Introduction to Algorithms	**Solution Set 1**
CS 4820, Spring 2019	

(1) *(5 points) For each positive integer n, let t_n denote the number of distinct ways to cover a rectangular $2 \times n$ grid with non-overlapping dominoes. What is the value of t_n? Prove the correctness of your answer using mathematical induction.*

Hint: *You are allowed to use* `https://oeis.org/`. *This may help if you know how to calculate the elements of the sequence $t_1, t_2, t_3, \ldots$ but you don't know what terms to use to describe the sequence.*

Figure 1: $t_1 = 1$

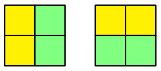

Figure 2: $t_2 = 2$

Figure 3: $t_3 = 3$

Solution. t_n is equal to F_{n+1} the $(n+1)^{\text{th}}$ element of the Fibonacci sequence $1, 1, 2, 3, 5, 8, 13, \ldots$ defined by $F_1 = F_2 = 1$ and $F_{n+2} = F_{n+1} + F_n$ for $n > 0$.

To prove this, we make use of the following lemma.

Lemma 1. *For $n \geq 1$, every domino tiling of a $2 \times (n+1)$ rectangle is either*

1. *a single vertical domino covering the first column, followed by a domino tiling of the remaining n columns; or*
2. *two horizontal dominoes covering the first two columns, followed by a domino tiling of the remaining $n-1$ columns.*

Proof. Consider any domino tiling of a $2 \times (n+1)$ rectangle. The top left corner must be covered by one of the dominoes. If it is a vertical domino, the structure of the tiling is as described in case 1. If it is a horizontal domino, then the lower left corner must be a covered by a different domino. Since the second domino does not overlap the first, it is also a horizontal domino, hence the structure of the tiling is as described in case 2. □

Corollary 1. *For $n \geq 1$ the equation $t_{n+1} = t_n + t_{n-1}$ holds.*

Now, to prove that $t_n = F_{n+1}$ for all $n \geq 1$, we use induction on n, with the induction hypothesis, "For all $n \geq 1$, $t_n = F_{n+1}$ and $t_{n+1} = F_{n+2}$." In the base case $n = 1$ this requires showing that $t_1 = 1$ and $t_2 = 2$. Figures 1 and 2 on the problem set establish these two facts. For the induction step, we assume the induction hypothesis for $n-1$ — that is, we assume $t_{n-1} = F_n$ and $t_n = F_{n+1}$ — and we must prove that $t_n = F_{n+1}$ and $t_{n+1} = F_{n+2}$. The equation $t_n = F_{n+1}$ is already assumed as part of the induction hypothesis, and the equation $t_{n+1} = F_{n+2}$ follows from Corollary 1 and the induction hypothesis, as follows:

$$t_{n+1} = t_n + t_{n-1} = F_{n+1} + F_n = F_{n+2}.$$

Remark: An alternative closed-form expression for t_n is

$$t_n = \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^{n+1} - \left(\frac{1-\sqrt{5}}{2} \right)^{n+1} \right].$$

Discussion and suggested hints. In writing the "reference solution" above, I chose to use a lemma and corollary before starting the inductive argument. That's just a matter of style; proofs don't need to name certain steps as lemmas and corollaries in order to be considered correct, but often identifying steps as lemmas or corollaries can aid the reader in understanding the logic of the proof.

The most natural way to write the proof (which is not the way it's written above) is to assert the induction hypothesis that $t_n = F_{n+1}$ and then cover *two base cases* $n = 1, n = 2$, before moving on to the induction step. A proof with the simpler induction hypothesis and the two base cases is perfectly valid, but most often the principle of mathematical induction is defined to have one base case, not two. For this reason (so as not to cause some students to question whether "induction" means something different from what they thought it meant) I chose to format the proof above in a slightly unnatural way, strengthening the induction hypothesis to assert $t_n = F_{n+1}$ *and* $t_{n+1} = F_{n+2}$, so that the argument would adhere to the strict only-one-base-case definition of induction.

(2) *(10 points) Suppose we are given an instance of the stable matching problem, consisting of a set of n applicants* $\{x_1, \ldots, x_n\}$ *and a set of n employers* $\{y_1, \ldots, y_n\}$*, together with a list for each entity (applicant or employer) that ranks the entities of the opposite type from best to worst. This exercise concerns algorithms to solve the following problem:* decide whether there exists a stable perfect matching in which x_n is matched to y_n.

(2a) *A simple algorithm for this problem is the following: remove* x_n *from every employer's preference list, and remove* y_n *from every applicant's preference list. Run the Gale-Shapley algorithm (say, with employers proposing) to find a stable perfect matching,* M*, of the applicant set* $\{x_1, \ldots, x_{n-1}\}$ *and employer set* $\{y_1, \ldots, y_{n-1}\}$*. If* $M \cup \{(x_n, y_n)\}$ *is a stable perfect matching of the original* $2n$ *entities (with their original unmodified preference lists) then answer "yes"; otherwise, answer "no". Give an explicit input instance on which this algorithm outputs the wrong answer.*

(2b) *Design a polynomial-time algorithm to decide whether there exists a stable perfect matching in which* x_n *is matched to* y_n*. Prove that your algorithm always outputs the correct answer and analyze its running time.*

Hint: *The solved exercises at the end of Chapter 1 in the textbook may provide a useful subroutine for your algorithm.*

Solution. **(2a)** Consider the following preference lists.

$$\begin{array}{llll} x_1: & y_2 > y_3 > y_1 & y_1: & x_1 > x_2 > x_3 \\ x_2: & y_1 > y_2 > y_3 & y_2: & x_2 > x_1 > x_3 \\ x_3: & y_3 > y_1 > y_2 & y_3: & x_1 > x_3 > x_2 \end{array}$$

If we run the algorithm proposed in part (2a), then $M = \{(y_1, x_1), (y_2, x_2)\}$ and $M \cup \{(y_3, x_3)\} = \{(y_1, x_1), (y_2, x_2), (y_3, x_3)\}$, which is not a stable matching because (y_3, x_1) is an unstable pair. However, there exists a stable matching in which x_3 and y_3 are matched, namely $\{(y_1, x_2), (y_2, x_1), (y_3, x_3)\}$.

(2b) To solve this problem we will reduce to the problem of computing a stable matching with forbidden pairs, Solved Exercise 2 in Chapter 1 of the Kleinberg-Tardos textbook.

The algorithm works as follows. Modify the preference lists of $x_1, \ldots, x_{n-1}$ and $y_1, \ldots, y_{n-1}$ by removing x_n and y_n, as in the first step of the incorrect algorithm in part (2a). Then define the following set, F, of *forbidden pairs*: a pair (y_i, x_j) belongs to F if one of the following properties holds.

- y_n prefers x_j to x_n and x_j prefers y_n to y_i; or
- x_n prefers y_i to y_n and y_i prefers x_n to x_j.

Run the modified Gale-Shapley algorithm on page 21 of the book with applicant set $\{x_1, \ldots, x_{n-1}\}$, employer set $\{y_1, \ldots, y_{n-1}\}$, and forbidden pair set F. Let M be the matching that the algorithm produces. If M is a perfect matching between $\{x_1, \ldots, x_{n-1}\}$ and $\{y_1, \ldots, y_{n-1}\}$, answer YES; otherwise answer NO.

First let's analyze the running time of this algorithm. To implement the algorithm efficiently it helps to start with a preprocessing step that constructs two $n \times n$ arrays, A and E, such that $A[i, j]$ is the ranking of y_j on x_i's preference list (1 if y_j is the highest-ranked, n if y_j is the lowest-ranked) and $E[i, j]$ is the ranking of x_i on y_j's preference list. Constructing these two arrays from the input data takes $O(n^2)$ time, and after this preprocessing we can answer preference queries (e.g., "Does y_i prefer x_j to x_k?") in constant time. Now, let's account for the running time to prepare the input to the modified Gale-Shapley algorithm. Deleting x_n and y_n from everyone else's preference lists takes $O(n^2)$. Finding the set F of forbidden pairs also takes $O(n^2)$, since testing whether a given pair is forbidden requires at most 4 preference queries, and we have explained how preference queries can be implemented in constant time. Finally, the modified Gale-Shapley algorithm runs in $O(n^2)$ time since it is exactly the same as the ordinary $O(n^2)$ Gale-Shapley algorithm except that an additional constant-time test for $(y, x) \notin F$ is performed before y proposes to x, and the proposal is only performed if this test passes.

For the proof of correctness, we must show two things.

1. *If the algorithm outputs* YES*, there exists a stable perfect matching in which x_n is matched to y_n.* In fact, if the algorithm outputs YES then the matching M computed by the modified Gale-Shapley algorithm is a perfect matching between $\{x_1, \ldots, x_{n-1}\}$ and $\{y_1, \ldots, y_{n-1}\}$, so $M' = M \cup \{(y_n, x_n)\}$ is a perfect matching between $\{x_1, \ldots, x_n\}$ and $\{y_1, \ldots, y_n\}$. There are no unstable pairs of the form (y_n, x_j) because if such an unstable pair existed, then letting (y_i, x_j) denote the pair in M that contains x_j we observe that (y_i, x_j) satisfies the first criterion defining a forbidden pair, and hence it would not be included in M. Similarly, there are no unstable pairs of the form (y_i, x_n) because if such an unstable pair existed it would imply that M contains a pair satisfying the second criterion defining a forbidden pair. Finally, we must consider pairs of the form (y_i, x_j) where $i, j < n$. There are two cases to consider. If $(y_i, x_j) \notin F$ then (y_i, x_j) cannot be unstable in M', because the partners of y_i and x_j in M' are the same as their partners in M, and we are assured that M has no unstable pairs. If $(y_i, x_j) \in F$ then assume y_i prefers x_n to x_j and x_n prefers y_i to y_n; the other case is handled symmetrically. The fact that x_n prefers y_i to y_n implies that for every x_k who ranks below x_n in y_i's preference list, $(y_i, x_k) \in F$. Consequently, letting x_k denote the applicant who is paired with y_i in M, it must be the case that y_i prefers x_k to x_n, who in turn is preferred to x_j. Since y_i prefers x_k to x_j, the pair (y_i, x_j) is not unstable in M'.

2. *If there exists a stable perfect matching in which x_n is matched to y_n, the algorithm outputs* YES. If M' is a stable perfect matching that contains (y_n, x_n), then the stability property of M' ensures that the matching $M = M' \setminus \{(x_n, y_n)\}$ is a stable perfect matching between $\{x_1, \ldots, x_{n-1}\}$ and $\{y_1, \ldots, y_{n-1}\}$, and furthermore, again due to the stability property of M', none of the elements of M belongs to the set F of forbidden pairs. To prove that our algorithm outputs YES we must show that the modified Gale-Shapley algorithm outputs a perfect matching, and to do so we will use a proof by contradiction. Suppose that the modified Gale-Shapley algorithm fails to output a perfect matching. Then there must exist some employer y_i who is not matched, and y_i must have proposed to every x_j such that $(y_i, x_j) \notin F$. In particular, letting x_j denote the applicant matched to y_i in M, we know that $(y_i, x_j) \notin F$ so y_i must have proposed to x_j. Thus, at the moment when the modified Gale-Shapley algorithm terminates, the following property holds:

   $(*)$ *There exists a pair $(y_i, x_j) \in M$ such that y_i has proposed to x_j but they are not currently matched.*

   Let T denote the first moment during the execution of the algorithm at which property $(*)$ holds. The reason why y_i is not matched to x_j at this moment is either:

(a) y_i just proposed to x_j, but x_j was already matched to y_k which is preferred to y_i; or

(b) y_i and x_j had been matched, but x_j received an offer from y_k which is preferred to y_i.

In both cases, y_k must have proposed to x_j. Now consider the applicant x_ℓ who is matched to y_k in M. Since M is stable and x_j prefers y_k to y_i, we know that y_k prefers x_ℓ to x_j. Since y_k proposes to applicants in order of preference, this means that y_k proposed to x_ℓ before x_j. Hence, before y_k proposed to x_j, it was already the case that y_k had earlier proposed to x_ℓ and was no longer matcherd to x_ℓ, implying that property $(*)$ was satisfied at a time *strictly earlier than* T, which contradicts our choice of time T. This is the desired contradiction; we may conclude that if there exists a stable perfect matching M' containing (y_n, x_n) then the modified Gale-Shapley algorithm will produce a perfect matching and our algorithm will output YES.

Remarks. First let's talk about the process of solving (2a). You have to ask yourself: if the algorithm is capable of producing an incorrect answer, is it because it produces a false positive or a false negative? The algorithm only claims there is a stable perfect matching containing (y_n, x_n) if it has actually found one, so there can be no false positives. Thus, we are trying to construct an example that triggers a false negative. What does that mean? There must be a stable perfect matching containing (y_n, x_n) but running the (2a) algorithm doesn't find it. In order for that to happen there must be at least two *distinct* stable perfect matchings between $\{x_1, \ldots, x_{n-1}\}$ and $\{y_1, \ldots, y_{n-1}\}$. This implies, first of all, that $n - 1 \geq 2$ — so don't go looking for counterexamples with $n = 2$! — and also that in order to solve (2a) we first need to understand how to construct a stable matching problem instance in which there are at least two distinct solutions. The book furnishes an example for $n = 2$ where both perfect matchings are stable. Use that as the basis for the counterexample. Then it's just a matter of inserting x_3, y_3 into people's preference lists so that the first perfect matching becomes unstable when we insert (y_3, x_3) into it, but the second one remains stable. This takes some trial and error.

For (2b), it's easiest to solve this problem after having solved (2a). Once you understand the potential for false negatives, you realize that what went on in the (2a) counterexample is that *there are some pairs that should never have been inserted into the matching between* $\{x_1, \ldots, x_{n-1}\}$ *and* $\{y_1, \ldots, y_{n-1}\}$*, because we could have anticipated in advance that they would lead to an unstable pair.* The hint already tells you to look at the solved exercises of Chapter 1, so you know that it's possible to solve stable matching with forbidden pairs. The counterexample to (2a) motivates the forbidden-pair criterion used in the solution to (2b). Then it's just a matter of proving that the (2b) algorithm is correct. This is actually a bit subtle: the issue is that although we know the constructed "stable matching with forbidden pairs" instance has a stable *perfect* matching, we need to prove that the modified Gale-Shapley algorithm will not output a stable matching that leaves some entities unmatched. Proving this property is not trivial. The proof given above is not the only way to go about it, but it is designed to imitate the proof in the book that the Gale-Shapley algorithm with men proposing matches each man with his best valid partner.

(3) *(10 points)* Here is one example of a block of code to insert into `Framework.java` to implement the Gale-Shapley algorithm.

```
// Compute employers' and applicants' rankings.
// Arank[i][k] is the position of employer k on applicant i's list.
// Erank[i][k] is the position of applicant k on employer i's list.
int Arank[][] = new int[n][n];
int Erank[][] = new int[n][n];
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
    Arank[i][APrefs[i][j]] = j;
    Erank[i][EPrefs[i][j]] = j;
```

```
  }
}



// Array to store how far each employer has gone down their proposal list
// Epos[i] = j means that i has proposed to the first j applicants on their list
int Epos[] = new int[n];

// Array to store the current match of each applicant.
// Amatch[j] is the number of the employer to whom applicant j is matched.
// Amatch[j] = -1 indicates applicant j is unmatched.
int Amatch[] = new int[n];

// Deque to hold employers who are still looking to fill their position.
Deque<Integer> q = new LinkedList<Integer>();

// Initially:
// -- every employer is unmatched
// -- every employer has proposed to zero applicants
// -- every applicant is unmatched
for (int i = 0; i < n; i++) {
  q.addFirst(i);
  Epos[i] = 0;
  Amatch[i] = -1;
}

while (q.peekFirst() != null) { // queue of unmatched employers is not empty
  int i = q.pollFirst(); // remove first employer from queue
  int j = EPrefs[i][Epos[i]]; // highest-ranked applicant to whom i has not proposed

  // i proposes to j
  Epos[i] = Epos[i] + 1;
  int k = Amatch[j];
  if (k == -1) { // if j is not currently matched
    Amatch[j] = i; // match to i
  }

  // else j is matched to some other employer, k
  else if (Arank[j][i] < Arank[j][k]) { // does j prefer i to k?
    Amatch[j] = i; // if so, replace j's match with i
    q.addFirst(k); // ... and k joins the queue of unmatched employers
  }
  else {
    q.addFirst(i); // if not, i returns to the queue of unmatched employers
  }
} // end of while loop; now every employer is matched

for (int i = 0; i < n; i++) { // add all matched pairs to MatchedPairsList
  MatchedPair pair = new MatchedPair(i,Amatch[i]);
```

```
  MatchedPairsList.add(pair);
}
```