# SI 335 Spring 2014: Problem Set 4

**Due**: Wednesday, April 30

**Your scheduled presentation time**:

**Group member**:

**Group member**:

**Group member**:

**Group member**:

**Instructions**: Review the course honor policy: you may not use any human sources outside your group, and must document anything you used that's not on the course webpage.

This cover sheet must be the front page of what you hand in. Use separate paper for the your written solutions outline and make sure they are neatly done and in order. Staple the entire packet together.

**Comments or suggestions about this problem set:**

**Comments or suggestions about the course so far:**

**Citations** (be specific about websites):

**Grading rubric:**

**A**: Solution meets the stated requirements and is completely correct. Presentation is clear, confident, and concise.
**B**: The main idea of the solution is correct, and the presentation was fairly clear. There may be a few small mistakes in the solution, or some faltering or missteps in the explanation.
**C**: The solution is acceptable, but there are significant flaws or differences from the stated requirements. Group members have difficulty explaining or analyzing their proposed solution.
**D**: Group members fail to present a solution that correctly solves the problem. However, there is clear evidence of significant work and progess towards a solution.
**F**: Little to no evidence of progress towards understanding the problem or producing a correct solution.

| Problem | Final assessment |
|---|---|
| 1 | |
| 2 | |
| 3 | |
| 4 | |

## 0.1 Known NP-Complete Problems

For your reference, here is a list of the **NP**-Complete Decision Problems that were presented in class. In any of your solutions, you may use the fact that these problems are **NP**-complete.

- `LONGPATH(G,u,v,k)`
  Input: Graph $G = (V, E)$, vertices $u$ and $v$, integer $k$
  Output: Does $G$ contain a path from $u$ to $v$ of length at least $k$?
- `VC(G,k)`
  Input: Graph $G = (V, E)$, integer $k$.
  Output: Does $G$ have a vertex cover (subset of $V$) containing at most $k$ vertices?
- `HITSET(L,k)`
  Input: List $L$ of sets $S_1, S_2, \ldots, S_m$, and an integer $k$
  Output: Is there a "hitting set" $H$ with size at most $k$ such that $H$ contains at least one member of every set $S_i$?
- `HAMCYCLE(G)`
  Input: Graph $G = (V, E)$
  Output: Does $G$ contain a cycle (path with same starting and ending vertex) that touches every node exactly once?
- `CIRCUIT-SAT(C)`
  Input: Boolean circuit $C$ with $m$ inputs and one output
  Output: Is there a setting of the $m$ inputs to True/False that makes the output stabilize to True?
- `3-SAT(F)`
  Input: Boolean formula $F$ in conjunctive normal form (product of sums) with three literals in every clause
  Output: Does $F$ have a "satisfying assignment" (setting of every variable to True/False so that the entire formula is True)?
- `SPLIT-EVENLY(S,k)`
  Input: Set $S$ of integers
  Output: Can $S$ be partitioned into two subsets $A$ and $B$ such that difference between the sums of the numbers in $A$ and $B$ is at most $k$?

# 1 Hungry Hungry Mids

This question is about the following *computational* problem:

King Hall has a bunch of random leftover food items: a single hamburger patty, a bottle of ketchup, a bowl of mashed potatoes, a dill pickle, etc. Each leftover food item has a certain number of calories in it. The question is how many complete meals can be made from these leftover items, with the only restriction being that each "meal" must contain at least a certain number of calories.

Formally, the problem is defined as follows:

> `COMPUTE-MAX-MEALS(L,k)`
> **Input**: List $L$ of integers, and a single integer $k$. Each integer in $L$ is between 1 and $k-1$.
> **Output**: A partition of $L$ into $r$ subsets $M_1, M_2, \ldots, M_r$ such that the sum of the numbers in each $M_i$ is at least $k$, and the number of subsets $r$ is as large as possible.

For example, if $L = (5, 3, 3, 8, 6, 10, 11, 5, 7, 4)$ and $k = 20$, then an optimum solution has $r = 3$ and the subsets are $M_1 = (10, 8, 3)$, $M_2 = (11, 5, 5)$, and $M_3 = (7, 4, 6, 3)$.

a) Describe a *greedy* algorithm to solve the `COMPUTE-MAX-MEALS` problem. This should be a fairly simple idea.

b) Give a counterexample to show that your greedy algorithm does not always produce the best possible solution. That is, you give a sample input $L$, show what your greedy algorithm would produce, and then come up with an even better solution which demonstrates the greedy algorithm is not optimal.

c) Come up with a *decision version* of this problem. Call your decision problem `MEALS`.

Remember that your decision problem should be roughly the same difficulty as the original problem, up to polynomial-time reductions. In particular, you have to choose your decision problem carefully so that parts (e) and (f) below are at least *possible* to prove (even if you don't prove them yourself).

You are encouraged to e-mail your idea for this part to your instructor if you're not sure, before proceeding.

d) Show that your `MEALS` problem is in **NP**, using the steps we went over in class for an **NP** proof.

e) Present a polynomial-time reduction from your `MEALS` problem to `COMPUTE-MAX-MEALS`.

f) Show that your `MEALS` problem is **NP**-hard by presenting a reduction from a known **NP**-complete problem to your `MEALS` problem. You may use any of the **NP**-complete problems in the list at the beginning of this assignment for your reduction.

(Hint: what other decision problem involves the sums of numbers in different subsets?)

g) State what we know about your `MEALS` problem and the `COMPUTE-MAX-MEALS` problem, assuming (d) through (f) have been proven.

## 2 Party Planner

This question is about the following *computational* problem:

You are planning a party and want to invite a bunch of your friends. Unfortunately, some of your friends and acquaintences don't get along with each other, and bad things will happen if they both show up for the party. So, given the histories of bad blood among your friends, you want to invite the largest group of friends possible to your party, without inviting any two people that don't get along.

Formally, the problem is defined as follows:

> `COMPUTE-MAX-PARTY(F,E)`
> **Input**: A list of friends $F$, and a list of pairs of enemies $E$, each pair containing two elements from $F$.
> **Output**: A subset of $P$ of $F$, as large as possible, such that no two elements in $P$ are enemies, i.e., for every pair in $E$, at most one of the pair is in $P$.

For example, if $F = \{1, 2, 3, 4, 5\}$ and $E = \{(1, 3), (2, 3), (1, 5), (4, 5)\}$, then an optimum solution is $P = \{1, 2, 4\}$.

a) Describe a *greedy* algorithm to solve the `COMPUTE-MAX-PARTY` problem. Your algorithm should be fairly simple.

b) Give a counterexample to show that your greedy algorithm does not always produce the best possible solution.

c) Come up with a *decision version* of this problem. Call your decision problem `PARTY`.

Remember that your decision problem should be roughly the same difficulty as the original problem, up to polynomial-time reductions. In particular, you have to choose your decision problem carefully so that parts (e) and (f) below are at least *possible* to prove (even if you don't prove them yourself).

You are encouraged to e-mail your idea for this part to your instructor if you're not sure, before proceeding.

d) Show that your `PARTY` problem is in **NP**, using the steps we went over in class for an **NP** proof.

e) Present a polynomial-time reduction from your `PARTY` problem to `COMPUTE-MAX-PARTY`.

f) Show that your `PARTY` problem is **NP**-hard by presenting a reduction from a known **NP**-complete problem to your `PARTY` problem. You may use any of the **NP**-complete problems in the list at the beginning of this assignment for your reduction.

(Hint: think about choosing the minimal number of people that *won't* be invited to the party. What decision problem(s) involve finding the smallest possible something?)

g) **NO CREDIT GIVEN IF YOU ALREADY ANSWERED 1(g).**

State what we know about your `PARTY` problem and the `COMPUTE-MAX-PARTY` problem, assuming (d) through (f) have been proven.

## 3 Shoe Matching

a.k.a. the multiple Cinderellas problem.

**Scenario**:

You and a group of friends all decide to jump in the lake for a swim. Naturally, you all leave your shoes up on the beach. And for some reason, you all have exactly the same looking shoes.

While you are in the water, a practical joker removes all the labels from the shoes and mixes them all around. So when you and your friends get out of the water, you are met with a big pile of shoes and you have no idea whose is whose. All you can do is start trying them on until everyone finds their own pair.

Knowing that you are an algorithms expert, your friends turn to you for the quickest way to sort all this out.

**Specifics**:

You *must* solve the problem under the following assumptions. No "side-channel" solutions like saying that you all find the practical joker, take his money, and buy yourselves new sneakers.

1) There are $n$ friends and $2n$ shoes, consisting of $n$ left shoes and $n$ right shoes.
2) You can immediately tell the difference between a left shoe and a right shoe. Otherwise, there is no way to tell the difference between any shoes besides trying them on.
3) You cannot compare shoes to shoes directly. You cannot compare feet to feet directly.
4) An individual can try on one shoe at a time.
5) After trying on a shoe, the individual knows whether it was too small, too big, or just right.
6) For each of the $n$ friends, there is exactly one shoe in the pile (no more, no less) that fits each of their feet perfectly.

**Tasks**

a) Devise an algorithm to match everyone with their shoes. You should be able to describe your algorithm in words or using pseudocode, or both.

b) Analyze the running time of your algorithm, in terms of the number of shoes that must be tried on, in total, by everyone. Try to come up with a big-Theta bound.

**Hint**: Perhaps you would like to `partition` somehow.

# 4 Selection (up to 5% BONUS for your group)

Consider the following variant on the selection problem: Given a list $A$ of $n$ elements, and an integer $k$, find the $k$th largest DISTINCT element in $A$, counting from 0. This is like the original selection problem, but ignoring duplicated elements.

Show that, at least in the comparison model, there can't possibly be a $O(n)$-time algorithm for this modified selection problem that ignores duplicate elements.