Paul Winzer
CS 520
Hood College
Spring 2018

Homework 4

1. [COMPLETE]
    a. There are 24 possible solutions, or 4!. In deciding how to assign these tasks, anyone can get the first task (4 possible choices), then only three people can get the next task (3 for every 4 in the first choice, so 3 * 4 = 12), then only two people can get the third task (2 for every 12 choices in the last step, so 2 * 12 = 24). At that point only one person remains to do the last task, and 1 * 24 = 24.
    b. I choose to assign the tasks to the same order of people each time, as such: Mother, Father, Daughter, Son. Just clarifying, because you could choose to delegate the same order of tasks to a different order of people. See Figure 1 below for state space representation of the entire search. Non-terminal nodes represent incomplete solutions that are investigated further. Terminal nodes are either green, white, or red. Read from left to right, green terminal nodes represent new complete optimal solutions. White terminal nodes represent complete solutions that are not optimal. Red terminal nodes represent incomplete solutions, the complete solution supersets of which are not investigated further because the costs of the incomplete solutions have already met or exceeded the cost of the globally optimal solution. The first complete solution we find is assigned as the global maximum. Then next is cheaper so it replaces the global maximum. From there, only one complete solution is explored and it is more expensive than the existing solution so we do not replace it as the globally optimal solution.
    c. This algorithm generated 19 partial solutions. For this problem and other problems of this size or smaller, it's not a huge win to implement a Branch and Bound strategy, because the absolute number of operations avoided is small compared to the potential total of operations of a full space search. The best case scenario for a problem this size (that is, the first solution searched turns out to be globally optimal and we can abandon all other searches as early as possible), results in 6 partial solutions, substantially less than the 24 possible complete solutions, but these 24 solutions don't take very long to generate if we don't Bound the search (or if we explore the data in the "unluckiest" order). However, for larger problems the savings associated with Bounding are potentially very significant because of the combinatorial explosion of a factorial number of potential solutions. In summary, this strategy is worth implementing on problems of significant size, and a job assignment problem with a 4x4 possibility matrix is trivial.
2. [COMPLETE] We don't have enough information. Considering the black box algorithm with time complexity $T = O(n^2)$ completes in 24 seconds with an input size of 16, the largest data set that the algorithm can complete in under 60 seconds is potentially very

large and cannot be determined without more information about the constant overhead cost of the algorithm. It's possible that an algorithm whose time complexity is $T = O(n^2)$ is more accurately determined to be $T_{seconds} = \frac{n^2}{256} + 23$. In this case the constant value (23) heavily influences the run time at relatively smaller input sizes. Plugging 16 in for n, we get $T = 24$ seconds. Plugging in 100 for T, we solve and get $n \approx 140.399$. But what if the algorithm's time complexity is actually $T_{seconds} = \frac{n^2}{512} + 23.5$. In that case, $n = 16$ still yields $T = 24$ but now $T = 100$ means that $n \approx 197.909$. And if the complexity is $T_{seconds} = \frac{n^2}{1024} + 23.75$, an input of $n \approx 279.428$ will complete in $T = 100$ seconds. So while all these are $T = O(n^2)$ time complexity, continually dividing the leading coefficient by a larger value and increasing the constant cost to a value infinitely close to 24 will satisfy the ($n = 16$, $T = 24$) constraint but allow us to continually raise the input size n and calculate algorithm completion time at 100 seconds.

3. [PART COMPLETE] Firstly, the Miller-Rabin primality test uses modular arithmetic to probabilistically test whether a number is prime or composite. It can determine the compositeness of a number with absolute certainty. However, it cannot be absolutely certain about the primality of a number, but only strongly suggest that a number is prime. The mathematics that form the basis of the algorithm are founded on modular arithmetic and its congruence, the properties of pseudoprimes, Fermat's Little Theorem, and the Generalized Riemann Hypothesis. Modular congruence means that two numbers produce the same value when modularly divided by the same number. For example, $3 \equiv 7 \ (mod\ 4)$ because when 3 mod 4 and 7 mod 4 both produce the remainder 3. In fact, any positive integer that is one less than a factor of 4 will produce 3 when mod 4, and so all such integers are congruent (mod 4).

   To begin, every if *n* is a prime number greater than 2, and therefore also odd, then *n -1* must be even, and can be represented as $2^s \times d$, where *d* is odd and therefore the value of *s* has been maximized by factoring all 2's out of *n-1*. Both *s* and *d* are positive integers, as well, because of prime *n* is greater than 2. For every *a* in the finite field (Z/nZ)*, and for some *r* greater than or equal to 0 and less than or equal to *s-1*, either of the following statements must be true: $a^d \equiv 1 \ (mod\ n)$ OR $a^{2^r \cdot d} \equiv -1 \ (mod\ n)$. The necessary truth of either of these statements can be proven by Fermat's Little Theorem: for any prime number *n*, $a^{n-1} \equiv 1 \ (mod\ n)$. By continually taking the square root of the left side of this congruence, we arrive at either 1 or -1. Either way we arrive at one of the equalities above that was said to have been true.

   The Miller-Rabin test leverages the contrapositive of the logic above. Remember that the contrapositive of a statement $a \rightarrow b$ is $\neg b \rightarrow \neg a$. That is, if 'a implies b' is true, then 'not b implies not a' is also true. And similarly, if the former is false then the latter must be false. They are logically equivalent. So the contrapositive of the above statement is that the two congruences are incongruent, or $a^d \neq 1 \ (mod\ n)$ OR $a^{2^r \cdot d} \neq -1 \ (mod\ n)$. (Those congruence symbols should have three lines but Google Docs doesn't show that character in the equation tool.) If we can see that the first incongruence is true or that the second is true for all r where $0 \leq r \leq s-1$, then *n* is not

prime. There are, however, values for *a* that are considered 'strong liars', that is, for a composite number *n*, *a* will show that the above equalities can be show, despite the fact the *n* is not prime. It's proven that at least ¾ of possible values for *a* will witness that composite *n* is actually composite, so running the Miller-Rabin test algorithm multiple times with different values for *a* greatly decreases the chances of being lied to by a strong liar value.

As an example, let's test the primality of the integer 221. We calculate $n - 1 = 220$, and the highest power of 2 that goes into 4 or 2^2, so we can write 220 as $2^2 \cdot 55$. So our *s* = 2 and *d* = 55. The we have to pick a random number for *a*, let's use 137. First we test $a^d \equiv 1(mod\ n)$, and $137^{55} mod\ 221 = 188.$ This does not equal 1, so our first equality hasn't been realized. Nor has our second equality for an *r* value of 0, because $a^{2^0 \cdot d} = a^d$. So we try with *r* =1. But $137^{110} mod\ 221 = 205$, and 205 is not modularly congruent to -1 with regard to our prime tester 221. We don't check any more values for *r* because we have only check for values $0 \le r \le s - 1$. Because none of our equalities were realized, we can say that 221 is certainly composite. If any of our values for *r* had resulted in one of the above equalities, we could say that 221 was probably prime, but we would want to run the algorithm again, because the more *a* values we test, the more certain we can be about primality.

The algorithm's obvious shortcoming is that is probabilistic, that is, strong liars for pseudoprimes are not affirmatively prime determinants. The time complexity of this algorithm is $O(k\ log^3 n)$ where k is the number of times that we execute our algorithm to become more and more certain that the primes are primes.

[Implementation included in attached files]

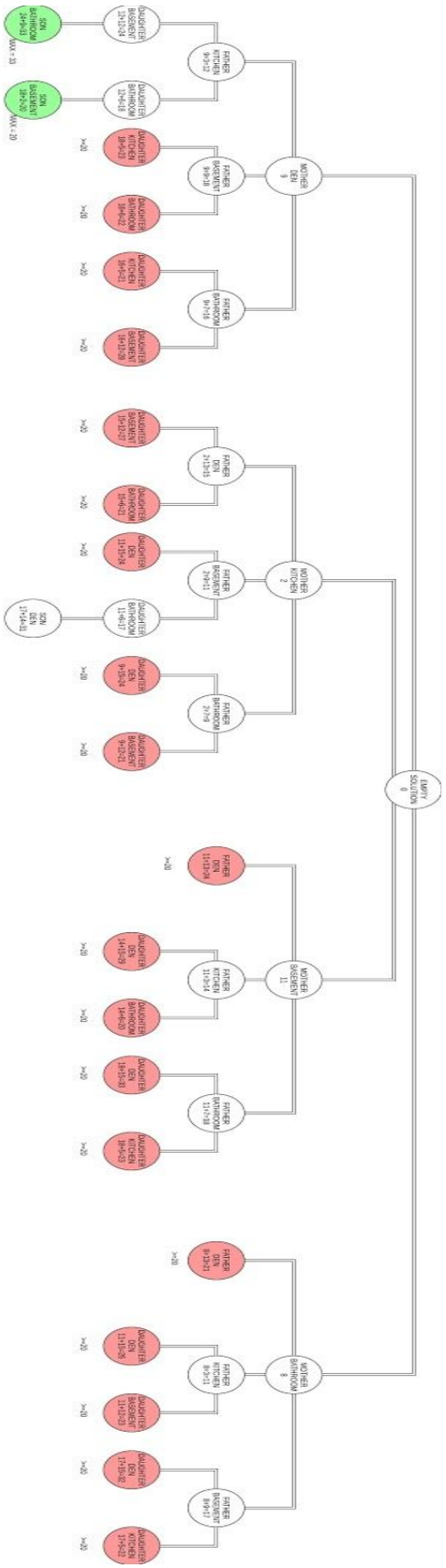

Figure 1 (From Problem 1)