Introduction to Algorithms **Solution Set 3**
CS 4820, Spring 2018

(1) *(10 points)*
Suppose we are given a graph $G = (V, E)$ with costs $(c_e)_{e \in E}$ on the edges and costs $(c_v)_{v \in V}$ on the vertices. We wish to designate a subset of the vertices as active *and a subset of the edges as* eligible*, such that every inactive vertex can be joined to at least one active vertex by a path made up of eligible edges. Design an algorithm to choose the set of active vertices, A, and the set of eligible edges, F, so as to minimize their combined cost, $\sum_{v \in A} c_v + \sum_{e \in F} c_e$.*

Remark: *You may assume that G is a connected graph. In particular, if G has n vertices and m edges, you may assume $m \geq n - 1$.*

Solution. The easiest way to solve this problem is by reducing it to the minimum spanning tree problem on a slightly larger graph. Adjoin a new vertex — let's call it the root — and connect the root to every other vertex v with an edge whose cost is c_v. Call this augmented graph G^+. Compute a minimum spanning tree of G^+, denoted by $MST(G^+)$, and define A to be the set of vertices v that are adjacent to the root in $MST(G^+)$ and define F to be the set of all edges of $MST(G^+)$ that are not incident to the root.

The running time is $O(m \log n)$ and the proof of correctness is a consequence of the following lemma.

Lemma 1. *Every connected spanning subgraph of G^+ can be converted into a pair (A, F) meeting the problem constraints by the above transformation, and every pair (A, F) meeting the problem constraints can be converted into a connected spanning subgraph of G^+ by the inverse transformation. These transformations preserve cost.*

Proof. Let r denote the root node of G^+. Suppose H is a connected spanning subgraph of G^+. Define A to be the set of vertices v that are adjacent to r in H, and define F to be the set of edges of H that are not incident to the root. For any vertex u in G, there is a path from u to r in H because H is a connected spanning subgraph of G^+. Deleting the final edge of this path, we obtain a path from u to a vertex in A, and the (possibly empty) edge set of this path is a subset of F. This verifies that (A, F) satisfies the problem constraints. Conversely, if (A, F) satisfies the problem constraints and we apply the reverse transformation which yields a subgraph H of G^+ with vertex set $V(G^+)$ and edge set $F \cup \{(v, r) \mid v \in A\}$, then H is a spanning subgraph of G^+ because its vertex set if $V(G^+)$, and it is connected because for every vertex $u \neq r$ H contains a path from u to r; indeed, such a path can be constructed by taking a path from u to a vertex $v \in A$ made up of edges in F, and appending the edge (v, r) to this path.

The fact that the transformation and its inverse are cost-preserving is an immediate consequence of the way we defined edge costs in H. $\square$

Concluding the proof, the lemma implies that a minimum cost pair (A, F) satisfying the problem constraints must be obtained from a mininum cost connected spanning subgraph H of G^+ — or equivalently, a minimum spanning tree of G^+ — by performing the transformation described in the algorithm above: define A to be the set of vertices of H adjacent to r, and F to be the set of edges of H not incident to r.

(2) *In class we've been talking about applications of dynamic programming to optimization. There are also many applications of dynamic programming to counting, and to calculating probabilities. This exercise explores one such application. Recall that an instance of the* interval scheduling *problem consists of n intervals $I_1, I_2, \ldots, I_n$, where each interval I_k (for $k = 1, \ldots, n$) is a closed interval $[s_k, f_k]$ with start time s_k and finish time $f_k > s_k$. A set of intervals is* non-conflicting *if no two of its elments overlap.*

In this exercise we assume we are given an instance of the interval scheduling problem such that the numbers $s_1, s_2, \ldots, s_n, f_1, f_2, \ldots, f_n$ are all distinct, and such that the intervals are ordered by increasing finish time: $f_1 < f_2 < \cdots < f_n$.

(2a) (7 points)
Design an algorithm to count how many subsets of $\{I_1, I_2, \ldots, I_n\}$ are non-conflicting. Remember that the empty set and one-element sets are always non-conflicting.

(2b) (3 points)
Let Ω denote the collection of all non-conflicting subsets of $\{I_1, \ldots, I_n\}$. Given the list of intervals $I_1, \ldots, I_n$, and an index k in the range $1 \leq k \leq n$, design an algorithm to compute the probability that a uniformly random element of Ω contains interval I_k.

In your solution to (2b), you may omit the running time analysis. The algorithm you design must still have running time bounded by a polynomial function of n, but you don't need to include the analysis of running time in your write-up. You are also free to use the algorithm from part (2a) as a subroutine in part (2b), even if you didn't succeed in solving (2a).

Solution. For part (2a), the following dynamic programming algorithm fills in an array A whose entry $A[k]$ records the number of non-conflicting subsets of $\{I_1, \ldots, I_k\}$.

```
                                   // Initialize array of predecessor pointers.
for $k = 1, \ldots, n$ do
    if there exists $j < k$ such that $f_j < s_k$ then
        Let $p[k]$ be the largest such $j$.
    else
        Let $p[k] = 0$.
    end if
end for
                                   // Initialize dynamic programming table.
Initialize $A[0] = 1$.
Initialize $A[k] = $ NULL for $k = 1, \ldots, n$.
                                   // Main loop: fill in dynamic programming table.
for $k = 1, \ldots, n$ do
    $A[k] = A[k-1] + A[p[k]]$
end for
Output $A[n]$.
```

In class on Friday, February 9, Professor Kleinberg presented a way to implement the initial step of filling in the predecessor array, $p[0], \ldots, p[n]$, in $O(n \log n)$ time by sorting the set

$\{s_1, \ldots, s_n, f_1, \ldots, f_n\}$, and scanning through the sorted list while keeping track of the index j of the most recently scanned finish time f_j. (Initially $j = 0$ until the scan encounters f_1.) Every time a start time s_k is scanned, we set $p[k] = j$. Every time a finish time is scanned, we increment the value of j.

Other than this $O(n \log n)$ preprocessing step, the rest of the algorithm runs in time $O(n)$ because it consists of two loops — an initialization loop and a main loop — which both run for n iterations while doing $O(1)$ work per iteration. Hence the total running time of the algorithm is $O(n \log n)$.

To prove correctness, we use strong induction. The induction hypothesis is that $A[k]$ equals the number of non-conflicting subsets of intervals $\{I_1, \ldots, I_k\}$. In the base case $k = 0$, there is exactly one such subset, namely the empty set. Since the algorithm initializes $A[0] = 1$, the induction hypothesis is satisfied in the base case. For $k > 0$, there are two types of non-conflicting subsets of $\{I_1, \ldots, I_k\}$ — those that contain I_k and those that do not. The number of subsets of the first type is $A[k-1]$, by the induction hypothesis. The subsets of the second type are in one-to-one correspondence with non-conflicting subsets of $\{I_1, \ldots, I_{p[k]}\}$; the correspondence is defined by adjoining I_k to a non-conflicting subset of $\{I_1, \ldots, I_{p[k]}\}$. Hence the number of non-conflicting subsets of the second type is $A[p[k]]$, and the total number of non-conflicting subsets of $\{I_1, \ldots, I_k\}$ is $A[k-1] + A[p[k]]$. Since the algorithm sets $A[k]$ equal to this value, the induction step is confirmed.

For part (2b), let A denote the subset of $\{I_1, \ldots, I_n\}$ consisting of intervals that finish before s_k and let B denote the subset consisting of intervals that start after f_k. Let Ω_A and Ω_B denote the collections of non-conflicting subsets of A and of B, respectively. For any sets $S_A \in \Omega_A$ and $S_B \in \Omega_B$, the union $S_A \cup \{I_k\} \cup S_B$ is a non-conflicting subset of $\{I_1, \ldots, I_n\}$ that contains I_k. Conversely, every non-conflicting subset of $\{I_1, \ldots, I_n\}$ that contains I_k is formed in this way. In other words, a one-to-one correspondence between $\Omega_A \times \Omega_B$ and the set of non-conflicting subsets of $\{I_1, \ldots, I_n\}$ that contain I_k can be defined using the mapping $(S_A, S_B) \mapsto S_A \cup \{k\} \cup S_B$. This counting argument proves the correctness of the formula

$$\Pr_{S \sim \mathrm{Unif}(\Omega)}(k \in S) = \frac{|\Omega_A| \times |\Omega_B|}{|\Omega|}.$$

The two terms in the numerator, as well as the denominator, can all be computed by applying the algorithm from (2a) to the interval sets A, B, and $\{I_1, \ldots, I_n\}$. Hence, the time required to compute the right side of the formula is equal to the time required to call the algorithm three times on inputs is size at most n, plus the time to perform one multiplication and one division operation. In other words, the running time is $O(n \log n)$.

(3) *(10 points)*
In the TROMINO TILING *problem one is given a subset of a square grid, and one must decide whether the given subset can be tiled (i.e., covered without overlaps) by L-shaped trominoes.*

The input to the problem can be given in the form of a $k \times n$ array of 0's and 1's. Denoting this array by A, the entry $A[i,j]$ equals 1 if the grid cell in location (i,j) belongs to the subset to be tiled, and if not then $A[i,j] = 0$.

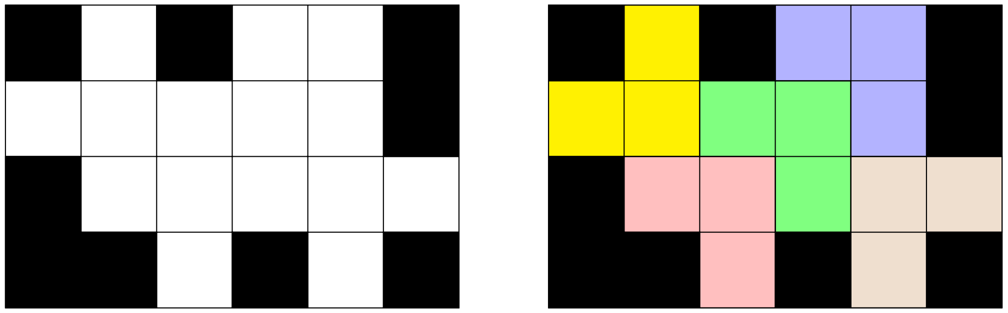
Design an algorithm to solve TROMINO TILING *in time $O(n) \cdot 2^{O(k)}$. Your algorithm only needs to output a simple "yes" or "no" answer indicating whether or not the given subset of the grid can be tiled by L-shaped trominoes. In describing your algorithm, you may assume you have access to a subroutine called* WIDTHTWO *that solves the case when n (the width of the grid)* *is equal to 2, and k (the grid's height) is an arbitrary positive integer. The running time of the* WidthTwo *subroutine is $O(k)$. You are* not *responsible for designing or analyzing the* WidthTwo *subroutine*[1]*; you are welcome to just assume it exists and treat it as a "black box."*

Note that the running time of the algorithm you are being asked to design is **not polynomial in the input size**. *It is linear in n but exponential in k. Partial credit will be awarded for algorithms whose running time depends super-exponentially on k, or has super-linear but still polynomial dependence on n. No credit will be awarded for algorithms whose running time is exponential in n.*

Example. *In the following example $k = 4$ and $n = 6$, and the matrix A is given by*

$$A = \begin{pmatrix} 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 \end{pmatrix}.$$

The region to be tiled is the set of white squares in the left figure below. The right figure illustrates a tromino tiling of the region.



Solution 1. Our solution will make use of a two-dimensional dynamic programming table $T[i, J]$ where the first index, i, is an integer in the range $0, \ldots, n$, and the second index, J, is a subset of $\{0, \ldots, k-1\}$. The value stored at $T[i, J]$ will be TRUE if there is a tromino tiling of the region represented by the first i columns of A, together with all the grid cells in column $i+1$ whose row number is an element of J. Otherwise the value stored at $T[i, J]$ will be FALSE. (Of course, as always, if we haven't computed $T[i, J]$ yet, then we will temporarily store the value NULL there.)

In a typical programming language, array entries need to be indexed by integers, not sets. A subset $J \subseteq \{0, \ldots, k-1\}$ can be represented by the binary digits of a number j in the range $0, 1, \ldots, 2^k - 1$. In the following pseudocode we use this binary encoding trick to translate sets into numbers when using them to index array entries. On the other hand, we don't bother translating sets into numbers when passing them as function arguments. In particular, we will assume that the WidthTwo function accepts two arguments, both of which are subsets of $\{0, 1, \ldots, k-1\}$, and that WidthTwo(I, J) outputs the answer — (TRUE or FALSE — to the tromino tiling problem in which I represents the set of squares that belong to the first column

[1]If you're curious, though, you might find it to be an instructive exercise to try designing an algorithm for the WidthTwo subroutine that runs in $O(k)$.

of the region to be tiled, and J represents the set of squares belonging to the second column of that region.

```
function SetToInt($J$)                                   // $J$ is a subset of $\{0,\dots,k-1\}$
    return $\sum_{i\in J} 2^i$
end function

for $i = 0, 1, \dots, n$ do
    for all subsets $J \subseteq \{0,\dots,k-1\}$ do
        $j$ = SetToInt($J$)
        Initialize $T[i,j]$ = FALSE
        if $i = 0$ and $J = \emptyset$ then
            $T[i,j]$ = TRUE
        else if $i > 0$ then
            Let $R = \{r \mid A[r, i-1] = 1\}$.
            for all partitions of $R$ into two sets, $X$ and $Y$ do
                $x$ = SetToInt($X$)
                $T[i,j] = T[i,j] \vee (T[i-1,x] \wedge$ WidthTwo($Y, J$)).
            end for
        end if
    end for
end for
Output $T[n,\emptyset]$.
```

The running time is $O(n) \cdot 2^{O(k)}$ because there are three nested loops with $n+1$, 2^k, and 2^k (or fewer) iterations, respectively, so the total number of iterations of the inner loop is bounded above by $(n+1) \cdot 4^k$, while the amount of work per inner loop iteration is $O(k)$ due to calling SetToInt and WidthTwo.

The correctness of the algorithm is justified by the following lemma.

Lemma 2. *A subset S of the $k \times \ell$ grid can be tiled by trominoes if and only if it can be partitioned into two subsets, Y and Z, such that:*

1. *Y contains the intersection of S with the n^{th} column*
2. *Y is contained in the intersection of S with the final two columns.*
3. *Y and Z can each be tiled by trominoes.*

Proof. Consider any tromino tiling of S. Let Y be the union of all tiles that include at least one square in the n^{th} column, and let Z be the union of all other tiles. Then it is clear that Y and Z satisfy the three properties stated in the lemma. □

The correctness of the algorithm for tromino tiling now follows by induction on i, the number of iterations of the outer loop. The induction hypothesis is that for every set J represented by a k-bit binary number j = SetToInt(J), the value of $T[i,j]$ equals TRUE if and only if the region represented by the first i columns of A, together with all the grid cells in column $i+1$ whose row number is an element of J, has a tromino tiling.‘The base case says that a region that fits in one column ($i = 0$) can be tiled if and only if it is the empty set. The induction step is justified using the lemma above.

Finally, although the problem didn't ask for an $O(k)$ implementation of the WidthTwo subroutine, it's now clear how to implement it: form a $k \times 2$ matrix of 0's and 1's from the two set-valued arguments of WidthTwo by interpreting the first argument as specifying the set of 1's in the first column and the second argument as specifying the set of 1's in the second column. Transpose this into a $2 \times k$ matrix and solve the associated tromino tiling problem using the algorithm above. The algorithm will call the subroutine WidthTwo at various times during its execution, but every such call will correspond to a 2×2 instance of the tromino tiling problem, and such problems are trivial to solve in constant time using a table lookup.

Solution 2. Suppose for a minute that we can maintain a dynamic programming table U of size 2^{kn}, indexed by $k \times n$ matrices of 0's and 1's, which stores a Boolean value representing the answer to the tromino tiling problem defined by that matrix. Even initializing such a table would take $O(2^{kn})$ time, which would exceed the allowable running time in this problem. But just for the moment, let's ignore that. Then the following algorithm outputs a correct answer to the tromino tiling problem.

```
function TrominoTiling($A$)                    // $A$ is a $k \times n$ matrix of 0's and 1's
    if $U[A] \neq$ NULL then
        return $U[A]$
    end if
    if every entry of $A$ equals 0 then
        $U[A] =$ TRUE
    else
        Initialize $U[A] =$ FALSE
        Let $j$ be the highest-numbered column of $A$ containing a 1.
        for all tromino-shaped regions $T$ that intersect the $j^{\text{th}}$ column do
            if $T$ is contained in the region to be tiled then
                Let $B$ be the matrix obtained from $A$ by setting the three entries corresponding to $T$ to zero.
                $U[A] = U[A] \vee$ TrominoTiling($B$)
            end if
        end for
        return $U[A]$
    end if
end function
```

The justification for the algorithm is pretty simple: if a region can be tiled by trominoes, there must be at least one tile that touches the last column of the region. Consider all ways of deleting such a tile from the region, and check whether any of the “deleted regions” thus formed can be tiled by trominoes.

As far as running time is concerned, the following lemma is crucial.

Lemma 3. *When an execution of the* TrominoTiling *algorithm starts by calling* TrominoTiling*(A), if we consider any matrix B such that* TrominoTiling*(B) is called at any time during the algorithm's execution, then there exists a j such that:*

1. *The first $j-2$ columns of B match the first $j-2$ columns of A.*

2. *In every column of B numbered higher than j, all of the entries are zero.*

Proof. Whenever the algorithm calls TrominoTiling(B), it has just formed B by deleting a tromino-shaped tile from some other region. The tromino-shaped tile belonged to two columns, say j and $j-1$. Since the algorithm always deletes tiles from the highest-numbered occupied column, we know that in every column of B numbered higher than j, all of the entries are zero. We also know that all previously deleted tiles — in the stack of recursive function calls leading up to calling TrominoTiling(B) — must have intersected a column numbered j or higher. Hence, the first $j-2$ columns of A have not been modified in any of these function calls, which ensures that the first $j-2$ columns of B matches those of A. $\square$

It is evident that the number of matrices B satisfying the criteria in the lemma is bounded by $n \cdot 4^k$, since such a matrix can be specified by giving the value of j and the entries of columns $j-1$ and j. Hence, the total number of times TrominoTiling is called during its execution is at most $n \cdot 4^k$. A single call to TrominoTiling takes only $O(nk)$ time, so ironically, the only thing preventing TrominoTiling from running in poly(n) time is the step of initializing the dynamic programming table as an array of size 2^{kn}.

To make the running time polynomial in n, we need to shrink the data structure used for storing the dynamic programming table. A naïve way to do this is to store the dynamic programming table as a linked list of ordered pairs. Every time we finish computing $U[B]$ for a matrix B, we store the pair $(B, U[B])$ in the linked list. In any line of the pseudocode above when we access a value such as $U[A]$ or $U[B]$, we do this by performing a linear scan through the entire linked list, returning the value once it is found, or returning NULL if no value is found. This means that querying the dynamic programming table involves scanning through a linked list whose size is potentially $O(nk \cdot n \cdot 4^k)$, which inflates the running time, but even after this inflation we have a running time of $O(n^4 k^2 16^k)$, or something like that.

To improve the running time to linear in n, we need to store the dynamic programming table in a more efficient form. A suitable way of storing it is a two-dimensional array $T[i,j]$ where j denotes the highest-numbered column containing a non-zero entry of the matrix, and i is a binary number whose $2k$ digits represent the contents of columns $j-1$ and j.