# CS 170 HW 8

## Due on 2018-10-14, at 9:59 pm

## 1 Study Group

List the names and SIDs of the members in your study group.

## 2 A Dice Game

Consider the following 2-player game played with a 6-sided die. On your turn, you can decide either to roll the die or to pass. If you roll the die and get a 1, your turn immediately ends and you get 1 point. If you instead get some other number, it gets added to a running total and your turn continues (i.e. you can again decide whether to roll or pass). If you pass, then you get either 1 point or the running total number of points, whichever is larger, and it becomes your opponent's turn. For example, if you roll 3, 4, 1 you get only 1 point, but if you roll 3, 4, 2 and then decide to pass you get 9 points. The first player to get to $N$ points wins, for some positive $N$.

Alice and Bob are playing the above game. Let $W(x, y, z)$ be the probability that Alice wins given that it is currently Alice's turn, Alice's score (in the bank) is $x$, Bob's score is $y$ and Alice's running total is $z$.

(a) Give a recursive formula for the winning probability $W(x, y, z)$.

(b) Based on the recursive formula you gave in the previous part, design an $O(N^3)$ dynamic programming algorithm to compute $W(x, y, z)$. Briefly describe your algorithm, prove its correctness and runtime.

**Solution:**

(a) *Hint if students struggle:* Work out the probabilities $R$ and $P$ that the current player will win if they decide to roll and pass respectively.

If the current player rolls and gets a 1, they'll win with probability $1 - W(y, x + 1, 0)$. If they roll and get a different value $v$, then $v$ gets added to the running total and it's still their turn, so they'll win with probability $W(x, y, z + v)$. Since each value of the die has probability $1/6$, this means

$$R = \frac{1}{6}\left(1 - W(y, x + 1, 0)\right) + \frac{1}{6}\sum_{v=2}^{6} W(x, y, z + v).$$

If instead they pass, they get $\max(1, z)$ points and it becomes their opponent's turn. So we have

$$P = 1 - W(y, x + \max(1, z), 0).$$

Finally, $W(x, y, z) = \max(R, P)$, and substituting the expressions above gives a recursive formula for $W$.

(b) We convert the recursive formula above into a recursive algorithm. The base cases are as indicated in part (1): when a player has a large enough running total to win immediately by passing. Note that in the recursive formula, every recursive term either increases the number of points one of the players has, or increases the running total. So the recursion is guaranteed to terminate. Using memoization to keep track of the values of $W$ we have already computed gives a dynamic programming algorithm.

To work out the runtime of this algorithm when you need $N$ points to win, notice that we never need to compute $W(x, y, z)$ where any of $x$, $y$, or $z$ is $N + 6$ or larger. This is because if either player had that many points they would have already won on the previous turn, and if $z$ was that large the current player could have won before the last die roll by passing. So the algorithm will compute $W$ for at most $(N+6)^3$ different game positions, and therefore its runtime is $O(N^3)$.

## 3 Knightmare

Give an algorithm to find the number of ways you can place knights on an $N$ by $M$ ($M < N$) chessboard such that no two knights can attack each other (there can be any number of knights on the board, including zero knights). Clearly describe your algorithm and prove its correctness. The runtime should be $O(2^{3M} M \cdot N)$.

**Solution:**

We use length $M$ bit strings to represent the configuration of rows of the chessboard (1 means there is knight and otherwise 0).

The main idea of the algorithm is as follows: we solve the subproblem of the number of valid configurations of $(n-1) \times M$ chessboard and use it to solve the $n \times M$ case. Note that as we iteratively incrementing $n$, a knight in the $n$-th rows can only affect configurations of rows $n+1$ and $n+2$. So we can denote $K(n, u, v)$ as the number of possible configurations of the first $n$ rows with $u$ being the $(n-1)$-th row and $v$ being the $n$-th row, and then use dynamic programming to solve this problem. Note that we can precompute and memorize all the valid configurations of three consecutive rows (i.e. all the valid configurations of the $3 \times M$ table).

**Pseudocode:** We say bit strings $u, v$ is valid no two knights can attack each other in the $2 \times M$ table represented by $u$ and $v$. Similarly we say bit strings $u, v, w$ is valid no two knights can attack each other in the $3 \times M$ table represented by $u, v$ and $w$.

Initialize $K(\cdot, \cdot, \cdot) := 0$
**for** all size $M$ bitstrings $v, w$ **do**
  Initialize $K(2, v, w) := 1$ if $v, w$ is valid else 0
**for** $n = 3$ to $N$ **do**
  **for** all size $M$ bitstrings $u, v, w$ if $u, v, w$ is valid **do**
    $K(n, v, w)$ += $K(n-1, u, v)$
  **return** $\sum_{v,w} K(N, v, w)$

**Proof of Correctness:** By definition, $K(2, v, w) = 1$ if the $M$ by 2 chessboard configuration defined by $v$ and $w$ is legitimate. Otherwise, $K(2, v, w) = 0$.

For $n > 2$, we have

$$K(n, v, w) = \sum_u K(n-1, u, v),$$

where we are summing over all possible configurations $u$ for the third-last row of a chessboard whose last rows are specified by $v$ and $w$.

To bound the total runtime, first note that for each row, we iterate over all possible configurations of knights in the row and for each such configuration, we perform a sum over all possible valid configurations of the previous two rows. The time to check if a configuration is valid is $O(M)$. Therefore, the time taken to compute the sub-problems for a single row is $O(2^{3M}M)$ which gives us an overall runtime of $O(2^{3M}MN)$.

## 4 Triangulating a polygon

You are given a convex polygon, $P$, of $n$ vertices, $(x_1, y_1), \ldots, (x_n, y_n)$. A triangulation of $P$ is a collection of $n-3$ diagonals of $P$ such that no two diagonals intersect inside the polygon. A triangulation splits the polygon's interior into $n-2$ disjoint triangles. The cost of a triangulation is defined to be the sum of the lengths of the diagonals forming the triangulation. Your task is to devise a dynamic programming algorithm to compute the minimum cost triangulation of a given polygon. Please provide a three part solution describing your algorithm, a proof of correctness and a runtime analysis. (*Hint*: First order the points $(x_1, y_1), \ldots, (x_n, y_n)$ in a clock-wise manner and index each sub-problem with a pair of indices $1 \leq i < j \leq n$).

**Solution:** We will denote the points as $p_1 = (x_1, y_1), \ldots, p_n = (x_n, y_n)$ and assume that the points are arranged in a clock-wise manner. Furthermore, we will denote the distance between $p_i$ and $p_j$ by $d(p_i, p_j)$.

**Main Idea:**

We will index our sub-problems, $A(i, j)$, with pairs of indices $i, j$ such that $1 \leq i < j \leq n$. The subproblem, $A(i, j)$, will denote the minimum cost triangulation of the polygon formed by the vertices $(x_i, y_i) \ldots (x_j, y_j)$. Our recursive call will be as follows:

$$A(i,j) = \begin{cases} \min(\min_{k \in \{i+2, \ldots, j-2\}}(A(i,k) + A(k,j) + d(p_i, p_k) + d(p_k, p_j)), \\ \qquad d(p_j, p_{i+1}) + A(i+1, j), d(p_i, p_{j-1}) + A(i, j-1)), & \text{if } j - i > 2 \\ 0, & \text{otherwise} \end{cases}$$

At the end of the algorithm, we will output the value $A(1, n)$.

**Runtime Analysis:**

Notice, that the cost to compute each sub-problem is $O(n)$ and each sub-problem needs to be computed exactly once. This means we can bound the total runtime by $O(n^3)$ as there are $\binom{n}{2}$ choices for $i$ and $j$.

**Proof of Correctness:**

We will prove the correctness of our algorithm by induction on the value of $j - i$.

Base Case: $j - i \leq 2$ In the base case, we have three or fewer vertices in our polygon. This implies our polygon is already triangulated.

Inductive Step: Suppose now that we wish to find a triangulation of the polygon, $P_{ij}$, formed by the vertices $p_i, \ldots, p_j$. Now consider the edge $(j, i)$. In any triangulation of $P_{ij}$, the edge $(j, i)$ has to be part of some triangle. The third vertex of the triangle is one of the vertices, $p_{i+1}, \ldots, p_{j-1}$. The cases where the third vertex of the triangle is one of the vertices $p_{i+1}$ or $p_{j-1}$ is covered by the last two terms in the recursive formula as in both cases we add either the edge $(j, i+1)$ or $(i, j-1)$. For all the other possibilities (where $k \in \{i+2, \ldots, j-2\}$), we add both edges $(i, k)$ and $(j, k)$ which is covered by the first term of the recursion. In particular, for the optimal triangulation of $P_{ij}$, the edge $(j, i)$ is part of some triangle, say $i, j, k^*$. We see that this triangulation consists of a triangulation of $P_{ik^*}$ and $P_{k^*j}$ along with the edges $(i, k^*)$ and $(k^*, j)$. We see that the cost computed by our algorithm is at most the cost of this triangulation. The correctness of our algorithm is evident from the fact that we may construct a triangulation from the choice of $k$ minimizing the recursive formula.

## 5 Three Partition

Given a list of positive numbers, $a_1, \ldots, a_n$, can we partition $\{1, \ldots, n\}$ into 3 disjoint subsets, $I, J, K$ such that:

$$\sum_{i \in I} a_i = \sum_{j \in J} a_j = \sum_{k \in K} a_k = \frac{\sum_{i=1}^{n} a_i}{3}$$

Devise and analyze a dynamic programming solution to the above problem that runs in time polynomial in $\sum_{i=1}^{n} a_i$ and $n$.

**Solution: Main Idea:**

Our algorithm will consist of sub-problems indexed by $i, j, k$, $A(i, j, k)$, which denotes whether it is possible to find two disjoint subsets $I_k$ and $J_k$ of $\{1, \ldots, k\}$ such that:

$$\sum_{m \in I_k} a_m = i, \qquad \sum_{l \in J_k} a_l = j$$

Therefore, the solution to each sub-problem is either True or False. Using $w$ to denote $\sum_{i=1}^{n} a_i$, we may assume that $i, j$ are in the range from $0 - w$ and that $k$ ranges from 0 to $n$. To compute the value of $A(i, j, k)$ using the following recursion:

$$A(i, j, 0) = \begin{cases} True, & \text{if } i = j = 0 \\ False, & \text{otherwise} \end{cases} \qquad A(i, j, k) = A(i - a_k, j, k-1) \vee A(i, j - a_k, k-1) \vee A(i, j, k-1)$$

Our Dynamic Programming algorithm now starts with the base case where $k = 0$ and computes the value for $A(i, j, 0)$ for all combinations of $i$ and $j$. It then recursively computes the values of $A(i, j, k)$ for all combinations of $i$ and $j$ by using the previously computed values for $A(i, j, k-1)$. Our algorithm finally returns $A(w/3, w/3, n)$.

**Runtime Analysis:**

Notice that each subproblem takes constant time to solve given solutions all the previous subproblems. The runtime of our algorithm is bounded by $w^2 n$.

**Proof of Correctness:**

For the case, $k = 0$, our algorithm trivially computes the right answers to the subproblems. Now, assuming the values $A(i, j, k)$ are computed correctly for all combinations of $i$, $j$ and $k$ up to $l-1$. Suppose now, we wish to compute the value of $A(i, j, l)$. Notice that if the algorithm computes True, there definitely exists a positive solution to the subproblem. If $A(i - a_l, j, l - 1)$ is True, our solution is simply $\{l\} \cup I_{l-1}, J_{l-1}$ where $I_{l-1}, J_{l-1}$ are solutions to $A(i - a_l, j, k - 1)$. Similarly, $A(i, j - a_l, l - 1)$ correspond to adding $l$ to $J_{l-1}$ and not including it in either respectively. On the other hand, if a combination, $I_l, J_l$ exists, then either $l \in I_l$ or $l \in J_l$ or $l \notin I_l, J_l$. In either case, $A(i, j, l)$ evaluates to True as one of $A(i - a_l, j, l - 1), A(i, j - a_l, l - 1), A(i, j, l - 1)$ have to be True. This proves the correctness of the algorithm.

## 6 2-SAT

Please provide solutions to parts (d), (e) and (f) of Question 3.28 from
`http://algorithmics.lsi.upc.edu/docs/Dasgupta-Papadimitriou-Vazirani.pdf`.

**Solution:** Let $\varphi$ be a formula acting on $n$ literals $x_1, \ldots, x_n$. Construct a graph with $2n$ vertices representing the set of literals and their negations. For each clause $(a \vee b)$ of $\varphi$ add the edges $\neg a \Rightarrow b$ and $\neg b \Rightarrow a$. Use the strongly connected components algorithm and for each $i$, check if there is a SCC containing both $x_i$ and $\neg x_i$. If any such component is found, report unsatisfiable. Otherwise, report satisfiable.

In the case that the algorithm reports unsatisfiable, notice that the edges of the graph are necessary implications. Thus, if some $x_i$ and $\neg x_i$ are in the same component, there is a chain of implications which is equivalent to $x_i \to \neg x_i$ and a different chain which is equivalent to $\neg x_i \to x_i$, i.e. there is a contradiction in the set of clauses.

In the case that the algorithm reports satisfiable, there is a simple satisfying assignment: Take any sink component, and assign variables so all the literals in this component are True. Because of how we define the graph, there is a corresponding source component which has the negations of all literals in this component. Remove this source/sink component pair, and repeat the process until the graph is empty. Since we set components to true in reverse topological order, there is no implication from a true literal to a false literal. Since no literal and its negation are in the same SCC, we never try to set a variable to be both true and false. So this produces an assignment satisfying all clauses.

(Note: A common mistake is to report unsatisfiable if there is a path from $x_i$ to $\neg x_i$ in this graph, even if there is no path from $\neg x_i$ to $x_i$. Even if there is a series of implications which combined give $x_i \to \neg x_i$, unless we also know $\neg x_i \to x_i$ we could set $x_i$ to False and still possibly satisfy the clauses. For example, consider the 2-SAT formula $(\neg a \vee b) \wedge (\neg a \vee \neg b)$. These clauses are equivalent to $a \to b, b \to \neg a$, which implies $a \to \neg a$, but this 2-SAT formula is still easily satisfiable.)