Cairo University
Faculty of Computers and Artificial Intelligence
Software Engineering Program

Algorithms Design and Analysis
Lecture Task 02

Name: Khalid Ibrahim Abdallah Shawki ID: 20206018

Q1: A primality test is an algorithm for determining whether an input number is prime or not, present 4 different methods

A primality test is an algorithm for determining whether an input number is prime. Among other fields of mathematics, it is used for cryptography.

4 Methods to Solve primality test:

1. School Method
2. Fermat Method
3. Miller-Rabin Method
4. Solovay Strassen Method

1. School Method

We Iterate through all number from 2 to n-1 and for every number check if it divides n, If we find any number that divides we return false. The complexity is O (n).

2. Fermat Method

a) Repeat following k times:
   a. Pick a randomly in the range [2, n - 2]
   b. If gcd(a, n) ≠ 1, then return false
   c. If an-1 ≢ 1 (mod n), then return false
b) Return true [probably prime].

3. Miller-robin Method

It returns false if n is composite and returns true if n is probably prime. k is an input parameter that determines accuracy level. Higher value of k indicates more accuracy.

bool isPrime(int n, int k)
1) Handle base cases for n < 3
2) If n is even, return false.
3) Find an odd number d such that n-1 can be written as d*2r.
Note that since n is odd, (n-1) must be even and r must be
greater than 0.
4) Do following k times
if (millerTest(n, d) == false)
return false
5) Return true.

This function is called for all k trials. It returns false if n is composite and returns true if n is probably prime.
d is an odd number such that d*2r = n-1 for some r>=1
bool millerTest(int n, int d)

1) Pick a random number 'a' in range [2, n-2]
2) Compute: x = pow(a, d) % n
3) If x == 1 or x == n-1, return true.

Below loop mainly runs 'r-1' times.
4) Do following while d doesn't become n-1.
a) x = (x*x) % n.
b) If (x == 1) return false.
c) If (x == n-1) return true.

4. Solovay–Strassen Method

The Solovay–Strassen primality test is a probabilistic test to determine if a number is composite or probably prime. Before diving into the code we will need to understand some key terms and concepts to be able to code this algorithm.

```
Algorithm for Jacobian:
Step 1    //base cases omitted
Step 2     if a>n then
Step 3         return (a mod n)/n
Step 4     else

Step 5         return (-1)^((a - 1)/2)((n - 1)/2)(a/n)
Step 6     endif
```

```
Algorithm for Solovay-Strassen:
Step 1    Pick a random element a < n
Step 2    if gcd(a, n) > 1 then
Step 3        return COMPOSITE
Step 4    end if
Step 5    Compute a^(n - 1)/2 using repeated squaring
          and (a/n) using Jacobian algorithm.
Step 6    if (a/n) not equal to a^(n - 1)/2 then
Step 7        return composite
Step 8    else
Step 9        return prime
Step 10   endif
```

Q2: in knapsack problem discuss how the backtracking algorithm work?

Given *n* positive weights w_i, n positive profits p_i , and a positive number M which is the knapsack capacity, the 0/1 knapsack problem calls for choosing a subset of the weights such that

$$S_{i = 1 \text{ to } k} w_i x_i \text{ £ } M \text{ and}$$
$$S_{i = 1 \text{ to } k} p_i x_i \text{ is maximizd}$$

The x's constitute a zero-one valued vector.

The solution space for this problen consists of the 2^n distinct ways to assign zero or one values to the x's.
Thus the solution space is the same as that for the sum of the subsets problem.

Bounding function is needed to help kill some live nodes without actually expanding them.
A good bounding function for this problem is obtained by using an upper bound on the value of the best feasible solution obtainable by expanding the given live node and any of its descendants. If this upper bound is not higher than the value of the best solution determined so far then that live node may be killed.

Here we use the fixed tuple size formulation.
If at node Z the values of x_i , 1 £ i £ k have already been determined, then an upper bound for Z can be obtained by relaxing the requirement x_i = 0 or 1 to 0 £ x_i £ 1 for k+1 £ i £ n and use the greedy method to solve the relaxed problem.

Procedure **Bound(p,w,k,M)** determines an upper bound on the best solution obtainable by expanding any node Z at level k+1 of the state space tree.

The object weights and profits are W(i) and P(i).
p = $S_{i = 1 \text{ to } k}$ P(i)X(i) and it is assumed that P(i)/W(i) ³ P(i+1)/W(i+1), 1 £ i £ n

```
procedure BOUND(p,w,k,M)
// p: the current profit total
// w: the current weight total
// k : the index of the last removed item
// M : the knapsack size
// the return result is a new profit

global  n , P(1:n) , W(1:n)
integer k, i l real b,c,p,w, M
      b := p ;  c := w
      for  i := k+1 to n do
           c := c + W(i)
           if c < M then b := b + P(j)
                    else  return (b + (1 - (c - M)/W(i))*P(i))
           endif
      repeat
      return (b)
end BOUND

procedure Knapsack(M,n,W,P, fw,fp,X)
// M : the size of the knapsack
// n : the number of the weights and profits
// W(1:n) : the weights
// P(1:n) : the corresponding profits ;  P(i)/W(i) ³  P(i+1)/W(i+1),
// fw : the final weight of the knapsack
// fp : the final maximum profit
// X(1:n), either zero or one ; X(k) = 0  if W(k) is not in the knapsack else X(k) = 1


1.  integer n,k, Y(1:n), i , X(1:n) ; real M, W(1:n), P(1:n), fw, fp, cw, cp ;
2.    cw := cp := 0  ; k := 1 ; fp := -1    // cw = current weight, cp = current profit
3.  loop
4.  while  k £ n  and cw + W(k) £ M do // place k into knapsack
5.       cw := cw + W(k) ; cp := cp + P(k) ; Y(k) := 1 ; k := k+1
6.  repeat
7.  if k > n then fp := cp; fw := cw ; k := n ; X := Y   // update the solution
8.  else  Y(k) := 0     // M is exceeded so object k does not fit
9.  endif
10. while BOUND(cp,cw,k,M) £ fp do // after fp is set above, BOUND = fp
11.      while k <>  0 and Y(k) <>  1 do
12.          k := k -1   // find the last weight included in the knapsack
13.      repeat
14.      if k = 0 then return   endif    // the algorithm ends here
15.      Y(k) := 0 ; cw := cw - W(k) ; cp := cp - P(k)  // remove the k-th item
16.  repeat
17.  k := k+1
18.  repeat
19.  end knapsack
```

3. show the stpes of solving money change problem by dynamic programming

Example: coins = {1, 2, 5}
Amount = 11

0	1	2	3	4	5	6	7	8	9	10	11
0	12	12	12	12	12	12	12	12	12	12	12

1) Min (0+1, 12) = 1

0	1	2	3	4	5	6	7	8	9	10	11
0	1	12	12	12	12	12	12	12	12	12	12

2) Min (1+1, 12) = 2

0	1	2	3	4	5	6	7	8	9	10	11
0	1	1	12	12	12	12	12	12	12	12	12

3) Min (1+1, 12) = 2
Min (1+1, 2) = 12

0	1	2	3	4	5	6	7	8	9	10	11
0	1	1	2	12	12	12	12	12	12	12	12

4) Min (2+1, 12) = 3
Min (1+1, 3) = 2

0	1	2	3	4	5	6	7	8	9	10	11
0	1	1	2	2	1	12	12	12	12	12	12

5) Min (2+1, 12) = 3
Min (2+1, 3) = 3
Min (0+1, 3) = 1

0	1	2	3	4	5	6	7	8	9	10	11
0	1	1	2	2	1	12	12	12	12	12	12

6) Min (1+1, 12) = 2
Min (2+1, 2) = 2
Min (1+1, 2) = 2

0	1	2	3	4	5	6	7	8	9	10	11
0	1	1	2	2	1	2	12	12	12	12	12

7) Min (2+1, 12) = 3
Min (1+1, 2) = 2
Min (1+1, 2) = 2

0	1	2	3	4	5	6	7	8	9	10	11
0	1	1	2	2	1	2	2	12	12	12	12

8) Min (2+1, 12) = 3
Min (2+1, 3) = 3

0	1	2	3	4	5	6	7	8	9	10	11
0	1	1	2	2	1	2	2	3	12	12	12

9) Min (3+1, 12) = 4
Min (2+1, 4) = 3
Min (2+1,3) = 3

0	1	2	3	4	5	6	7	8	9	10	11
0	1	1	2	2	1	2	2	3	3	12	12

10) Min (3+1, 12) = 4
Min (3+1, 4) = 3
Min (1+1, 3) = 3

0	1	2	3	4	5	6	7	8	9	10	11
0	1	1	2	2	1	2	2	3	3	2	12

11) Min (2+1, 12) = 3
Min (3+1, 3) = 3
Min (2+1, 3) = 3

0	1	2	3	4	5	6	7	8	9	10	11
0	1	1	2	2	1	2	2	3	3	2	3