University of Nevada, Las Vegas Las Vegas Computer Science 477/677 Fall 2019

Answers to Assignment 8 Due Monday November 18, 2019

1. (Cuckoo hash table problem.) We define two hash functions, h_1 and h_2, on 4-letter words as follows. Define [letter] to be the place, from one to 26, of that letter in the Roman alphabet, that is:

   [A] = 1, [B] = 2, ... [M] = 13 ... [Z] = 26.

   Let $h_1(\text{word}) = ([\text{first letter}] * 2 + [\text{second letter}] * 4)\,\%\,7$
   and $h_2(\text{word}) = (h_1(\text{word}) + ([\text{third letter}] + [\text{fourth letter}] * 5)\,\%\,6 + 1)\,\%\,7$

   For example, if the work is "John," we have:

   $h_1(\text{John}) = ([\text{J}] * 2 + [\text{O}] * 4)\,\%\,7 = (10 * 2 + 15 * 4)\,\%\,7 = 3$
   $h_2(\text{John}) = (h_1(\text{John}) + ([\text{H}] + [\text{N}] * 5)\,\%\,6 + 1)\,\%\,7 = (3 + (8 + 70)\,\%\,6 + 1)\,\%\,7 = 4$

   (a) Find a word such that h_1 and h_2 of that word are equal.

   There is no such word. The two hash functions are designed so that they will never be equal.

   (b) Enter the following 4-letter words into a cuckoo hash table of size 7, using the two hash functions h_1 and h_2 defined above. Show the steps. That is, don't erase ejected words, simply cross them out.

	h_1	h_2
Faye	2	5
Trip	0	6
Thom	2	5
Suzy	3	5
Bess	3	4
Leah	2	1
Ping	5	0

0	~~Trip~~	Ping				
1	Leah					
2	~~Faye~~	~~Thom~~	~~Faye~~	~~Leah~~	~~Thom~~	Faye
3	~~Suzy~~	~~Bess~~	Suzy			
4	Bess					
5	~~Faye~~	~~Suzy~~	~~Thom~~	~~Faye~~	~~Ping~~	Thom
6	Trip					

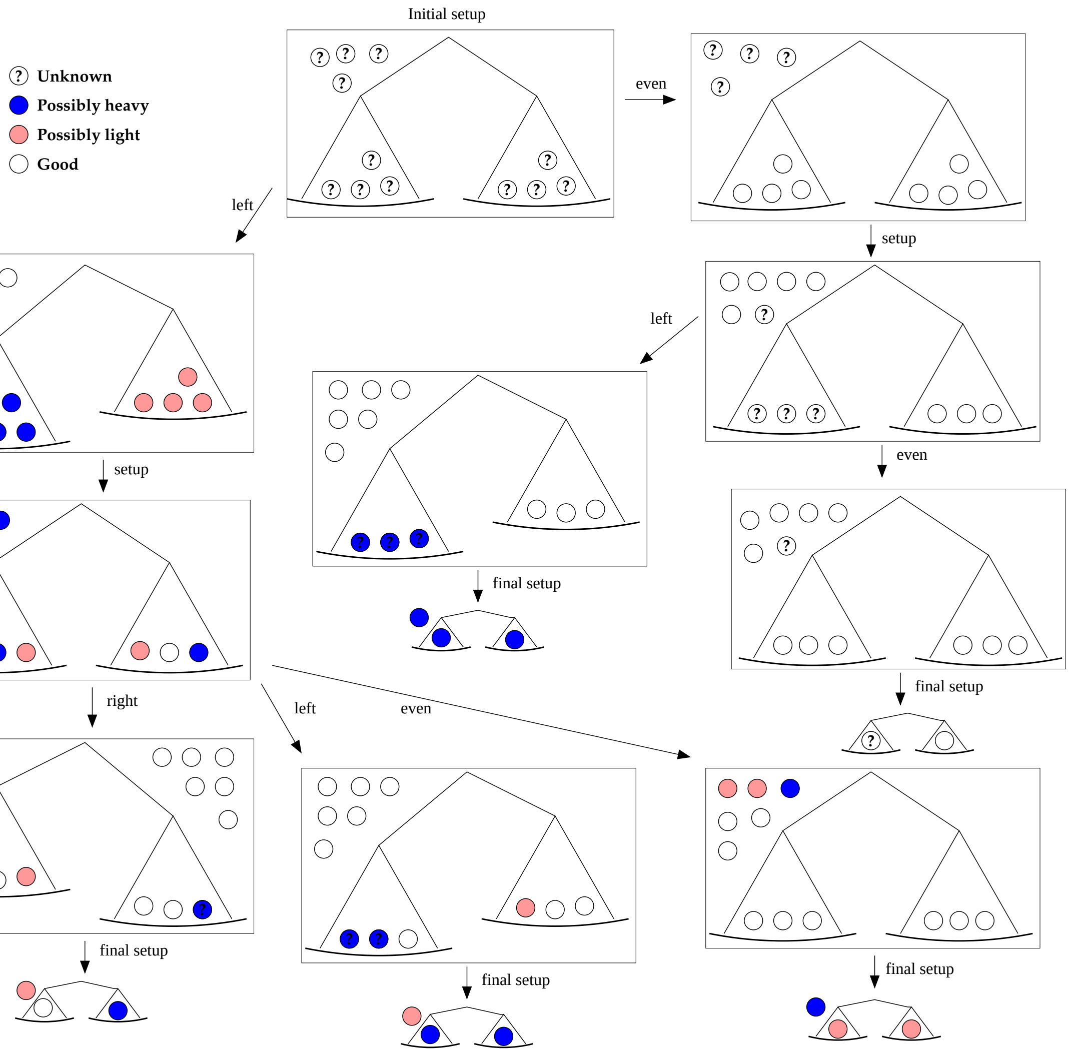
2. You are given 12 pennies, all of which are exactly the same weight, except for one "bad" penny, which is either slightly too heavy or slightly too light. You have a balance scale; if you put pennies on each side, the scales gives you one of three answers: the left side is heavier, the right side is heavier, or the two sides have equal weight.

   (a) Give an algorithm which deternies which penny is bad, and whether it is too heavy or too light, using only three weighings.

   There are 24 possible outcomes. Three weighings can, in theory, distinguish $3^3 = 27$ outcomes, so you should be able to do it, and in fact, you can. The first thing to do is to put k pennies on each side, but what is k? There must be at most 9 outcomes after the first weighing. If $k = 3$ and the scale balances, the bad penny is one of the 6 remaining, so there are 12 remaining outcomes. If $k = 5$ and the scale does not balance, there are 10 remaining outcomes. So k can be neither 3 nor 5. If $k = 4$ and the scale balances, there are 8 remaining outcomes, while if the scale does not balance, there are also 8 remaining outcomes. Thus the correct first weighing is to put 4 on each side.

   But what do you do next? The harder case is when the scales do not balance. Suppose the left side goes down and the right side up. Then either one of the 4 "left" pennies is too heavy, or one of the 4 "right" pennies is too light. The other 4 pennies are known to be "good."

You put two "left" pennies and one "right" penny on the left, and put one "left" penny, one "right" penny, and one "good" penny on the right. What then?



Solution for Twelve Pennies

(b) Considier the same problem with 13 pennies. There are 26 possible outcomes, so you should be able to do it in three weighings, but you can't. Use information theory to prove that it's impossible.

If you put 4 or fewer pennies on each side in the initial setup and it balances, there are still at least 10 possibilities. If you put 5 our more pennies on each side and it does not balance, there are still at least 10 possibilites. With two remaining weighings, you can only distinguish 9 possibilities, so the problem has no solution.

(c) Suppose there are 13 pennies, one of which is too heavy or too light, and you also have a 14th penny which you know is good. Show how you can find the bad penny and determine whether it's too heavy or too light in three weighings.

Put 5 of the thirteen questionable pennies on one side, four on the other along with the good penny.

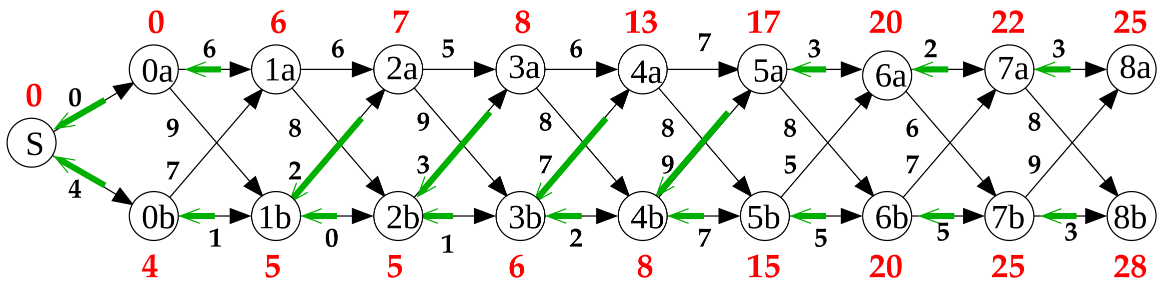
3. It is rather easy to compute the maximum (or minimum, sum, or product) of a list of n numbers in $O(\log n)$ time using n processors, by the "tournament" method. Can you show how to do it in $O(\log n)$ time using $\frac{n}{\log n}$ processors?

   Partition the list into sublists, each of size $\log n$. Assign one processor to each sublist to find the (local) maximum of that sublist, for a total of $\frac{n}{\log n}$ processors. Use those same processors to find the maximum of all those local maxima in $O(\log n)$ time.

   Problems 4 and 6 can also be worked with $\frac{n}{\log n}$ processors in $O(\log n)$ time.

4. A layered graph is a graph whose vertices are partitioned into consecutive layers, and where each edge only connects adjacent layers. For example, the graph G given below is a weighted direected layered graph, with nine layers.

   (a) Use dynamic programming to work the single source shortest path problem for G, where S is the source. Show minimum lengths and backpointers.



	S	0a	0b	1a	1b	2a	2b	3a	3b	4a	4b	5a	5b	6a	6b	7a	7b	8a	8b
length	0	0	4	6	5	7	5	8	6	13	8	17	15	20	20	22	25	25	28
back		S	S	0a	0b	1b	1b	2b	2b	3b	3b	4b	4b	5a	5b	6a	6b	7a	7b

Alternatively, back(6a) = 5b.

You know how to multiply matrices. For example: $\begin{pmatrix} 2 & -1 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} 1 & 2 \\ 0 & -1 \end{pmatrix} = \begin{pmatrix} 2 & 5 \\ 1 & 2 \end{pmatrix}$

This is the usual kind of matrix multiplication, called "$(+,*)$" matrix multiplication, because each entry in the product matrix is the sum $(+)$ of products $(*)$. That is

$$\begin{array}{rr} 2*1+(-1)*0=2 & 2*2+(-1)*(-1)=5 \\ 1*1+0*0=1 & 1*2+0*(-1)=2 \end{array}$$

But there is another very useful kind, called $(\min,+)$ matrix multiplication. Each entry of the product matrix is the minimum of sums.[1] For example: $\begin{pmatrix} 2 & -1 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} 1 & 2 \\ 0 & -1 \end{pmatrix} = \begin{pmatrix} -1 & 1 \\ 0 & -1 \end{pmatrix}$ Since

$$\begin{array}{rr} \min\{2+1,(-1)+0\}=-1 & \min\{2+2,(-1)+(-1)\}=1 \\ \min\{1+1,0+0\}=0 & \min\{1+2,0+(-1)\}=-1 \end{array}$$

[1]We can use any two operators, such as $(\max,+)$, provided the second operator distributes over the first operator

Relating (min,+) Multipication to the Layered Graph Problem. We can define the initial vector of our problem to be the vector of minimum costs to the 0th layer, namely $\text{Vec}[0] = (0, 4)$. After working the dynamic program, we know that $\text{Vec}[1] = (6, 5)$.

Let $M[0,1] = \begin{pmatrix} 6 & 9 \\ 7 & 1 \end{pmatrix}$, the matrix representing the arcs from Layer 0 to Layer 1. Then $\text{Vec}[0]M[0,1] = (0,4)\begin{pmatrix} 6 & 9 \\ 7 & 1 \end{pmatrix} = (6,5) = \text{Vec}[1]$. Similarly, $\text{Vec}[0]M[0,1]M[1,2]M[2,3]M[3,4] = (13,8) = \text{Vec}[4]$.

This formulation does not save any time. However, we can shortcut the computation by using n parallel processors to compute the product $M[0,i] = M[0,1]M[1,2]\cdots M[i-1,i]$ for each $i \leq n$, in $O(\log n)$ time, where $n = 8$ in our example. Then $Vec[0]M[0,i] = \text{Vec}[i]$ For example

$$M[0,2] = M[1]M[2] = \begin{pmatrix} 6 & 9 \\ 7 & 1 \end{pmatrix}\begin{pmatrix} 6 & 8 \\ 2 & 0 \end{pmatrix} = \begin{pmatrix} 11 & 9 \\ 3 & 1 \end{pmatrix}$$

$$\text{Vec}[0]M[0,2] = (0,4)\begin{pmatrix} 11 & 9 \\ 3 & 1 \end{pmatrix} = (11,5) = \text{Vec}[2]$$

(b) Using the example layered graph G, Compute the following $(\min, +)$ matrix products. (We have already computed $M[0,2]$.)

$M[2,4] = M[2,3]M[3,4] = \begin{pmatrix} 5 & 9 \\ 3 & 1 \end{pmatrix}\begin{pmatrix} 6 & 8 \\ 7 & 2 \end{pmatrix} = \begin{pmatrix} 11 & 11 \\ 8 & 3 \end{pmatrix}$

$M[4,6] = M[4,5]M[5,6] = \begin{pmatrix} 7 & 8 \\ 9 & 7 \end{pmatrix}\begin{pmatrix} 3 & 8 \\ 5 & 5 \end{pmatrix} = \begin{pmatrix} 10 & 13 \\ 12 & 12 \end{pmatrix}$

$M[6,8] = M[6,7]M[7,8] = \begin{pmatrix} 2 & 6 \\ 7 & 5 \end{pmatrix}\begin{pmatrix} 3 & 8 \\ 9 & 3 \end{pmatrix} = \begin{pmatrix} 5 & 9 \\ 10 & 8 \end{pmatrix}$

$M[0,4] = M[0,2]M[2,4] = \begin{pmatrix} 11 & 9 \\ 3 & 1 \end{pmatrix}\begin{pmatrix} 11 & 11 \\ 8 & 3 \end{pmatrix} = \begin{pmatrix} 17 & 12 \\ 9 & 4 \end{pmatrix}$

$M[4,8] = M[4,6]M[6,8] = \begin{pmatrix} 10 & 13 \\ 12 & 12 \end{pmatrix}\begin{pmatrix} 5 & 9 \\ 10 & 8 \end{pmatrix} = \begin{pmatrix} 15 & 19 \\ 17 & 20 \end{pmatrix}$

$M[0,8] = M[0,4]M[4,8] = \begin{pmatrix} 17 & 12 \\ 9 & 4 \end{pmatrix}\begin{pmatrix} 15 & 19 \\ 17 & 20 \end{pmatrix} = \begin{pmatrix} 29 & 32 \\ 21 & 24 \end{pmatrix}$

$\text{Vec}[8] = \text{Vec}[0]M[0,8] = (0,4)\begin{pmatrix} 29 & 32 \\ 21 & 24 \end{pmatrix} = (25, 28).$

(c) For an instance of the problem of arbitrary size, can you compute $\text{Vec}[i]$ for all i in $O(\log n)$ time with n processors? (Hint: Yes.)

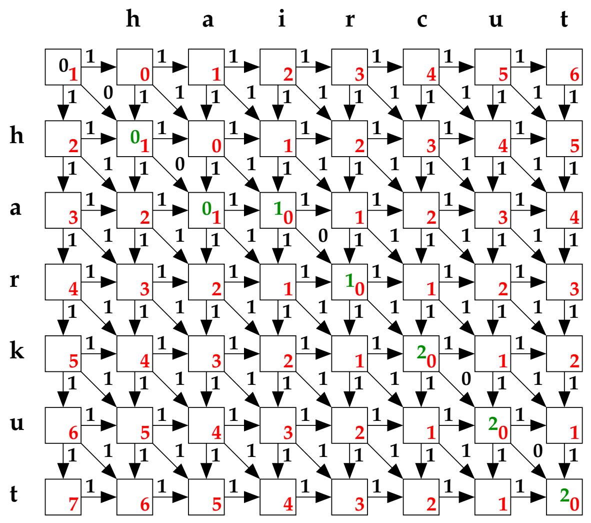
5. This is an "extra difficulty" problem. The Levenstein edit distance between two strings of lengths m and n, respectively, can be found in $O(nm)$ time using the dynamic program we went over in class. However, filling out that huge matrix seems awfully wasteful if the edit distance is small.

   Let h be the actual Levenstein distance between two words of lengths n and m. Use the A^* algorithm to find the edit of minimum Levenstein cost in $O(n(h+1)\log n)$ time.

The standard dynamic program for finding the distance between strings x and y of lengths n and m, respectively is to construct D, an $(n+1)\times(m+1)$ array of integers, where $D[i,j]$ is the edit distance from the prefix $x[0..i]$ to the prefix $y[0..j]$. $D[0,0]$ is initialized to 0, and $D[n,m]$ contains the edit distance. The entries are defined by

$$D[i,j] = \min \begin{cases} D[i-1,j]+1 \\ D[i,j-1]+1 \\ D[i-1,j-1]+\epsilon \end{cases}$$

where $\epsilon = 0$ if the $i^{\rm th}$ symbol of x equals the $j^{\rm th}$ symbol of y, 0 otherwise. Thus, finding the edit distance is an instance of the single pair minpath problem, where there are $(n+1)(m+1)$ vertices and $3nm+n+m-1$ weighted arcs, as shown in the figure, where we compute the edit distance from "`harkut`" to "`haircut`." The minimum path consists of the vertices with green numerals, and has length 2.



Using the A* Algorithm. It is a waste of computation time to fill in the entire matrix, since all the important action takes place near the main diagonal. We can define a heuristic $h[i,j] = |n-m-i+j|$. Values of h are shown in red in the figure. That heuristic would be the value of $D[i,j]$ if both x and y were over a unary alphabet, and hence is guaranteed to be no greater than $D[i,j]$. Using this heuristic, the A^* algorithm computes the edit distance, which is 2, much faster, since it only computes values near the main diagonal.

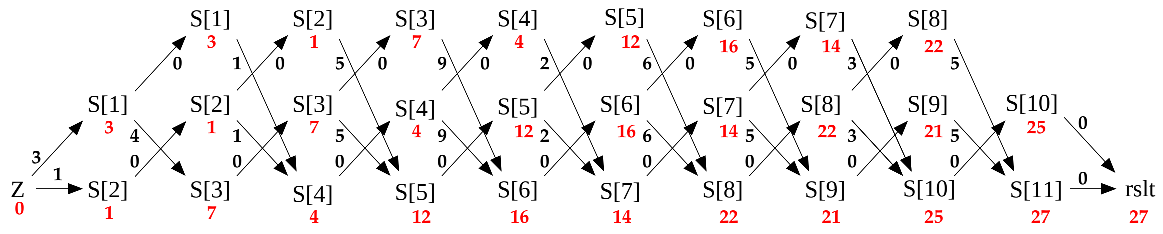
6. This is an "extra difficulty" problem. In the last homework, one problem was to design a dynamic programming algorithm that inputs a sequence of integers, and finds that subsequence which has the largest total, subject to the condition that no two consecutive terms of the original sequence are in the subsequence. For example, if the input sequence is 3,1,4,1,5,9,2,6,5,3,5, your algorithm will find the subsequence 3,4,9,6,5, whose terms add up to 27.

   As we have shown in class, the problem can be worked in $O(n)$ sequential time. The problem can be reduced to the problem of finding the shortest path through a weighted layered directed graph, as in problem 4 of this homework. We can find the best subsequence in $O(\log n)$ time using n processors.

   Let $x[1], \ldots x[n]$ be the sequence. Let $S[i]$ be the maximum sum of any legal subsequence which includes the $i^{\rm th}$ term of the sequence. For convenience, we write $S[0] = 0$. Recall the sequential algorithm:

```
S[0] = 0;
S[1] = x[1];
S[2] = x[2];
for(int i = 3; i <= n; i++)
 S[i] = max(S[i-3]+x[i],S[i-2]+x[i});
rslt = max(S[n-1],S[n]);
cout << "The maximum total of any legal subsequence is " << rslt << endl;
```

One layer of the graph contains those variables that the algorithm needs for later steps. Thus, a layer contains up to three nodes. For example, one layer contains $S[2]$, $S[3]$, and $S[4]$, since $S[2]$ and $S[3]$ are used in the next step, and $S[4]$ will be needed later.



In the figure, Z has the value zero, and `rslt` is the result, the maximum length of a path starting at Z.

In the matrices which represent steps, since we are trying to maximize weights, we use $-\infty$ for the weight of an edge which does not appear in the figure.[2] $\begin{pmatrix} S[1] & S[2] & S[3] \end{pmatrix} = \begin{pmatrix} 3 & 1 & 7 \end{pmatrix}$, thus

$$\begin{pmatrix} S[2] & S[3] & S[4] \end{pmatrix} = \begin{pmatrix} 3 & 1 & 7 \end{pmatrix} \begin{pmatrix} -\infty & -\infty & 1 \\ 0 & -\infty & 1 \\ -\infty & 0 & -\infty \end{pmatrix} = \begin{pmatrix} 1 & 7 & 4 \end{pmatrix}$$

Similarly, by multiplying $\begin{pmatrix} S[1] & S[2] & S[3] \end{pmatrix}$ by

$$\begin{pmatrix} -\infty & -\infty & 1 \\ 0 & -\infty & 1 \\ -\infty & 0 & -\infty \end{pmatrix} \begin{pmatrix} -\infty & -\infty & 5 \\ 0 & -\infty & 5 \\ -\infty & 0 & -\infty \end{pmatrix} \begin{pmatrix} -\infty & -\infty & 9 \\ 0 & -\infty & 9 \\ -\infty & 0 & -\infty \end{pmatrix} \begin{pmatrix} -\infty & -\infty & 2 \\ 0 & -\infty & 2 \\ -\infty & 0 & -\infty \end{pmatrix} = \begin{pmatrix} -\infty & 10 & 3 \\ 5 & 10 & 7 \\ 5 & 9 & 7 \end{pmatrix}$$

we obtain $\begin{pmatrix} 12 & 16 & 14 \end{pmatrix} = \begin{pmatrix} S[5] & S[6] & S[7] \end{pmatrix}$.

Matrices on the left and right end of the computation can be smaller. The entire example computation can be written out as the $(\max, +)$ product of 12 matrices of various shapes.

$$\begin{pmatrix} 0 \end{pmatrix} \begin{pmatrix} 3 & 1 \end{pmatrix} \begin{pmatrix} 0 & -\infty & 4 \\ -\infty & 0 & -\infty \end{pmatrix} \begin{pmatrix} \text{seven} \\ 3 \times 3 \\ \text{matrices} \end{pmatrix} \begin{pmatrix} -\infty & 5 \\ -\infty & 5 \\ 0 & -\infty \end{pmatrix} \begin{pmatrix} 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 27 \end{pmatrix}$$

For an arbitrary input sequence of length n, the computation can be done by n parallel processors in $O(\log n)$ time, since matrix multiplication is associative.

[2]As opposed to ∞, which is used when we are trying to minimize.