CSC373 Assignment1 1003326046 Ruijie Sun & 1002183059 Wenjie Hao

Q1

a)

$x = x_2 \cdot 2^{\frac{2}{3}n} + x_1 \cdot 2^{\frac{1}{3}n} + x_0$

$y = y_2 \cdot 2^{\frac{2}{3}n} + y_1 \cdot 2^{\frac{1}{3}n} + y_0$

b) **def multiply(x, y, n):**

ADD 0's to left of x and y if n is not multiple of 3.

if n = 1:

return x $\times$ y

find x2, x1, x0, y2, y1, y0

num1 = multiply((x1+x2), (y1+y2), $\left\lceil \frac{1}{3} n \right\rceil$)

num2 = multiply(x1, y1, $\left\lceil \frac{1}{3} n \right\rceil$)

num3 = multiply(x2, y2, $\left\lceil \frac{1}{3} n \right\rceil$)

num4 = multiply((x0+x1), (y0+y1), $\left\lceil \frac{1}{3} n \right\rceil$)

num5 = multiply(x0, y0, $\left\lceil \frac{1}{3} n \right\rceil$)

num6 = multiply(x0, y2, $\left\lceil \frac{1}{3} n \right\rceil$)

num7 = multiply(x2, y0, $\left\lceil \frac{1}{3} n \right\rceil$)

coeff1 = num3

coeff2 = num1 – num2 – num3

coeff3 = num2 + num6 + num7

coeff4 = num4 – num 2 – num5

coeff5 = num5

res = coeff1$\times 2^{\frac{4}{3}n}$ + coeff2$\times 2^{n}$ + coeff3$\times 2^{\frac{2}{3}n}$ + coeff4$\times 2^{\frac{1}{3}n}$ + coeff5

return res

Prove by induction:

Define predicate P(n): "The algorithm holds for n-bits integer x and y"

Base case (n = 1):

x1 = x2 = y1 = y2 = 0

x0 = x, y0 = y

By the algorithm, multiply(x, y) will be x0×y0, which is equal to x×y.

The P(1) holds.

Inductive step:

Assume $\forall k < n, P(k)\ holds.\ \ We\ want\ to\ show\ P(n)\ holds.$

$$x \times y = (x_2 \cdot 2^{\frac{2}{3}n} + x_1 \cdot 2^{\frac{1}{3}n} + x_0) \times (y_2 \cdot 2^{\frac{2}{3}n} + y_1 \cdot 2^{\frac{1}{3}n} + y_0)$$

$$= x_2 \cdot y_2 \cdot 2^{\frac{4}{3}n} + (x_2 \cdot y_1 + x_1 \cdot y_2) \cdot 2^{n} + (x_2 \cdot y_0 + x_1 \cdot y_1 + x_0 \cdot y_2) \cdot 2^{\frac{2}{3}n} + (x_1 \cdot y_0) \cdot 2^{\frac{1}{3}n} + x_0 \cdot y_0$$

$$= x_2 \cdot y_2 \cdot 2^{\frac{4}{3}n} + ((x1 + x2)(y1 + y2) - x1 \cdot y1 - x2 \cdot y2) \cdot 2^{n} + (x2 \cdot y0 + x1 \cdot y1 + x0 \cdot y2) \cdot 2^{\frac{2}{3}n} + ((x0 + x1)(y0 + y1) - x1 \cdot y1 - x0 \cdot y0) \cdot 2^{\frac{1}{3}n} + x0 \cdot y0$$

Since the inductive assumption, then the P(n) holds.

Runtime:

T(n) = 7 T($\frac{1}{3}n$) + $\theta(n)$,

a = 7, b = 3, $n^{log_b^a} = n^{log_3^7} = n^{1.77}$

By master theorem, it falls into case 1, so T(n) = $\theta(n^{1.77})$

c) It is more slower since $n^{log_3^7} > n^{log_2^3}$.

Q2

a) Pseudocode：

```
S: given interval set{(s1,f1),(s2,f2)…(sn,fn)}

n: size of S

def Schedule(S,n):

    sort S by starting time of each interval such that s1 ≤ s2 ≤ . . .
≤ sn

    res = 0

    for i = 1 to n:

        if job i is compatible with some processor j: #use  min-
priority queue to check

            put job i into processor j

        else:

            create new processor res+1

            put job i into processor res+1
```

```
            res +=1

        return res
```

b)

Define Depth: The depth of a set of open intervals is the maximum number of jobs that contain any given time.

And we observe that the depth is the the minimum number of processor we need to schedule given intervals.

Prove by contradiction:

Given a interval set S, the depth is d.

Assume above algorithm is not optimal, then res = Schedule(S, n) > d. Let job k be the first one which is added into processor d+1. Based on above algorithm, there must be d jobs in previous d processors which are conflict with job k. Then there are at least d+1 jobs in the set S which are mutually overlapping which is contradicted to the given set depth d.

c)

def improve_schedule(S, n):

Merge sort S by starting time of each interval such that $s_1 \le s_2 \le . . . \le s_n$

```
q = min_priority_queue
res = 0
for i = 1 to n:
    if q is empty:
        create a processor P1 with attribute last-end-time
        add job i into p1 and set p1’s last-end-time as f1
```

```
            add processor P1 to min_priority_queue q


        else:
            t = min_priority_queue.find_min()
            if Si < t.last-end_time:
                create a new processor P_res+1 with attribute last-end-
                time
                add job i into processor P_res+1 and set P_res+1.last-
                end-time as fi
                add processor P_res+1 to min_priority_queue q
                res +=1
            else:
                add job i into processor t and set t.last-end_time as
                fi
    return res
```

sorting cost = θ(nlogn)

There is only one for loop and for each loop the insert processor into min_priority_queue operation costs θ(logn). Other costs θ(1) in each loop.So there is θ(nlogn) cost for the loop.

Therefore T(n) = θ(nlogn)

Q3.

a)

S： given interval set{(s1, f1), (s2, f2)…(sn, fn)}

n： size of S

def Scheduling(S, n):

Sort intervals in S such that $f_1 \leq f_2 \leq . . . \leq f_n$.

// Tracking the last-end time for processor1 and processor2

end1 = 0

end2 = 0

//Create schedules, A1, A2

A1 = {}

A2 = {}

for i:=1 to n do:

pros = find the processor with the minimum gap (min((si-end1), (si-end2))) if the minimum gap is non-negative.

if the processor is not found above:

pros = None

if the pros is not None

Add the job i to the found pros

Update pros's last end time to be fi.

return (A1, A2)

b)

Let A1 and A2 be the schedules that hold jobs. Let A1-i and A2-i be the schedules after checking i-th jobs.

Let optimal schedule sets be OPT1 and OPT2.

Let OPT1-i and OPT2-i be the optimal schedule sets after checking i-th jobs.

Let End1-i and End2-i be the last finishing time in A1-i and A2-i.

Define P(k): "A1-k = OPT1-k and A2-k = OPT2-k" $\forall k \in \mathbb{Z}, 0 \leq k \leq n$

Base case1: k=0

This is vacuously true.

Base case2: k=1

Our algorism selects the job with the shortest finishing time, and thus. P(1) holds.

Inductive steps:

Assume P(k) holds such that A1-k = OPT1-k and A2-k = OPT2-k" $\forall k \in \mathbb{Z}, 0 \leq k \leq n$.

WTS: P(k+1) holds.

Case 1: The algorism doesn't add (k+1)-th job to either A1-(k) or A2-(k).

This means (k+1)-th job overlaps with a job in either A1-(k) or A2-(k). Since A1-(k)=OPT1-k and A2-(k) = OPT2-k by inductive hypothesis, k+1-th jobs doesn't belong to any OPT schedule sets. Therefore, A1-(k+1)=OPT1-k+1 and A2-(k+1) = OPT2-k+1.

Case 2: A1-(k+1) = A1-(k) $\cup$ (k+1)

Case 2.1: (k+1)-th job ∈ OPT1-k+1, so the P(k+1) holds.

Case 2.2: (k+1)-th job ∈ OPT2-k+1.

Since A1-k = OPT1-k and A2-k = OPT2-k by induction hypothesis, the reason why we add (k+1)-th job into A1 instead of A2 (we could add it into A2 because OPT2 have done it) must be End2-k ≤ End1-k ≤ S(k+1).

Besides, we can rewrite OPT1 = A1-k ∪ M1 and OPT2 = A2-k ∪ M2 where M1, M2 ⊆ {k+1,…n} and we know that k+1 $\in M2$.

Since every job in M1 starts at time ≥ End1-k, then every job in M1 starts at time ≥ End2-k.

Since every job in M2 starts at $\geq S(k+1)$, then every job in M2 starts at ≥ End1-k.

Therefore, if we interchange M1 and M2, there will be no conflict: OPT1 = A1-k ∪ M2 and OPT2 = A2-k ∪ M1. Since the size is same as before, this is still optimal. Therefore, A1-k+1 = OPT1-k+1 and A2-k+1 = OPT2-k+1. So P(k+1) holds.

Case 2.3: (k+1)-th job ∉ OPT1 and (k+1)-th job ∉ OPT2

There must be a job j which is conflict with (k+1)-th job in OPT1 where j > k+1, otherwise (k+1)-th job is part of OPT1. Then fj ≥ fk+1 which means (k+1)-th job is compatible to the job after job j in OPT1. Let OPT1 = OPT1 $\cup (k+1) - \text{j}$, {OPT1, OPT2} is still optimal solution. Therefore A1-k+1 = OPT1-k+1 and A2-k+1 = OPT2-k+1. So P(k+1) holds.

Case 3: A2-(k+1) = A2-(k) ∪ (k+1)

This is symmetric to case 2.

c)

```
S: given interval set{(s1, f1), (s2, f2)…(sn, fn)}

n: size of S
```

```
def improve_schedule(S, n):

    Merge sort S by starting time of each interval such that s1 ≤ s2
    ≤ . . . ≤ sn

    // Tracking the last-end time for processor1 and processor2
    End1 = 0
    End2 = 0


    A1 = {}
    A2 = {}


    for i:=1 to n do:
        res = min((si-end1), (si-end2))
        if (res) < 0:
            res = NONE
        else if (res == (si-end1)):
            A1.add(job i)
            End1 = fi
        else:
            A2.add(job i)
            End2 = fi
```

Sorting costs θ(nlogn)

The single loop costs θ(n) using two array to store allocated jobs. (As each iteration costs $\theta(1)$)

So total cost is θ(nlogn)

Q4:

a)

Natural greedy algorism:

Repeatedly takes the largest coin that is less than the currently target from the remaining coins.

When it doesn't work:

Let $c1 = 40, c2 = 20, c3 = 15, c4 = 10, c5 = 1, c6 = 1, c7 = 1, c8 = 1, c9 = 1$.

Let $A = 45$.

By the natural greedy algorism above, we take one c1 and five coins c5-9, and in total k is equal to 6.

However, the optimal solution should be one c2, one c3 and one c4. The optimal k for $A = 45$ should be 3.

b)

Let input coins $S = \{c1, c2, \ldots cm\}$, positive amount be A

Step1: Describe the recursive structure for the sub-problem

For every optimal solution O, either 'coin m' is in OPT or not.

If coin m is in O, then $O - \{cm\}$ is the optimal solution for input $(A - cm), S - \{cm\}$

If coin m is not in O, then O is the optimal solution for input $A, S - \{cm\}$

Step2: Define an array that stores optimal values for sub-problem

Let $M = [0\ldots m, 0\ldots A]$, where $M[k, a]$ represents the optimal value for input $\{c1,c2\ldots ck\}$ and amount a. When there is no solution for this input, $M(k, a) = \infty$.

Step3: Give a recurrence relation for the array values.

M(k, 0) = 0   #if the amount is 0, no coin is needed

M(0, a) = ∞ # impossible to make a amount without using any coins

M(k, a) = M(k-1, a)   if ck >a

M(k, a) = min{M(k-1, a), 1+ M(k-1, a - ck)}   if ck ≤ a

Step4: bottom to up algorithm

```
def bottom_to_up(S, A):
    m = len(S)
    M = [0…m,   0…A]
    M(0, 0) = 0
    for a = 1 to A:
        M(0, a) = ∞

    for k = 1 to m:
        M[k, 0] = 0
        for a = 0 to A:
            if ck > a:
                M(k, a) = M(k-1, a)
            else:
                M(k, a) = min(M(k-1, a), 1+ M(k-1, a - ck))

    Return M
```

Step 5: Optimal Solution

```
def optimal_sol(S, A):
    m = len(S)
    M = bottom_to_up(S, A)
    a = A
    Q = {}
    if (M[m, A] != ∞):
        for k = m to 1:
            if (M[k, a] != M[k-1, a]):
                Q UNION {k}
                a = a – c_k
    else:
        return {}
    return Q
```

c)

The worst-running time of our algorism is $\theta(mA)$ for the nested for loop when building the M array in bottom_to_up function.

Q5:

Step1: Describe the recursive structure for the sub-problem

For every optimal path j1, j2, j3, …, jk, the coordinate when i=2 on the optimal path must be (2, j2-1), or (2, j2), or (2, j2+1). The maximum drill hardness left at i=2 is d-H[1, j1]. Among these three paths starting from (2, j2-1), (2, j2) and (2, j2+1), we select the one with the maximum amount of gold in total.

Step2: Define an array that stores optimal values for sub-problem

Let M = [1…m+1, 0…n+1, 0…d]

Let M[i, j, h] be the maximum amount of gold that can be extracted from the path starting from (i, j) with drill hardness h.

Step3: Give a recurrence relation for the array values.

Case 1: Reach to the outside of the available region

M[i, 0, h] = M[i, n+1, h] = -1 for $1 \le i \le m+1$, $0 \le h \le d$

Case 2: Reach to the maximum depth

M[m+1, j, h] = 0 for $1 \le j \le n$, $0 \le h \le d$

Case 3: If there is no drill hardness available(h=0).

M[i, j, 0] = 0 for $1 \le i \le m$, $1 \le j \le n$

Case 4: h < H[i, j] (the available drill hardness is not sufficient to drill at M[i, j])

M[i, j, h] = 0 for $1 \le i \le m$, $1 \le j \le n$

Case 5: h >= H[i, j] (the available drill hardness is sufficient to drill at M[i, j])

for $1 \le i \le m$, $1 \le j \le n$

M[i, j, h] = G[i, j] + max{M[i+1, j-1, h-H[i, j]], M[i+1, j, h-H[i, j]], M[i+1, j+1, h-

H[i, j]]}

Step 4: bottom to up

```
Def bottom_to_up(H, G, d):
    #create a 3d-array to store values
    M = [1…m+1, 0…n+1, 0…d]
    for h = 0 to d:
        for p = 1 to n:
            M[m+1, p, h] = 0 // deal with bottom case
        M[m+1, n+1, h] = -1 // deal with right corner case
        M[m+1, 0, h] = -1    // deal with left corner case
        for i = m to 1:
            M[i, 0, h] = -1
            M[i, n+1, h] = -1
            for j = 1 to n:
                if h < H[i, j] :
                    M[i, j, h] = 0
                else:
                    M[i, j, h] = G[i, j] + max(M[i+1, j-1, h-H[i, j]],
                                               M[i+1, j, h-H[i, j]],
                                               M[i+1, j+1, h-H[i, j]]}

Return M
```

T(n) = $\theta(dmn)$

Step 5: optimal solution

```
def optimal_solution ( H, G, d):
    cur_depth = 1
    S = []
    M = bottom_to_up(H, G, d)
```

```
//find the start point j'
max = 0
j' = none
for 1 ≤ i ≤ n:
    if M(cur_depth, i, d) > max:
        j'=i
        max = M(cur_depth, i, d)
add j' into S

While M(cur_depth, j', d) > 0:
    d = d - H(cur_depth, j')
    cur_depth += 1
    //find the next one in the optimal path
    Set j' as one of {j'-1, j', j'+1} which maximize M(cur_depth, j', d)
    add j' into S

return S
```

bottom_to_up costs $\theta(dmn)$

Finding the starting point costs $\theta(n)$

Finding the other point on the path costs $\theta(m)$

So total cost is $\theta(dmn)$