Quiz 2

Cover Sheet

<table>
<thead>
<tr>
<th>Problem</th>
<th>Points</th>
<th>Grade</th>
<th>Initials</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>100</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Name: ___________________________________________

Circle your recitation:

R01  R02  R03  R04  R05  R06
F10  F11  F12  F1   F2   F3
Joe  Joe  Khanh Khanh Emily Emily

R07  R08  R09  R10
F11  F12  F1   F2
Matt Matt Geoff Geoff
This page intentionally left blank.
INSTRUCTIONS
This take-home quiz contains 5 problems worth a total of 100 points. Your quiz solutions are due at the beginning of class on Tuesday, November 23, 2010. Late quizzes will not be accepted unless you obtain a Dean’s support or make prior arrangements with the course staff. You must hand in your own quiz solutions in person.

Guide to this quiz: For problems that ask you to design an efficient algorithm for a certain problem, your goal is to find the most efficient algorithm possible. Generally, the faster your algorithm, the more points you receive. For two asymptotically equal bounds, worst-case bounds are better than expected or amortized bounds. The best possible solution will receive full points if well written, but ample partial credit will be given for correct solutions, especially if they are well written. Bonus points may be awarded for exceptionally efficient or elegant solutions.

Plan your time wisely. Do not overwork, and get enough sleep. Your very first step should be to write up the most obvious algorithm for every problem, even if it is exponential time, and then work on improving your solutions, writing up each improved algorithm as you obtain it. In this way, at all times, you have a complete quiz that you could hand in.

Policy on academic honesty: The rules for this take-home quiz are like those for an in-class quiz, except that you may take the quiz home with you. As during an in-class quiz, you may not communicate with any person except members of the 6.046 staff about any aspect of the quiz, even if you have already handed in your quiz solutions. In addition, you may not discuss any aspect of the quiz with anyone except the course staff until November 30, 2010. Note that this date is after the due date of the quiz.

This take-home quiz is “limited open book.” You may use your course notes, the CLRS textbook, and any of the materials posted on the course web page, but no other sources whatsoever may be consulted. For example, you may not use notes or solutions to problem sets, exams, etc. from other times that this course or other related courses have been taught. You may not use any materials on the World-Wide Web, including OCW. You probably won’t find information in these other sources that will help directly with these problems, but you may not use them regardless.

If at any time you feel that you may have violated this policy, it is imperative that you contact the course staff immediately. If you have any questions about what resources may or may not be used during the quiz, please send email to 6046-staff@csail.mit.edu.

Write-ups: Write your name and circle your recitation section on the cover sheet. Write your answers up on separate pieces of paper, and submit all your answers stapled together with the cover sheet of the quiz on top. Your name, your recitation, and the problem number should be at the top of every piece of paper in your solutions.

Your solutions are due, in hardcopy, at the start of lecture on Tuesday the 23rd of November. There is no electronic submission option for this test.

Your write-up for a problem should start with a topic paragraph that provides an executive summary of your solution. This executive summary should describe the problem you are solving, the techniques you use to solve it, any important assumptions you make, and the asymptotic bounds
on the running time your algorithm achieves, including whether they are worst-case, expected, or amortized.

Write your solutions cleanly and concisely to maximize the chance that we understand them. When describing an algorithm, give an English description of the main idea of the algorithm. Adopt suitable notation. Use pseudocode if necessary to clarify your solution. Give examples, draw figures, and state invariants. A long-winded description of an algorithm’s execution should not replace a succinct description of the algorithm itself.

Provide short and convincing arguments for the correctness of your solutions. Do not regurgitate material presented in class. Cite algorithms and theorems from CLRS, lecture, and recitation to simplify your solutions. Do not waste effort proving facts that can simply be cited.

Be explicit about running time and algorithms. For example, don’t just say that you sort $n$ numbers, state that you are using MERGE-SORT, which sorts the $n$ numbers in $O(n \lg n)$ time in the worst case. If the problem contains multiple variables, analyze your algorithm in terms of all the variables, to the extent possible.

Part of the goal of this quiz is to test your engineering common sense. If you find that a question is unclear or ambiguous, make reasonable assumptions in order to solve the problem, and state clearly in your write-up what assumptions you have made. Be careful what you assume, however, because you will receive little credit if you make a strong assumption that renders a problem trivial.

**Bugs:** If you think that you’ve found a bug, send email to 6046-staff@csail.mit.edu. Corrections and clarifications will be sent to the class via email and posted on the class website. Check your email and the class website daily to avoid missing potentially important announcements.

**Good Luck!**
Problem 1. Scheduling [20 points] (4 parts)
You are advising a high-school science team that is working on a large number of projects simultaneously. Your team has a set of \( n \) projects and a set of \( m \) physical tools. Each project specifies a set of tools that it needs, and can only be completed if all those tools are available for use on this project. Each tool might be required by many projects, but naturally can only be used for at most a single project. Your job is to distribute tools to projects so as to complete as many projects as possible. More precisely, given a set of projects, a set of tools, a description of the set of tools necessary for each project, and an integer \( k \), you would like to know if it is possible to distribute tools to projects so that at least \( k \) projects can be completed.

Consider the following variants and restrictions of the general problem described above. For each one state whether the variant problem is solvable in polynomial time or whether it is NP-hard. Justify your answer by either describing a polynomial-time algorithm or sketching a proof of the problem’s NP-hardness.

(a) [5 points] The problem when \( k \) is constant.

Solution: Since \( k \) is constant, we can try all possible combinations of \( k \) projects to complete. There are \( \binom{n}{k} = O(n^k) \) such combinations, and for each one we can verify in \( O(m) \) time whether any two of the \( k \) projects require the same tool (there are \( k \) sets, but a total of at most \( m \) required tools, otherwise we can trivially conclude that it is impossible). Hence we have an \( O(n^k m) \) time algorithm, which is polynomial in the case \( k \) is fixed.

(b) [5 points] The special case of the problem when there are two types of tools (say, power tools and computers), and each project requires at most one tool of each type (perhaps one project requires the power drill and the iPad, and another requires the circular saw and the Thinkpad).

Solution: This is essentially a bipartite matching problem. Let one set of vertices consist of tools of the first type, and the other set consist of tools of the second type. We will have an edge for each project connecting the two tools it needs of the two types, creating dummy vertices as needed for projects that only require one tool. Then \( k \) projects can be completed iff this graph has a matching of size \( k \). We have \( O(n) \) vertices on each side, with at most \( O(m) \) additional dummy vertices, and \( O(m) \) edges, so we can solve the distribution problem in \( O(m^2 + n) \) time.

(c) [5 points] The special case (of the original problem) when each tool is needed by at most two projects (but each project may need many tools).

Solution: We make the problem even more restrictive, by allowing each tool to only be requested by \textit{exactly} two projects, and show that even this is NP-hard. Now, consider
any undirected graph $G$ with $n$ vertices and $m$ edges. If we consider each vertex a project and each edge a tool connecting the two projects that requested it, then we have an instance of this restricted distribution problem. Moreover, $G$ has an independent set of size $k$ iff the resulting tools can be distributed to complete $k$ projects. This is therefore a polynomial time reduction from maximum independent set, which is NP-hard, so this special case of the distribution problem is NP-hard as well.

(d) [5 points] An $O(n/\log n)$-approximate optimization version of the problem in part (c): find a distribution of tools to complete at least $\text{OPT}/O(n/\log n)$ projects, where $\text{OPT}$ is the largest number of projects that can be completed.

Solution: Each tool that is requested by only one project can be given to that project ($O(m)$ time). For the remaining, we can follow the reduction in part (c) in reverse to construct a graph $G$, which has an independent set of size $k$ iff $k$ projects can be satisfied in our problem. This take $O(m + n)$ time. Now, if we complement $G$ (i.e., remove all edges that were there and add an edge wherever there wasn’t one) to make $G'$, then $G'$ has a clique of size $k$ iff $k$ projects can be satisfied. This step takes $O(n^2)$ time. Finally, let $\text{OPT}$ be the size of a maximum clique in $G'$, so that $\text{OPT}$ is also the largest number of projects that can be satisfied. Using the $O(n/\log n)$-approximation algorithm for maximum clique from lecture, which takes $O(n^2 \log n)$ time ($n/\log n$ blocks, $n$ different subsets in each, each of which subset requires us to check up to $O(\log^2 n)$ edges), we can find a clique of size at least $\text{OPT}/O(n/\log n)$ in $G'$, which translates directly to a distribution satisfying the same number of projects. The total running time is $O(n^2 \log n + m)$. 


Problem 2. Spy Games [20 points]

You are an enemy agent inside the borders of the country of Elbonia.

Your assignment is to make it as difficult as possible for the Elbonian rulers to send couriers on horseback from their capital city of Anar to its distant outpost of Chy along the country’s network of roads.

Each road $r$ in Elbonia connects two cities, and has an associated time $t_r$ (in minutes) needed to travel it by horseback. A courier will always take the fastest route from Anar to Chy. You have a budget of $B$ dollars to hire workers who can covertly degrade the quality of the roads at night (by digging potholes, etc.). Spending $y_r$ dollars on any road $r$ will increase the time needed to travel it by $y_r$ minutes. Given your complete knowledge of Elbonian roads (including their travel times) and your budget $B$, how can you efficiently calculate a way to allocate your money so that the fastest route from Anar to Chy will take as much time as possible?

Solution: We represent the map of Elbonian cities and roads as an undirected graph $G = (V, E)$, where vertices correspond to cities, and an edge $r = (u, v)$ corresponds to a road connecting cities $u$ and $v$. The weight of each edge $r$ will be $t_r + y_r$ (the original travel time plus the amount by which we degraded the quality of the road). We will assume that travel times and dollar amounts are real, non-negative numbers.

Our goal is to find a way to allocate our money (i.e., assign a $y_r$ to each road $r$) so that the length of the shortest path from Anar to Chy is maximized, subject to our budget constraint.

We can formulate this problem as the following linear program, similar to the linear program for single-pair shortest paths given in CLRS Section 29.2.

\[
\begin{align*}
\text{maximize} & \quad d_{Chy} \\
\text{subject to} & \quad d_v \leq d_u + t_r + y_r \quad \text{for each edge } r = (u, v) \in E \\
& \quad d_u \leq d_v + t_r + y_r \quad \text{for each edge } r = (u, v) \in E \\
& \quad d_{Anar} = 0 \\
& \quad y_r \geq 0 \quad \text{for each edge } r \in E \\
& \quad \sum_{r \in E} y_r \leq B
\end{align*}
\]

Correctness: For any particular choice of $y_r$ values, the length of the shortest path from Anar to Chy is the maximum value of $d_{Chy}$ subject to the first three constraints above (this is what the linear program for single-pair shortest paths computes). Our linear program maximizes the length of the shortest path from Anar to Chy over all possible choices of the $y_r$’s, subject to the constraints that the $y_r$’s are non-negative and add up to at most our budget $B$. 
Running Time: Our linear program has $V + E$ variables and $3E + 2$ constraints. Linear programming can be solved in (worst-case) polynomial time using interior-point methods or the ellipsoid algorithm.

Common Alternate Solutions and Errors: Many solutions used some type of iterative, greedy approach. A common approach was to repeatedly find the current shortest path from Anar to Chy, and degrade an edge on that path, where the edge is chosen according to some heuristic (e.g., the edge with the smallest weight, the edge incident on Chy, or the edge contained in the most paths from Anar to Chy). These heuristics were generally incorrect; see below for counterexamples. Also, these solutions generally did not specify how to handle ties (i.e., more than one shortest path, or more than one edge satisfying the heuristic).

Some iterative solutions involved degrading edges in a min-cut of the original graph, or degrading edges in a min-cut of the subgraph consisting of all shortest paths from Anar to Chy. These solutions were incorrect; see the third counterexample below.

Many solutions had a running time that was polynomial in $B$, not $\log(B)$ (the size of $B$), so the running time was only pseudopolynomial.

Some solutions involved enumerating all paths from Anar to Chy. However, this can take exponential time, since a graph can have an exponential number of paths (see second counterexample below).

Some solutions incorrectly stated that linear programming can be solved in polynomial time by the simplex algorithm. The running time of the simplex algorithm can be exponential in the worst case.

Figure 1: A counterexample for many heuristic approaches.

Figure 2: A graph with an exponential number of paths.
Figure 3: A counterexample for min-cut approaches. Suppose the budget $B = 2$. The optimal solution is to spend 1 on $AB$ and 1 on $DC$, increasing the length of the shortest path from 3 to 5.

Problem 3. Reductions [20 points]

Let $A$ be an NP-complete decision problem. Let $B$ be a decision problem in P. Let $C$ be a decision problem that is in NP, but may or may not be NP-complete.

Suppose we have a polynomial-time computable function $f$ that takes as input instances (either yes-instances or no-instances) of problem $A$. For each such instance $x$, the output of the function $f(x)$ will have one of two forms:

1. $f(x) = (y, 1)$
2. $f(x) = (z, 2)$.

We are guaranteed that if $f(x)$ is of type (1), then $x \in A$ iff $y \in B$, and if $f(x)$ is of type (2), then $x \in A$ iff $z \in C$, but there is no discernable pattern, rhyme or reason as to whether $f$ will produce an output of type (1) or type (2). Assuming $P \neq NP$, prove that $C$ is NP-complete. Explain where in your proof you used the assumption that $P \neq NP$.

Solution: Suppose first that $C$ is non-trivial; that is, there exist some input $z_+$ that is in $C$ and some input $z_-$ that is not in $C$. Consider the following algorithm $R$ that takes an instance $x$ of $A$ as input:

```plaintext
R(x)
1 Run $f(x)$
2 if $f(x) = (y, 1)$ and $y \in B$, return $z_+$
3 if $f(x) = (y, 1)$ and $y \notin B$, return $z_-$
4 if $f(x) = (z, 2)$, return $z$
```

First, we know that computing $f(x)$ takes polynomial time, as well as checking whether $y \in B$ (since $B$ is in P). Thus $R$ runs in polynomial time. Second, $x \in A$ iff $R(x) \in C$. It follows that $R$ is a polynomial time reduction from $A$ to $C$, and since $A$ is NP-hard, $C$ must be NP-hard as well. We already know $C$ is in NP, so it is in fact NP-complete. Note that we only needed to assume that $z_+$ and $z_-$ exist, without needing to find them: it suffices that the above $R$ exists; we do not need to actually be able to write it down!
Now, suppose, alternatively, that $C$ is trivial; that is, it either accepts every input or rejects every input. Then $f$ allows us to construct a polynomial algorithm for $A$: for input $x$, if $f(x) = (y, 1)$ (i.e., is of type (1)), we can check whether $y \in B$ in polynomial time and return that answer, while if $f(x) = (z, 2)$ (i.e., is of type (2)), then the answer to whether $z \in C$ is fixed so we can return that answer. Either way, we are able to answer whether $x \in A$ in polynomial time. But assuming $P \neq NP$, this contradicts the fact that $A$ is NP-complete.
Problem 4.  Train Sabotage [20 points] (2 parts)

You are a security consultant for a railroad company that manages one very long railroad line. There are \( n \) bridges on this line, spaced one mile apart. You are at bridge \( k_0 \) and have just received reliable intelligence that a terrorist organization has planted time bombs on all \( n \) bridges, scheduled to go off at times \((t_1, \ldots, t_n)\), given in minutes (with the current time being 0). Because of your military background, you can disable a bomb instantaneously (it takes no time). You have a train that travels at one mile per minute.

(a) [15 points] Give an efficient dynamic programming algorithm for finding whether there is a travel schedule to all of the bridges that allows you to disable all the bombs before any of them go off (to be precise, assume that if you arrive at a bridge the very instant it is due to go off, it blows up).

Solution: In order to design an efficient dynamic program for this problem, we will actually make it a little harder: instead of just determining if there is any way of disabling all the bombs in time, we will determine the shortest time in which we can do so, so that we can go home early for dinner (if it is not possible, this “shortest time” will be infinite).

A crucial observation is that if we are going to disable some bomb, and we pass another not-yet-disabled bomb along the way, we might as well disable it since it takes us 0 time to do so. So at any point in time during our travel, we will have a contiguous stretch of disabled bombs in the middle (this stretch will contain \( k_0 \), as well as wherever we happen to be), with all the not-yet-disabled bombs left and right of it.

As a consequence of this, whenever we disable a bomb, we know that we have to currently be either at the left or the right edge of the interval of disabled bombs—that is, since we never pass a live bomb without disabling it, if our disarmed bombs are in the interval \([a, b]\) then either \( a \) or \( b \) was the last bomb to be disabled.

We can therefore define our subproblems as follows:

\[
\text{MinTime}(a, b, \text{side}) \text{ is the shortest time in which we can disable all bombs from bridge } a \text{ to bridge } b \text{ inclusive (} 1 \leq a \leq k_0 \leq b \leq n\), and end up on the side denoted by side. Here side } \in \{ \text{left, right} \}, \text{ so if side } = \text{ left we end at } a \text{ and if side } = \text{ right we end at } b. \text{ (If the value is infinite, it means that no path to disable the bombs in that interval exists.)}
\]

We assume we disable the bomb at \( k_0 \) at time 0 (if \( t_{k_0} \leq 0 \), there is trivially no path to disable all the bombs in time). Thus, our base case is \( \text{MinTime}(k_0, k_0, \text{left}) = \text{MinTime}(k_0, k_0, \text{right}) = 0 \), since we are at both the left and right sides of the interval.

We can then build up larger intervals in a bottom-up fashion (alternatively, we could start with the top-level call and use recursion with memoization). As stated before, if we end at \( a \), then \( a \) was the last bomb to be disabled. Thus, we consider the subproblem on the interval \([a + 1, b]\). There are two possibilities: either we were on the left
side of that interval (i.e. at \(a + 1\)), in which case we are only one mile away, or we were at the right side of that interval (at \(b\)), in which case we must turn around and move \(b - a\) miles to get to \(a\). Recalling that we can move at one mile per minute, we wish to consider the faster of these times and check whether it allows us to get to \(a\) before the bomb at \(a\) explodes. Mathematically, this corresponds to

\[
\text{MIN\_TIME}(a, b, \text{left}) = \min(\text{MIN\_TIME}(a + 1, b, \text{left}) + 1, \text{MIN\_TIME}(a + 1, b, \text{right}) + (b - a)) \quad \text{if that value} < t_a, \\
\infty \quad \text{otherwise}
\]

Symmetrically,

\[
\text{MIN\_TIME}(a, b, \text{right}) = \min(\text{MIN\_TIME}(a, b - 1, \text{left}) + (b - a), \text{MIN\_TIME}(a, b - 1, \text{right}) + 1) \quad \text{if that value} < t_b, \\
\infty \quad \text{otherwise}
\]

If either \(\text{MIN\_TIME}(1, n, \text{left})\) or \(\text{MIN\_TIME}(1, n, \text{right})\) is finite, then there is a path that allows you to disable all bombs in time.

Each step of this algorithm only takes \(O(1)\) time given the solution to the smaller subproblems. The number of subproblems is the number of pairs \((a, b)\) times two (for the two directions). Since \(1 \leq a \leq k_0\) and \(k_0 \leq b \leq n\), the overall running time is \(O(2 \cdot k_0 \cdot (n - k_0 + 1)) = O(k_0(n - k_0)) = O(n^2)\).

**Common Alternate Solutions and Errors.** Many students attempted a greedy approach, such as ordering the bridges by detonation time or by difference between detonation time and distance from current position. The key point is that neither of these heuristics captures the situation where you need to move away from the “most urgent” bomb since going straight for it will mean you won’t reach another bomb in time. For example, consider the situation where there is a bomb 10 miles to the left that goes off in 15 minutes, and a bomb 1 mile to the right that goes off in 16 minutes. The only way to disarm both is to disarm the one on the right first. It is technically possible to combine this greedy approach with backtracking and memoize the results in order to solve the problem correctly, but this is more complicated and requires a precise description and analysis in order to ensure that it runs in polynomial time.

Many students also defined the recurrence as a decision problem, with the arguments as the left and right ends of the interval of defused bombs, the current position, and the current time. This can work if done correctly, but introduces a running-time dependence on the travel time. (A number of students correctly upper-bounded the travel time by \(O(n^2)\).)

Realizing that the defused bombs form a continuous interval is critical; without this observation, there are \(2^n\) possible subsets of defused bombs, which is too many subproblems for dynamic programming to handle efficiently.
(b) [5 points] Can your method from part (a) be easily modified to handle the variant where the bridges are separated by variable distances? If yes, give a sketch of the required modification. If no, explain why not.

**Solution:** Yes. The only place where the distance between bridges is used is to add to the travel time in the recursive call. If we denote the distance between bridges $x$ and $y$ as $d(x, y)$, we replace the $1$ and $b - a$ in the recursive call by the appropriate difference in distances, e.g.

$$
\text{MinTime}(a, b, \text{left}) = \min(\text{MinTime}(a + 1, b, \text{left}) + d(a, a + 1), \text{MinTime}(a + 1, b, \text{right}) + d(a, b)) \quad \text{if that value} < t_a,
$$

$$\infty \quad \text{otherwise}
$$

and similarly for the symmetric case.

If we are given the distances as absolute positions (i.e. bridge $x$ is at distance $d_x$ from the end of the line, we can calculate $d(a, b) = |d_b - d_a|$ in constant time for each pair $(a, b)$ we need, which does not add to the total running time.

If we are given the distances as relative positions (i.e. we are given $d(i, i+1)$ for all $i$), we can calculate absolute $d_x$ for all $x$ in linear time as a preprocessing step by simply taking a running total of the distances. Then we can proceed as above. This adds $O(n)$ preprocessing, so the total running time is still $O(k_0(n - k_0)) = O(n^2)$. 
Problem 5. Attacking Rooks/Knights [20 points]

(a) [10 points] You are given an $n \times n$ chessboard containing $k$ rooks, where there is at most one rook per square. Give an efficient algorithm to find the largest possible subset of the $k$ rooks such that no rook attacks any other rook. (A rook is said to attack another rook if the two rooks are in the same row or in the same column.)

Solution: We will reduce this problem to the maximum bipartite matching problem. Let $R$ denote the set of $n$ rows and $C$ denote the set of $n$ columns of the board. $R$ will be our set of left-vertices, and $C$ our set of right-vertices. A row $r \in R$ is connected to a column $c \in C$ by an edge if there is a rook at their intersecting square. This gives us a bipartite graph $G$. Observe that a subset of non-attacking rooks corresponds precisely to a matching in $G$, since two rooks can be included iff they don’t share a row or column, and two edges can be included in a matching iff they don’t share a vertex (which corresponds to a row or column in our setting). So our task of finding the largest such subset of rooks reduces to finding a maximum matching in $G$, which we know how to do in $O(\sqrt{nk})$ time (using Hopcroft-Karp).

To analyze the overall running time, we need to be a bit careful about how we read the input, and how we construct the graph $G$. Suppose, first, that the input (i.e., the chessboard with its rook configuration) is given to us as an $n \times n$ binary matrix with a 1 in positions where there is a rook. In this case, we need to read the whole matrix before we can know where all the rooks are, so just reading the input already takes us $O(n^2)$ time. Then, to construct $G$, we create the list of vertices in $O(n)$ time, and go through every row/column pair to create the list of edges in $O(n^2)$ time. This gives us an overall running time of $O(n^2 + \sqrt{nk})$. Alternatively, suppose the input is given to us as a list of coordinates where the rooks are located. In this case, reading the input only takes us $O(k)$ time, while constructing $G$ takes us $O(n)$ time to create the list of vertices and $O(k)$ time to reformat the list of rooks into a list of edges. This gives us an overall running time of $O(n + \sqrt{nk})$.

Remark: Several students proposed the heuristic of repeatedly removing a rook that attacks the largest number of other rooks, until the rooks remaining on the board are non-attacking. This, however, does not work. Consider the following counterexample:

![Counterexample](image.png)

while the optimal solution consists of all three rooks along the diagonal. Another similar heuristic suggested is to repeatedly include a rook attacking fewest other rooks and remove those that it attacks. This also does not work, as seen in the following
somewhat larger example (where underlined R’s are included):

while the optimal solution consists of all six rooks along the diagonal. Note that this is also a counterexample to other greedy rules such as repeatedly including a rook that has the smallest product of the number of other rooks in its row and the number of rooks in its column.

(b) [10 points] Replace rooks with knights in the above problem. (A knight is said to attack another knight if their row numbers differ by 1 and their column numbers differ by 2, or if their row numbers differ by 2 and their column numbers differ by 1.) You may use the following theorem:

**Theorem 1 (König’s Theorem)** In any bipartite graph \( G = (V, E) \), the size of a maximum matching in \( G \) is equal to the size of a minimum vertex cover of \( G \). Moreover, given a maximum matching, we can find a minimum vertex cover in \( O(V + E) \) time.

**Solution:** At a high level, we will reduce this problem to the maximum bipartite independent set problem, which we in turn reduce to minimum bipartite vertex cover, which, finally, we reduce to maximum bipartite matching (via König’s Theorem).

First, we reduce the non-attacking knights problem to the maximum bipartite independent set problem. To begin, observe that knights on black squares can only attack knights on white squares, and vice versa. We therefore let \( B \) be the set of knights on black squares, and \( W \) the set of knights on white squares. Letting \( B \) and \( W \) be our sets of left- and right-vertices, respectively, we connect a knight \( b \in B \) with a knight \( w \in W \) by an edge if they attack each other (in the case of rooks, this would have been the arguably more natural way of interpreting the chessboard as a graph; unfortunately, there it did not work because the rooks did not divide themselves naturally into two sets to make a bipartite graph). In the resulting graph \( G \) (which has \( k \) vertices...
and at most $4k$ edges since each knight can attack at most 8 others, an independent set corresponds precisely to a set of non-attacking knights. Therefore it suffices to find the maximum independent set in $G$.

Second, we already proved in lecture that maximum independent set reduces to minimum vertex cover on the same graph (this is fortunate, otherwise we might lose the fact that $G$ is bipartite). We simply need to complement the set of vertices (specifically, if we find a minimum vertex cover $S$ of $G$, then $B \cup W - S$ is a maximum independent set of $G$), which takes $O(k)$ time. So it remains to find a minimum vertex cover of $G$.

Third, and last, we can do this by first finding a maximum bipartite matching of $G$ in $O(k^{3/2})$ time, again using Hopcroft-Karp, and then invoking Kőnig’s Theorem to convert it into a minimum vertex cover in $O(k)$ time.

The second and third steps take a total of $O(k^{3/2})$ time. To analyze the running time of the first step, we again have to be careful about the format in which the input is given to us, and how we construct $G$. As in part (a), if the input is given to us as a binary matrix with a 1 in positions where there is a knight, then we cannot avoid an $O(n^2)$ overhead in reading the input. To construct $G$, we take $O(k)$ time to create the list of vertices, and $O(k)$ time to create the list of edges by checking, for each knight, the up to 8 positions it can attack to see if there are other knights there. Note that we should not simply consider all pairs of knights, since this will take $\Theta(k^2)$ time, which is up to $\Theta(n^3)$ when $k = \Theta(n^2)$. Using this trick, we obtain an overall running time of $O(n^2 + k^{3/2})$.

Alternatively, if the input is given to us as a list of coordinates where the knights are located, it only takes us $O(k)$ time to read the input, but constructing $G$ is trickier. Naïvely, we can create the list of vertices in $O(k)$ time, and go through all pairs of knights to construct the list of edges in $O(k^2)$ time. Note that we can no longer check, for each knight, the 8 positions it can attack, since we have no way to see if there is a knight in any given position other than to go through the entire list of coordinates. But when $k > n$ this would be even worse than the binary matrix case (which we can always convert to by creating that binary matrix ourselves from our list of coordinates). Luckily, we can do slightly better: sort our list of coordinates (by any ordering rule we like); then, for each knight, do 8 binary searches in our sorted list of coordinates to see if each of the 8 attackable positions contains another knight. Both steps take only $O(k \log k)$ time, so our overall running time is dominated by the previous $O(k^{3/2})$.

**Remark:** The “repeatedly-remove-most-offensive-rook/knight” heuristic above was also proposed for this problem, and it also fails here. Consider the following counterexample:

while the optimal solution consists of all seven knights in the middle three rows. The “repeatedly-include-least-offensive-rook/knight” heuristic also does not work, but a counterexample will take up too much space so we leave it as an exercise.