First Name: Zhicheng

Last Name: Yan

Student ID: 1002643142

First Name: Zhihong

Last Name: Wang

Student ID: 1002095207

First Name: Kecheng

Last Name: Li

Student ID: 1001504507

This submission has been prepared using LaTeX.

Question 1
(a) Solution:
For $x$, the solution is:
$x = 2^{\frac{2n}{3}} X_2 + 2^{\frac{n}{3}} X_1 + X_0$
By the way, for $y$, the relationship between $y$ and $Y_2, Y_1, Y_0$ will be:
$y = 2^{\frac{2n}{3}} Y_2 + 2^{\frac{n}{3}} Y_1 + Y_0$

(b) Solution:
By (a), we can easily get:

$$xy = 2^{\frac{4n}{3}} X_2Y_2 + 2^n(X_2Y_1 + X_1Y_2) + 2^{\frac{2n}{3}}(X_2Y_0 + X_1Y_1 + X_0Y_2) \\ +2^{\frac{n}{3}}(X_1Y_0 + X_0Y_1) + X_0Y_0$$

$X_2Y_1 + X_1Y_2 = (X_1 + X_2)(Y_1 + Y_2) - X_1Y_1 - X_2Y_2$
$X_2Y_0 + X_0Y_2 = (X_0 + X_2)(Y_0 + Y_2) - X_0Y_0 - X_2Y_2$
$X_1Y_0 + X_0Y_1 = (X_0 + X_1)(Y_0 + Y_1) - X_0Y_0 - X_1Y_1$

i) Algorithm Description
We split $x$ and $y$ into three equal parts $X_2$, $X_1$, $X_0$ and $Y_2$, $Y_1$, $Y_0$.
According to the relationship described from part (a), we do the addition: $X_0 + X_1$, $X_1 + X_2$, $X_0 + X_2$, $Y_0 + Y_1$, $Y_1 + Y_2$, and $Y_0 + Y_2$.
Then we recursively do:
$X_2Y_2$, $X_1Y_1$, $X_0Y_0$, $(X_0 + X_1)(Y_0 + Y_1)$, $(X_0 + X_2)(Y_0 + Y_2)$, and $(X_1 + X_2)(Y_1 + Y_2)$
and then we combine the results from the recursion, and get the result:

$$xy = 2^{\frac{4n}{3}} X_2Y_2 + 2^n((X_1 + X_2) \cdot (Y_1 + Y_2) - X_1Y_1 - X_2Y_2) \\ + 2^{\frac{2n}{3}}((X_0 + X_2) \cdot (Y_0 + Y_2) - X_0Y_0 - X_2Y_2 + X_1Y_1) \\ + 2^{\frac{n}{3}}((X_0 + X_1) \cdot (Y_0 + Y_1) - X_1Y_1 - X_0Y_0) + X_0Y_0.$$

We will compute $xy$ via the expression on the right. The additions take linear time, as do the multiplications by powers of 2 (which are merely left-shifts). The significant operations are the nine n/2-bit multiplications, $X_2Y_2, X_2Y_1, X_2Y_0, X_1Y_2, X_1Y_1, X_1Y_0, X_0Y_2, X_0Y_1, X_0Y_0$; these we can handle by nine recursive calls. Thus our method for multiplying n-bit numbers starts by making recursive calls to multiply these nine pairs of n/2-bit numbers (four subproblems of half the size), and then evaluates the preceding expression in O(n) time. Writing T(n) for the overall running time on n-bit inputs, we get the recurrence relation:

$T(n) = 9T(n/3) + O(n)$
Then as we wrote at part (b), the running time of resulting algorithm can be improved to:
$T(n) = 6T(n/3) + O(n)$

Precondition: x and y are two n-bit integers, n is a multiple of 3
Postcondition: It returns a bit integer which is x * y
ii) Pseudocode

```
Mult(x, y)
Equally split x into X2, X1, and X0
Equally split y into Y2, Y1, and Y0
S0 = X0 + X1
S1 = X1 + X2
S2 = X0 + X2
S3 = Y0 + Y1
S4 = Y1 + Y2
S5 = Y0 + Y2

M0 = Mult(X0, Y0)
M1 = Mult(X1, Y1)
M2 = Mult(X2, X2)
M3 = Mult(S0, S3)
M4 = Mult(S1, S4)
M5 = Mult(S2, S5)

R0 = M2
R1 = M4 − M1 − M2
R2 = M5 − M0 − M2 + M1
R3 = M3 − M0 − M1
R4 = M0
return R0 * 2^(4n/3) + R1 * 2^n +
                  R2 * 2^(2n/3) + R3 * 2^(n/3) + R4
```

iii) Show Correctness
Prove that $x \cdot y = 2^{\frac{4n}{3}} X_2Y_2 + 2^n((X_1 + X_2) \cdot (Y_1 + Y_2) - X_1Y_1 - X_2Y_2) + 2^{\frac{2n}{3}}((X_0 + X_2) \cdot (Y_0 + Y_2) - X_0Y_0 - X_2Y_2 + X_1Y_1) + 2^{\frac{n}{3}}((X_0 + X_1) \cdot (Y_0 + Y_1) - X_1Y_1 - X_0Y_0) + X_0Y_0$, where x and y are aribitrary n-bit integers, and n is a multiple of 3.
Let x and y be aribitrary n-bit integers, and n is a multiple of 3
Then $x$ and $y$ are splited to equal parts $X_2$, $X_1$, $X_0$ and $Y_2$, $Y_1$, $Y_0$ by part (a)
Then we can get

$$\begin{aligned}X_2Y_1 + X_1Y_2 &= X_1Y_1 + X_1Y_2 + X_2Y_1 + X_2Y_2 - X_1Y_1 - X_2Y_2 \\ &= (X_1 + X_2)\cdot(Y_1 + Y_2) - X_1Y_1 - X_2Y_2\end{aligned}$$

$$\begin{aligned}X_2Y_0 + X_0Y_2 &= X_0Y_0 + X_0Y_2 + X_2Y_0 + X_2Y_2 - X_0Y_0 - X_2Y_2 \\ &= (X_0 + X_2)\cdot(Y_0 + Y_2) - X_0Y_0 - X_2Y_2\end{aligned}$$

$$\begin{aligned}X_1Y_0 + X_0Y_1 &= X_0Y_0 + X_0Y_1 + X_1Y_0 + X_1Y_1 - X_0Y_0 - X_1Y_1 \\ &= (X_0 + X_1)\cdot(Y_0 + Y_1) - X_1Y_1 - X_0Y_0\end{aligned}$$

$$\begin{aligned}\text{Then, } x\cdot y &= (2^{\frac{2n}{3}}X_2 + 2^{\frac{n}{3}}X_1 + X_0)\cdot(2^{\frac{2n}{3}}Y_2 + 2^{\frac{n}{3}}Y_1 + Y_0) \\ &= 2^{\frac{4n}{3}}X_2Y_2 + 2^n(X_2Y_1 + X_1Y_2) + 2^{\frac{2n}{3}}(X_2Y_0 + X_0Y_2 + X_1Y_1) \\ &\quad +2^{\frac{n}{3}}(X_1Y_0 + X_0Y_1) + X_0Y_0 \\ &= 2^{\frac{4n}{3}}X_2Y_2 + 2^n((X_1 + X_2)\cdot(Y_1 + Y_2) - X_1Y_1 - X_2Y_2) \\ &\quad +2^{\frac{2n}{3}}((X_0 + X_2)\cdot(Y_0 + Y_2) - X_0Y_0 - X_2Y_2 + X_1Y_1) \\ &\quad +2^{\frac{n}{3}}((X_0 + X_1)\cdot(Y_0 + Y_1) - X_1Y_1 - X_0Y_0) + X_0Y_0\end{aligned}$$

Therefore, we can conclude that we can use $2^{\frac{4n}{3}}X_2Y_2 + 2^n((X_1 + X_2)\cdot(Y_1 + Y_2) - X_1Y_1 - X_2Y_2) + 2^{\frac{2n}{3}}((X_0 + X_2)\cdot(Y_0 + Y_2) - X_0Y_0 - X_2Y_2 + X_1Y_1) + 2^{\frac{n}{3}}((X_0 + X_1)\cdot(Y_0 + Y_1) - X_1Y_1 - X_0Y_0) + X_0Y_0$ to compute to $x\cdot y$
iv) Show complexity
We divide the original input to three equal parts, and use them to do the recursion 6 times. Also shift bits(line 22, multiply by the power of 2) takes and the additions(from line 3 to line 8) take $O(n)$ time. The function which shows the number of steps is

$$T(n) = 6T(\tfrac{n}{3}) + O(n)$$

By master theorem, $a = 6, b = 3, f(n) = n$, the complexity is $\Theta(n^{\log_3 6})$, which runs in time $o(n^2)$ ($\log_3 6 < 2$)

(c) Solution
The running time of our algorithm is $\Theta(n^{\log_3 6}) \approx \Theta(n^{1.63})$, which is slower than the divide-and-conquer algorithm which is $\Theta(n^{log_2 3}) \approx \Theta(n^{1.59})$

Question 2

(a). Algorithm design

Precondition: Given a set of tuples $R\left\{(s_1, f_1), (s_2, f_2), ..., (s_n, f_n)\right\}$, where $n > 0$ and all $s_i < f_i$ are nonnegative integers. The integers $s_i$ and $f_i$ represent the start and finish times, respectively, of job $i$.

Postcondition: Defines a schedule specifies for each job $i$ a positive integer $P(i)$ (the processor number for job $i$). It must be the case that if $i \neq j$ and $P(i) = P(j)$, then jobs $i$ and $j$ do not overlap. Returns a schedule that uses as few processors as possible, i.e., such that $max\left\{P(1), P(2), ..., P(n)\right\}$ is minimal.

Algorithm:

```
    Sort all jobs in order of starting time
    $d = 1$ // represents first processor
    $minpq.insert(1, f(1))$ //add first job to min heap
    for $i = 2$ to $n$:
        $q = minpq.minIndex()$ // get processor number of the earliest finishing requests
        if $s_i \geq minpq.Keyof(q)$:
            $minpq.increaseKey(q, f(i))$ // remain heap property
        else:
        $d = d + 1$ // add new processor
        $minpq.insert(d, f(i))$
    return all processors
```

(b). Check correctness

Proof with contradiction:

First, let us define depth $d$ of a set of jobs in $R$ is the maximum number of jobs have conflicts at anytime. Obviously, to schedule a set of jobs in $R$ needs at least d processors since conflict jobs need use different processors.

Let's suppose that we need d+1 processors to schedule all the jobs in set $R$. Assume the first job use processor d+1 is job $x$. The condition that job $x$ needs allocate processor $d + 1$ means that there are $d$ jobs start earlier than job $x$ and have conflicts with it. Which means that the depth of current set of jobs is at least $d + 1$. This contradicts the pre-setting that depth is $d$. In conclusion, the schedule made by our algorithm is optimal and using exactly $d$ processors.

(c). Implementation and time complexity
Pre-sorting for n jobs using merge sort takes O(n log n) time. Keep track of the earliest finishing time by min heap. In each node of the heap, we store the processor name, a list of jobs be scheduled in this processor and the finishing time of the last job as priority. For each job, either increase the key number, (i.e, assign it to a used available processor and remain heap property by sorting) or allocate to an new processor. Both actions take time $log(d)$ ($d$ is the depth as mentioned in (b)). So, it takes $O(nlog(n)) + O(nlog(d))$, which $O(nlog(n))$ dominates the time. So, the time complexity is $O(nlog(n))$, which is strictly less than $O(n^2)$.

Question 3.
(a). Solution:
Our greedy approach is pick the job that end earliest.
Precondition: Set of all the request jobs R (tuples with starting time and ending time)
Postcondition: The optimum number of jobs are scheduled to two processors, it return $A_1$ and $A_2$, where $|A_1| + |A_2|$ is maximum

```
Schedule(R):
    Sort the requests R by their ending time and put it
        sequentially in Q
    Set A1 = empty, and label ending time of A1 be 0
    Set A2 = empty, and label ending time of A2 be 0
    while Q is not empty
        q = dequeue(Q)
        if q not overlaps with requests in A1 and A2:
            add q to the processor with request has later
                labeled ending time
            update the processor's labeled ending time to q's
                ending time
        else if q overlaps with only one processor:
            add q to the other processor
    end while
    return (A1, A2)
```

(b) Solution
Prove that the algorithm above is guaranteed to compute an optimal schedule.

Let $Alg(n)$ be number of jobs scheduled to these two processors by the algorithm above, where $n$ is number of jobs in the input
Let $Opt(n)$ be number of jobs scheduled to these two processors by optimum scheduling, where $n$ is number of jobs in the input.
We will prove the algorithm gets the optimal number of jobs scheduled for the input of $n$ job intervals by induction(i.e. $Alg(n) \geq Opt(n)$, where $n$ is the number of jobs in the input)

Base Case: We want to show that the algorithm gets the optimal solution when the input just has only one job.
When the algorithm just already dealt with the only job, in the algorithm above processor 1 is scheduled the job since when dealing with scheduling

that job, the labeled ending time for both processors are the same which is zero. By the algorithm designed, the job will be scheduled to processor 1. Then the total number of jobs scheduled is 1 ($Alg(n) = 1$). For optimum solution, the only job is assigned to processor 1 or processor 2. The number of jobs scheduled in these two processors are 1 ($Opt(n) = 1$). So $Alg(n) = Opt(n) = 1 \Rightarrow Alg(n) \geq Opt(n)$, this case it holds.

Inductive Step:
Inductive Hypothesis: Assume the number of jobs scheduled by the algorithm above meets the number of jobs scheduled by the optimal solution to these two processors when finished dealing with $k^{th}$ job requests in the input, where $Alg(k) \geq Opt(k)$
Want to Prove: $Alg(k+1) \geq Opt(k+1)$, when scheduling the $k+1^{th}$ job

Case 1: Assume the algorithm and the optimal solution choose the same job to schedule to the same processor(the same job picked with no overlapping with that processor)
In this case, they schedule the same job request to the same processor, by inductive hypothesis, from the first job to the $k^{th}$ job, $Alg(k) \geq Opt(k)$. The algorithm choose the job which has the earliest ending time to schedule by its greedy property. Then after scheduling the $k+1^{th}$ job, the total number of jobs scheduled to two processors by the algorithm increased by 1, so $Alg(k+1) = Alg(k) + 1$. The optimal solution do the same thing as the algorithm above(choose the job with earliest ending time) by the assumption of this case, so $Opt(k+1) = Opt(k) + 1$. Also from the inductive hypothesis we can get $Alg(k) \geq Opt(k) \Rightarrow Alg(k) + 1 \geq Opt(k) + 1$. Therefore $Alg(k+1) = Alg(k) + 1 \geq Opt(k) + 1 = Opt(k+1) \Rightarrow Alg(k+1) \geq Opt(k+1)$. So, $Alg(k+1) \geq Opt(k+1)$ holds.

Case 2: Assume the algorithm and the optimal solution schedule the same job to the different processor(the same job picked with no overlapping with both processors)
In this case, they schedule the same job request to the different processor, from inductive hypothesis, from the first job to the $k^{th}$ job, $Alg(k) \geq Opt(k)$. The algorithm choose the job which has the earliest ending time to schedule by its greedy property. Then after scheduling the $k+1^{th}$ job, the total number of jobs scheduled to two processors by the algorithm increased by 1, so $Alg(k+1) = Alg(k) + 1$. The optimal solution scheduled

the job(still the earliest job by the assumption of this case) to the different processor but total number of jobs scheduled to these two processors is still increased by 1, so $Opt(k+1) = Opt(k) + 1$. Also from the inductive hypothesis we can get $Alg(k) \geq Opt(k) \Rightarrow Alg(k) + 1 \geq Opt(k) + 1$. Therefore, $Alg(k+1) = Alg(k) + 1 \geq Opt(k) + 1 = Opt(k+1) \Rightarrow Alg(k+1) \geq Opt(k+1)$. So, $Alg(k+1) \geq Opt(k+1)$ holds.

Case 3: Assume the algorithm and the optimal solution schedule the different job to the same processor(the different jobs they picked with no overlapping with both processors in either way)
In this case, they schedule the different job request to the same processor, by inductive hypothesis, from the first job to the $k^{th}$ job, $Alg(k) \geq Opt(k)$. Then after scheduling the $k+1^{th}$ job, the total number of jobs scheduled to two processors by the algorithm increased by 1, so $Alg(k+1) = Alg(k) + 1$. The optimal solution chose the different job(the job has later ending time compare with the job picked by the algorithm) to schedule but total number of jobs scheduled to these two processors is also increase by 1, so $Opt(k+1) = Opt(k) + 1$. Also from the inductive hypothesis we can get $Alg(k) \geq Opt(k) \Rightarrow Alg(k) + 1 \geq Opt(k) + 1$. Therefore, $Alg(k+1) = Alg(k) + 1 \geq Opt(k) + 1 = Opt(k+1) \Rightarrow Alg(k+1) \geq Opt(k+1)$. So, $Alg(k+1) \geq Opt(k+1)$ holds.

Case 4: Assume the algorithm and the optimal solution schedule the different job to the different processor(the different jobs they picked with no overlapping with both processors in either way)
In this case, they schedule the different job request to the different processor, by inductive hypothesis, from the first job to the $k^{th}$ job, $Alg(k) \geq Opt(k)$. Then after scheduling the $k+1^{th}$ job, the total number of jobs scheduled to two processors by the algorithm increased by 1, so $Alg(k+1) = Alg(k) + 1$. The optimal solution scheduled different job(the job has later ending time compare with the job picked by the algorithm) to the different processor but total number of jobs scheduled to these two processors is also increase by 1, so $Opt(k+1) = Opt(k) + 1$. Also from the inductive hypothesis we can get $Alg(k) \geq Opt(k) \Rightarrow Alg(k) + 1 \geq Opt(k) + 1$. Then $Algo(k+1) \geq Opt(k+1)$ holds.

Case 5: Assume the algorithm and the optimal solution schedule the same job, but this job has overlap with other jobs which already scheculed by

the algorithm and the optimum solution in both processors
In this case, if the algorithm and the optimal solution has overlap with both processors, then both ways cannot schedule this same job and get $Opt(k+1) = Opt(k) \leq Alg(k)$. $Alg(k+1) = Alg(k)$, so $Alg(k+1) \geq Alg(k) \geq Opt(k) = Opt(k+1) \Rightarrow Alg(k+1) \geq Opt(k+1)$ Then $Algo(k+1) \geq Opt(k+1)$ holds.

Case 6: Assume the algorithm and the optimal solution schedule the different job, but the different jobs picked by optimal and algorithm have overlap with other jobs which already scheculed by the algorithm and the optimum solution respectively in both processors
In this case, if the algorithm and the optimal solution has overlap with both processors, then both ways cannot schedule their related different job and get $Opt(k+1) = Opt(k) \leq Alg(k)$. $Alg(k+1) = Alg(k)$, so $Alg(k+1) \geq Alg(k) \geq Opt(k) = Opt(k+1) \Rightarrow Alg(k+1) \geq Opt(k+1)$ Then $Algo(k+1) \geq Opt(k+1)$ holds.

Therefore, $Alg(k+1) \geq Opt(k+1)$ holds for all the cases.
By induction, we have shown that $Alg(n) \geq Opt(n)$, so the algorithm gets the optimal solution.

(c). Solution:
The complexity is $\mathcal{O}(n \log n)$ time. Sorting all the job intervals takes $\mathcal{O}(n \log n)$ time. Schedule each request job(determine whether put it in processor 1, processor 2, or declined) takes constant time from line 6 to line 11(each iteration of the while loop). Totally need to deal with $n$ jobs. So for scheduling all the jobs takes $\mathcal{O}(n)$ time. Overall, it takes $\mathcal{O}(n \log n + n)$, which can be concluded to $\mathcal{O}(n \log n)$ time.

Question 4

(a). Solution:

A Natural Greedy Strategy: Always selects the largest coin denomination which does not exceed the remaining amount. As long as the remaining amount is greater than zero, the process is repeated.
The time complexity of this strategy is O(m) (m is the number of coins) as the number of coins is added once for every denomination.

A Example to show this greedy strategy does not work:

Input: A list of positive integer coin values:
$\{c_1 = 1, c_2 = 1, c_3 = 5, c_4 = 5, c_5 = 8\}$
and positive integer amount $A = 10$.
Output: A selection of coins $\{i_1, i_2, ..., i_k\} \subseteq \{1, 2, ..., m\}$ such that $c_{i_1} + c_{i_2} + ... + c_{i_k} = A$ and $k$ is as small as possible. If it is impossible to make change for amount $A$ exactly, then the output should be the empty set $\emptyset$.

Obviously, we can see the best solution is $\{5, 5\}$ ($k_1 = 2$) because $5 + 5 = 10$, but according to the Natural Greedy Strategy we assume at the beginning, the strategy will pick 8 first because 8 is the largest coin that the strategy can choose. Then, by this Natural Greedy Strategy, the solution should be $\{8, 1, 1\}$ ($k_2 = 3$) because $8 + 1 + 1 = 10$, but this solution is obviously not the best ($k_1 = 2 < k_2 = 3$). So this strategy does not work.

(b). Solution:

Dynamic Programming Algorithm:

Input: A list of positive integer coin values $C = \{c_1, c_2, ..., c_m\}$ and positive integer amount $A$.
We assume the input list is non empty. If the input is empty, then is problem will be vacuously true.

Output: A selection of coins $\{i_1, i_2, ..., i_k\} \subseteq \{1, 2, ..., m\}$ such that $c_{i_1} + c_{i_2} + ... + c_{i_k} = A$ and $k$ is as small as possible. If it is impossible to make change for amount $A$ exactly, then the output should be the empty set $\emptyset$.

i) Algorithm Description:

First, we have a list of positive integer coin values $C$ and a positive integer amount $A$ as inputs. In order to achieve the Output we mentioned before, we should initialize three list: $X$ ($A+1$ items, index $i$ from 0 to $A$), $Y$ (also $A+1$ items, index $i$ from 0 to $A$) and $Z$ (up to $m$ items, $m$ is the number of all coins) to solve the problem. Set all items in $X$ to be $\infty$ and all items in $Y$ to be $-1$ except $X[0] = 0$. We use $X$ to record the minimum number of coins that can reach $A$, use $Y$ to track what combination of coins can reach the minimum number of coins, and use $Z$ to save coin(s) we can use at current step.

Second, loop over $C$ by index $j$ (from 1 to $m$). In our algorithm, the loop of list $C$ also limit what coins we can use. (i.e. If loop to $C[1]$, then we can only use coin $C[1]$; if loop to $C[3]$, then we can use $C[1], C[2], C[3]$) For each value in $C$ during the loop, we should also go over $X$ and $Y$ by index $i$ and update $Z$. For $Z$, we append $C[j]$ into $Z$ at each time. For $X$ and $Y$, if $i$ is smaller than the value in $C$ (i.e. $i < C[j]$), then we will do nothing about $X$ and $Y$. Once $i = C[j]$, then we should let $X[i] = 1$ and $Y[i] = j$, which means we can use one coin (best case for this question) to reach $i$ and use $Y$ to record what coin we used. If $i > C[j]$, we will use $Z$ to check if we can reach the value $i$ by coins we have at current step (example: If we only have one coin which values 1, then we can never reach 2. We can use greedy approach we mentioned at part(a) to check if we can reach $i$). If we cannot reach $i$ by coins we have, then we will jump to $i+1$ without changing $X[i]$ and $Y[i]$. If we can reach $i$, then we should check if we have minimum number of coins to reach $i$. So, we use $X[i - C[j]] + 1 < X[i]$ to check this ($X[i - C[j]]$ means we can use coin $C[j]$ and a optimal value we reached before to get $i$, then we may add one coin at this case). If $X[i - C[j]] + 1 < X[i]$, then $i$ can be reached by smaller coin we already have. We will also update $X[i]$ to be $1 + X[i - C[j]]$ and $Y[i] = j$ too. We keep doing this until we get the end of $C$.

Finally, we use $Y$ to trace result and return a list of coins. If the last item of $Y$ is still -1, the this problem has no solution and return empty set. If not, we can get what coin is used in $A$, then do a back-checking until we find all coins that used to reach $A$.

ii) Pseudocode:
Precondition: a non empty list of positive integer coin values C and a positive integer amount A.
Postcondition: return a selection list of coins as Result.

MinimumCoinsCombination(C,A):

```
1  Initialize empty list Result, X, Y and Z;
2  X[0] is set to 0 and Y[0] is set to −1 at first.
3  For all int i form 1 to A:
4    X[i] = Infinity, Y[i] = −1;
5  For all int j form 1 to C.length (C.length = m in Q4):
6    append C[j] into Z;
7    For all int i form 1 to A:
8      if i == C[j]:
9        X[i] = 1, Y[i] = j;
10     if i > C[j]:
11       check if we can use coins in Z to reach i:
12         if X[i − C[j]] + 1 < X[i]:
13           X[i] = 1 + X[i − C[j]], Y[i] = j;
14 if Y[A] == −1:
15   return empty set;
16 int target = A;
17 while target >= 0:
18   int k = Y[target];
19   append C[k] into Result;
20   target = target − C[k];
21 return Result;
```

iii) Show Correctness:
We will show correctness by Pseudocode and induction based on the amount $A$.
Let optimal solutions to be a set $O$.

Base case:
$C$ is an one element list.
Line 1 to 4 will work because they are basic initialization.
Line 5 will loop once.
Line 6 will work because it is basic appending.
Line 7 will work because $A$ exists.
Line 8 to 13 will work because $C[j]$ always exists by line 5, $X[i]$ and $Y[i]$ always exist by line 1 to 4, and we can only do $i - C[j]$ after we checked

$i > C[j]$. We are able to check $Z$ like we mentioned at part(b). i).
Line 14 will work because it just checking the value of item in $Y$.
Line 16 to 20 will work because the integer *target* is a combination of coin(s) and its total value is equal to the input $A$. They will check what coins formed A. In the end, our algorithm will return at line 21 or 15.
Because there is only one element $c$ in the input list $C$, if $c = A$, our algorithm will return correct list of a coin at line 21; if $c$ is not equal to $A$, our algorithm will return empty set at line 15.
After all, our algorithm is correct and optimal for base case.

Inductive Step:
Inductive Hypothesis: Assume $X[i]$ is optimal for $0 \leq i < A$.
(i.e. $X[i] = O[i]$ for $0 \leq i < A$)
Want To Show: $X[A] = O[A]$ for $0 \leq i \leq A$.
Let optimal solutions $O$ be at $0 \leq i \leq A$.
$O[A] = 1 + O[A - c]$ # $c$ is a constant at here by code in part ii).
$= 1 + X[A - c]$ # By I.H., $A - c < A$
$= X[A]$
So, $O[A] = X[A]$.
Therefore, our algorithm is correct and optimal.

(c). Solution:
Worst-case running time: $O(Am^2)$
According to this question, we know the length of list of positive integer coin values $C$ is $m$ and the positive integer amount is $A$. We can get the worst-case running time by analyzing part (b). ii). Pseudocode.
By lectures, tutorials and textbooks, assume all lines except recursions or loops only require constant time.
Then line 1-2, 4, 6, 8-10, 12-16, 18-21 only require constant time.
Line 3 is a simple for loop from 1 to $A$. It will cost $A$.
Line 5 and 7 contain a nested loop. Line 5 is from 1 to $C.length = m$ and line 6 is from 1 to $A$. They will cost $Am$.
Line 11 will cost $m$ according to the greedy approach we mentioned at part(a).
Line 17 is a simple while loop. The cost of this loop is not stable but less or equal to line 2. In worst-case it will cost $A$ (when 1 is the only coin value).
Therefore, the worst-case running time is $O(Am^2)$.

Question 5

Solution:

Step 1: Optimal Substructure Definition.
Let $O_{mn}$ denote an optimal solution and OPT(m, n, h) be its maximum gold value.

Case 1: $Rock_{mn} \in O_{mn}$

$$\begin{aligned} OPT(m,n,h) = max(&OPT(m+1,n-1,h-H(m,n))+G(m,n),\\ &OPT(m+1,n,h-H(m,n))+G(m,n),\\ &OPT(m+1,n+1,h-H(m,n))+G(m,n)) \end{aligned}$$

Case 2: $Rock_{mn} \notin O_{mn}$

$OPT(m,n,h) = 0$

Step 2: Memorization. Define an appropriate array to store intermediate value.
M[0...m+1, 0...n+1, 0...d]
M[i, j, k] = the maximum gold values could obtained on rock with coordinates i, j and remaining hardness k.

Step 3: Recurrence relation in terms of the array(s) defined in Step 2.
M[i, j, k] = 0, if i = 0 or j =0
M[i,j,k] = max(M[i+1,j-1,k-H[i,j]]+G[i,j],M[i+1,j,k-H[i,j]]+G[i,j], M[i+1, j+1, k=H[i,j]]+G[i,j]), if $Rock_{mn} \in O_{mn}$
M[i,j,k] = 0, if $Rock_{mn} \notin O_{mn}$

Step 4: Bottom Up approach (iterative solution).
Drill maximum gold(DMG)

```
DMG(H, G, d):
    Define M[0...m+1, 0...n+1, 0...d]
    for i=0...m+1
        for j=0...n+1
            for k=0...d
                M[i,j,k] = 0
    for i = 1...m
        for j = 1...n
```

```
            for k = 0 ...d
                if k − H[i,j] >= 0:
                    M[i,j,k] = max(M(i+1,j−1,k−H(i,j)),
                                            M(i+1,J,K−H(i,j)),
                                        M(i+1,j+1,k−H(i,j)))+G[i,k])
                else:
                    M[i,j,k] = 0
    return max(M[1,j,k])
```

Step 5: Compute a path to actual solution.
The input i,j,k are the coordinators of result of DMG(H, G, d)

```
DMGPath(M, i, j, k)
    k = d−k // original hardness
    if i = 0 or i = m+1 or j = 0 or j=n+1 then
        return " "
    else if k − H[i,j]>=0 then
        a = i+1, b = j, c = k−H[i,j]
        if M[i+1, j−1,k−H[i,j]] >= M[i+1,j,k−H[i,j]] then
            b = j −1
        else if (M[i+1, j+1,k−H[i,j]] >= M[i+1,j,k−H[i,j]]
            and M[i+1, j+1,k−H[i,j]] >= M[i+1,j−1,k−H[i,j]])
                then
            b = j+ 1
        return DMGPath(M, a, b, c) + " j"
```

Check correctness
We will show correctness by Pseudocode and induction based on the given hardness d.Assume i and j bigger than 0.
Let optimal solutions to be a set Opt, our solutions denote as Alg.

Base case:
d is a constant, H and G are two 1*1 matrix
If $d \geq H[1,1]$,which means it's able to drill, then Opt = G[1,1]
By our algorithm in step4, it's easy to see it will goes to if statement in line 10, since maximum gold value from lower level is 0, get final maximum gold value equals G[1,1]. Our algorithm achieve the same result as Opt. Condition holds.
If $d < H[1,1]$, which means it's not able to drill, all algorithm output value is 0, condition holds.

Inductive Hypothesis: Assume there is a j, $0 < j < n+1$, st, Alg(i,j) is optimal for $0 \leq i < m$.(i.e. there's a j in interval 0 to n+1, st, Opt(i,j) =

Alg(i,j) for $0 \leq \; i < \; m$)(d is the given drill hardness, H and G are two matrices mentioned above)
Want to show that there's a j, $0 < \; j < \; n+1$, st, Opt(i,j) = Alg(i,j) for $0 \leq \; i \leq \; m$

Inductive Step:
Let optimal solution Opt be at $0 \leq \; i \leq \; m$
Case 1: If $rock_{mj}$ in Opt solution.
Opt(m,j) = max(Opt(m+1,j-1), Opt(m+1,j),Opt(m+1, j+1)) + G[m,j]
= max(Alg(m+1, j-1), Alg(m+1,j), Alg(m+1, j+1)) + G[m,j] by I.H, $0 \leq m-1 < m$
= Alg(m,j)
Case 2: if $rock_{mj} isnotintheOpt.$
Opt(m,j) = 0
= 0 by I.H, $0 \leq m-1 < m$
= Alg(m,j)
So, Opt(m,j) = Alg(m,j) for $0 \leq \; i \leq \; m.$
Since it's true for all j in the interval. Finding max will find the maximum gold value on the start level. Therefore, our algorithm is correct and optimal.

Time complexity:
$O((n*m*d)*3) = O(3nmd)$