists of one. For a player are LEN_CARDS/2 choices
    # in one game because there are two players for LEN_CARDS number
    # of cards. This leads to exponential choices. With bin() we convert
    # the decimal value into a binary.
    if sisterFirst
        numberOfChoices = floor(LEN_CARDS / 2)
    else
        numberOfChoices = ceil(LEN_CARDS / 2)
    lastSolutionStrategy = bin(pow(2, numberOfChoices))

    # Store the best solutions here.
    bestSolutionValue = 0
    bestSolutionStrategy = 0b0

    while solutionStrategy is not lastSolutionStrategy :
        newSolutionValue = applySolution(sisterFirst, solutionStrategy)
        if newSolutionValue > bestSolutionValue :
                bestSolutionValue = newSolutionValue
                bestSolutionStrategy = solutionStrategy
            solutionStrategy++

        return (bestSolutionStrategy, bestSolutionValue)

def applySolution(sisterFirst, solutionStrategy)
    # Save the current round. One round is one card choice
    # of the sister and one of the strategist.
    currentRound = 1
    # sum up the result for the strategist.
    sum = 0
    i = 0
    j = LEN_CARDS - 1

    while i != j
        if (sisterFirst):
            # Sister takes the local optimum
            if (CARDS[i] > CARDS[j]):
                i++
            else:
                j -= 1
        else:
            # Here we apply the solutionStrategy, the strategist
            # follows the solutionStrategy.
            # We choose the card depending on the bit in the
            # solution strategy. We get the correct bit with
            # the bitAtPos() function with the informal signature:
            # bitAtPos() : position -> binary number -> bit at position
            if bitAtPos(currentRound, solutionStrategy) == 0:
                sum = CARDS[i]
                i++
            else:
                sum = CARDS[j]
                j -= 1
        sisterFirst = not sisterFirst
        currentRound++

    return sum
```

4.2 Complexity

We have seen that in this game either the sister, or the strategist starts the game. It is already known that the sister will always pick the highest valued card, that's why the number of possible games will always be less when the sister starts, since she would always start with the same highest card, thereby restricting the number of possible moves. Whereas in the strategist's case, we could start with either of the card since we are trying to play strategically. Therefore we can move forward with the idea that for complexity analysis part, the derivation of the space/time complexity is only necessary for the worst case. Hence, in the complexity part we will only look at the cases in which the strategist starts, since there are more possible play combinations.

Let n be the total number of cards.

The idea is that in this game, half of the cards $\frac{n}{2}$ will go to the strategist, and half of the cards will go to the sister. If n is odd numbered, the strategist will get one card extra ($\frac{n}{2}$ + 1), since the strategist is starting the game.

Since at a time we have two possible options only, we can write the number of possible games as $2^{\frac{n}{2}}$. This is the case for the strategist starting only. When sister starts, she would be the limiting case and restrict the number of cards to half the number of cases, or in best case be equal to $2^{\frac{n}{2}}$. In order to accommodate for the worst case, well take the later case. Therefore, the total number of cases is $2 * (2^{\frac{n}{2}})$. In terms of big-O notation, we can write : $\mathcal{O}(2 * 2^{\frac{n}{2}}) = \mathcal{O}(2^{\frac{n}{2}+1}) \approx \mathcal{O}(2^n)$: time complexity is exponential.

5 Dynamic Programming solution

The dynamic programming techniques are primarily used for problem optimization when attempting to reduce the usually exponential complexity into a polynomial one.

Our problem solving process involves three components:

- **Simple Subproblems**: We must find a way of breaking the global problem into subproblems, each having the same structure to the global problem.
- **Subproblem Optimality**: In order to find the global optimum solution, an optimal solution for each subproblem was composed.
- **Subproblem Overlapping**: Optimal solutions to unrelated subproblems can contain common subproblems. Indeed, such overlap allows us to improve the efficiency of a dynamic programming algorithm by storing solutions to subproblems. This last property is particularly important for dynamic programming algorithms, because it allows them to take advantage of memoization, which is an optimization that allows us to avoid repeated recursive calls.

To make the problem easier, we make some assumptions about the behavior of players in some undefined situations :

- If the player sister has to choose between two cards of equal cost, she always picks the right-most card
- If the player strategist has to choose between two cards that yield equal potential scores, he picks the right-most card as well [1]

In the last part 5.2.1, we attempt to produce a more general version of our dynamic programming approach, getting rid of unnecessary assumptions.

5.1 First recursive solution

To begin the problem solving, we suggest a first solution that does not use dynamic programming techniques such as memoization and tabulation. It is a recursive method that finds the optimal solution for the strategist player, under the assumptions mentioned in the introduction of section 5.

[1] As shown later on, this assumption has no impact on accurately predicting win or loss.

5.1.1 Subproblem decomposition

We will consider for each configuration two variables `firstScore` and `secondScore` which are equal to the **final** score[2] of the player making respectively the first, and the second move from this configuration. To introduce this solution, let us consider the simple configuration described in figure 6.

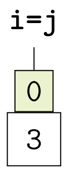

Figure 6: Configuration $(0, 0)$

Let rem_0, rem_1 be the remaining points that the first player, second player (respectively) are yet to earn in the subsequent configurations. After this configuration is played (no matter which of the player, sister or the strategist is playing), the resulting scores will be `firstScore = 3 +` rem_0 and `secondScore = 0 +` rem_1.

However, since all cards have been picked, we have $rem_0 = rem_1 = 0$ and the game ends. This simple observation can be extended in a configuration with two playing cards, like shown in figure 7 :

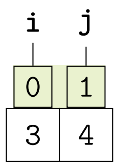

Figure 7: Configuration $(0, 1)$

If the first player in this configuration picks the right-most card, we have `firstScore = 4 +` rem_0 and `secondScore =` rem_1. Here however, a subsequent configuration exists, which is the configuration $(0, 0)$ described in figure 6. The second player in configuration $(0, 1)$ becomes the first player in subconfiguration $(0, 0)$, and we then have :

$$firstScore_{(0,1)} = CARDS[1] + rem_0 = CARDS[1] + secondScore_{(0,0)} = 4 + 0 = 4$$

$$secondScore_{(0,1)} = rem_1 = firstScore_{(0,0)} = CARDS[0] = 3$$

[2] That is, the score obtained after the whole game has been played from this configuration.

Now, if we extend again our configuration with two more cards, we get :

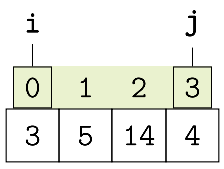

Figure 8: Configuration $(0,3)$

Let's assume that the first player picks the left-most card. His score for configuration $(0,3)$ will be equal to :

$$firstScore_{(0,3)} = CARDS[0] + rem_0 = 3 + rem_0$$

The second player's score will then simply be :

$$secondScore_{(0,3)} = rem_1$$

Since he didn't get to pick any card for this configuration.
However, for the next configuration we have :

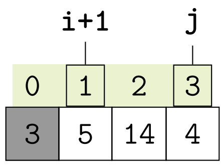

Figure 9: Configuration $(1,3)$

Let $firstScore_{(1,3)}$, $secondScore_{(1,3)}$ be the scores for the players in this configuration $(1,3)$. Again, we notice that the first player to pick a card for this configuration was the second player to pick a card in previous configuration $(0,3)$ (figure 9), and vice versa. This means that we have :

$$\begin{cases} firstScore_{(0,3)} = CARDS[0] + rem_0 = CARDS[0] + secondScore_{(1,3)} \\ secondScore_{(0,3)} = rem_1 = firstScore_{(1,3)} \end{cases} \tag{1}$$

More generally, with (i,j) being the initial configuration, (n,k) being the immediately subsequent configuration and $\alpha \in \{i,j\}$ being the choice of the first player in configuration (i,j), we have if $i \neq j$:

$$firstScore_{(i,j)} = \begin{cases} CARDS[i] + secondScore_{(i+1,j)}, & \text{if } \alpha = i \\ CARDS[j] + secondScore_{(i,j-1)}, & \text{else if } \alpha = j \end{cases} \tag{2}$$

$$secondScore_{(i,j)} = \begin{cases} firstScore_{(i+1,j)}, & \text{if } \alpha = i \\ firstScore_{(i,j-1)}, & \text{else if } \alpha = j \end{cases} \tag{3}$$

If $i = j$, as described in figure 6, we have :

$$firstScore_{(i,j)} = CARDS[i] \qquad secondScore_{(i,j)} = 0 \tag{4}$$

5.1.2 Recursive formula

Using the definitions and observations made in section 5.1.1, we can start writing the recursive formulas defining the resulting scores of configurations for both players, dealing with the two possible cases :

- The first player for configuration (i, j) is the sister
- The first player for configuration (i, j) is the strategist

If the first player for configuration (i, j) is the sister, she always takes the card between i and j that yields the highest value. Following our assumptions, if the two cards are of equal cost, she picks the right-most card.
We then have for the `firstScore`[3] :

$$firstScore_{(i,j)} = \begin{cases} CARDS[i] + secondScore_{(i+1,j)} & \text{if CARDS[i] > CARDS[j]} \\ CARDS[j] + secondScore_{(i,j-1)} & \text{otherwise} \end{cases} \tag{5}$$

If the first player is the strategist however, he must pick the card that yields a greater global score. This global score can be written as such :

$$firstScore_{(i,j)} = \max\Big(CARDS[i] + secondScore_{(i+1,j)},\ CARDS[j] + secondScore_{(i,j-1)}\Big) \tag{6}$$

5.1.3 Recursive implementation

Our implementation for this version of the program is very straightforward and comes directly from our recursion formula. We first define the global constants of our program.

- `CARDS` is an array containing all cards as described in the introduction
- `LEN_CARDS` is the total number of cards (equal to n)

[3] `secondScore` is still defined as in equation 3 with α being equal to i in first case, and j in the second

- SISTER_FIRST is a boolean, which is equal to true if the sister picks a card first

Let *getScores* be a function operating on configurations of list CARDS that takes as parameters :

- i,j : The configuration indexes
- sister_first : equals to true if the sister is the one to pick a card for this configuration

And returns a tuple :

$$getScore(i, j, sisterFirst) = (firstScore, \ secondScore)$$

(firstScore, secondScore) being the total score of the first and second players respectively for the configuration (i, j) if the sister start first or not, **in the optimal case for the strategist**. This means that if sisterFirst = true, firstScore is the sister's score, and firstScore > secondScore means that the sister won for configuration (i, j).

We then implement the recursive formulas defined in section 5.1.2 :

```python
def getScores(i, j, sisterFirst):
    if (i==j): return (CARDS[i], 0)
    if (sisterFirst):
        # Sister takes the card of superior cost (local optimum)
        if (CARDS[i] > CARDS[j]):
            scoreTuple = getScores(i+1, j, not sisterFirst)
            firstScore  = CARDS[i] + scoreTuple[1]
            secondScore = scoreTuple[0]
        else:
            scoreTuple = getScores(i, j-1, not sisterFirst)
            firstScore  = CARDS[j] + scoreTuple[1]
            secondScore = scoreTuple[0]
    else:
        leftCardScore = getScores(i+1, j, not sisterFirst)
        rightCardScore = getScores(i, j-1, not sisterFirst)

        # Strategist takes the card that leads to better score (global optimum)
        if (CARDS[i] + leftCardScore[1] > CARDS[j] + rightCardScore[1]):
            firstScore = CARDS[i] + leftCardScore[1]
            secondScore = leftCardScore[0]
        else:
            firstScore = CARDS[j] + rightCardScore[1]
            secondScore = rightCardScore[0]

    return (firstScore, secondScore)
```

The fact that the first player becomes the second player in the next round is implemented by flipping the boolean "`sisterFirst`" in the recursive calls to `getScores`.

To give a practical example for this function, we will re-use the following configuration :

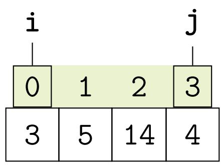

Figure 10: Configuration $(0, 3)$

On this configuration, whoever gets the card of cost 14 wins. If the first player is the sister she will always pick the 4 card, thus giving the strategist immediate win. If the strategist starts, he should pick 3 so that the sister is forced to pick either 5 or 4, thus giving him the win.

Thus, if the sister plays first, the strategist should always win with a score of 14 + 3 = 17, and if the strategist plays first he should win with the same score (picking 3, sister picks 5, strategist picks 14 and wins).

Running this function on this starting configuration with two values for `sisterFirst` gives us the scores in comments in the following snippet of code :

```
scores = getScores(0, LEN_CARDS - 1, False) # => expected : (17, 9)
scores = getScores(0, LEN_CARDS - 1, True)  # => expected : (9, 17)
```

Which are indeed the predicted results for this example (the strategist always wins with a score of 17).

5.1.4 Complexity

As an illustration, we will reuse this following simple configuration to illustrate a function call tree :

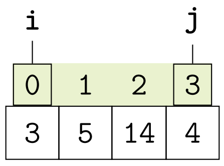

Figure 11: Configuration $(0, 3)$

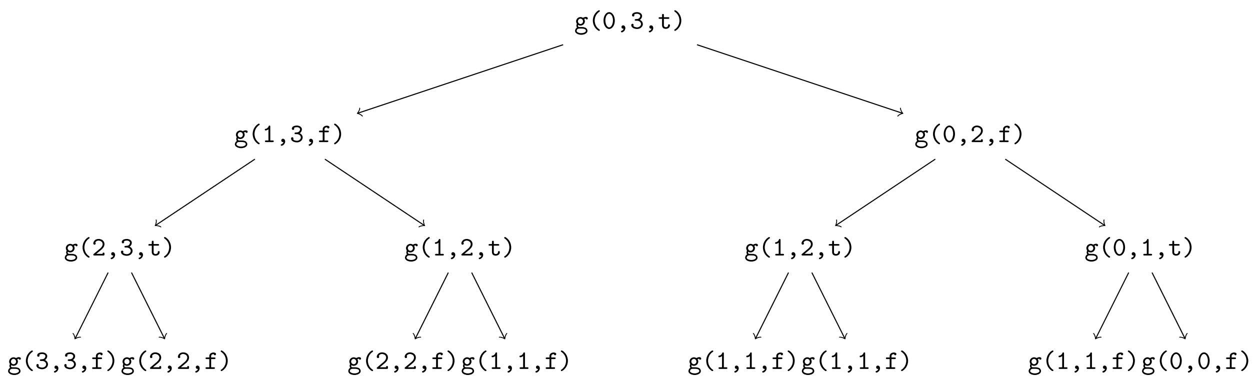

Figure 12: Recursion call tree, g is function getScores, t is true, f is false

In the above figure, given a 4-card deck, it yields 15 configurations because for every turn of player we have 2 choices until only a card is remaining. Hence, given a n-card deck we can say that the complexity is in $O(2^n - 2) \approx O(2^n)$ which is exponential. Recursion yields an exponential time caused by the recursive calls.

However, if we look through the above diagram we can find that the deeper recursive calls recompute a lot of values (e.g. `g(1,2,t)` is computed twice from both `g(1,3,f)` and `g(0,2,f)`) from primitive calls.

5.2 Memoization and Tabulation

To address such overhead from reptitive recursive calls, dynamic programming makes use of the subproblem decomposition by reusing them without recalculation. It uses either Memoization or Tabulation. Memoization is recursive like the aforementioned technique but more optimized such that it starts from the final(addressed) state through the states that computes the target in a top-down fashion. In contrary, Tabulation follows a bottom-up fashion by addressing iteratively(not recursively) the base case that gives our targeted configuration. Both techniques are widely used but it depends on the nature of the subproblem addressed.

5.2.1 Memoization

In order to improve the complexity of our first recursive version, we remove redundant calculations by caching the results of the sub-problems. To achieve this, we allocate a two-dimensional cache of size $n \times n$ that will contain the computed subconfigurations.

In practice, $cache[i][j]$ will yield the tuple `(firstScore, secondScore)` of configuration (i, j).

Below is the memoized version of our `getScores` function :

```
cache = [[0]*LEN_CARDS for _ in range(LEN_CARDS)]
def getScores(i, j, sisterFirst):
    if (cache[i][j] != 0): return cache[i][j]
    if (i==j): return (CARDS[i], 0)

    if (sisterFirst):
        # Sister takes the local optimum
        if (CARDS[i] > CARDS[j]):
            scoreTuple = getScores(i+1, j, not sisterFirst)
            firstScore  = CARDS[i] + scoreTuple[1]
            secondScore = scoreTuple[0]
        else:
            scoreTuple = getScores(i, j-1, not sisterFirst)
            firstScore  = CARDS[j] + scoreTuple[1]
            secondScore = scoreTuple[0]
    else:
        leftCutScore = getScores(i+1, j, not sisterFirst)
        rightCutScore = getScores(i, j-1, not sisterFirst)

        # Strategist takes the global optimum
        if (CARDS[i] + leftCutScore[1] > CARDS[j] + rightCutScore[1]):
            firstScore = CARDS[i] + leftCutScore[1]
            secondScore = leftCutScore[0]
        else:
            firstScore = CARDS[j] + rightCutScore[1]
            secondScore = rightCutScore[0]

    cache[i][j] = (firstScore, secondScore)
    return cache[i][j]
```

Some precisions :

- The default value we use for our cache is 0, because an initialized cache entry must contain a tuple object.
- Since configurations (i, j) with $i > j$ are impossible, half of our table below the diagonal is unused (as seen in figure 13).
- Because the sister makes choices only based on the two available cards, and not on the overall score of subconfigurations, some configurations will never be explored.

The last point is explained a bit more completely below. In our Python function, we have the following condition :

```
# Sister takes the local optimum
if (CARDS[i] > CARDS[j]): # alpha = i
    scoreTuple = getScores(i+1, j, not sisterFirst)
    firstScore  = CARDS[i] + scoreTuple[1]
    secondScore = scoreTuple[0]
else:                     # alpha = j
    scoreTuple = getScores(i, j-1, not sisterFirst)
    firstScore  = CARDS[j] + scoreTuple[1]
    secondScore = scoreTuple[0]
```

If case $\alpha = i$ is taken, then `getScores(i,j-1)` will never be computed and the subconfiguration $(i, j-1)$ will never be considered[4].

As an illustration, in table 2 below is an example of the resulting cache of the execution of our optimized recursive function, on the initial configuration described in figure 13 :

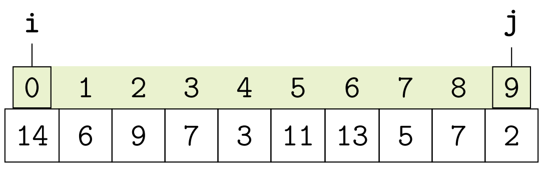

Figure 13: Configuration $(0, 9)$

[4] Please refer to figure 14 for the dependency graph further justifying this argument.

0	0	0	0	0	0	0	0	(43, 32)	(46, 31)
0	0	0	0	0	0	0	(29, 25)	(32, 29)	(31, 32)
0	0	0	0	(12, 7)	(18, 12)	(25, 12)	(25, 23)	(32, 23)	(32, 25)
0	0	0	(7, 0)	(7, 3)	(14, 3)	0	(23, 16)	(23, 23)	(25, 23)
0	0	0	0	(3, 0)	(11, 3)	(16, 3)	(16, 16)	(23, 16)	(23, 18)
0	0	0	0	0	(11, 0)	0	(16, 13)	0	(18, 20)
0	0	0	0	0	0	(13, 0)	(13, 5)	(18, 7)	(20, 7)
0	0	0	0	0	0	0	(5, 0)	(7, 5)	(7, 7)
0	0	0	0	0	0	0	0	(7, 0)	(7, 2)
0	0	0	0	0	0	0	0	0	(2, 0)

Table 2: Resulting entries for the memoization cache

As expected, on the diagonal we have entries of the form `(CARDS[i], 0)`. We can also see that entries below the diagonal are never initialized. Trying to use a data structure that would not leave half of the table uninitialized is not relevant towards space complexity, because it would still be in $O(n^2)$.

As aforementioned, some entries in the memoization cache don't have to be computed because of the sister's greedy strategy : this is one of the advantages of the top-down approach, as we only need to solve subproblems that are definitely required to obtain the final solution.

5.2.2 Memoization solution complexity

In general, you can bound the runtime of memoized functions by bounding the number of subproblems and multiplying by the maximum amount of non-recursive work performed for a subproblem. In some cases this wont be the tightest bound, but here all of the subproblems (which is $O(n^2)$) perform the same amount of non-recursive work which is $O(1)$ for accessing the cache which ends up having a $O(n^2)$ complexity.

5.2.3 Tabulation

Our current complexity is improving, however we would like to optimize our function a bit more by getting rid of recursion overhead. We start by making a dependency graph of our cache entries. In gray are the entries for the configurations in which the starting player for the global configuration (the upper-right one in the diagram) is the one to pick a card, and in black are the ones in which the second player becomes the first to make a move.

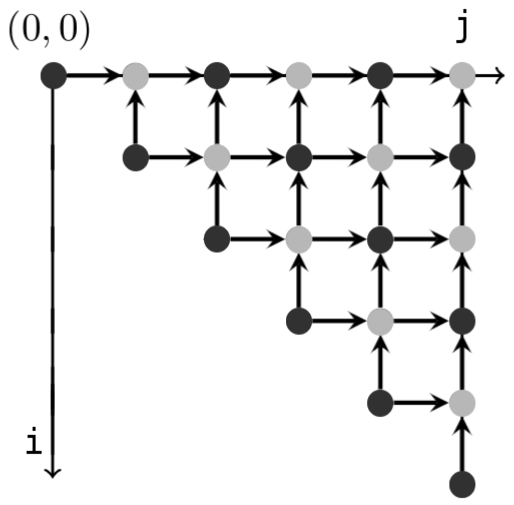

Figure 14: Dependency graph

Remark : In a top-down approach, like the one presented in section 5.2.1, computing all entries is not mandatory because the sister always picks the local optimum. Some entries of the cache for some subconfigurations will never have to be computed to get an optimal answer, like shown in table 2. However, in a bottom-up approach, we will have to compute all entries to get the scores of the final configuration.

This does not change the algorithmic complexity which will still be in $O(n^2)$, however it could potentially affect run time. It is a trade-off between getting rid of the recursion overhead and dramatically reducing space complexity (which will be in $O(n)$ for the optimized bottom-up approach), and computing some unnecessary configurations.

To get this bottom-up solution to work, we consider the bottom diagonal and move all the way up to the global configuration. The first diagonal is initialized easily, with :

$$cache[i][i] = (CARDS[i], 0),\ i \in [\![0,\ n-1]\!] \tag{7}$$

We then move up, computing the next stage using the recurrence formula, as described in figure 15. To know whether or not each diagonal corresponds to the first player or the second player picking a card (gray and black colors in the diagram), we just compute the following modulo of the j index of the current diagonal :

$$(j+n) \equiv_2 1 \implies sisterFirst_{diag_j} = sisterFirst_{(0,\ n-1)} \tag{8}$$

$$(j+n) \equiv_2 0 \implies sisterFirst_{diag_j} = \neg\, sisterFirst_{(0,\ n-1)} \tag{9}$$

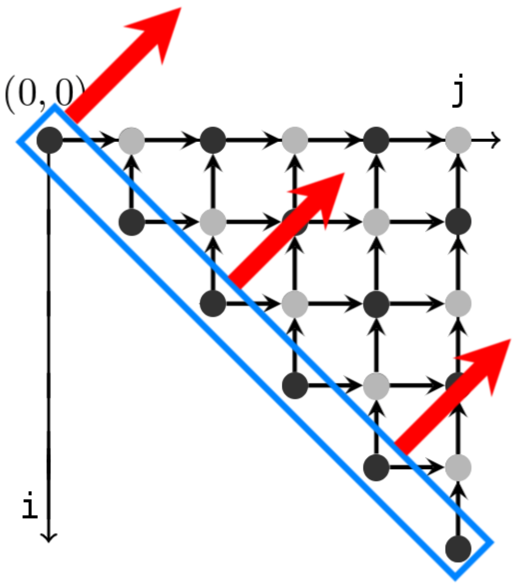

Figure 15: Following the diagonal

Below is the Python implementation of the bottom-up version of the program :

```python
cache = [[0]*LEN_CARDS for _ in range(LEN_CARDS)]
# Finding which player is first on diagonal {diag_index}
def isSisterFirst(diag_index, sisterFirst):
    if ((diag_index + LEN_CARDS)%2 == 1):
        return sisterFirst
    else:
        return not sisterFirst

# Updating a cell value on a diagonal
def updateCell(coord, sisterFirst):
    i,j = coord
    if (sisterFirst):
        # Sister takes the local optimum
        if (CARDS[i] > CARDS[j]):
            firstScore  = CARDS[i] + cache[i+1][j][1]
            secondScore = cache[i+1][j][0]
        else:
            firstScore  = CARDS[j] + cache[i][j-1][1]
            secondScore = cache[i][j-1][0]
    else:
        # Strategist takes the global optimum
        if (CARDS[i] + cache[i+1][j][1] > CARDS[j] + cache[i][j-1][1]):
            firstScore = CARDS[i] + cache[i+1][j][1]
            secondScore = cache[i+1][j][0]
        else:
            firstScore = CARDS[j] + cache[i][j-1][1]
            secondScore = cache[i][j-1][0]

    cache[i][j] = (firstScore, secondScore)

def getScores(i, j, sisterFirst):
    if (i==j): return (CARDS[i], 0)

    # Initializing the main diagonal
    for k in range(LEN_CARDS):
        cache[k][k] = (CARDS[k], 0)

    # Computing all cache values diagonal by diagonal
    for diag_index in range(1, LEN_CARDS):
        sisterFirstDiag = isSisterFirst(diag_index, sisterFirst)

        # Updating all cells in the diagonal
        it_coord = [0, diag_index]
        for _ in range(LEN_CARDS - diag_index):
            updateCell(it_coord, sisterFirstDiag)
            it_coord = [x+1 for x in it_coord]

    return cache[0][LEN_CARDS-1]
```

No more recursive calls are necessary to obtain the optimal score for the strategist, however we now need to compute every cache value. As an example, in table 3 is the cache obtained when executing this new bottom-up function on the configuration of the aformentioned figure 13 (as a reminder, table 2 was the final cache using the top-bottom solution).

This new version of the function returns exactly the same optimal scores and intermediate scores for subconfigurations, however it makes a lot more computations than the top-bottom version. The complexity however still remains the same, and we got rid of the recursion overhead.

(14, 0)	(14, 6)	(20, 9)	(23, 13)	(26, 13)	(27, 23)	(37, 26)	(39, 29)	(43, 32)	(46, 31)
0	(6, 0)	(9, 6)	(13, 9)	(13, 12)	(23, 13)	(26, 23)	(29, 25)	(32, 29)	(31, 32)
0	0	(9, 0)	(9, 7)	(12, 7)	(18, 12)	(25, 18)	(25, 23)	(32, 23)	(32, 25)
0	0	0	(7, 0)	(7, 3)	(14, 7)	(20, 14)	(23, 16)	(23, 23)	(25, 23)
0	0	0	0	(3, 0)	(11, 3)	(16, 11)	(16, 16)	(23, 16)	(23, 18)
0	0	0	0	0	(11, 0)	(13, 11)	(16, 13)	(20, 16)	(18, 20)
0	0	0	0	0	0	(13, 0)	(13, 5)	(18, 7)	(20, 7)
0	0	0	0	0	0	0	(5, 0)	(7, 5)	(7, 7)
0	0	0	0	0	0	0	0	(7, 0)	(7, 2)
0	0	0	0	0	0	0	0	0	(2, 0)

Table 3: Resulting entries for the memoization cache

This solution can be further down optimized in memory to $O(n)$ space complexity by storing only two subsequent diagonals at a time. In figure 16 is an illustration of the two diagonals that will be kept in memory at some given point of our iterative process.

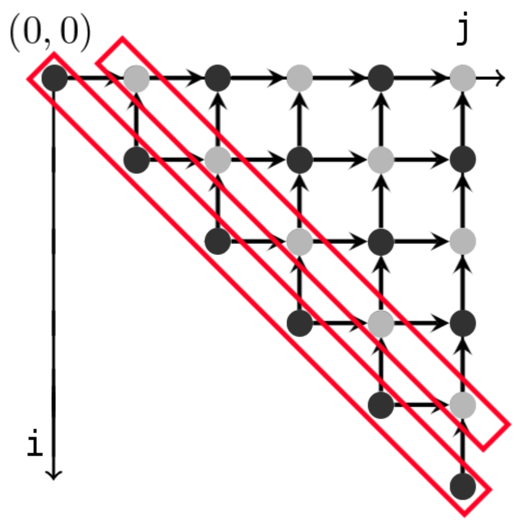

Figure 16: Graph dependency with the two diagonals highlighted

Let $diag_0$ and $diag_1$ be the two arrays of size `LEN_CARDS` (or n) storing the values highlighted in the figure. Since $diag_1$ is one entry smaller than $diag_2$, we only need to compute $n-1$ values. In general, for a diagonal of index j we must compute $n-j$ values.

To obtain $diag_j[k]$, we then need to know the values of $diag_{j-1}[k]$ and $diag_{j-1}[k+1]$. In practice, $diag_{j-1}[k]$ will be $cache[i][j-1]$, and $diag_{j-1}[k+1]$ will be $cache[i+1][j]$.

We will then only need to use two arrays of fixed size n to compute the final result, which knocks down our space complexity to $O(2n) = O(n)$.

Below is the implementation corresponding to this improvement (function `isSisterFirst` is as defined earlier) :

```
diags = [[0]*LEN_CARDS for _ in range(2)]

def updateCell(coord, k, sisterFirst, diag_i):
    i,j = coord
    print (k, diag_i)
    if (sisterFirst):
        # Sister takes the local optimum
        if (CARDS[i] > CARDS[j]):
            firstScore  = CARDS[i] + diags[diag_i][k+1][1]
            secondScore = diags[diag_i][k+1][0]
        else:
            firstScore  = CARDS[j] + diags[diag_i][k][1]
            secondScore = diags[diag_i][k][0]
    else:
        # Strategist takes the global optimum
        if (CARDS[i] + diags[diag_i][k+1][1] > CARDS[j] + diags[diag_i][k][1]):
            firstScore = CARDS[i] + diags[diag_i][k+1][1]
            secondScore = diags[diag_i][k+1][0]
        else:
            firstScore = CARDS[j] + diags[diag_i][k][1]
            secondScore = diags[diag_i][j-1][0]

    diags[1-diag_i][k] = (firstScore, secondScore)

def getScores(i, j, sisterFirst):
    if (i==j): return (CARDS[i], 0)

    # Initializing the first diagonal
    for k in range(LEN_CARDS):
        diags[0][k] = (CARDS[k], 0)

    # Computing cache values, starting from the diagonal
    diag_i = 0
    for diag_index in range(1, LEN_CARDS):
        sisterFirstDiag = isSisterFirst(diag_index, sisterFirst)

        # Updating all cells in the diagonal
        it_coord = [0, diag_index]
        for k in range(LEN_CARDS - diag_index):
            updateCell(it_coord, k, sisterFirstDiag, diag_i)
            it_coord = [x+1 for x in it_coord]

        # Flipping the cache entry to use for next diagonal
        diag_i = 1 - diag_i

    return diags[diag_i][0]
```

Here is a practical example of the resulting cache obtained for the example configuration described earlier in figure 2 :

(18, 13)	(17, 9)	(13, 9)	(16, 9)	(12, 9)	(9, 0)	(4, 0)
(25, 10)	(13, 17)	(17, 9)	(16, 12)	(9, 8)	(9, 4)	0

Table 4: Resulting entries for the diagonals optimized cache

In the first row, in coordinates [0,0] is the final score for both players. Since diagonals don't have to be cleared between computations, old values are simply overwritten and what we see in the cache are mostly values from diagonals computed in previous iterations.

5.2.4 Tabulation solution complexity

After replacing the recursive overheads in the memoized version, computing the time complexity gets easier. Indeed, the time-expensive computations are those we do while following the diagonal such as in figure 15. The first diagonal costs n computations, the second one $(n-1)$ and so on, leading to a total cost of $\sum_{k=1}^{n} \mathcal{O}(k) = \mathcal{O}(\frac{n(n+1)}{2})$: the complexity is thus quadratic in n. As mentioned earlier, the space complexity is in $\mathcal{O}(n)$.

5.3 General case

In the general case, the sister may not always pick the right-most card if she has to choose between two cards of equal cost. Going down the solution tree, if we get to a point for which we have $CARDS[i] = CARDS[j]$, and the sister is the one playing, we should be exploring both cases. However, in one case the optimal solution exploration might be losing or winning.

Thus, to model the best the problem, we should explore both options and compute probabilities of winning for each. As an example, if one subtree leads to a winning optimal solution, and the other leads to a losing one, assuming equiprobability of the sister's choice our program should return a 0.5 probability of winning, instead of a binary answer.

However, we did not have time to implement this version of the solution.

6 Conclusion

Our main goal in this APP is winning a card game regardless of which player starts. We first demonstrated how we could win using a greedy algorithm. However, as winning was not systematic even when we could, we extended our solution to cover all possible solution sets in order to be sure that we can get the best possible score, given the initial configuration of the cards. As this method was not efficient in terms of time complexity, dynamic programming was introduced as the optimum solution for winning (based on assumptions and conditions).

In general, the dynamic programming method is one of the few algorithmic techniques that can take problems for which the naive solutions require exponential time and produce polynomial time algorithms. In this APP, dynamic programming was used not only to solve the card game but also to lead us to an optimal game strategy.