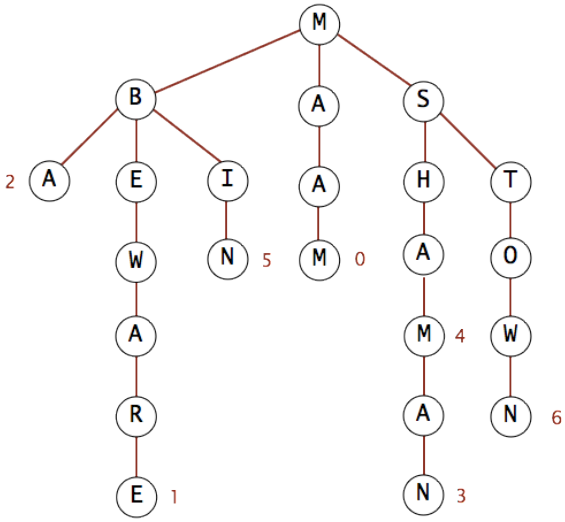
Spring 2013 Midterm Solutions (Beta Edition)

1. **Analysis of Algorithms.**
   a. OBESPIYWM
   b. OBEYSWIMP
   c. This graph is not a DAG, and thus there is no topological order.
2. MSTs.
   a. A-B A-H A-F B-E C-G C-I D-E C-E
   b. A-B A-H A-F B-E D-E E-C C-G C-I
   c. $x \leq 120$. Given a loop in a graph, we can always throw away the largest edge. We know all the rest of the edges are part of the MST from parts a and b.
3. **TSTs.**
   a. Height of TST (to the right) is 6.
   b. Worst case insertion order is to get SHAMAN or BEWARE to the bottom level. For example, A, BEWARE, IN, MAAM, TOWN, SHAMAN.



4. **Analysis of algorithms.**
   a. Linear: Given
   b. **G.** Each iteration of the inner loop is quadratic in the outer loop variable. The simplest way to do this is to realize we're just summing $\sum i^2$, which will just be $O(N^3)$ if we use the integration trick.
   c. **H.** Each iteration spawns two iterations. Thus by the time we get to the bottom level (where n=1), we've produced 2^n total calls to f3.
   d. **D.** This is the similar to the pattern that we saw in Mergesort and Quicksort, except that each recursive call does only a constant amount of work instead of a linear amount. It is the same as the pattern for bottom up heapification. At the top level, we do 1 unit of work; at the 2nd level, we do 2 units of work; at the 3rd level, we do 4 units, etc. The total amount of work is thus given by $1 + 2 + 4 + 8 + \cdots + N$. This sum is linear in N.
   e. **E.** This is the exact same pattern as Mergesort and Quicksort. If you want to think of it as a sum, then it's $N + N + \cdots N$, where are $\log_2 N$ summands.
   f. **B.** After the first iteration, $i = 2$. After the second, $i = 2^2$. After the third, $i = 2^{2^2}$, etc. This takes Log* N steps to reach N. If you weren't totally sure, you could have also observed that Log* N was the only answer between constant and Log N.
5. **Shortest Paths.**
   a. 35174
   b. Vertex 6 is the next to be relaxed, reducing vertex 4's distance to 1.0, and setting edgeTo[4] to 6.
   c. Choosing any weight > 10 ensures that vertex 6 relaxes before vertex 4, and thus edge $6 \rightarrow 4$ is guaranteed to get utilized. Alternate answer was to choose any weight ≤ 2, in which case the edge $6 \rightarrow 4$ is irrelevant.

d. Worksheet: 

6.

a. BBABBCBBABBB

b. No matter how many times you click `find next`, there's no need to rebuild the KMP DFA if things are as optimized to be as fast as possible. Thus we need only construct the DFA once.

c. This problem was a bit underspecified so we took a number of different answers. The two things that were not specified in the problem were:

   i. Given a text AAAAAAAAAA and a pattern AAA, would the first match start at the second A, or at the underlined A? Determines if best case is M+F or MF.

   ii. Suppose the user clicks `find next` after the text editor has scanned all the way to the end, will it start at the beginning of the text file after the next click of the `find next` button? Determines if worst case is FN or N.

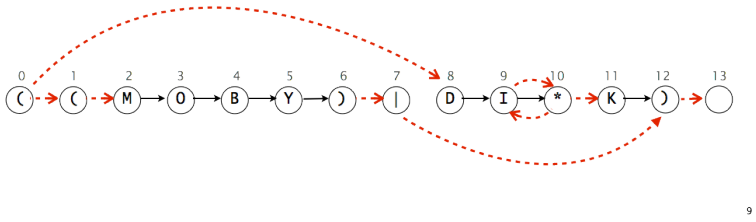
Given those under-specifications, we accepted:

Best Case: MF + Q, M+F
Worst Case: FN + Q, N + Q

Here Q is the construction time that you specified in part b. If you omitted Q, that was fine, since it would be dominated by the search time in all cases.

7. Regular Expressions.

a.



9

b.

   i. `(AB)*`

   ii. `(ABD*)+` or equivalently `(ABD*)(ABD*)*`

c. The DFA can be exponentially large (which will naturally take exponential time to construct)

8. MaxFlow, Reductions.

a. 11

b. 12

c. s, B

d.

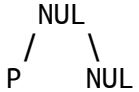
   i. Y. Reduction takes N^2 time.

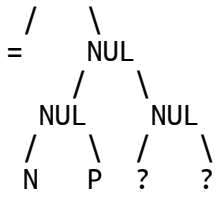
ii. N. There could be other solutions to OctopusWrangling.
iii. N. A lower bound is not a proof of existence. For example, I say that all NP complete problems can be solved in constant time or better. That is a lower bound.
iv. N. Reduction doesn't go backwards. Just because I know how to turn off the lights at my apartment doesn't mean I know how to disable the power grid to the city.

9. True. False. Pain.

a. F. If you think about how Kruskal's or Prim's algorithms work, all that matters is the relative order.
b. T. If this was false, then you could use Dijkstra's algorithm to handle negative edge weights just fine.
c. F. Finding a Hamilton path just requires that you find a topological sort. Alternate answer: There is no such thing as a Hamilton tour on a DAG (since you have to come back to the start!)
d. True OR False, depending on your interpretation of "algorithm completes". There is an infinitely small chance that you may end up never completing, in which case the statement is false. If you disregard this silly (but technically real) possibility, then the algorithm works fine, since it's a most direct application of the cut property.
e. F. Even if you ignore the silly possibility of getting infinitely unlucky, this algorithm still won't work. Consider the case where you have 4 vertices where the minimum spanning tree would be a Z. The top and bottom pairs of points will pick each other, and there will never be a case where they are able to reach out to their buddies on the other side. ☹
f. T. Rerunning Dijkstra's algorithm V times is basically the same thing as Bellman-Ford. You end up relaxing every vertex V times (once more than you need to in the worst case).
g. T. If the answer to "Does there exist a tour of city-set Q of length less than 10,000" is yes, then the proof (i.e. a putative tour of length less than 10,000) can be verified in polynomial time, we simply check the length of the proof in the tour. This problem is thus in NP. Note, it is NOT in P.
h. F. If the answer to "Is tour X the shortest tour of city-set Q?" is yes, we have no tour to check to prove that the proposition is true. This problem is not in P or NP (though it is in yet another complexity class we haven't discussed called co-NP).
i. F. It is impossible to create an algorithm for finding the Kolmogorov complexity of a given string.
j. F. Reducing problem X to 3SAT just means that the problem can be solved using 3SAT. To be NP complete, you have to shown that X is in NP and that every problem in NP reduces to X. Another way of thinking about this is that you have to show that X is as hard as the hardest problem in NP. Reducing X to 3SAT just says that 3SAT is at least as hard as X (wrong direction).

10. Recursive Code / BinaryStdIn.





11. Compression.

a. CHEEZ

b. 1 million as: LZW (gets shorter once codeword represents more than 8 as)
abacad…fg: Huffman, because no codeword used twice

c. We observe that if $L = n(n+1)/2$, then the number of bits used is exactly $8n$. In this case, we have that $n = \frac{-1+\sqrt{1+8L}}{2}$. The output of LZW thus uses $-4 + 4\sqrt{1+8L}$ bits. In ASCII, we had $8L$ bits, resulting in a compression ratio order of growth of $\frac{\sqrt{L}}{L}$.

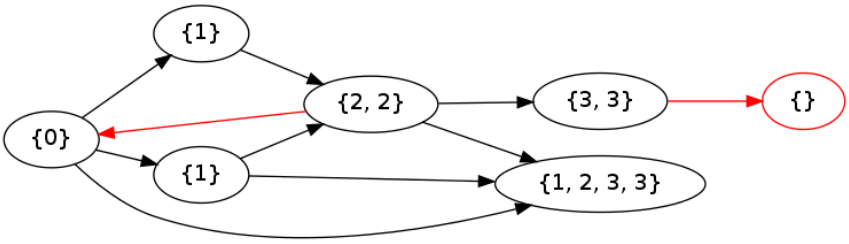
d. Printing out π requires only a finite length program (albeit infinite time). Printing out only π's first N digits requires $\log(N)$ bits to specify the number of digits of π to print.

12. Graph Algorithm Design. Since we're looking for shortest paths, it's pretty clear we're going to need to use some sort of BFS.

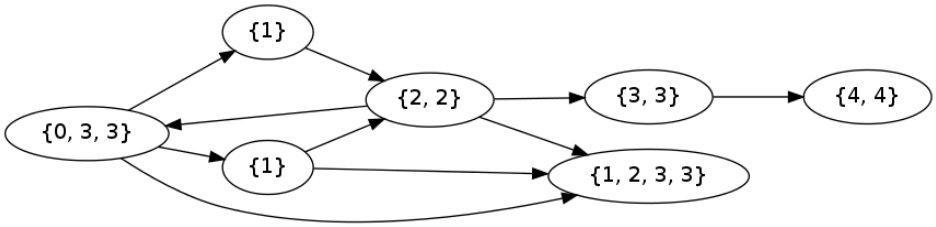
The first insight in this problem is that we don't need to worry at all about cycles. Since we're dealing with shortest paths, we know that any path involving a cycle will not count as a shortest path.

Another useful insight is that if we're using BFS, every incoming edge to vertex X that could possibly be part of a shortest path to X will be processed before vertex X is dequeued. This means that we can move on from vertex X as soon as X is dequeued (i.e. there's no need to wait for every incoming edge, so we can proceed in normal BFS order).

Given the two insights above, we have a pretty natural (but possibly slow) algorithm that we can use as a starting point. We simply give every vertex an empty Bag, where each entry in the Bag represents the length of a particular path to that vertex. We start the source vertex off with a 0 in its bag, and every other bag starts empty. When an edge $a \to b$ is processed (using BFS), the list inside a is incremented by 1 and appended to b's list. For example, consider the graph below, where vertices in red have not yet been dequeued, and edges in red have not yet been processed.

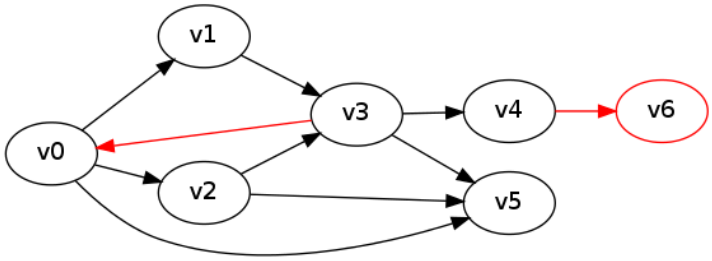


When this algorithm completes, we're left with the graph shown below. To determine the number of shortest paths to a node, we simply count the number of occurrences of the minimum inside the node.



This algorithm is too slow, because in a fully connected graph, we'd have to build V lists of length V, which is nonlinear. To improve performance, we can take advantage of the fact that we don't really need to keep anything other than the number of shortest paths for each node. We can accomplish this by adding a third array to BFS called `pathCount` of length V.

In this improved algorithm we start by setting pathCount inside each node to 0. When processing an edge $a \rightarrow b$, `if distTo[b] = distTo[a] + 1`, we increment `pathCount[b]` by `pathCount[a]` (since vertex a provides a new set of shortest `distTo[a]`). If `distTo[b] > distTo[a] + 1`, we've found a new shortest path, and set `pathCount[b]` to `pathCount[a]`. If `distTo[b] < distTo[a] + 1`, we do nothing since the path(s) under consideration is too long to be considered. Think of distTo[b] as the "old path", and distTo[a] + 1 as the "new path".



For example, for the first graph above, our graph state would be given by:

	distTo	edgeTo	pathCount
v0	0	(don't care)	1
v1	1	(don't care)	1
v2	1	(don't care)	1
v3	2	(don't care)	2
v4	3	(don't care)	2
v5	1	(don't care)	1
v6	∞	(don't care)	0

As an example, when the edge from $v4 \rightarrow v6$ is processed, the algorithm will see that `distTo[v6] > distTo[v4] + 1`, thus `pathCount[v6]` will be set equal to `pathCount[v4]`.

13. F

a. Only Rabin-Karp can be generalized nicely. Attempts to build a KMP DFA that inhabits multiple states results in a run time identical to just searching for K patterns independently. Attempts to build a Boyer-Moore like algorithm that individually tracks matches with each pattern also have runtime equal to searching for K patterns independently.
b. The simplest approach is to simply pre-hash all the patterns and store hash in a set. This set can be implemented either as a hash set or as an LLRB. In Java, we could do this by creating a Set<Integer>.
c. The procedure is precisely the same as normal Rabin-Karp, except that instead of checking equality, one checks to see if the Set contains the hash of the current M characters.
d. Calculating K hashes takes KM time. If we use a hash set, then storing these K hashes takes K time on average under the uniform hashing assumption. Construction thus takes a total of KM time. If you stated worst case performance, then $K \log K + KM$ was also acceptable for a hash table.

   When processing the string, assuming that $N \gg M$, we have to perform N set accesses. If we're using a hash set, this results in a total run time of N, or $N \log K$ if you consider the worst case under the uniform hashing assumption.

   If using an LLRB of hashes, then the build and process time are $KM + K \log K$ and $N \log K$ respectively.

14. Legume grime pop.

a. A hash table `originals` that maps strings to Bags of strings.

   Also acceptable: any symbol table with string keys that can be constructed in time linear in the number of strings, e.g., an R-way trie. A red-black tree or TST would take $O(N \log N)$ time to construct.

b. Key idea: avoid generating every permutation of each word (which is $O(L!)$).

   - Read each word `word` from input, insertion sort it to generate `dorw`, and add `dorw` to the Bag `originals[w]` (create the Bag if it doesn't exist). Keep track of the maximum Bag length. $O(NL^2)$.
   - Iterate through `originals`. Stop when you find a Bag whose length equals the maximum. Print the words in this Bag. $O(N)$.

Sorting each word using key-indexed counting is asymptotically faster at $O(NLR)$, where R is the alphabet size, but insertion sort which is $O(NL^2)$ is likely faster in practice for typical English words.

c. Key idea: avoid an exponential search for all possible word ladders. Here are two solutions with varying tradeoffs between simplicity, speed and generalizability.

Both solutions use a subroutine `neighbors(dorw)` that assumes `originals` already exists – Given a dorw of length k, generate the k dorws of length $k-1$ and return the ones that are valid (i.e., those that are keys in `originals`). $O(L^2)$.

Solution 1.

- Create a DAG where nodes are dorws and there is an edge from each dorw to each of its neighbors. N invocations of `neighbors`, $O(NL^2)$.
- Find the longest path in this DAG.
    - This can be done by creating a virtual root node with an edge to every root (finding the roots is $O(N)$), assigning a weight of -1 to each edge, and computing the *shortest* paths from the virtual root using topological sort. $O(N)$.
- Look up the dorws on this longest path (in reverse) in `originals` and print a sequence of original words.

Solution 2.

- Initialize a hash table `rung_height` from dorws to `int`s with all values set to 0. The `rung_height` of `dorw` is the max rung height of `dorw` in any anagram ladder that it can appear in. Lowest rung is 0.
- Create a sorted array of dorws in increasing order of length. $O(N)$ by key-indexed counting.
- For each dorw `dorw` In this array:

```
    rung_height[dorw] = max(rung_height[nbr] for nbr in
neighbors[dorw])
```

    // if `neighbors[dorw]` is empty do nothing, as `dorw` must be the bottom rung in any ladder. Note that the neighbors have already been processed because of sortedness.

  N invocations of `neighbors`, $O(NL^2)$.
- Find the dorw with maximum `rung_height`, and iteratively find a sequence of neighbors with `rung_height` of each neighbor one less than the previous. $O(N + poly(L))$.

- Look up the dorws in this ladder (in reverse) in `originals` and print a sequence of original words.

  Solution 1 is more elegant is but probably slower in practice (and consumes more memory) due to graph creation, despite having the same asymptotic runtime as Solution 2.