CPSC 320 2019S2: Assignment 1

Please submit this assignment via GradeScope at `https://gradescope.com`. Be sure to identify everyone in your group if you're making a group submission (which we encourage!).

Submit by the deadline **Tuesday July 9 at 11PM**. For credit, your group must make a **single** submission via one group member's account, marking all other group members in that submission **using GradeScope's interface**. Your group's submission must:

- Be on time.
- Consist of a single, clearly legible file uploadable to GradeScope with clearly indicated solutions to the problems. (PDFs produced via LaTeX, Word, Google Docs, or other editing software work well. Scanned documents will likely work well. **High-quality** photographs are OK if we agree they're legible.)
- Include prominent numbering that corresponds to the numbering used in this assignment handout. Put these **in order** starting each problem on a new page, ideally. If not, **very clearly** and prominently indicate which problem is answered where!
- Include at the start of the document the **ugrad.cs.ubc.ca e-mail addresses** of each member of your team. Please **do not** include names on the assignment. If you don't mind private information being stored outside of Canada and want an extra double-check on your identity, include you student number rather than your name.
- Include at the start of the document the statement: "All group members have read and followed the guidelines for academic conduct in CPSC 320. As part of those rules, when collaborating with anyone outside my group, (1) I and my collaborators took no record but names (and GradeScope information) away, and (2) after a suitable break, my group created the assignment I am submitting without help from anyone other than the course staff." (Go read those guidelines!)
- Include at the start of the document your outside-group collaborators' ugrad.cs.ubc.ca IDs, but **not** their names. (Be sure to get those IDs when you collaborate!)

1 Trading Lunch Bags

A standard approach in text analysis – for machine learning, machine translation, spam detection, and so on – is transforming a document into a "bag of words" representation. A "bag of words" is a data structure that maps the unique words in the document to the number of times each one appears. For example, the phrase "For lunch I would like a chocolate bar, a jelly donut, and a chocolate donut" would become a 'bag' like [a:3, and:1, bar:1, chocolate:2, donut:2, for:1, I:1, jelly:1, like:1, lunch:1, would:1], with each word and its number of appearances. (We put the words in sorted order, but that's not necessary.) Note that the phrase has 15 words total but only 11 unique words, because "a" occurs three times and "chocolate" and "donut" occur twice.

You want to create an algorithm that, given a list of words W containing m occurrences of n unique words,, produces a complete list of tuples (pairs) of words and their numbers of occurrences. The result should have n entries (one for each unique word) and the total number of occurrences across all entries should be m, but these entries can be in any order.

1. The following algorithm gives a brute force approach to convert a list into a big of words. Give and briefly justify a good asymptotic bound on the worst-case runtime of this algorithm in terms of m (the length of W). You may assume that deleting from W and appending to `bagOfWords` both take $O(1)$ time.

```
PROCEDURE ConvertToBag_BF(W):
   bagOfWords = [ ]
   While W is not empty:
      Let w be the first element of W
      Count the number of occurences of w in W (by scanning W from start to
          end) and call this result c_w
      Append w:c_w to bagOfWords
      Delete all occurrences of w from W
   Return bagOfWords
```

2. Give and briefly justify a good asymptotic bound on the **best-case** runtime of `ConvertToBag_BF` in terms of m.

Making a Hash of the Bag

Next we consider using a hash table for this problem. Here are some useful standard operations for a hash table using chaining:

- `ConstructHashTable(e)` creates and returns an empty hash table containing e entries. Takes time proportional to e.
- `Contains(T, k)` returns true if hash table T contains an entry for the key k and false otherwise. Takes expected constant time, worst-case linear time.
- `Insert(T, k, v)` inserts key k with value v into hash table T. If k is already in T, overwrites its value with v. Takes expected constant time, worst-case linear time.
- `Find(T, k)` finds key k in hash table T and returns the value associated with it. If k is not in T, produces an error instead. Takes expected constant time, worst-case linear time.
- `Entries(T)` returns a list of all the key/value pairs in hash table T. If the table currently has n keys and its underlying aray has e entries, this takes time proportional to $n + e$.

We can use this to define a new algorithm for the problem:

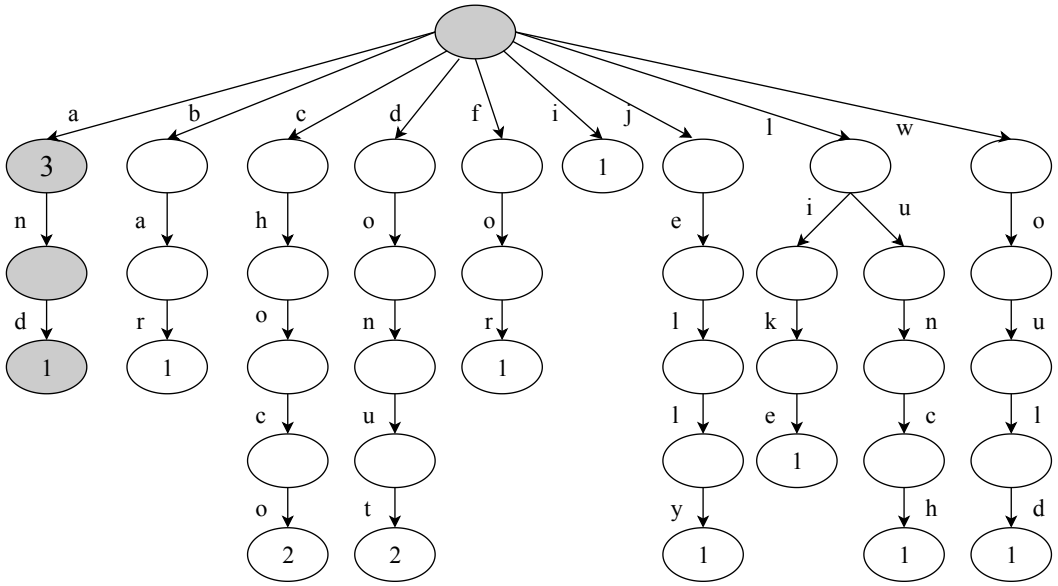
```
PROCEDURE ConvertToBag_HT(W):
  T = ConstructHashTable(|W|)    // |W| is likely bigger than we need but not incorrect
  For w in W:
      If Contains(T, w):
          c_w = Find(T, w)
          Insert(T, w, c_w+1)
      Else:
          Insert(T, w, 1)
  Return Entries(T)
```

3. Give and briefly justify a good asymptotic bound on the worst-case runtime of a call to `ConvertToBag_HT` in terms of m (the length of W) **and** n **(the number of unique words in** W**)**.

Trieing Different Data Structures

A *trie* is a tree data structure for storing key/value pairs where a key can be represented as a word (a string of letters). The trie has a fixed alphabet (in our case, the 26 letters of the English alphabet). The root node of the trie represents the empty string. Each node stores a list of child pointers, one for each letter in the alphabet. Each node that represents a complete word stores the value associated with that word. (Practically speaking, nodes that don't represent complete words store some kind of "null" for their value indicating that no complete word ends at that node.)

Below is a trie that represents the bag of words [a:3, and:1, bar:1, choco:2,[1] donut:2, for:1, i:1, jelly:1, like:1, lunch:1, would:1]:



We've shaded the leftmost path in this trie. The first shaded node under the root represents the word "a", which appears 3 times. The bottom node along that shaded path represents the word "and" – because we follow pointers from the root labelled "a", "n", and "d" to reach it – which appears 1 time. The word "an" wasn't in our bag, so the second node along the path has no value.

4. Alter the sketch above to add the following to the trie:

- Add the word "do" with value 2
- Add the word "four" with value 4

5. We described an upper-bound on the worst-case runtime of operations on the hash tables in terms of the number of keys stored in the table. That's not the most convenient variable to describe the runtime of operations on the trie, however. In terms of what quantity (variable) do operations on the trie run in worst-case linear time?

6. **[BONUS, WORTH 1 COURSE BONUS POINT]** We can, however, give a **lower** bound (an Ω bound) on the worst-case runtime of operations on a trie in terms of **just** the number of keys stored in the trie. Give and briefly explain the bound.

[1] Because chocolate is delicious but it makes the trie drawing too tall. A similar argument can be made for not using the British spelling of "doughnut."

2 SMP Proofpourri

Each of the following problem represents a SMP scenario (for the classic stable marriage problem with n men, n women, and complete preference lists) and a statement about that scenario. Each statement may be **always** true, **sometimes** true, or **never** true. Select the best of these three choices and then:

- If the statement is **always** true, (a) give and very briefly explain an example instance in which it is true and (b) prove that it is always true.
- If the statement is **never** true, (a) give and very briefly explain an example instance in which it is false and (b) prove that it is always false.
- If the statement is **sometimes** true, (a) give and very briefly explain an example in which it is true and (b) give and very briefly explain an example instance in which it is false.

Here are the problems:

1. **Scenario:** an SMP instance with $n \geq 2$. **Statement:** there is exactly one possible stable matching.
2. **Scenario:** an SMP instance with $n \geq 2$. **Statement:** there exists a stable matching in which nobody gets their first choice of partner.
3. **Scenario:** an SMP instance with $n \geq 2$ where two women have the same first choice of partner. **Statement:** the Gale-Shapley algorithm run with women proposing terminates after n iterations.

3 Utilitarian Marriage

Here we consider a variant of the stable marriage problem where, rather than having all women **rank** all men and vice versa, we have each women **rate** each man (and vice versa). To rate man m_j, woman w_i assigns him a positive integer that we will call the **utility**, and denote this by $w_i(m_j)$. We say that w_i prefers m_j to m_k if and only if she rates m_j higher than m_k; that is, if $w_i(m_j) > w_i(m_k)$. We will insist that a woman's ratings for any two men are distinct (i.e., that there can be no ties where $w_i(m_j) = w_i(m_k)$ for two different men m_j and m_k), and similarly for men rating women.

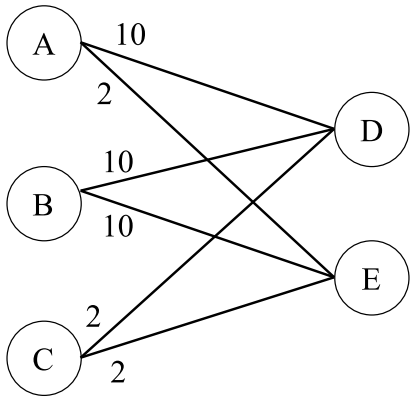
1. Given a list L of one woman's ratings of all the men – where $L[1]$ is her rating for m_1, $L[2]$ is her rating for m_2, etc. – give an algorithm to convert that into a preference list. Again, assume all ratings are distinct.
2. Give a **small** instance of the utility-rating problem where an **unstable** solution is **much better** in terms of utilities than the stable one. For full credit, you must:
   - Give the preference lists corresponding to your utility-rating problem.
   - Give a stable matching for this instance.
   - Explain why the unstable matching is so much better in terms of utilities.

Maximum Marriage

Consider the **maximum bipartite matching** problem: given a weighted bipartite graph, compute a maximal matching. A *matching* is a set of edges such that no two edges share a vertex and a *maximal matching* is a matching of maximum weight, where the weight of a matching is the sum of the weights of its edges.[2]

For example, in the graph below, the maximal matching consists of the edges (A,D) and (B,E), with total weight 20.

[2]You may be unfamiliar with some of the terms in this paragraph. If so, look them up!



3. Give a reduction from the utility-based marriage problem to maximum matching on a weighted, bipartite graph.

4. Give at least one measure of "goodness" of a solution for which your reduction produces an optimal result. Briefly explain why your reduction produces an optimal result.

5. Give at least one measure of "goodness" of a solution for which you reduction does **not** produce an optimal result. Use a small example to illustrate how the reduction fails.

4 It's an Applicant's Market

You're in charge of the CS department's internship program: your job is to match CS undergraduate students with companies for summer internships. By astonishingly good luck, you have n employers for n student applicants. The job market for software developers is booming: as a result, every employer desperately wants someone to fill their open internship spot, but **not** every student desperately wants to get an internship, because they can likely find a job outside the internship program if need be (or they can take summer courses, backpack through Asia, etc.). As a result, every employer has a complete preference list of all n students, but some students are unwilling to work for some employers and therefore have incomplete preference lists. A student would rather work for an employer on her preference list than be unmatched, but would rather be unmatched than work for an employer **not** on her preference list.

For example, for $n = 3$, a student s_1 may have the preference list $[e_2, e_3]$, which indicates that she is unwilling to work employer e_1. She would prefer to work for e_2 or e_3 than be unmatched, but would rather be unmatched than work for e_1.

An employer would prefer to be matched with **any** student than to be unmatched.

1. We will need to expand our definition of "instability" for this problem. The definition we saw in class still applies to the CS internship problem: namely, when e_i is matched with a student and s_j is matched with an employer on their preference list, but e_i and s_j prefer each other to their current partner. We can also consider it to be an instability when a student is matched with an employer not on his preference list: in this case, the student has an incentive to break the imposed matching (by quitting) and be happier as a result.

Describe three new types of instability that can occur between: (a) an unmatched student and a matched employer; (b) a matched student and an unmatched employer; and (c) an unmatched student and an unmatched employer.

2. Give an briefly explain a small example in which no stable perfect matching (i.e., a stable matching where every employer and every student is paired up) is possible.

Modifying Gale-Shapley

We will now modify Gale-Shapley to solve SMP with the change that not every student needs to list every employer in their preference list.

Here is the Gale-Shapley algorithm, with employers "proposing":

```
1:  procedure GALE-SHAPLEY(E, S)
2:      initialize all employers in E and students in S to unmatched
3:      while an unmatched employer with at least one student on its preference list remains do
4:          choose such an employer e ∈ E
5:          make offer to next student s ∈ S on e's preference list
6:          if s is unmatched then
7:              Match e with s                                  ▷ s accepts e's offer
8:          else if s prefers e to their current employer e' then
9:              Unmatch s and e'                                ▷ s rejects e'
10:             Match e with s                                  ▷ s accepts e's offer
11:         end if
12:         cross s off e's preference list
13:     end while
14:     report the set of matched pairs as the final matching
15: end procedure
```

With a small change, we can apply this algorithm and ensure that the (not necessarily perfect) matching produced never matches a student with an employer not in his preference list.

3. Make the small change necessary to the algorithm above.

4. Prove that, in your modified G-S algorithm, a student can never end up matched with an employer not on his or her preference list.

Special-Case Reductions

5. Suppose that **every** student refuses to work for the **same** employers. Without loss of generality[3], suppose that every student's preference list includes employers $e_1, e_2, \ldots, e_j$ and excludes employers e_i for $i > j$, where $j \leq n$.

Give a reduction from this special case of the CS internship problem to SMP. (Remember that your reduction must describe both how to convert the CS intership instance to an SMP instance **and** how to convert the solution from SMP back to a solution for the CS internship problem.)

6. **[BONUS, WORTH 1 COURSE BONUS POINT]** Prove the correctness of your reduction. In other words: prove that if the matching returned by the SMP solver is correct (contains no instabilities), then the matching returned by your reduction will also be correct (contain no instabilities).

[3] We can say "without loss of generality" here because, no matter which employers are excluded, we can just reorder the indices so that the rejected employers are at the end of the list.