CSCI 3104 Spring 2017 **Problem Set 10**

Matthew Niemiec (06/24)
Mehmet Karaoglu (07/30)

1. (40 pts total) Every young wizard learns the classic NP-complete problem of determining whether some unweighted, undirected graph $G = (V, E)$ contains a Hamiltonian cycle, i.e., a cycle that visits each vertex exactly once (except for the starting/ending vertex, which is visited exactly twice). The following figure illustrates a graph and a Hamiltonian cycle on it (see Chapter 35.5.3 in CLRS for a proof of NP-completeness).

   (a) Ginny Weasley is working on a particularly tricky instance of this problem for her Witchcraft and Algorithms class, and she believes she has written down a "witness" for a particular graph G in the form of a path P on its vertices. Explain how she should verify in polynomial time whether P is or is not a Hamiltonian cycle. (And hence, demonstrate that the problem of Hamiltonian Cycle is in the complexity class NP.)

   To determine whether a "witness" path P is a Hamiltonian Cycle, we can first simply check whether there exists an edge (v_i, v_{i+1}) in the graph G for $i = 1, 2, 3...n$ (n is the number of vertices in P), where $v_1, v_2, v_3...v_n$ are all of the vertices in P, and if $v_n = v_1$, v_1 being the source vertex and v_n being the last visited vertex of the "witness" path, since a cycle needs to start and end on the same vertex. We would also need to check if every vertex in P gets visited exactly once - except for the source vertex, which should get visited exactly twice. If we have an adjacency-list representation of G, then we can check whether there exists an edge (v_i, v_{i+1}) in G for each v naively in $O(E)$ time, where E is the set of all edges in the graph. We can check if each vertex is visited exactly once in $O(1)$ time for each v if we use an array to store number of visits for each vertex. Since we need to do these checks for all n vertices, we can verify this "witness" in $O(nE)$ time. This shows us that the time complexity is polynomial, and therefore, the method of determining if a cycle is Hamiltonian is in the NP complexity class.

   (b) For the final exam in Ginny's class, each student must visit the Oracle's Well in the Forbidden Forest. For every bronze Knut a young wizard tosses into the Well, the Oracle will give a yes or no response as to whether a particular graph contains a Hamiltonian cycle. Ginny is given an arbitrary graph G, and she must find a Hamiltonian path on it to pass her exam.

Describe an algorithm that will allow Ginny to use the Oracle to solve this problem by asking it a series of questions,

each involving a modified version of the original graph G. Her solution must not cost more Knuts than a number that grows polynomially as a function of the number of vertices in G. (Hence, prove that if we can solve the Hamiltonian cycle *decision* problem in polynomial time, we can solve its *search* problem as well.)

The first step of our algorithm would be to ask the Oracle if the original graph G contains a Hamiltonian Cycle, because if it doesn't, there is no point of searching the graph for one. If it *does* contain a Hamiltonian Cycle, we can first select a vertex $v \in V$ of G and remove an edge (v, u) that is coming out of v. Then we ask the Oracle if the resulting graph G' is a Hamiltonian Cycle. We keep on removing all of the edges that come out of v and asking the Oracle if it is a Hamiltonian cycle until the Oracle replies with an answer of "no." Then, we know that the edge that we just removed has to be part of the Hamiltonian Cycle, and we add that edge back into the graph. We move onto the the vertex u, which is connected to v by the edge that we just found was part of the Hamiltonian Cycle, and keep on removing edges and asking the Oracle if the resulting graph is a Hamiltonian Cycle until we end up back at our starting vertex. This graph will ask the Oracle $O(E)$ times, which also implies that the solution has polynomial time complexity with respect to the number of vertices in G.

2. (25 pts) In the review session for his Deep Wizarding class, Dumbledore reminds everyone that the logical definition of NP requires that the number of *bits* in the witness w is polynomial in the number of bits of the input n. That is, $|w| = poly(n)$. With a smile, he says that in beginner wizarding, witnesses are usually only logarithmic in size, i.e., $|w| = O(\log n)$.

   (a) Because you are a model student, Dumbledore asks you to prove, in front of the whole class, that any such property is in the complexity class P.

   We know that a brute force algorithm is exponential in time. For example, if there are n witnesses, then we have to check $O(e^n)$ possibilities. Now, however, if each witness is $O(\log n) = O(\lg n)$, then the number of possible witnesses we have to check is 2 to the length of a witnesses, which is $O(2^{\lg n})$, which, by logarithmic properties, is $O(n)$, a polynomial runtime in the worst case with the worst approach.

   (b) Well done, Dumbledore says. Now, explain why the logical definition of NP implies exponential time for any brute force algorithm.

   There are n bits in a witness, and, by the definition of NP given above, each witness has polynomial, or $O(n^c)$, where c is a constant, bites. Then thinking about the definition of binary, there are $O(2^{n^c})$ possible combinations of witnesses. Checking each one takes $\Omega(1)$ time, so that means that we have a runtime of no less than $O(2^{n^c})$, which is exponential.

3. (15 pts total) Determine whether each claim is true or false. Justify your determination. If the claim is false, state the correct asymptotic relationship as O, Θ, or Ω.

(a) $n + \pi = O(n^2)$

True. More specifically, $n + \pi = \Theta(n + \pi) = \Theta(n) = O(n^2)$

(b) $4^{2n} = O(4^n)$

False. $4^{2n} = (4^n)^2 = \Theta((4^n)^2) = \omega(4^n) \neq O(4^n)$

(c) $n^n = \Theta(n!)$

Here we need the fact that $n^{n-1} \geq n!$, since $n \times n \times ... \times n \times 1 \geq n \times (n-1) \times ... \times 1$. Note they both have n terms, the last of which is 1. Only the terms of n^n are at least as large as those of $n!$.

False. $n^n = n \times n^{n-1} \geq n \times n! = \Theta(n \;\times n!) = \omega(n!) \neq \Theta(n!)$

(d) $1/(3n) = \Omega(1)$

False. $1/(3n) = \Theta(1/(3n)) = \Theta((1/3)(1/n)) = \Theta(1/n)$

Since $\lim_{n\to\infty} 1/(1/n) = \infty$

$\Theta(1/n) = o(1) \neq \Omega(1)$

(e) $\ln^2 n = \Theta(\lg^2 n)$

True. $\ln^2 n = (\ln n)^2 = (\frac{\lg n}{\lg e})^2 = (\frac{1}{\lg e})^2(\lg n)^2 = \Theta((\frac{1}{\lg e})^2(\lg n)^2) = \Theta((\lg n)^2) = \Theta(\lg^2 n)$

4. (10 pts) Consider the following function:

```
def foo(n) {
     if (n > 1) {
        print( ''hello'' )
        foo(n/4)
        foo(n/4)
}}
```

In terms of the input n, determine how many times is "hello" printed. Write down a recurrence and solve using the Master method.

The recurrence relation for this function is: $\underline{foo(n) = 2foo(n/4) + 1}$

We can see that this recurrence relation is solvable by the Master method because it is in the form of $T(n) = aT(n/b) + f(n)$, where $a = 2$ and $b = 4$. One of the properties of the Master method is that if $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$. For our function `foo(n)`, $f(n) = 1 = O(n^{\log_4 2 - (1/2)}) = O(n^{(1/2)-(1/2)}) = O(n^0) = O(1)$. Therefore, we can conclude that $foo(n) = \Theta(n^{\log_4 2}) = \underline{\Theta(n^{1/2})}$. More specifically, the formula is $foo(n) = n^{1/2} - 1$, where `foo(n)` outputs the number of "hello" statements that will be printed.

5. (10 pts) Minerva McGonagall gives you two arrays A and B, which both contain n integers sorted in ascending order. Give an algorithm that will determine in quadratic time or better whether they have an element in common. Using a loop invariant, prove that your algorithm is correct.

```
def CommonElement(A, B):
    hasCommonElement = false
    i = 0
    j = 1
    while i < A.length and j < B.length:
         if A[i] == B[j] then return true
         else if A[i] < B[j] then i++
         else j++
     return false
```

We know that the arrays are sorted, so if we start at the beginning of both array, we are comparing the smallest numbers. If they are the same, then that confirms the presence of a common element, and it will return true.

Initialization: When i and j are both 0, we know that either the first element in the arrays are the same or they're not. If they are, we end, and if they're not, then we know that there is no element smaller than the first element in the element with the larger first element. Therefore, if there is a match, we must find an element larger than the first element in the array with the smaller first number.

Maintenance: If A[i] equals B[j], then the algorithm returns. If not, then we know that there is no element in the array with the larger number that equals the smaller number being compared. If there were, it would have been found by going through the array of the smaller number. Therefore, the only way to create a match is no see if there is a larger element in the array of the smaller element that equals the larger element of the two, so we increment the array of the smaller element.

Termination: At some point either a match will be found, or one of the indices will equal the length of its corresponding array, causing an the loop to exit. This is valid because the element currently being checked in the array not going out of bounds must, by the algorithm, be larger than the last element of the array going out of bounds. Therefore searching for a larger number is guaranteed to not produce a common element.

In the worst case, this loop will increment both i and j to n, so it will run $2n$ times, so it runs in $O(n)$ time, which is polynomial.