Coins in a Line

February 9, 2011 by 1337c0d3r 75 Replies

> There are n coins in a line. (Assume n is even). Two players take turns to take a coin from one of the ends of the line until there are no more coins left. The player with the larger amount of money wins.
>
> 1. Would you rather go first or second? Does it matter?
> 2. Assume that you go first, describe an algorithm to compute the maximum amount of money you can win.

This is an interesting problem itself, and different solutions from multiple perspectives are provided in this post.

U.S. coins in various denominations in a line. Two players take turn to pick a coin from one of the ends until no more coins are left. Whoever with the larger amount of money wins.

Hints:
If you go first, is there a strategy you can follow which prevents you from losing? Try to consider how it matters when the number of coins are odd vs. even.

Solution for (1):
Going first will guarantee that you will not lose. By following the strategy below, you will always win the game (or get a possible tie).

- Count the sum of all coins that are odd-numbered. (Call this **X**)
- Count the sum of all coins that are even-numbered. (Call this **Y**)
- If **X** > **Y**, take the left-most coin first. Choose all odd-numbered coins in subsequent moves.
- If **X** < **Y**, take the right-most coin first. Choose all even-numbered coins in subsequent moves.
- If **X** == **Y**, you will guarantee to get a tie if you stick with taking only even-numbered/odd-numbered coins.

You might be wondering how you can always choose odd-numbered/even-numbered coins. Let me illustrate this using an example where you have 10 coins:

If you take the coin numbered 1 (the left-most coin), your opponent can only have the choice of taking coin numbered 2 or 10 (which are both even-numbered coins). On the other hand, if you choose to take the coin numbered 10 (the right-most coin), your opponent can only take coin numbered 1 or 9 (which are odd-numbered coins).

Notice that the total number of coins change from even to odd and vice-versa when player takes turn each time. Therefore, by going first and depending on the coin you choose, you are essentially forcing your opponent to take either only even-numbered or odd-numbered coins.

Now that you have found a non-losing strategy, could you compute the maximum amount of money you can win?

Hints:
One misconception is to think that the above non-losing strategy would generate the maximum amount of money as well. This is probably incorrect. Could you find a counter example? (You might need at least 6 coins to find a counter example).

Assume that you are finding the maximum amount of money in a certain range (ie, from coins numbered i to j, inclusive). Could you express it as a recursive formula? Find ways to make it as efficient as possible.

Solution for (2):
Although the simple strategy illustrated in **Solution (1)** guarantees you not to lose, it does not guarantee that it is optimal in any way.

Here, we use a good counter example to better see why this is so. Assume the coins are laid out as below:

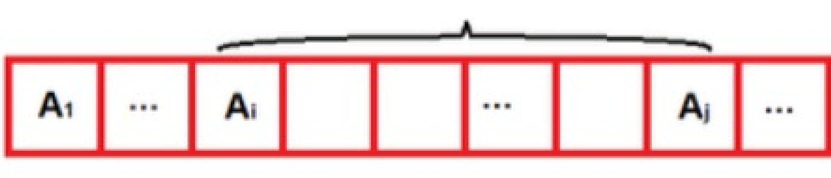
```
{ 3, 2, 2, 3, 1, 2 }
```

Following our previous non-losing strategy, we would count the sum of odd-numbered coins, **X** = 3 + 2 + 1 = **6**, and the sum of even-numbered coins, **Y** = 2 + 3 + 2 = **7**. As **Y** > **X**, we would take the last coin first and end up winning with the total amount of **7** by taking only even-numbered coins.

However, let us try another way by taking the first coin (valued at 3, denote by **(3)**) instead. The opponent is left with two possible choices, the left coin **(2)** and the right coin **(2)**, both valued at 2. No matter which coin the opponent chose, you can always take the other coin **(2)** next and the configuration of the coins becomes: **{ 2, 3, 1 }**. Now, the coin in the middle **(3)** would be yours to keep for sure. Therefore, you win the game by a total amount of 3 + 2 + 3 = **8**, which proves that the previous non-losing strategy is not necessarily optimal.

To solve this problem in an optimal way, we need to find efficient means in enumerating all possibilities. This is when Dynamic Programming (DP) kicks in and become so powerful that you start to feel magical.

First, we would need some observations to establish a recurrence relation, which is essential as our first step in solving DP problems.



The remaining coins are { A_i ... A_j } and it is your turn. Let P(i, j) denotes the maximum amount of money you can get. Should you choose A_i or A_j?

Assume that P(i, j) denotes the maximum amount of money you can win when the remaining coins are { A_i, …, A_j }, and it is your turn now. You have two choices, either take A_i or A_j. First, let us focus on the case where you take A_i, so that the remaining coins become { A_{i+1} … A_j }. Since the opponent is as smart as you, he must choose the best way that yields the maximum for him, where the maximum amount he can get is denoted by P(i+1, j).

Therefore, if you choose A_i, the maximum amount you can get is:

```
P1 = Sum{Ai ... Aj} - P(i+1, j)
```

Similarly, if you choose A_j, the maximum amount you can get is:

```
P2 = Sum{Ai ... Aj} - P(i, j-1)
```

Therefore,

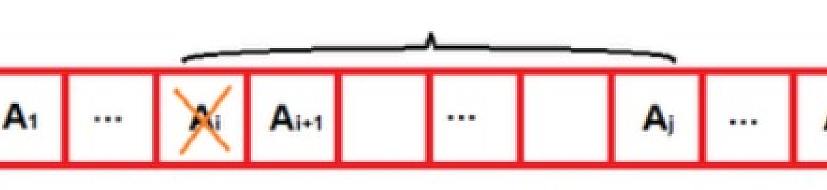
```
P(i, j) = max { P1, P2 }
        = max { Sum{Ai ... Aj} - P(i+1, j),
                Sum{Ai ... Aj} - P(i, j-1) }
```

In fact, we are able to simplify the above relation further to (Why?):

```
P(i, j) = Sum{Ai ... Aj} - min { P(i+1, j), P(i, j-1) }
```

Although the above recurrence relation is easy to understand, we need to compute the value of Sum{A_i … A_j} in each step, which is not very efficient. To avoid this problem, we can store values of Sum{A_i … A_j} in a table and avoid re-computations by computing in a certain order. Try to figure this out by yourself. (Hint: You would first compute P(1,1), P(2,2), … P(n, n) and work your way up).

A Better Solution:
There is another solution which does not rely on computing and storing results of Sum{A_i … A_j}, therefore is more efficient in terms of time and space. Let us rewind back to the case where you take A_i, and the remaining coins become { A_{i+1} … A_j }.



You took A_i from the coins { A_i … A_j }. The opponent will choose either A_{i+1} or A_j. Which one would he choose?

Let us look one extra step ahead this time by considering the two coins the opponent will possibly take, A_{i+1} and A_j. If the opponent takes A_{i+1}, the remaining coins are { A_{i+2} … A_j }, which our maximum is denoted by P(i+2, j). On the other hand, if the opponent takes A_j, our maximum is P(i+1, j-1). Since the opponent is as smart as you, he would have chosen the choice that yields the minimum amount to you.

Therefore, the maximum amount you can get when you choose A_i is:

```
P1 = Ai + min { P(i+2, j), P(i+1, j-1) }
```

Similarly, the maximum amount you can get when you choose A_j is:

```
P2 = Aj + min { P(i+1, j-1), P(i, j-2) }
```

Therefore,

```
P(i, j) = max { P1, P2 }
        = max { Ai + min { P(i+2, j),   P(i+1, j-1) },
                Aj + min { P(i+1, j-1), P(i,   j-2) } }
```

Although the above recurrence relation could be implemented in few lines of code, its complexity is exponential. The reason is that each recursive call branches into a total of four separate recursive calls, and it could be n levels deep from the very first call). Memoization provides an efficient way by avoiding re-computations using intermediate results stored in a table. Below is the code which runs in $O(n^2)$ time and takes $O(n^2)$ space.

Edit:
Updated code with a new function *printMoves* which prints out all the moves you and the opponent make (assuming both of you are taking the coins in an optimal way).

```
const int MAX_N = 100;

void printMoves(int P[][MAX_N], int A[], int N) {
  int sum1 = 0, sum2 = 0;
  int m = 0, n = N-1;
  bool myTurn = true;
  while (m <= n) {
    int P1 = P[m+1][n]; // If take A[m], opponent can get...
    int P2 = P[m][n-1]; // If take A[n]
    cout << (myTurn ? "I" : "You") << " take coin no. ";
    if (P1 <= P2) {
      cout << m+1 << " (" << A[m] << ")";
      m++;
    } else {
      cout << n+1 << " (" << A[n] << ")";
      n--;
    }
    cout << (myTurn ? ", " : ".\n");
    myTurn = !myTurn;
  }
  cout << "\nThe total amount of money (maximum) I get is " << P[0][N-1] << ".\n";
}

int maxMoney(int A[], int N) {
  int P[MAX_N][MAX_N] = {0};
  int a, b, c;
  for (int i = 0; i < N; i++) {
    for (int m = 0, n = i; n < N; m++, n++) {
      assert(m < N); assert(n < N);
      a = ((m+2 <= N-1)             ? P[m+2][n] : 0);
      b = ((m+1 <= N-1 && n-1 >= 0) ? P[m+1][n-1] : 0);
      c = ((n-2 >= 0)               ? P[m][n-2] : 0);
      P[m][n] = max(A[m] + min(a,b),
                    A[n] + min(b,c));
    }
  }
  printMoves(P, A, N);
  return P[0][N-1];
}
```

Further Thoughts:
Assume that your opponent is so dumb that you are able to manipulate him into choosing the coins you want him to choose. Now, what is the maximum possible amount of money you can win?