1.
a = 4841247740021026788214420074996258540545281

b = 7120104115728581516059224292256265185280 01

n = a * b = 34470187958985406639333538918593330781169908420316995503493 2

933955664207902800691328 1

$\emptyset(n) = (a-1)(b-1)$ =34470187958985406639333538918593330781169852887

735479564643895192141378571429478400 00

The smallest valid RSA public key should be relatively prime to $\emptyset(n)$, so I write a python procedure to compute this value [1]. The principle is that starting from 2, add 1 each time until the greatest common divisor of it and $\emptyset(n)$ equals 1 (relatively prime). Inputting the parameter $\emptyset(n)$, we get the smallest valid RSA public key e 179.

I write a python procedure to compute the multiplicative inverse using extended Euclid algorithm [2].

The inverse d of 179 mod $\emptyset(n)$ is:

38514176490486487865177138456528861208011008813112267669993178985632 82
52194742775419

So, the public key is

{179 (public key),

34470187958985406639333538918593330781169908420316995503493293395566 42
079028006913281 (n)},

The private key is

{38514176490486487865177138456528861208011008813112267669993178985632 82
52194742775419 (private key),

34470187958985406639333538918593330781169908420316995503493293395566 40
79028006913281 (n)}

2.
https://asecuritysite.com/encryption/random3

This webpage is an online prime number generator, I entered 1024 which means it will generate prime number of approximately 2^{1024} ($>10^{256}$).

a =

142067892537710038404728203368355245008715878078991689417267746803540137968406393230902896111023520403849882954090477365027830148120971559436319701479923864343284491573463263596198109898190549888399454023923106284617885316115699288809409587947685548660627266243020382061171313779173459763477684321653625 6907593

b =

397230441808518770638457397772646427603802979607012244064362853378191450315397130929319847724362872954104798117162012613318832177451053958713005834695888790224003576508637846811611221470808187671802610742448863772353880327028111045361712510641682079966181174796885491099496129341163321862865764284488282 8279

n = a * b =

5643369171955972550742291655346337369828313868176659207049306724895849492235000938973263052427377353912320911436213525896819039221666981632209771239550443541518564138548119142789995772155085324407980751295384682237712367896381151957937853698793987452967269976080051490782381123192328309953967796750479119434137174848700743264270013280889405885321856003698555642046141751883371500944319785655255954111369662702308195069903285548306489916669565676596431816073830633329205082569391920188985281933948062598563923073549788906782975140820819902641902555473287002521663288093693089719399015653300047886785682559486168222447

$\emptyset(n) = (a-1)(b-1) =$

56433691719559725507422916553463373698283138681766592070493067248958494922350009389732630524273773539123209114362135258968190392216669816322097712395504435415185641385481191427899957721550853244079807512953846822377123678963811519579378536987939874529672699760800514907823811231923283099539677967504791194339553839119821813488014393377437859975527598276588627282224377197420122179443736793314210732279098550030478323040966069219467765508034887212888115311243178899635202333450648719116260498886766939429842079753817962449297017920023095100116354249322633079187725081838007199977362649274491180657222629145010824865 76

The smallest valid RSA public key should be relatively prime to $\emptyset(n)$. By implementing SmallestPublicKey.py [1] we get the smallest valid RSA public key 5.

By implementing ExtendEuclid.py [2] we get the inverse d of 5 mod $\emptyset(n)$ is:

451469533756477804059383332427706989586265109454132736563944537991667959378800075117861044194190188312985672914897082071745523137733358530576781699164035483321485131083849531423199661772406825952638460103630774579016989431710492156635028295903518996237381598086404119262590489855386264796317423740038329554716430712958574507904115147019502879804220786212709018257795017579360977435549894346513685858232788400243826584327728553755742124064279097703104922489945431197081618667605189752930083991094135515438736638030543699594376143360184760800930833994581064633501800654704057599818901194195929445257781033160086598926I

https://www.random.org/integers/

This webpage is an online random number generator, I used it to generate a random message m ($>10^{500}$) and a secret random number r between 0 and n-1:

m =

989341685881536206614879811226442176696794054183020113469687580534700488037777246497088795021641632603311279612984798417044537830394798241345256954980879751731525312571158114896194426315035791665639180362095298518921819575423824559093693272525402946572747524569730265327077859600082056160131745427634112489135044693876086916612762856183039921824068265551615711386910110799037683036465436615934031928287349232508877540167468968229931495901393252735488402915395226252851600289713026918253466980403281610701173519553982

secret random number r between 0 and n-1:

115109923663824572071580769927096633717042988572358250587609110188213330225309255762076120236760450432204287598626983342300896045653146087051541702713792 79

The blinded message $m_b = m \times r^e =$

199944811124900359263285134766585484562033380639515997686080578562160729014525151983686637056284276474024010506093430986818682275005205935609832477251611104790445831018613618113513856053380158617782068561671530217667968935153016577614854159418761177269126101211189598117054666596588275663520582308772326712654959098261722279072047920491109250770813738037838371902445243649265858372813678538719494962414555774752664974806984677771035217287794022234463343161980540742084585253221974218603413248340964463845897435693031797508321071905780033559591944907468943615992939480236065142961852046018524719659747188240146413574731500198441071204529983324965244686981742027547594807919965768860401884107399382084228304306751937339082633736624300074948333077659581740736055049543643703200004391464586109337617799353446752183118312123210014564960406374511669118081422544763048792109254297209002885453828730591191111991936986905166911307098606670578630818877583640337282757438817288245580769857825405030084388564385779280764210788971174940899185948146695934882467523391I

2121149710813666915226515434756893763622352229690981742030849540313944
4267856622050967523095006694955712955731199642565601148572600886122380
1054253194914523877895806356493488361080779784608737515231773359832097
265807998823362115862818

By implementing Mod.py [3] we get the ciphertext s_b of m_b encrypted by secret key d:

$s_b = {m_b}^d$ mod n =

1904355323618778421509433650983970574299644816471371382582674645240622
1097468712528560686229997967490017927613685566606997316229211316348385
3272912479778412040705384951832556312661388676972222904402666549159506
1047506377517883039702322043690081931124268638282411315589811682062532 7
4997431619378051017413064084532963275936121072386574905625781846813890
6518803876759400447525723254822002195990633149319351292584221491167510
0194801779764006978209874146406566308210170885608430215654374822319933
7806608318718705333928370896360187774655355196342204111866110629010887 5
437345872115088940607640564185034958814595424167440239

By implementing Mod.py [3] we get the correct ciphertext s of plaintext m encrypted by secret key d:

s = m^d mod n =

8394965568379727066182483008470502092498603603584110061615783875316593
9716511045509189859478530152468722247927674860724698514181331180097704
6584724244874228025521296858884399274048829205656040618389208486293809
3182431487264992092623981064128498390257381699101064269400306715828017
0782565922183501422568508152502030558652159866330304745024271046485298
0734201020508838786671024347968144126026615298744855125724156148676710
4297924745234796286918887296686167249031763469828218878523433168611396
8012455868068434380738024873482667609578740761295119117038980713105374
688619784415619273315497535286918804435397494102060312 2

By implementing ExtendEuclid.py [2] we get the inverse r^{-1} of r mod n is:

2969204593263916667513352194730924843641692120889681054689502948828543
7127689038016045891108097213925669645114794969647878873518485201897085
7919360273845688426911022302076913899441730497034410568767191693730622
5690129824367561461181777697421211093015815359801498428903563739461 67
5189138488206424710590477845847452281733024360748445592205373919856704
2160530272804123221859230797198861485594556954573663566961531445553059
5915770478851554335827175457302487146926826415109706267097421944776110
2030086977217921726815666705194417741056269882347144289945936185103471
4290966872678457138247430568346812407496927181986414534 3

The ciphertext s_b of blinded message m_b multiply by the inverse r^{-1} of r mod n and the ciphertext s of plaintext m should be congruent modulo n. By implementing Mod.py [3] we get:

$s_b r^{-1} \bmod \mathrm{n} =$

8394965568379727066182483008470502092498603603584110061615783875316593
9716511045509189859478530152468722247927674860724698514181331180097704
6584724244874228025521296858884399274048829205656040618389208486293809
3182431487264992092623981064128498390257381699101064269400306715828017
0782565922183501422568508152502030558652159866330304745024271046485298
0734201020508838786671024347968144126026615298744855125724156148676710
4297924745234796286918887296686167249031763469828218878523433168611396
8012455868068434380738024873482667609578740761295119117038980713105374
688619784415619273315497535286918804435397494102060312 2 = s

In this way, we can use the blinded message to recover the original plaintext.

3.

Man-in-the-middle attack:

Suppose A and B want to exchange keys, attacker can disguise himself as B in order to communicate with A. By doing so, the attacker shares a secret key k1 with A. After that, the attacker disguises himself as A to exchange data with B so that he gets another secret key k2 with B.

Attack success.

(a) Message Authentication Codes

By applying MAC, the secret key for MAC is only shared by A and B which means for A (B), only B (A) can generate the correct MAC. The adversary does not know this secret key, so he cannot generate a correct MAC for a given message. As a result, when the attacker wants to disguise himself as A or B to send a wrong message, the receiver B or A will find that the message has been modified through MAC authentication.

Attack failed.

(b) Public Key Digital Signatures

Digital signature method is more secure than MAC, because only person with a private key can generate the correct signature. If the adversary wants to disguise himself as B, he cannot generate the correct signature of B, so the receiver A can realize that the message has been modified or is totally wrong.

Attack failed.

(c) Hash functions.

There is no secret key shared by A and B, so the adversary can also successfully disguise by generating the correct hash value. The attack method is the same as above.

Attack success.

4.

This method is really easy to break. Because each alphabet character is encrypted separately, there are only 26 different cipher characters in ciphertext corresponding to letter a to letter z. The most efficient attack against this method is chosen ciphertext attack (CCA). Different from the attack method in textbook, we can simply choose these 26 different cipher characters and get the plaintext. The cipher characters in the ciphertext and the alphabet characters in the plaintext are one-to-one correspondence.

If you cannot use this attack method, cryptanalytic attack is also practical. Because ciphertext reflects the frequency data of the original alphabet. Without confusion and diffusion, attacker can easily break this encryption system by using statistical characteristics.

A simple countermeasure to these attack above is to encrypt the plaintext in blocks. Notice that the binary value of each block should be less than n. In addition, you can also add secure hash function to enhance security (integrity).

5.

(a)

Step 1: A sends a message to responder B. The message includes identity of A (ID_a) and a unique identifier N_a encrypted by K_a. N_a is a nonce generated by A and K_a is the master key known only to KDC and A.

Step 2: B sends a message to KDC which can be divided into two parts. One is the message just received from A, the other includes identity of B (ID_b) and a unique identifier N_b encrypted by K_b. N_b is a nonce generated by B and K_b is the master key known only to KDC and B.

Step 3: The KDC responds a message which can be divided into two prats as well. One is for B encrypted by K_b and it contains three items which are the session key K_s, identity of A, nonce N_b. The other part is for A encrypted by K_a which contains the session key K_s, identity ID_b and nonce N_a.

Step 4: B stores the session key Ks for future use and sends a message originated by KDC for A.

(b)

Consider the most extreme case, there are n entities want to communicate in pairs, so there are n(n-1)/2 session keys needed simultaneously. Let’s assume n equals 10000. For a certain entity, it should store its master key shared with KDC and 9999 session keys shared with other entities for each connection. For KDC, if it doesn’t need to store the session key history, it should distribute 50 million session keys for each connection and only needs to store 10000 master keys for these 10000 entities.

(c)

As we can see clearly, the alternative key distribution method has 4 steps and the scheme in the lecture (textbook) has 5 steps, so the efficiency of alternative key distribution method is better (4 < 5). The fourth and fifth steps of the scheme in the lecture (textbook) is added for authentication. By sending nonce N_2 encrypted by the session key K_s to the initiator A, only A can respond a message with $f(N_2)$ encrypted by K_s. So the scheme in the lecture (textbook) provides authentication but the alternative method does not. In the alternative method, two nonce N_a and N_b are used so that they can assure the message is not a replay. As for the method in lecture (textbook), two nonce N_1 and N_2 are used to assure the message is not a replay. So both of the methods can resist replay attack.

The session key in both of the methods is encrypted by K_a and K_b all the time, so the confidentiality is assured. Due to alternative method cannot provide authentication, I think the method in the lecture (textbook) is more secure than the alternative method.

6.

This hash function is weak, so it's very easy to find another message which has the same hash value with M.

(a) One-way property

For a very large n, the hash function has one-way property. Even though n is not a large number whose factorization is unknown, it's also really hard to find the plaintext. For example, let's assume n is 10 and the hash value is 7. It's almost impossible to find the original message on the basis of these known information.

(b) Second preimage resistance

Second preimage resistance means that given a message, it's computationally infeasible to find another message with the same hash value. In this case, this hash function doesn't satisfy the requirement. Let's assume the hash value h, you can simply find another message whose quadratic sum is h or h + xn (x-positive integer). For instance, h is 123456789 and n is very large. You can always easily find a message $\{b_1, b_2, \dots, b_y\}$ which $\sum_{i=1}^{t}(b_i)^2$ = 123456789 or 123456789 + xn (x-positive integer).

(c) Collision resistance

Because this hash function doesn't satisfy second preimage resistance, it also doesn't satisfy collision resistance. In this case, you can choose whatever message you want and easily find collisions. For example, a message {4, 2} which quadratic sum is 20. It must have the same hash value with {4,1,1,1,1}, {3,3,1,1}, etc.

Appendix

(1) SmallestPublicKey.py

```
def gcd(int1, int2):
    if int1 % int2 != 0:
        return gcd(int2, int1 % int2)
    return int2

def smallestPublicKey(integer):
    a = 2
    while gcd(integer, a) != 1:
        a += 1
    return a
```

(2) ExtendedEuclid.py

```
def extendedEuclid(a, b):
    remainder = -1
    x1 = 1
    x2 = 0
    y1 = 0
    y2 = 1
    if a > b:
        divisor = b
    else:
        dividend = b
    divisor = a
    while remainder != 0:
        gcd = remainder
        quotient = dividend // divisor
        remainder = dividend % divisor
        temp1 = x1 - x2 * quotient
        x1 = x2
        x2 = temp1
        temp2 = y1 - y2 * quotient
        y1 = y2
        y2 = temp2
        dividend = divisor
        divisor = remainder
    if a > b:
        return [x1,y1,gcd]
    else:
        return [y1,x1,gcd]
```

(3) Mod.py

```
def mod(m, d, n):
    i = 0
    ex = 1
    r = m % n
    dictionary = {i: [r, ex]}
    while ex < d:
        i += 1
        ex *= 2
        r = (r ** 2) % n
        dictionary[i] = [r, ex]

    result = 1
    while d > 0:
        if d - dictionary[i][1] >= 0:
            d = d - dictionary[i][1]
            result = (result * dictionary[i][0]) % n
        i = i - 1
    return result
```