CS 170 | Efficient Algorithms and Intractable Problems
Spring 2017 | Prasad Raghavendra and Sanjam Garg

Homework 6

Instructions: You are welcome to form small groups (up to 4 people total) to work through the homework, but you **must** write up all solutions by yourself. List your study partners for homework on the first page, or "none" if you had no partners.

If using LaTeX (which we recommend), you may use the homework template linked on this Piazza post to get started.

Begin each problem on a new page. Clearly label where each problem and subproblem begin. The problems must be submitted in order (all of P1 must be before P2, etc). For questions asking you to give an algorithm, respond in what we will refer to as the *four-part format* for algorithms: main idea, pseudocode, proof of correctness, and running time analysis.

Read the Homework FAQ Piazza post on Piazza before doing the homework for more explanation on the four-part format and other clarifications for our homework expectations.

No late homeworks will be accepted. No exceptions. This is not out of a desire to be harsh, but rather out of fairness to all students in this large course. Out of a total of approximately 12 homework assignments, the lowest two scores will be dropped.

Special Questions:

- *Shortcut questions*: Short questions are usually easy questions that give you opportunities to practice basic materials. However, we understand that some of you are very familiar with the topics and do not want to spend too much time on easy questions. Therefore, we design shortcut questions for this purpose. A shortcut question usually has multiple parts that build upon each other and are ordered by their difficulty level. You can work on those in order or start from wherever you like. However you only need to submit the last part you are able to solve. For example, if a question has 5 parts (a, b, c, d, e), you are confident about part e, you should submit part e without any of the previous four parts. If you are confident about d but not sure about e, you should submit d for grading purposes. Please clearly indicate in your submission which part you are submitting.
- *Redemption questions*: It is important that you carefully read the posted solutions, even for problems you got right. To encourage this, you have the option of submitting a redemption file, a few paragraphs in which you explain, for each problem you choose to cover, what you did wrong and what the right idea was in your own words (not cutting and pasting from the solution!), and appending it to your homework. For example, suppose that as you review your solutions to HW1, you realize you had misunderstood question 3 and answered it incorrectly. You would write down what you just learned, and then submit it in your HW2 assignment the following week. Because these are mainly for your benefit, feel free to format them however is most useful for you.
- *Extra credit questions*: We might have some extra credit questions in the homework for people who really enjoy the materials. However, please note that you should do the extra credit problems only if you really enjoy working on these problems and want an extra challenge. It is likely not the most efficient manner in which to maximize your score.

Due Tuesday, March 14, at 11:59am

This homework emphasizes dynamic programming problems. In your solutions, note the following:

- When you give the main idea for a dynamic programming solution, you need to explicitly write out a recurrence relation and explain its interpretation.
- When you give the pseudocode for a dynamic programming solution, it is not sufficient to simply state "solve using memoization" or the like; your pseudocode should explain how results are stored.

1. (★★★★★ level) Hacking for Justice (shortcut question)

In an alternate universe, the students of CS170 found a certain problem on HW6 to be extremely difficult. Initially, no one was able to find the solution. However, some subset of the students managed to download the solution PDF to their laptops. These students began to send the PDF to others via email, who then sent the PDF to others, and so on. Eventually, all of the students had the solution PDF. Uh oh!

After much effort, a TA has figured out the full history of the solution PDF sharing, and constructed a directed acyclic graph $G = (V,E)$ to represent it. V represents the students, and an edge E from v_1 to v_2 represents that v_1 sent the solution file to v_2. All sharing is one-way, and you know that there are no cycles.

If the TA could go back in time, and completely block off all communications to/from one student's laptop, which student should be blocked to minimize the number of students who received the PDF? Assume that blocking one person will not cause anyone else to share with more people than they did before. Answer this question in part (c), or try (a) and (b) for inspiration.

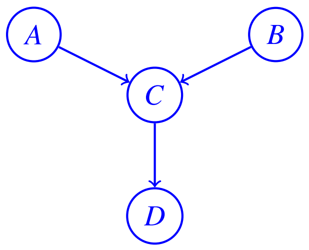
One more important detail: the TA notices that for any two different paths from A to B, the two paths differ by at most k vertices, where k is a small number. You can assume k is about equal to $\log(|V|+|E|)$.

A useful bound: the TA also notices that if you take a depth-k BFS search starting at any vertex, you will traverse $O(k)$ vertices in the search, and the BFS search runs in $O(k)$ time. Using this property will make the problem easier.

Solution: This problem was broken. The k detail was introduced in an attempt to both fix the problem and not drastically increase the difficulty, but was done incorrectly. The following solution is a rough draft, and staff attention is likely more productive elsewhere.

(a) **(10% credit)** Is it always the case that the best student to block is one of the original students who first downloaded the PDF? If so, explain why in 2-3 sentences. If not, give a small counterexample.

Solution: No, it's not always the case. Consider the following case:



Blocking off one of the source nodes A or B will block 1 student from having the PDF. Blocking off C will result in 2 students losing PDF. So in this case, the best student to block is C, which is not one of the original source nodes.

(b) **(50% credit)** Write a recursive definition for $F(s)$, the number of students that won't have the PDF if you choose to block s.

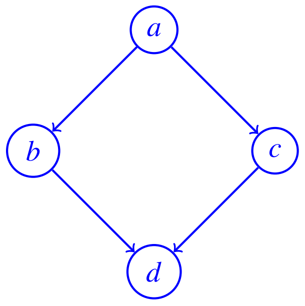
Solution:

The originally initially intended solution was a simple recurrence relation that depended on only the immediate children of s whose indegree was 1, but this is incorrect:

Let $\{c_i\}$ be all children of s with 1 incoming edge.

$$F(s) = 1 + \sum_i F(c_i)$$

But this equation doesn't hold if the children paths are not disjoint. Consider this case with overlapping paths:



$$\begin{tikzcd} & a \arrow[dl] \arrow[dr] & \\ b \arrow[dr] & & c \arrow[dl] \\ & d & \end{tikzcd}$$

If we were to apply the disjoint recursive definition, we get:

$$F(d) = 1$$
$$F(b) = 1$$
$$F(c) = 1$$
$$F(a) = 1 + F(b) + F(c) = 3$$

However, this is incorrect. $F(a)$ should be 4.

The goal behind introducing the k-detail was to constrain the graph so there there was still a DP solution, and to also strongly nuance students to set out to solve each subproblem not in $O(1)$ time, but in $O(k)$ time instead. However, the detail was not introduced correctly. The correct detail should have been:

"if P_1 and P_2 are two different paths from A to B, then both paths not only differ by at most k vertices, **but also, both paths are at most k vertices long**".

This was not noticed until Friday, and to be respectful of students' time, the problem was trashed instead of hotfixed [again].

If the correct detail had been given, then we can have the following solution. Essentially, you spend $O(k)$ time to solve for a single subproblem. Before we talk about that, we should talk about how to use something somewhat-cheaper than a BFS-search to compute locally affected nodes.

Side explanation of non-DP counting:

Suppose that we are intending to brute-force count how many nodes will be blocked if we block s. One way to do this is to delete s and all outgoing edges touching s. Then, check the children of s to see if any of them just lost their last incoming edge, and now have an indegree of 0. Such children do not have another source for the PDF, so they are effectively blocked too. We can then repeat the process of deleting source nodes on these children. If we reach a point where no children have indegree 0, then this means we have deleted all the effected nodes if we block s. In fact, if we were to do this to completion for every node, this would give us a $O((|V|+|E|)^2)$ runtime, with an easy proof. However, there is also a DP solution with a more impressive runtime.

How to use this with DP to reach $O(k \cdot (|V|+|E|))$ time:

We can run source-deletion, not until completion, but just for k iterations. For each source node deleted, we add 1 to our count. If we run out of source nodes, we are done for this node. If there are still children source node(s) after k iterations, stop. We are guaranteed that remaining paths are disjoint (share no vertices) from each other, because of the k property (for why, see the section below). So we can define a few variables from this k-depth source deletion. Then we can define our recursive relation based on these variables:

$$d = \text{number of source nodes deleted in a depth-}k\text{ source deletion}$$

$$\{c_i\} = \text{all remaining children source nodes after the depth-}k\text{ source deletion}$$

$$F(s) = d + \sum_i F(c_i)$$

We can store this value, so that if s is eventually some children source node for its own ancestor s', the ancestor can quickly compute $F(s')$.

Why a search of depth-k guarantees disjoint paths:

This argument is similar to the following: Suppose that we know that in Neverland, it rains at least once every 10 days. So if we observe 9 days of no rain in a row, then if we are in Neverland, I know it will rain tomorrow. If it doesn't rain tomorrow, I know we aren't in Neverland. By a similar agument, if we have BFS searched outwards from node A to a depth of k, then we should have considered all paths of length k or less. By the given k property, we know that all other unexplored paths should be disjoint.

Therefore, after a BFS search of depth-k, we can consider all resulting children source nodes that were k vertices away from s, and use their pre-computed values to return the answer.

(c) **(100% credit)** Design an algorithm to determine which student to block off. Give a 4-part solution.

Solution:

Main Idea

This solution will build off of (b). Also note that this solution solves a problem different from the one presented originally.

G is a DAG, so we can linearize it. In reverse topological order, calculate $F(s)$ for all s, in order from sink nodes to the source nodes. Then block the student s' with the highest $F(s')$. The recurrence relation is as given in the previous part.

Each $F(s)$ can be computed in $O(k)$ time, by running a depth-k source deletion process. Then after doing so, we've guaranteed disjoint and correct subproblems, so we can rely on their values $F(c_i)$.

Runtime

Linearizing takes $O(|V|+|E|)$. The computation of $F(s)$ for any single node s takes $O(k)$. Overall runtime is therefore $O(|V|+|E|) \cdot k)$.

Pseudocode

(incomplete)

Proof

We will first prove that the algorithm for $F(s)$ is correct. If it is always correct, we can of course find the vertex s' such that $F(s')$ is maximized after computing F for all vertices.

For this proof, it is handy to consider all the vertices in reverse-topological order, and use strong induction. The base case is if we have a leaf node with no children. Then blocking this node will result in a value of 1. Next, consider that we are blocking a student s, and assume that all subproblems for nodes that are descendants of s have already been solved correctly.

For a student s, if we block off all incoming and outgoing communications to/from s, of course s will not receive the PDF. Let's also consider the 3 possible cases for after running the depth-k deletion as described in (b).

- Case 1:
  After running the depth-k deletion, if no children source nodes remain, then of course all source nodes that were deleted in the depth-k deletion we deprived of the PDF, and no one else was deprived. So:

  $$F(s) = d$$

  .
- Case 2:
  After running the depth-k deletion, if 1 child source node c remains after depth-k deletion, then if this child's $F(c)$ is correct, then our $F(s)$ will be correct as well.

  $$F(s) = d + F(c)$$
- Case 3:
  If at least 2 child source nodes $\{c_i\}$ remain after depth-k deletion, we know that they are disjoint. If they are disjoint, we can blindly depend on their pre-computed subproblems $F(c_i)$, because:
  (1) We know that no node within k distance of s has an edge going to any descendent of $F(c_i)$, or else this would contradict the disjoint paths claim.
  (2) We assume that the pre-computed subproblem of $F(c_i)$ is correct.

  An issue arises if during the depth-k deletion, some outgoing edges were deleted such that a resulting child source node after the depth-k deletion could hold a $F(c_i)$ that is smaller than the true result of blocking not just c_i, but also some ancestor of c_i. However, for this to happen would mean a contradiction for the k-edit distance property.

Since they are disjoint, we have:

$$d = \text{number of source nodes deleted in a depth-}k\text{ source deletion}$$

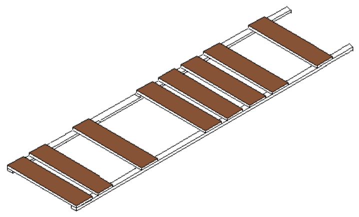
$$\{c_i\} = \text{all remaining children source nodes after the depth-}k\text{ source deletion}$$

$$F(s) = d + \sum_i F(c_i)$$

Therefore, by construction, the recursive definition of $F(s)$ will hold. So as long as we compute $F(s)$ only when we have already computed F for all children of s, then F can compute with no issues. In reverse-topological order, any parent is guaranteed to appear after all of its children (if it has any children). So since we compute F in reverse-topological order, then F will always be able to compute for the next vertex in the ordering.

2. (★★★★ level) Bridge Hop

You notice a bridge constructed of a single row of planks. Originally there had been n planks; unfortunately, some of them are now missing, and you're no longer sure if you can make it to the other side. For convenience, you define an array $V[1..n]$ so that $V[i]$ = TRUE iff the ith plank is present. You're at one side of the bridge, standing still; in other words, your *hop length* is 0 planks. Your bridge-hopping skills are as follows: with each hop, you can increase or decrease your hop length by 1, or keep it constant.



For example, the image above has planks at indices [1, 2, 4, 7, 8, 9, 10, 12], and you could get to the other side with the following hops: [0, 1, 2, 4, 7, 10, 12, 14].

Clarifications: You start at location 0, just before the first plank. Arriving at any location greater than n means you've successfully crossed. Due to your winged shoes, there is no maximum hop length. But you can only hop forward (hop length cannot be negative).

Devise an efficient algorithm to determine whether or not you can make it to the other side.

For this problem, you should know how to do the proof of correctness, but need not include it in your submission. You should submit the main idea, pseudocode, and runtime.

Solution:

Main idea:

As in the previous problems, we have to decide what information to include in the subproblem definitions to make the recurrence easy to compute. We decide to define subproblems as $P(i,h)$, which is TRUE if we can get to location i with our last hop of length h and FALSE otherwise. Then, the recurrence is

$$P(i,h) = V[i] \wedge [P(i-h,h) \vee P(i-h,h-1) \vee P(i-h,h+1)]$$

because we can only land at location i with hop length h if there's a plank there, and our hop length on the previous step was $h-1$, h, or $h+1$.

Note that because our hop length can only increase by 1 at each step, the maximum hop length is n.

Note: It's also possible to do this problem in the other direction, with the recurrence answering the question "can we get to the other side from this location at this speed?"

Pseudocode:

```
procedure canHop($V[1..n]$):
1. Set $V[n+1..2n]$ to TRUE.
2. Set $P(i,h)$ to FALSE in the range $(-n..-1, 0..n+1)$, $(0, 1..n+1)$, $(1..n, 0)$, and $(0..n, n+1)$.
3. Set $P(0,0)$ to TRUE.
4. For $i := 1, \dots, 2n$:
5.     For $h := 1, \dots, n$:
6.         Set $P(i,h) := V[i] \wedge [P(i-h,h) \vee P(i-h,h-1) \vee P(i-h,h+1)]$
7.         If $i > n$ and $P(i,h)$ return TRUE
8. return FALSE
```

The bounds of our iteration are pretty loose, in the sense that we're checking some values of $P(\cdot)$ we know will be FALSE, like $P(1,2)$, or in general $P(i,h)$ where $h > i$, for example. It's possible to add a bit of additional logic and speed up the algorithm by a constant factor, but the asymptotic runtime will not change.

Proof of correctness:

Base cases:

First, we always start at location 0, just before the bridge starts, at hop length 0, so we initialize this to TRUE.

Because we chose to write a simple recurrence, we have to ensure that impossible states are set to FALSE, so we don't lookup a missing value. Thus, we enforce min and max values on the hop length: we never need to stop after we get started (the only thing to do at that point would be to hop length back up to 1), so we can set hop lengths of 0 to FALSE after the starting point. Also, we need some upper bound so that the recurrence does not error when checking $P(i-h,h+1)$ at the max value of s, so we'll set hop lengths of $n+1$ to FALSE (we can never even reach hop length n, so $n+1$ can be safely set to FALSE). We also have to initialize $P(i,h)$ to FALSE at locations all the way back to $-n$, because our naive recurrence will consult them. Finally, we'll be looking at locations after the end of the bridge in our iteration, so we set $V[\cdot]$ to TRUE in all such plausible locations. This is fine as there's solid ground everywhere after the bridge ends.

Recurrence:

Having done all this legwork ensuring the recurrence never errors out, now we can confirm our recurrence is correct. By the definition of the problem, you can only speed up or slow down by at most one unit with each step, so that if you want to reach location i with hop length h you must have gotten to $i-h$ with a hop length equal to or one more/less than h. And there must be a plank at i. We set $P(i,h)$ to TRUE iff both of these conditions hold.

Now, let's consider when we return TRUE. We want to see if it's possible to cross the bridge, which means we must be able to reach some location larger than n at any hop length. It's impossible to speed up to faster than hop length n, so it's impossible for your first step after the bridge ends to fall after location $2n$. Thus, we can safely stop looking there. Thus, we simply return TRUE if we can successfully reach any location $i > n$.

Note that just checking location $n+1$ would not have been enough in all cases.

Running time:

The running time is $O(n^2)$, because we iterate through this number of cases in the nested for-loop in constant time for each case, and there are also $O(n^2)$ base cases.

3. **(★★★★ level) Longest Palindrome Substring** A substring is *palindromic* if it is the same whether read left to right or right to left. For example, "bob" and "racecar" are palindromes, but "cat" is not. Devise an algorithm that takes a sequence $x[1..n]$ and returns the length of the longest palindromic substring. Its running time should be $O(n^2)$.

   For this problem, you should know how to do the proof of correctness, but need not include it in your submission. You should submit the main idea, pseudocode, and runtime.

   Solution:

   Main Idea:

   Let $x = x_1 \dots x_n$ be the string, and let $P[i,j]$ be a boolean representing whether the substring $x_i \dots x_j$ forms a palindrome. Then we have the following recurrence:

$$P[i,j] = \begin{cases} P[i+1,j-1] & \text{if } x_i == x_j \\ \texttt{false} & \text{otherwise} \end{cases}$$

   With base cases:

$$\begin{aligned} S[i,i] &= \texttt{true} && \forall i = 1 \dots n \\ S[i,i+1] &= (x_i == x_{i+1}) && \forall i = 1 \dots n-1 \end{aligned}$$

Once we've computed the matrix $P[i,j]$ for $i = 1 \dots n, j = i \dots n$, we can then iterate over the its entries to pick out the largest difference $j-i+1$ such that $P[i,j]$ is `true` – this is the length of the longest palindromic substring.

Pseudocode:

```
procedure LONGESTPALINDROMICSUBSTRING($x[1 \dots n]$)
    $P \leftarrow$ empty boolean matrix of size $n$ by $n$.
    $P[i,i] \leftarrow 1$ for $i = 1 \dots n$.
    $P[i,i+1] \leftarrow (x_i == x_{i+1})$ for $i = 1 \dots n-1$.
    for $i = n-2 \dots 1$ do
        for $j = i+2 \dots n$ do
            if $x_i == x_j$ then
                $P[i,j] \leftarrow P[i+1,j-1]$.
            else $P[i,j] \leftarrow 0$
    $M \leftarrow 0$.
    for $i = 1 \dots n$ do
        for $j = i \dots n$ do
            if $P[i,j]$ and $j-i+1 > M$ then
                $M \leftarrow j-i+1$.
    return $M$.
```

Runtime:

The matrix $P[i,j]$ has $\frac{1}{2}n(n+1) = O(n^2)$ entries that need to be filled, each of which takes constant time. The overall runtime is $O(n^2)$.

4. (★★★★ level) A Sisyphean Task

Suppose that you have n boulders, each with a positive integer weight w_i. You'd like to determine if there is any set of boulders that together weight exactly k pounds. You may want to review the solution to the Knapsack Problem for inspiration.

For this problem, you should know how to do the proof of correctness, but need not include it in your submission. You should submit the main idea, pseudocode, and runtime.

(a) Design an algorithm to do this.

Solution:

Main idea: This is very much like the knapsack problem, except we want our items to sum to *exactly* k instead of being less than or equal to k.

Let $S(i,k')$ be true if and only if there is some subset of $W[1..i]$ that sums to k'. Then, we have the following recurrence relation:

$$S(i,k') = S(i-1,k') \vee S(i-1,k'-w_i)$$

Intuitively, this comes from the observation that a subset of $W[1..i]$ summing to k' either includes a_i or it doesn't. If it includes a_i, then we are left looking for a subset of $W[1..i-1]$ summing to $k'-w_i$. If it doesn't include a_i, then this means we need a subset of $W[1..i-1]$ that sums to k'. Thus, $S(i,k')$ is true if and only if at least one of $S(i-1,k')$ and $S(i-1,k'-w_i)$ is true. Our base cases are:

- $S(0,0) = true$ because the empty set sums to 0.
- $S(0,k') = false$ for $k' > 0$.

- $S(i,k') = false$ for $k' < 0$.

The final answer we are looking for is $S(n,k)$. This gives us nk subproblems, each of which takes cosntant time to solve.

Pseudocode:

```
1'. Set $S[0,0] = true$ and $S[0,k'] = false$ for $0 < k' \leq k$.
2'. For $i := 1,\dots,n$:
3'.     For $k' := 0,\dots,k$:
4'.         If $S[i-1,k']$:
5'.             $S[i,k'] = true$.
6'.         Else if $k' \geq w_i$ and $S[i-1,k'-w_i]$:        // Base case: false if $k' - w_i < 0$.
7'.             $S[i,k'] = true$.
8'.         Else:
9'.             $S[i,k'] = false$.
10'. Return $S[n,k]$.
```

Proof of correctness: The recurrence relation as described in the main idea is correct. To show this, we observe that if $S(i,k')$ is true, meaning there is some subset I of $W[1..i]$ that sums to k', then I either contains w_i or it doesn't. If $w_i \in I$, then removing w_i from I yields a I', which is a subset of $W[1..i-1]$ and must sum to $k' - w_i$. In this case, $S(i-1,k'-w_i)$ must be true. If $w_i \notin I$, then I is a subset of $W[1..i-1]$ that sums to k', so $S(i-1,k')$ must be true. To show the other direction, we observe that if $S(i-1,k'-w_i)$ is true, then $S(i,k')$ must be true because if I' is a subset of $W[1..i-1]$ that sums to $k' - w_i$, then $I = I \cup \{w_i\}$ is a subset of $W[1..i]$ that sums to k'. If $S(i-1,k')$ is true, then $S(i,k')$ must be true because if I' is a subset of $W[1..i-1]$ that sums to k', then I' is also a subset of $W[1..i]$ that sums to k'.

We observe that the algorithm correctly computes this recurrence relation, using the fact that each w_i must be strictly positive, so each iteration of the loop only relies on previously computed values. By definition $S(n,k)$ gives us the answer we want.

Running time analysis: This runs in $O(nk)$. There are nk iterations of the inner for loop, each of which takes a constant time.

(b) Is your algorithm polynomial in the *size* of the input? Remember that size is in terms of how many bits we need.

Solution:

No. This algorithm runs in time proportional to k, but to represent k in the input, we only need $\log_2 k$ bits. This may seem like a technicality, but it's actually very important – it means that it wouldn't take very long for somebody to write down an input that would make this algorithm run for a very long time. For instance, writing down the number 2^{100} takes just 100 bits, but would make our running time proportional to 2^{100}, which is more than the age of the universe in seconds (intractably large). This algorithm is what is known as *pseudopolynomial*, meaning it is polynomial in the input *values*, not the input *size*.

5. (★★★★★ level) Advertising network

You are an advertising network tasked with making bids on Facebook's mobile ad units for your customers. On each day, you make a bid of $\$\theta$, $\theta \in [0,1]$, and win that day's auction with probability θ. If you don't win, then you don't spend any money. As per your contract with your customers, you *need* to win at least k out of n auctions that will occur this fiscal year.

An obvious approach may be to bid \$1 for each of the first k days, then nothing for the rest of the $n-k$ days – this strategy will cost you $\$k$. However, it may not be optimal! Consider this alternative for $n = 30, k = 4$:

- Bid \$0.50 for each of the first 26 days, or until you've won 4 auctions.
- Bid \$1.00 for each of the next 4 days, if you didn't win them yet.
- This strategy will have expected cost \$2.00[1], and worst case cost \$4.00 – a strictly superior strategy.

Give an optimal strategy to minimize expected cost while maintaining your contract, for any k, n. You only need to explain the main idea; no need for proof, runtime, or pseudocode.

Here are some hints:

- You'll need to use probability; specifically, the linearity of expectation [2].
- Parameterize your strategy as a set of variables $\theta_{n,k}$, and notice that you need to minimize some function that can be written in terms of $\theta_{n,k}$.
- To actually solve said optimization problem, you can assume you have access to a quadratic program solver that can minimize any quadratic function of a single variable, in time that is efficient.

Solution:

We can formulate the problem as a dynamic program.

Let $\theta_{n,k}$ be the optimal amount to bid \$$\theta$ for a particular n, k. Let $E[n,k]$ be the expected cost needed to win k out of a remaining n auctions. Then we have the recurrence

$$E[n,k] = \theta_{n,k}(\theta_{n,k} + E[n-1,k-1]) + (1-\theta_{n,k})E[n-1,k]$$

Interpretation: the first term is the probability of winning, times the expected remaining cost for the next $n-1$ days needing to win $k-1$ times. The second term is the probability of not winning, times the expected cost for the next $n-1$ days needing to win k times. The base cases are

$$\begin{aligned} E[n,k] &= k && \forall n = k && \text{(must win all remaining bids)} \\ \theta_{n,k} &= 1 && \forall n = k && \text{(must win all remaining bids)} \\ E[n,0] &= 0 && \forall n && \text{(no need to bid)} \\ \theta_{n,0} &= 0 && \forall n && \text{(no need to bid)} \end{aligned}$$

Each $E[n,k]$ models a quadratic program which we can then solve for each $\theta_{n,k}$. If we go through each $E[n,k]$ and solve them sequentially (dynamic programming style) from the bottom up, we'd solve for the optimal set of θ's for any situation parameterized by n, k, and therefore have the optimal strategy.

6. (★★★★★★ level) Knightmare (extra credit question)

Give an algorithm to find the number of ways you can place knights on an N by M chessboard such that no two knights can attack each other [3] (there can be any number of knights on the board, including zero knights). The runtime should be $O(2^{3M} \cdot N)$ (or symmetrically, switch the variables).

Note that even though this question is extra credit (and thus only worth 1 pt), the staff *strongly recommends* that you at least attempt it and understand the solution. It will be *very* helpful practice for your upcoming midterm, which we remind you will be Monday March 20, 2017.

Solution:

[1] Rounded to the nearest cent.
[2] https://en.wikipedia.org/wiki/Expected_value#Linearity
[3] Knights attack according to https://en.wikipedia.org/wiki/Knight_(chess)#Movement

Main Idea: A knight can reach at most two columns away; therefore we can compute subproblems while looking at only the previous two columns. Without loss of generality, we assume that $M < N$. We define $K(n,v,w)$ to be the number of ways we can put knights on an M by n chessboard such that the last M by 1 column has knights placed exactly in the positions specified in v, and the second-last M by 1 column has knights placed exactly in the positions specified in w (so v and w are length M bit vectors). We want to compute the sum of $K(N,v,w)$ over all length M bit vectors v and w.

Pseudocode:

```
procedure KnightCounts($M,N$)
    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
            $K(n,v,w)$ += $K(n-1,w,u)$
    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

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

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

We can precompute all valid M by 2 configurations and M by 3 configurations in $O(2^{3M}M)$ time because we can just check that each square with a knight is not being attacked by any other square with a knight. A square can only be under attack by at most 8 other squares so this check takes constant time.

Running Time: We have $2^{2M}N$ subproblems $K(n,v,w)$ we wish to compute and each recurrence takes $O(2^M)$ time for a total running time of $O(2^{3M}N)$.