**NATIONAL UNIVERSITY OF SINGAPORE**

# CS2020 - Data Structures and Algorithms (Accelerated)

(Semester 2 AY2015/16)

Time Allowed: 2 Hour

## Instructions

- Write your Student Number below, and on every page. Do not write your name.
- The assessment contains 7 multi-part problems. You have 120 minutes to earn 100 points.
- The assessment contains 26 pages, including the cover page and 5 pages of scratch paper.
- The assessment is closed book. You may bring two double-sided sheet of A4 paper to the assessment. You may not use a calculator, your mobile phone, or any other electronic device.
- Write your solutions in the space provided. If you need more space, please use the scratch paper.
- Show your work. Partial credit will be given. Please be neat.
- You may use (unchanged) any algorithm or data structure that we have studied in class, without restating it. If you want to modify the algorithm, you must explain exactly how your version works.
- Draw pictures frequently to illustrate your ideas.
- Good luck!

**Student Number.:** ________________________________

| Problem # | Name | Possible Points | Achieved Points |
|---|---|---|---|
| 1 | Things that are not true | 15 | |
| 2 | Drawing pictures | 16 | |
| 3 | Java Snippets | 15 | |
| 4 | What is going on? | 07 | |
| 5 | Restaurant redux | 15 | |
| 6 | Radio Stations | 16 | |
| 7 | Social networking | 16 | |
| **Total:** | | 100 | |

**Problem 0. Before we begin.** [0 points]

Circle the image that best represents how you feel right now.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 😀 | 😆 | 😄 | 😎 | 🤔 | 😐 | 😴 | 😬 | 😠 | 😞 |

**Problem 1. Things that are not true.** [15 points]

Each of the following "claims" is false. Give a counter-example showing why each is false.

**Problem 1.a.** Every algorithm with $\Theta(n^2)$ running time is slower than every algorithm with $\Theta(n \log n)$ running time on all inputs.

**Solution:** There are many possible answers. One example: InsertionSort is an example of an $O(n^2)$ algorithm. MergeSort is an $O(n \log n$ algorithm. On a sorted list, InsertionSort is faster than MergeSort. Other examples might note that for small cases, e.g., $n = 2$, we know that $100n \log n = O(n \log n)$ will be larger than $2n^2 = O(n^2)$.

**Problem 1.b.** For every graph $G$, for every source node $s$ in $G$, the shortest path tree rooted at $s$ always includes the lightest edge in the graph $G$. (Recall, the shortest path tree is the tree consisting of the shortest paths from $s$ to every other node in the graph.)

**Solution:** Imagine a triangle consisting of three nodes: $s, A, B$. The edges $(s, A)$ and $(s, B)$ are of weight 1000 and are the shortest path trees. The edge from $(A, B)$ has weight 1 and is not included in the shortest path tree.

**Problem 1.c.** Every undirected graph $G$ with designated root $r$ has a tree $T$ that is both a shortest path tree and a minimum spanning tree.

**Solution:** The three node triangle in the previous part is an example. The shortest path tree from $s$ cannot include edge $(A, B)$, while the MST for the same graph *must* include edge $(A, B)$.

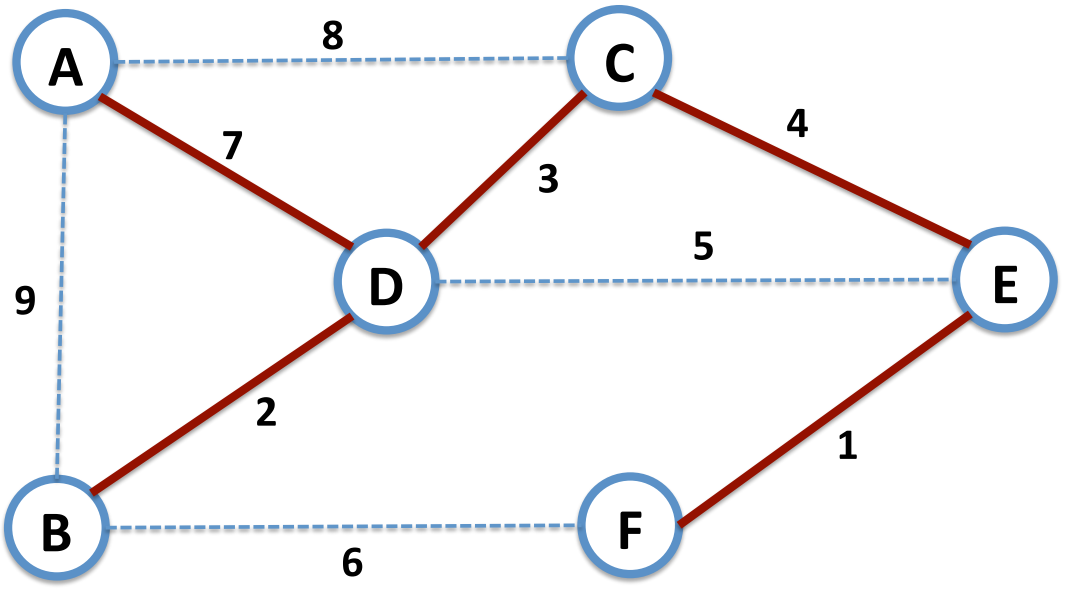
**Problem 2. Drawing Pictures** [16 points]

**Problem 2.a.** Execute Kruskal's Algorithm on the graph below. In the table, fill in the the edges that are added to the minimum spanning tree, in order. For each edge, write down the number associated with that edge. (That is, the weight of the edge acts as its name.) There are more spaces in the table than you need.

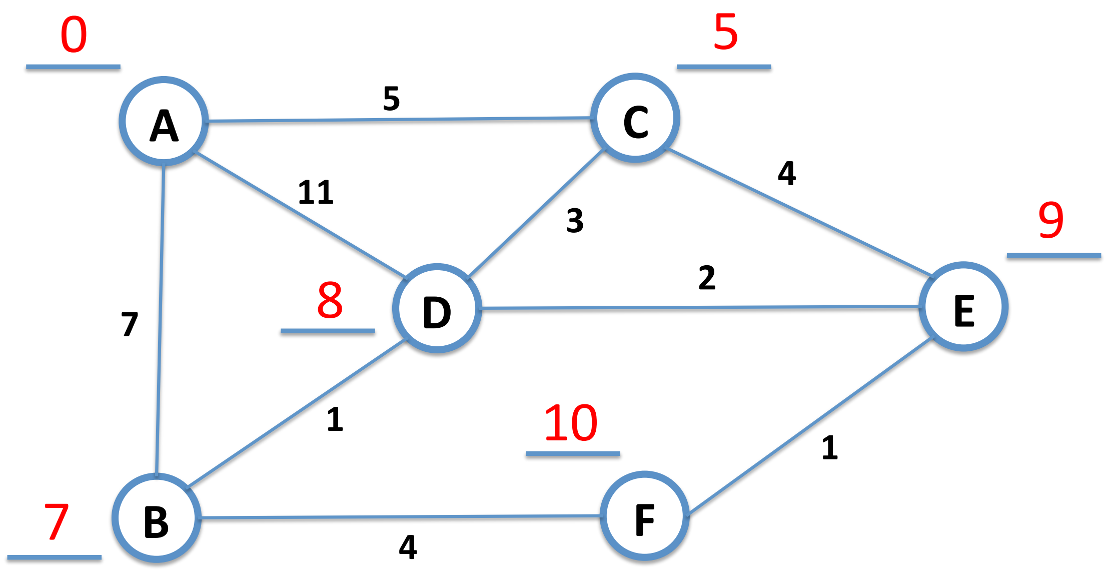
**Solution:**

**Edges added by Kruskal's, in order:**

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 7 | | | | | |



**Problem 2.b.** Execute Dijkstra's Algorithm on the graph below, starting with node $A$ as the source. For each iteration of the algorithm, fill in the state of the priority queue. (You may not need all the space provided.) Write the final distance next to the node in the graph.



**Solution:**

| Step 1 | | Step 2 | | Step 3 | | Step 4 | |
|---|---|---|---|---|---|---|---|
| Priority | Key | Priority | Key | Priority | Key | Priority | Key |
| 0 | A | 5 | C | 7 | B | 8 | D |
| | | 7 | B | 8 | D | 9 | E |
| | | 11 | D | 9 | E | 11 | F |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |

| Step 5 | | Step 6 | | Step 7 | | Step 8 | |
|---|---|---|---|---|---|---|---|
| Priority | Key | Priority | Key | Priority | Key | Priority | Key |
| 9 | E | 10 | F | | | | |
| 11 | F | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |

**Problem 3. Java Snippets** [18 points]

For each of the following examples, specify whether the code is: (a) correct, (b) has a syntax error/compile error, (c) crashes/throws an exception, or (d) enters an infinite loop. The code is considered correct if it compiles, runs without crashing or throwing an exception, and terminates. (It does not matter whether it actually seems to calculate the answer you expect it to.) Circle "correct" if it is correct for **all** inputs matching the stated assumptions. Otherwise, circle the proper answer for each part.

**Problem 3.a.** Assume that `IHardExam` and `IEasyExam` are existing interfaces and `Exam` is an existing class. Consider the following class definitions:

```
class FinalExam extends Exam implements IHardExam {

}
```

| | | | |
|---|---|---|---|
| **Correct** | **Syntax Error or Compiler Error** | **Crashes or throws an exception** | **Infinite loop** |

**Briefly explain why (in one sentence):**

**Solution: In general, IHardExam may contain methods that are not implemented in class FinalExam, and so this is a compiler/syntax error. If you stated an assumption that IHardExam is empty, i.e., containing no methods, then this is correct (and that is an acceptable answer, as long as you stated that assumption). Generally, your explanation should be consistent with the option you chose.**

**Problem 3.b.** Assume that `bicycles` is already defined as an array of `Bicyle` (a class that is already defined). The following code written inside a method is supposed to print out the string representation of the all the bicyles in the array.

```
int length = bicycles.length;
for (int i=0; i<= length; i++) {
    System.out.println(bicycles[i].toString());
}
```

| | | | |
|---|---|---|---|
| **Correct** | **Syntax Error or Compiler Error** | **Crashes or throws an exception** | **Infinite loop** |

**Briefly explain why (in one sentence):**

**Solution: Index out of bounds (throws an exception).**

**Problem 3.c.** The following code (which is part of an otherwise correct method) is designed to print out some (integer) divisions:

```
int i = 1733;
int max = i;
while (i > 0) {
    i = i/2;
    int d = (max/i);
    System.out.println(d);
}
```

| | | | |
|---|---|---|---|
| **Correct** | **Syntax Error or Compiler Error** | **Crashes or throws an exception** | **Infinite loop** |

**Briefly explain why (in one sentence):**

**Solution: Crash or exception, because $i$ can be 0.**

**Problem 3.d.** The following two methods are both contained in the same class:

```
int average(int x, int y){
   return (x+y)/2;
}

static int average(int x, int y, int z){
   return (x+y+z)/3;
}
```

| | | | |
|---|---|---|---|
| **Correct** | **Syntax Error or Compiler Error** | **Crashes or throws an exception** | **Infinite loop** |

**Briefly explain why (in one sentence):**

**Solution:** **Correct.**

**Problem 3.e.** The following code calculates the exact value of the recurrence: $T(n) = 2T(n/3) + n^2$. Assume $n$ is a power of 3.

```
int calculateRecurrence(int n) {
   // Calculate the non-recursive part:
   int valueA = n*n;

   // Calculate the recursive part:
   int valueB = calculateRecurrence(n/3);

   // Return the answer:
   return 2*valueB + n*n;
}
```

| | | | |
|---|---|---|---|
| **Correct** | **Syntax Error or Compiler Error** | **Crashes or throws an exception** | **Infinite loop** |

**Briefly explain why (in one sentence):**

**Solution:** Is either an infinite loop (as there no base case to the recursion), or throws an exception (heap overflow). There are two possible answers, so the explanation should match the item circled.

**Problem 4. What is going on?** [7 points]

Consider the following block of (mostly useless) Java code:

```
class PrintSome {

  static class StoreInt {
      public int m_int = 0;

      StoreInt(int x) {
          m_int = x;
      }
  }

  static int[] calculateValue(int[] values, int key, StoreInt j) {
     for (int i=0; i<values.length; i++) {
         values[i]*=key;
     }
     key = values[j.m_int];
     j = new StoreInt(58);
     return values;
  }

  public static void main(String[] args) {
       int j = 10;
       int[] array = {1,2,3};
       StoreInt  k = new StoreInt(2);
       int[] newArray = calculateValue(array, j, k);
       for (int i=0; i<array.length; i++) {
          System.out.println(array[i]);
       }
       System.out.println(j);
       System.out.println(k.m_int);
  }
}
```

**What does this code print out?**

**Solution:** It outputs the following:

**Line 1:** `10`

**Line 2:** `20`

**Line 3:** `30`

**Line 4:** `10`

**Line 5:** `2`

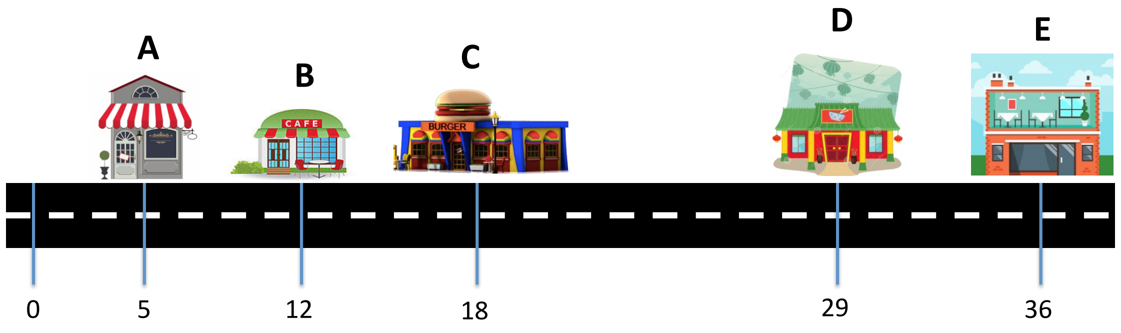
**Problem 5. Restaurant Redux** [15 points]

Imagine a very long road, with restaurants all along the road. You have been hired by `RestaurantsRUs` to build a web site that your clients can use to find the nearest restaurants to their location at any given time. Your data structure will consist of two parts:

- The *preprocessing phase*: In this phase, you receive as input an array $R$ consisting of $n$ restaurants, each with a name and a location. The location 0 is the far left end of the road, and the location is the distance along the road from the left. You may process the input data in any way you like, preparing your data structure. (There is nothing output from this phase.)
- The *query phase*: In this phase, you must respond to queries of the form `findNearest(d, k)` which finds the $k$ nearest restaurants to location $d$. Your output should be a list of $k$ restaurants in *sorted* order of distance from $d$.

Your data structure should store the $n$ restaurants in $O(n)$ space (independent of the length of the road). The query operation should run in $O(k + \log n)$ time (where $k$ is the parameter in the input to the query), and the preprocessing phase should be as efficient as possible.

**Example:**



`findNearest(22, 3)` $\rightarrow$ $C, D, B$
`findNearest(20, 3)` $\rightarrow$ $C, B, D$
`findNearest(20, 5)` $\rightarrow$ $C, B, D, A, E$

**Problem 5.a.** Describe your data structure and the preprocessing phase.

**Solution:** A simple solution is store the restaurants in $R$ in an array in sorted order. To preprocess the data, you need only to sort the array $R$ by location.

**Problem 5.b.** What is the running time of the preprocessing phase?

**Solution:** Sorting takes $O(n \log n)$ time.

**Problem 5.c.** Describe how the query operation works? Explain its running time.

**Solution:** The basic idea is to find the $k$ items larger than the query location, the $k$ items smaller than the query location, and then merge the two lists to a new sorted list of $2k$ items. The answer to the query is the first $k$ items in this list. The query operation can be implemented via the following steps:

- Use binary search to find the closest restaurant that is at a location $\geq$ the query location. This takes $O(\log n)$ time. Assume it returns location $\ell$ in the sorted array.
- Create a sorted list $A$ of the first $k$ restaurants after the query location, i.e., the restaurants at locations $[\ell, \ell+1, \ldots, \ell+k-1]$. Notice this list is sorted in increasing distance from the query location. This takes $O(k)$ time.
- Create a sorted list $B$ of the first $k$ restaurants before the query location, i.e., the restaurants at locations $[\ell-1, \ell-1, \ldots, \ell-k]$. Store this list in increasing distance from the query location. This takes $O(k)$ time.
- Merge the two lists $A$ and $B$ using the linear time merging algorithm (from class), resulting in a single list $C$ sorted in increasing distance from the query location. This takes $O(k)$ time.
- Return the first $k$ elements in the list $C$, in order. This takes $O(k)$ time.

The reason this works is that the $k$ nearest restaurants must be included in the set of $2k$ restaurants under consideration, and because the list of restaurants is already sorted, we can simply merge the two sub-lists (instead of having to sort again).

**Problem 5.d.** What if your data structure is required to support an additional operation: `addRestaurant(name, location)`? How would you modify the data structure above to *efficiently* support adding new restaurants? (Give a brief explanation.)

**Solution:** Instead of storing the restaurants in a sorted array, use an AVL. Augment the AVL tree so that each item has a pointer to its successor and predecessor in the tree. It takes $O(n \log n)$ time to build such a tree. Now we can insert new items into this tree in $O(\log n)$ time. As before, we can easily search for the successor of the query location, and find the $k$ restaurants larger and the $k$ restaurants smaller than the query location, and merge them.

**Problem 6. Radio Stations** [15 points]

You are given a set $V$ that consists of $n$ radio stations, including a special station $s \in V$ that is the source of all the radio signals. Some stations are close enough to be connected. For each such pair of stations, there is a maintenance fee to connect the two stations. You are given this cost as a function $f(u, w)$ where $u, w \in V$. Assume that the costs are unique (i.e., no two costs are the same), and assume the graph induced by the stations is connected. Assume that you have already computed a minimum spanning tree $T_0$ of the radio stations, thus minimizing the cost to distribute the signal to all nodes.

**Problem 6.a.** A startup company wants to test a new breakthrough technology for connecting stations, and offers, for free, to connect your source to another station. The new technology is strong enough to reach *any* station. You can pick one of the stations other than the source and connect this station to the source with NO maintenance cost! Free!

Briefly describe an algorithm for choosing which station to connect so as to minimize the maintenance fee. (Recall that you have already calculated $T_0$.) Explain why your algorithm works. We will label the resulting tree with the new free link $T_1$.

**Solution:** When a new edge is added, it creates a cycle in the MST, and we can remove the heaviest edge on the cycle to create a new MST. Thus, on adding the new edge, the MST will remain the same except for the new edge that is added and one existing edge that is removed. That is, the cost of the new MST will be equal to the cost of the old MST minus the weight of one edge. Given that, it is clear that we want to remove the heaviest edge on the MST. Let edge $(u, v)$ be the heaviest edge in $T_0$. Delete edge $(u, v)$. Now, we perform a DFS from the source $s$ until we reach either $u$ or $v$. (We cannot reach both, as by deleting $(u, v)$ we have disconnected the MST.) If the DFS from $s$ reaches $u$, then we create a new connection $(s, v)$; if the DFS from $s$ reaches $v$, then we create a new connection $(s, u)$.
Once the new connection is created, we have a path from $u$ to $v$ again (via $s$), and hence the resulting graph is connected. Moreover, if we add back edge $(u, v)$, it is obviously the heaviest edge on the cycle created, and so the new graph is an MST.

What is the (asymptotic) worst-case running time of your algorithm?

**Solution:** The running time is $O(V)$, i.e., the cost of finding the heaviest edge in the MST (which has $O(V)$ edges) and the cost of performing a DFS in the broken MST.

**Problem 6.b.** It turns out that the new (free) connection interferes with one of the existing links. That is, the link between two stations $A$ and $B$ becomes (permanently) impossible. Briefly describe an algorithm for rebuilding the minimum spanning tree without the link from $A$ to $B$. (Recall that you have already calculated the tree $T_1$ in the previous part.) Explain why your algorithm works and give the running time.

**Solution:** Removing edge $(A, B)$ divides the graph into two components. The MST of the new graph must include the lightest edge connecting these two components. To see that the resulting tree is an MST, considering adding some edge $e = (u, v)$ that is not in the MST. There are two cases: either $e$ connects the two components created by removing $(A, B)$ or it does not.
If $e$ does connect those two components, then we know that it used to be the heaviest edge on the cycle containing $(A, B)$; the only question is whether or not the new edge is heavier. But we know that the new edge must be lighter than $e$, because we chose the lightest edge to connect the two components. Hence $e$ is the heaviest edge on the cycle and does not belong in the MST.
If $e$ does not connect the two components (i.e., it forms a cycle within one of the two components), then we know immediately that $e$ is the heaviest edge on the cycle—since it was the heaviest edge on the cycle before the new edge was added.
In either case, we know that every edge not in the tree is the heaviest edge on some cycle, and hence should not be in the MST.
In order to find the lightest edge across the cut:

- Starting at $A$, do a DFS labelling each node as part of component 1.
- Starting at $B$, do a DFS labelling each node as part of component 2.
- Iterate through all the edges in the graph, finding the lightest that has one node in component 1 and one node in component 2.

The running time of this procedure is $O(E)$.

What is the (asymptotic) worst-case running time of your algorithm?

**Solution:** The running time is $O(E)$, where $E$ is the set of pairs of stations that can be connected in the initial graph.

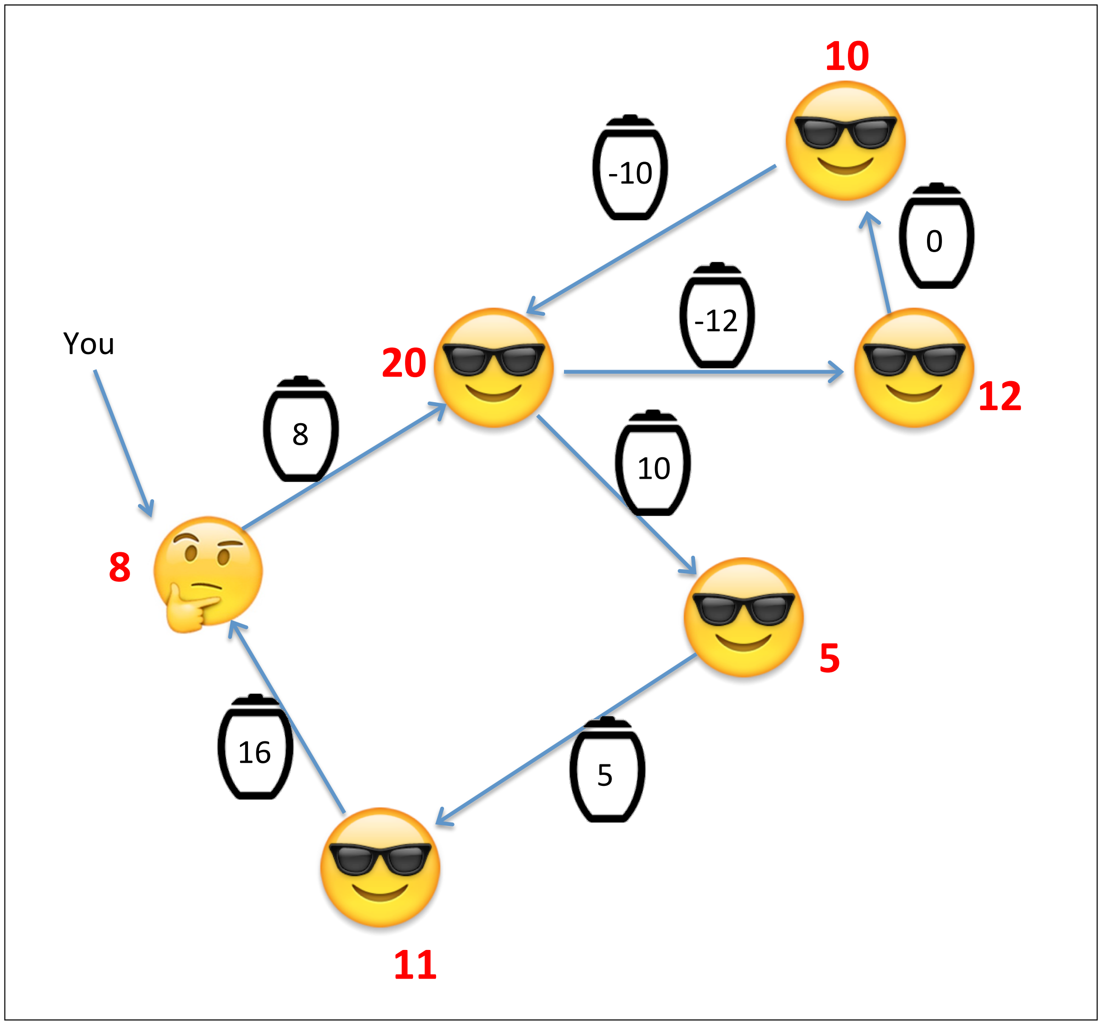
**Problem 7. Social Networking**

Imagine a whole bunch of friends standing around room. You are given a graph $G = (V, E)$ representing the relationships of the people in the room: $V$ is the set of people in the room, and the directed edges in $E$ imply friendship. That is, if $(u, v) \in E$, then $u$ is friends with $v$. (Notice that edges, and friendship, are directed.) You are bored—and almost broke—and propose a game.

- First, each person in the room is assigned a positive number. Let $n[u]$ be the number assigned to person $u$.
- You happen to have a large metal pot in your bag. You begin the game with the pot.
- Your job is to choose a path for the pot. It must traverse the social network, going from one friend to the next, until it eventually returns to you. To be more precise, the pot should traverse a path $(u_0, u_1, \ldots, u_k, u_0)$, where you are $u_0$, and each edge $(u_i, u_{i+1}) \in E$ and $(u_k, u_0) \in E$. The pot may visit a person more than once (i.e., there is no requirement that the $u_i$ be distinct)—however the game ends as soon as the pot returns to you.
- You start out by putting $n[u_0]$ dollars in the pot.
- The next person, $u_1$, takes $n[u_1]$ dollars out of the pot.
- The next person, $u_2$, puts $n[u_2]$ dollars in the pot.
- And so on. As the pot proceeds, each person alternates putting $n[u_i]$ dollars in the pot or taking $n[u_i]$ dollars out of the pot.
- We allow for "negative money" in the pot. For example, if the pot contains 10 dollars and someone removes 20 dollars, then the pot has $-10$ dollars in it. (This is implemented via an "IOU" promissory note system.)
- Eventually, the pot returns to you, at which point the game is over. You keep whatever money is in the pot.

Luckily, you know the graph $G$ and you know all the numbers $n[u]$. Even more luckily, you get to specify the route of the pot. Your goal is to double your money. (Assume that each of your friends has enough money to play the game without going broke.) That is, you want to ensure that when the pot returns to you, it has at least $2n[u_0]$ dollars in it.

Give an algorithm to decide whether or not this is possible. Your algorithm should output *Yes, play the game!* if it is possible to choose a route to double your money, and it should output *No, do not play!* if it is not possible. (You do not need to output the actual path of the pot.)



**Example.** This is an example of a social network (where all your friends wear sunglasses). The pot starts with you, and you add eight dollars. It then proceeds to your friend who removes 20 dollars, leave $-12$ dollars. There are then two options as to how you might route the pot. You route the pot toward your friend who adds 12 dollars, leaving zero. The next person removes 10, leaving $-10$. It then returns to the first friend, but this time he *adds* 20 dollars, leaving 10 dollars in the pot. It then proceeds to your friend who removes 5 dollars (leaving 5), and your friend who adds 11 dollars. The pot then returns to you with 16 dollars. You win! You have doubled your original money (i.e., the 8 dollars you put in initially).

**Problem 7.a.** Describe your algorithm briefly in one or two sentences.

**Solution:** Build a new graph consisting of two copies of the original graph, one for even steps and one for odd steps. Then use Bellman-Ford to determine if the longest path in the graph has sufficient value, or to detect a positive weight cycle.

**Problem 7.b.** Draw a picture illustrating how your algorithm/construction works on a small example (e.g., an example with four people).

**Problem 7.c.** Assume there are $n$ people in the room, and $m$ edges in the graph $G$. What is the running time of your algorithm as a function of $n$ and $m$? $\boxed{\phantom{xxxxxx}}$

**Solution:** The new graph constructed contains $O(n)$ nodes and $O(m)$ edges and the new graph can be constructed in $O(n+m)$ time. Hence the running time is $O(nm)$.

**Problem 7.d.** Give more details, as necessary, explaining how your algorithm works.

**Solution:** First, we construct a new graph that consists of two copies of the original graph $G$. That is, for each node in the original graph, we create two nodes $u$ and $u'$ in the new graph. Connect $u$ in the first graph with $v'$ in the second graph if edge $(u, v) \in E$. Set the value of this edge equal to $n_u$. (This represents amount $n_u$ being added to the pot.) Connect $u'$ in the second graph to $v$ in the first graph if edge $(u, v) \in E$. Set the value of this edge equal to $-n_u$. (This represents amount $n_u$ being taken out of the pot.) The resulting graph is bipartite: there are no edges between nodes in the first graph; there are also no edges between nodes in the second graph.

✓ We modify this new graph in one further manner. We create a new node $u_T$ with no outgoing edges. For every edge $(u, u_0)$ or $(u, u'_0)$ that has weight $w$, we delete that edge and replace it with an edge $(u, u_T$ with weight $w$. That ~~is,~~ node $u_0$ in either graph has no incoming edges, as all edges to $u_0$ have been redirected to $u_T$. Now, a path from $u_0$ to $u_T$ represents a legal cycle in the original social network graph.

Notice we can construct this graph in $O(n+m)$ time.

Notice that if there is any path from $u_0$ to $u_T$ that contains a positive weight cycle, then we are done. (A positive weight cycle that is not part of a path from $u_0$ to $u_T$ is not helpful.) So we run $n+1$ iterations of Bellman-Ford on the new graph, modified to find a maximum length path (rather than a minimum length path). In the end, we check if the estimate at $u_T$ has changed in the final iteration; if so, then here is a positive weight cycle on the path, and we output, "Yes, play the game!"

✓ If not, we look at the length of the longest path from $u_0$ to $u_T$. If the path is of lengh at least $2n_{u_0}$, then output "Yes, play the game!" Otherwise, output "No, do not play."

## **Problem 8. After you are done.** [0 points]

Circle the image that best represents how you feel right now.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 😀 | 😆 | 😄 | 😎 | 🤔 | 😐 | 😴 | 😁 | 😠 | 😞 |

## Scratch Paper

## Scratch Paper

## Scratch Paper

## Scratch Paper

**End of Paper**