CS 341, Winter 2015
Andrew Arnold, Timothy Chan, Doug Stinson

Assignment 4 (due Tuesday, March 17, 4pm)

Please read `https://www.student.cs.uwaterloo.ca/~cs341/policies.html` for general instructions, and `https://www.student.cs.uwaterloo.ca/~cs341/prog.html` for programming guidelines.

1. [*15 marks*] We are given n items, where item i has value v_i and weight w_i, where the v_i's and w_i's are positive integers. Suppose we have *two* knapsacks, each can store up to a weight of W. We want to select items to put into the two knapsacks, maximizing the total value of the selected items. In other words, we want to select two disjoint subsets $A, B \in \{1, \ldots, n\}$ to maximize $\sum_{k \in A} v_k + \sum_{k \in B} v_k$, subject to the constraints $\sum_{k \in A} w_k \leq W$ and $\sum_{k \in B} w_k \leq W$.

   Design a dynamic programming algorithm that solves the problem in $O(nW^2)$ time. Provide a complete description: (a) first define subproblems precisely, (b) give base cases, (c) derive the recursive formula (with justification), (d) write pseudocode for computing the optimal value, (e) write pseudocode for outputting the optimal subsets, and (f) give an analysis of the runtime and space usage.

   [Hint: Use a three-dimensional table...]

   Answer:

   (a) Let $C[i, x, y]$, $0 \leq i \leq n$, $0 \leq x, y \leq W$ be the maximal value of $\sum_{k \in A} v_k + \sum_{k \in B} v_k$ subject to constraints $A, B \subset \{1, \ldots, i\}$, $A \cap B = \emptyset$, $\sum_{k \in A} w_k \leq x$ and $\sum_{k \in B} w_k \leq y$. The target value is thus $C[n, W, W]$.

   (b) Base cases: If $i = 0$ then we must have $A = B = \emptyset$, thus $C[0, x, y] = 0, \forall x, y$. Similarly if $x = y = 0$, then we cannot carry any items, thus $C[i, 0, 0] = 0, \forall i$.

   (c) Recursive formula: Consider $C[i, x, y]$. A solution giving the optimal value to this subproblem must fall under one of three cases: The first knapsack contains item i, the second knapsack contains item i, or none of the knapsacks contain item i. As the first (second) knapsack can only carry item i if $x \geq w_i$ ($y \geq w_i$), this gives a recursive formula:

$$C[i, x, y] = \begin{cases} \max(C[i-1, x-w_i, y] + v_i, C[i-1, x, y-w_i] + v_i, C[i-1, x, y]) & \text{if } w_i \leq x, y, \\ \max(C[i-1, x-w_i, y] + v_i, C[i-1, x, y]) & \text{if } y < w_i \leq x, \\ \max(C[i-1, x, y-w_i] + v_i, C[i-1, x, y]) & \text{if } x < w_i \leq y, \\ C[i-1, x, y] & \text{else.} \end{cases}$$

   (d)

```
1.  for i ← 0 to n do
2.      for x ← 0 to W do
3.          for y ← 0 to W do
4.              if i = 0 or x = y = 0 then C[i, x, y] ← 0
5.              else if w_i ≤ x, y then
6.                  C[i, x, y] ← max(C[i−1, x−w_i, y] + v_i, C[i−1, x, y−w_i] + v_i, C[i−1, x, y])
7.              else if w_i ≤ x then
8.                  C[i, x, y] ← max(C[i−1, x−w_i, y] + v_i, C[i−1, x, y])
9.              else if w_i ≤ y then
10.                 C[i, x, y] ← max(C[i−1, x, y−w_i] + v_i, C[i−1, x, y])
11.             else C[i, x, y] ← C[i−1, x, y]
12. Return C[n, W, W]
```

(e) Idea: one could use an auxillary table $\pi[i, x, y]$ in order to keep track of the three cases from part (b) that gives us the optimal value for $C[i, x, y]$. Alternatively, we can merely inspect the table C to determine one case corresponding to an optimal solution. Note that while multiple cases may apply, we are only interested in constructing one optimal solution.

```
1. Tabulate C using the algorithm from part (d)
2. (i, x, y) ← (n, W, W), A ← {}, B ← {}
3. while C[i, x, y] > 0 do
4.     if x ≥ w_i and C[i, x, y] = C[i − 1, x − w_i, y] + v_i then
5.         A ← A ∪ {i}
6.         (i, x, y) ← (i − 1, x − w_i, y)
7.     else if y ≥ w_i and C[i, x, y] = C[i − 1, x, y − w_i] + v_i then
8.         B ← B ∪ {i}
9.         (i, x, y) ← (i − 1, x, y − w_i)
10.    else (i, x, y) ← (i − 1, x, y)
11. return A, B
```

(f) In part (d), lines 4-11 take $\Theta(1)$ time to construct one table entry $C[i, x, y]$. There are $\Theta(nW^2)$ such table entries, thus the algorithm from part (d) takes $\Theta(nW^2)$ time.

In part (e), each iteration of the while loop takes $\Theta(1)$ time. As i is decremented in every iteration of the while loop on lne 3, the total cost is $\mathcal{O}(n)$, which is dominated by that of part (d).

The space complexity is $\Theta(nW^2)$ in order to store the table C. This assumes that table entries are word-sized (i.e., constant sized).

2. [*15 marks*] A *histogram rectangle* is a rectangle whose bottom edge lies on the x-axis.

Given n points $p_1 = (x_1, y_1), \ldots, p_n = (x_n, y_n)$ (with $y_i > 0$ for all i) and an integer k, we want to find k nonoverlapping histogram rectangles that contain all n points, while minimizing the total area of the rectangles.

See the following example (the solution below is not necessarily optimal). Note that we allow "thin" rectangles with area 0 (like the third one from the left).

Design a dynamic programming algorithm that solves the problem in $O(kn^2)$ time. Provide a complete description: (a) first define subproblems precisely, (b) give base cases, (c) derive the recursive formula (with justification), (d) write pseudocode for computing the optimal area, (e) write pseudocode for outputting the vertices of the optimal polygon, and (f) give an analysis of the runtime and space usage.

[Hint: First sort the points $p_1, \ldots, p_n$ by x-coordinates. For each $i = 0, \ldots, n$ and $j = 0, \ldots, k$ or $j = 1, \ldots, k$, consider the subproblem of finding j nonoverlapping histogram rectangles that contain $p_1, \ldots, p_i$.]

Answer: Suppose that the p_i are sorted by x-coordinate, such that $x_1 < x_2 < \cdots < x_n$.

(a) Per the hint, we let $C[i, j]$ be the least area of j nonoverlapping histogram rectangles containing $p_1, \ldots, p_i$, for $1 \leq i \leq n$ and $0 \leq j \leq k$.

(b) We let $C[0, j] = 0$ for all j, as we can trivially select j thin rectangles. We also let $C[i, 0] = \infty$ for $i > 0$, as we are taking a minimum over the set of all solutions, and there are no solutions for $C[i, 0]$. Alternatively, if your base case is $C[i, 1]$, you will have to take the rectangle starting at $x = x_1$, ending at x_i, with height large enough to contain $p_1, \ldots, p_i$, i.e., $C[i, 1] = (x_i - x_1) \cdot \max_{1 \le \ell \le i} y_i$.

(c) Now consider an optimal solution $C[i, j]$, where $i > 0$. In such a solution, if the rightmost rectangle contains p_i, the rectangle should begin at some x_ℓ, $\ell \le i$ and end at x_i. The remaining $\ell - 1$ rectangles must contain $p_1, \ldots, p_{\ell-1}$. This gives a recursive formula

$$C[i, j] = \min_{1 \le \ell \le i} \left(C[\ell - 1, j - 1] + (x_i - x_\ell) \cdot \max_{\ell \le m \le i} y_m \right)$$

(Note if the rightmost rectangle in an optimal solution does not cover a point, we have that $C[i, j] = C[i, j - 1]$; however, as $C[i, j - 1] \ge C[i - 1, j - 1] + (x_i - x_i)y_i$, we already obtain a solution at least as good by the formula above. In other words, there always exists an optimal solution for $C[i, j]$ whereby the rightmost rectangle contains p_i.)

(d) We compute the minimum in the recursive formula in decreasing order of ℓ, such that the inner maximum is computed with constant cost per ℓ. We maintain a table π, where $\pi[i, j]$ stores the ℓ which gives us the minimum value of $C[i, j]$, and the m which gives us the maximum value of y_m.

```
0.  sort and relabel p_i such that x_1 < x_2 < ··· < x_n
1.  for j ← 0 to k do C[0, j] ← 0
2.  for i ← 1 to n do
3.      C[i, 0] ← ∞
4.      for j = 1 to k do
5.          max ← 0, min ← ∞
6.          for ℓ ← i down to 1
7.              if y_ℓ > max then max ← y_ℓ, m ← ℓ
8.              if C[ℓ − 1, j − 1] + (x_i − x_ℓ) · max < min then
9.                  min ← C[ℓ − 1, j − 1] + (x_i − x_ℓ) · max
10.                 π[i, j] ← (ℓ, m)
11.         C[i, j] ← min
12. return C[n, k]
```

(e) We will use the auxillary table π to determine the choices of ℓ and m which gave the minimum and maximum appearing in part (b). We present a recursive version of this algorithm such that the rectangles of the optimal solution are printed in order from left to right.

```
PrintSoln(π, i, j)
1. Compute π via the algorithm from part (d)
2. if i = 0 or j = 0 then return
3. (ℓ, m) ← π[i, j]
4. PrintSoln(π, ℓ − 1, j − 1)
5. print rectangle j: (x_ℓ, 0), (x_ℓ, y_m), (x_i, y_m), (x_i, 0)
```

Then in order to output the optimal solution we would run the algorithm from part (d) and then run PrintSoln(π, n, k). If $k \leq n$, the optimal solution must contain exactly k rectangles each containing at least one point, otherwise we would have one rectangle spanning at least two points, that could be split into two rectangles of smaller area.

Alternatively, one could iteratively print the rectangles from right to left:

1. Compute π via the algorithm from part (d)
2. $(i, j) \leftarrow (n, k)$
3. while $i > 0$ and $j > 0$ do
4. $\quad (\ell, m) \leftarrow \pi[i, j]$
5. $\quad$ print rectangle j: $(x_\ell, 0), (x_\ell, y_m), (x_i, y_m), (x_i, 0)$
6. $\quad (i, j) \leftarrow (\ell - 1, j - 1)$

(f) For the algorithm given in part (d), the for loop from lines 6-10 takes $\mathcal{O}(i)$ time. The for loop from lines 4-11 thus takes $\Theta(\sum_{j=1}^{k} i) = \Theta(ki)$ time. Thus the outer for-loop takes $\Theta(\sum_{i=1}^{n} ki) = \Theta(kn^2)$ time. Sorting the p_i takes $\mathcal{O}(n \log n)$ time. Thus the algorithm takes $\mathcal{O}(kn^2)$ time total.

The tables C and π have $\Theta(kn)$ entries, thus the space cost is $\Theta(kn)$. Again this assumes that table entries are word-sized.

Ignoring the cost of recursion, PrintSoln costs $\Theta(1)$. When it recurses, PrintSoln decrements both parameters i and j. It follows that PrintSoln(n, k) takes $\mathcal{O}(\min(k, n))$ time. If $k \geq n$, the orignal problem is trivial. In the interesting case that $k < n$, this gives us a cost $\mathcal{O}(k)$. The cost of the iterative algorithm is similar.

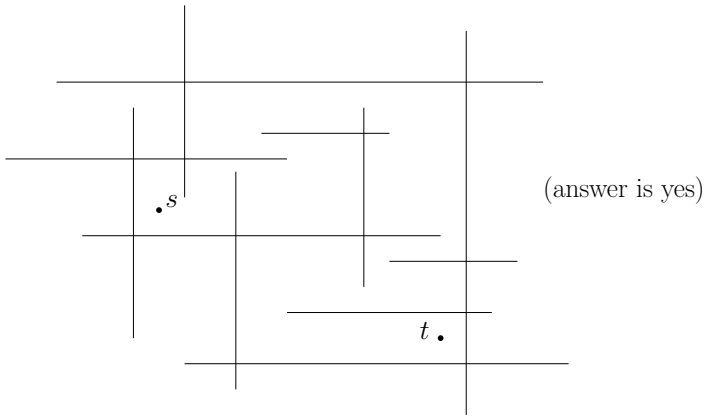
3. [*10 marks*] We are given a set of n line segments in 2D, where each line segment is either vertical (with endpoints (x_i, y_i) and (x_i, y_i') for some x_i, y_i, y_i') or or horizontal (with endpoints (x_i, y_i) and (x_i', y_i) for some x_i, x_i', y_i). We are also given two points $s = (x_s, y_s)$ and $t = (x_t, y_t)$. The problem is to decide whether there is a way to travel from s to t without crossing any of the given line segments.

Describe how to solve this problem in $O(n^2)$ time. [Hint: Break the 2D plane into rectangles in some manner using the line segments and their coordinates, and construct a graph whose vertices correspond to these rectangles.]

Answer: Let ℓ be the number of horizontal line segments and $m = n - \ell$ the number of vertical line segments given as input.

Then let $u_1 < \cdots < u_\ell$, and $v_1 < \cdots < v_m$, where $0 \leq \ell, m < n$, comprise the (by assumption distinct) values of y-coordinates and x-coordinates given by the horizontal and vertical line segments respectively. These points can be obtained by a linear scan of the inputs to build lists of the values u_i and v_j, and to sort the u_i and v_j separately, with cost $\mathcal{O}(\ell \log \ell + m \log m) = \mathcal{O}(n \log n)$.

The equations $y = u_i$, $1 \leq i \leq \ell$ break the 2D plane into $\ell + 1$ rows. Similarly the lines $x = v_j$, $1 \leq j \leq m$ break the plane into $m + 1$ columns. Let $r_{i,j}$ be the rectangle contained in the intersection of the $(i + 1)$-th row and the $(j + 1)$-th column, for $0 \leq i \leq \ell$, $0 \leq j \leq m$.



We will let each rectangle $r_{i,j}$ contain its boundary, such that adjacent rectangles share their common boundary.

We will let the $r_{i,j}$ be the vertices of our undirected graph G. For a pair of adjacent rectangles r and r', we add an undirected edge (r, r') to G if and only if their shared boundary contains a point not on a line segment. E.g. If r and r' share a vertical boundary $x = 5$ from $y = 0$ to $y = 10$, then we would check whether the vertical line segment at $x = 5$ begins at a point higher than $y = 0$ or ends below $y = 10$. If so, we would add an undirected edge from (r, r') to G, otherwise, not. Note that all adjacent rectangles on the periphery share borders in the plane that extend infinitely, and thus will share an edge.

As an aid for students, we present pseudocode for constructing the graph, assuming the values u_i and v_j are already obtained and sorted.

1. $V \leftarrow \{r_{i,j} \mid 0 \le i \le \ell, 0 \le j \le m\}$
2. $E \leftarrow$ Edge set given by adjacency lists
3. for $i \leftarrow 0$ to $\ell - 1$ do //edges over common borders running horizontally
4. add $(r_{i,0}, r_{i+1,0})$ and $(r_{i,m}, r_{i+1,m})$ to E
5. $(x, x') \leftarrow$ left and right x-coord's of horiz. line segment on $y = u_{i+1}$
6. for $j \leftarrow 1$ to $m - 1$ do
7. if $x > v_j$ or $x' < v_{j+1}$ then
8. add $(r_{i,j}, r_{i+1,j})$ to E
9. for $j \leftarrow 0$ to $m - 1$ do //edges over common borders running vertically
10. add $(r_{0,j}, r_{0,j+1})$ and $(r_{\ell,j}, r_{\ell,j+1})$ to E
11. $(y, y') \leftarrow$ bottom and top y-coord's of vert. line segment on $x = v_{j+1}$
12. for $i \leftarrow 1$ to $\ell - 1$ do
13. if $y > u_i$ or $y' < u_{i+1}$ then
14. add $(r_{i,j}, r_{i,j+1})$ to E
15. Return (V, E)

Once we construct our graph we perform a depth-first or breadth-first search starting from a rectangle containing s (if s lies on a boundary, it may be contained by multiple rectangles). If

the search comes across a rectangle containing t, there exists a path from s to t. Otherwise, there does not. We make the reasonable assumption that s and t do not lie on line segments.

The cost of naively adding an edge to an adjacency list is proportional to the length of that adjacency list. Since none of the vertices will have more than four edges, this is $\mathcal{O}(1)$ for our purposes. It follows that the pseudocode above costs $\mathcal{O}(\ell m)$, which in the worst case in $\mathcal{O}(n^2)$. Both BFS and DFS will cost $\mathcal{O}(|V| + |E|)$. We have $|V| = n^2$ and $|E| \leq 4n^2$, thus this cost is $\mathcal{O}(n^2)$, which dominates the cost of the algorithm. Space complexity, similarly, is $\mathcal{O}(n^2)$ in order to store G using adjacency lists (note as $|V| = x^2$, an adjacency graph would require $\mathcal{O}(n^4)$ storage).

4. [*20 marks*] *Programming question.* Given a set S of n circular disks in 2D, we want to partition S into two subsets S_1 and S_2 so that no two disks in S_1 intersect and no two disks in S_2 intersect, or determine that such a partition does not exist. Write a program to solve this problem in $O(n^2)$ time.

   The input to your program is given in the following format:

   $$n \; x_0 \; y_0 \; r_0 \; x_1 \; y_1 \; r_1 \; \cdots \; x_{n-1} \; y_{n-1} \; r_{n-1}$$

   where (x_i, y_i) denotes the center of the i-th disk and r_i denotes the radius of the i-th disk; x_i, y_i, r_i are all positive integers.

   The output should be in the following format:

   $$c_0 \; c_1 \; \cdots \; c_{n-1}$$

   where $c_i = 1$ if the i-th disk is in S_1 and $c_i = 2$ if the i-th disk is in S_2. Do not include any other information or text in the output. The output may not be unique.

   Hand in a printed copy of your program and electronically submit the source file, named a4.cpp or a4.java.

   [Hint: the problem reduces to bipartiteness testing, or 2-coloring, which can be solved by either BFS or DFS, as discussed in class. In the underlying graph, the vertices are the disks, and there is an edge between disks i and j if the two disks intersect, i.e., $\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \leq r_i + r_j$ (the square root operation can be avoided by squaring both sides). Since the desired time complexity is $O(n^2)$, you can use a simpler adjacency matrix representation of the graph instead of adjacency lists; in fact, you can even avoid constructing the graph explicitly by modifying BFS/DFS directly.]