# CPSC 320 2016W2: Assignment 1

January 13, 2017

Please submit this assignment via GradeScope at https://gradescope.com. Detailed instructions about how to do that are pinned to the top of our Piazza board: https://piazza.com/ubc.ca/winterterm22016/cpsc320/. Briefly, your GradeScope account **must** use the "GradeScope Student #" we distributed in your Connect gradebook so that we can link your account with you!

Submit by the deadline **Friday 20 Jan at 10PM**. For credit, your group must make a **single** submission via one group member's account, marking all other group members in that submission. 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 (not the individual quiz postings). 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 **GradeScope Student #s** of each member of your team. (No names are necessary.)
- 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!)
- **FOR THE FIRST ASSIGNMENT ONLY:** Include at the start of the document a note generally acknowledging if you had collaborators but without listing any names or identifying information.

  (On subsequent assignments, you will include collaborators' GradeScope account name, but we didn't let you know that soon enough to ask it of you here!)

## 1 Save the Last Dance for Someone Other Than Me

You're arranging a formal dance evening. $n$ people have signed up to lead and $n$ people to follow. There will be a series of $k$ dances (with $k \leq n$), each of which will match each leader with one follower. However, the same leader and follower will never be paired for more than one dance. (So, every person will dance with $k$ different people across the $k$ dances.) Your job is to create the series of $k$ matchings for the $k$ dances.

As with SMP, you have complete preference lists over the followers for each leader and complete preference lists over the leaders for each follower. (Where leaders and followers in this problem play similar roles to men and women—or vice versa—in SMP.)

This work is licensed under a Creative Commons Attribution 4.0 International License.
For license purposes, the author is the University of British Columbia.

### 1.1 A First Step

Write pseudocode for a simple algorithm that—given $n$, the $2n$ preference lists, and $k$—produces a valid list of $k$ perfect matchings for the dances, **completely ignoring the preferences**.

*Hint:* create your first matching by pairing $l_1$ and $f_1$, $l_2$ and $f_2$, and so on. How can you adjust these matching to create dance 2, dance 3, . . . , dance $k$.

### 1.2 Definitely Not the Two-Step

Someone suggests that for $k = 2$, we can run Gale-Shapley for the first matching and then simply have each partner in a pair from the first round move their first round partner to the end of their preference lists and run Gale-Shapley again for the second round. (I.e., pairs from the first round mutually declare each other their least favorite options for the second round.)

Work through the following counterexample to the correctness of this strategy—with $n = 3$ and the preferences of the leaders $(l_1, l_2, l_3)$ on the left and the followers $(f_1, f_2, f_3)$ on the right—to show that it does **not** necessarily work. Specifically: (a) identify the first dance's matchings, (b) write the new preference lists for the second dance, (c) identify the second dance's matchings, and (d) briefly explain why these matchings do not meet the requirements of the problem.

```
l1: 2 1 3   f1: 2 1 3
l2: 1 2 3   f2: 1 2 3
l3: 3 1 2   f3: 3 1 2
```

### 1.3 Almost the Two-Step

Someone suggests that for $k = 2$, we can run Gale-Shapley for the first matching and then simply have each partner in a pair from the first round move their first round partner to the end of their preference lists and run Gale-Shapley again for the second round. (I.e., pairs from the first round mutually declare each other their least favorite options for the second round.)

Finish this proof that this strategy never re-pairs **more than a single pair** in the second round.

**Proof:** Imagine that a couple **is** re-paired in the second round (which is possible) and that—without loss of generality, since the proof's structure remains the same if we make the opposite assumption—we run Gale-Shapley with leaders proposing. The couple that are re-paired listed each other as their least-preferred partners. The leader in that couple must have proposed to **everyone else** on their list prior to proposing to their eventual partner (because G-S has them propose in order, one after another, down their preference list).

At the moment when the leader's second-to-last offer is rejected, therefore, everyone but the last person on their preference list must already be engaged because. . .

## 2 THX 1138

A (conveniently-sized) set of $2n$ actors are applying for $n$ roles in a play. For each role, an actor is needed as well as an *understudy*, someone who plays the role if the main actor has problems. Each actor has a preference list over the roles. Each one would rather have any of the roles than any of the understudy positions; however, if they have to take an understudy position, they have the same preference order for these as for the roles. The casting director is in charge of hiring the actors. The casting director has a list of preferences over the actors for each role (and has the same list of preferences for each understudy position as for the corresponding role). We call this problem THX (the THeatre eXtension to the Stable Marriage Problem).

We want to find a solution to THX that is *stable*.

This work is licensed under a Creative Commons Attribution 4.0 International License.
For license purposes, the author is the University of British Columbia.

### 2.1 Small Example

Here is an example with $n = 2$.

- $a_1$ prefers $r_1$ to $r_2$ (and therefore $u_1$ to $u_2$)
- $a_2$ prefers $r_1$ to $r_2$
- $a_3$ prefers $r_2$ to $r_1$
- $a_4$ prefers $r_1$ to $r_2$
- For $r_1$, the casting director's preference order is: $a_1$, $a_3$, $a_2$, $a_4$
- For $r_2$, the casting director's preference order is: $a_1$, $a_4$, $a_3$, $a_2$

Give a stable matching of actors to roles and understudy positions for this problem.

### 2.2 Reducing THX to SMP

Give a clear and correct reduction from THX to SMP. Be sure to describe both how to convert any instance of THX to an instance of SMP and how to convert the corresponding solution to the SMP instance into a solution to the original THX instance.

Ensure the rationale for your reduction is clear in your answer. (We are **not** asking for a proof of correctness; think of it instead as documentation so we can understand the intention of your reduction.)

## 3 a Bag of Make Me Words

A standard part of text analysis—for machine learning, machine translation, sentiment analysis, 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 "make me a hot dog, a chili dog, and a bag of words" would become a 'bag' like: [a:3, and:1, bag:1, chili:1, dog:2, hot:1, make:1, me:1, of:1, words: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 13 words total but only 10 unique words (because "a" is repeated twice and "dog" once in the original phrase).

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

Most approaches we consider will use hash tables. For our purposes, such a hash table supports 6 operations. Here they are described for a standard hash table using chaining:

**ConstructHashTable($e$)** creates and returns a hash table of size $e$. (The hash table will be empty, but its underlying array will have $e$ entries. Takes time proportional to $e$.)

**Size($T$)** returns the number of keys in hash table $T$. (Takes constant time.)

**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.)

This work is licensed under a Creative Commons Attribution 4.0 International License.
For license purposes, the author is the University of British Columbia.

**ENTRIES($T$)** returns a list of all the key/value pairs in hash table $T$. (If the table currently has $n$ keys (i.e., SIZE($T$) $= n$) and its underlying array has $e$ entries, this takes time proportional to $n + e$.)

## 3.1 Making a Hash of the Bag

We're going to consider an approach that uses a standard hash table using chaining. Here is the planned approach:

```
procedure ConvertToBag(W)
    T ← ConstructHashTable(|W|)
    for each word w in W do
        c_w ← Find(T, w)
        Insert(T, w, c_w + 1)
    end for
    return Entries(T)
end procedure
```

This algorithm has a small bug.

1. Give the *shortest possible* input $W$ we could pass to CONVERTTOBAG to illustrate the bug. (Substantial partial credit is available for "almost-shortest" answers.)
2. Indicate what the implementation of CONVERTTOBAG above does and also what a correct impementation **should** do for your input.
3. Fix the bug in the pseudocode above.[1]

## 3.2 Asymptotic Bound

Here is a correct approach to this problem using a standard hash table with chaining:

```
procedure ConvertToBag(W)
    T ← ConstructHashTable(|W|)        ▷ |W| is likely bigger than we need but not incorrect.
    for each word w in W do
        if Contains(T, w) then
            c_w ← Find(T, w)
            Insert(T, w, c_w + 1)
        else
            Insert(T, w, 1)
        end if
    end for
    return Entries(T)
end procedure
```

Give and **briefly** justify—including annotating the algorithm above to indicate how each part contributes to your bound—a good asymptotic bound on the **worst-case** runtime of a call to CONVERTTOBAG in terms of $m$—the length of $W$, i.e., $m = |W|$—and $n$—the number of unique words in $W$.

## 3.3 Some Do and Others Trie

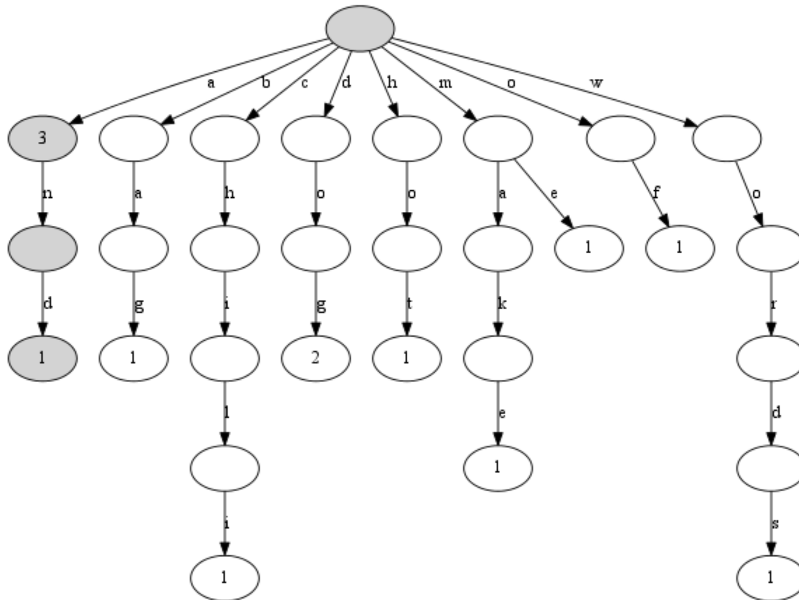
A *trie* is a tree data structure for storing key/value pairs where a key can be represented as a word (a string of letters). The trie has a fixed alphabet, like the 26 letters of the English alphabet. The root node of the trie represents the empty string. Each node stores a list of child pointers, one for each letter in the alphabet. Each node that represents a complete word stores the value associated with that word.

[1] Only for the quiz, **not** required for the assignment.

This work is licensed under a Creative Commons Attribution 4.0 International License.
For license purposes, the author is the University of British Columbia.

(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.)

For example, here is a trie that represents the bag of words [a:3, and:1, bag:1, chili:1, dog:2, hot:1, make:1, me:1, of:1, words: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 then "d" to reach it), which appears 1 time. The word "an" wasn't in our bag, and thus the second node along the path has no value.

1. Alter the sketch above to add the following to the trie:
    - add the key "do" with value 4
    - add the key "chime" with value 2
2. 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?
3. **[WORTH 1 BONUS POINT ON THE ASSIGNMENT AND COURSE; NOT REQUIRED OR GRADED FOR THE QUIZ.]** We **can**, however, give a lower-bound (an $\Omega$ 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.

This work is licensed under a Creative Commons Attribution 4.0 International License.
For license purposes, the author is the University of British Columbia.