Algorithm Assignment 3

Chunnan Sheng
z5100764

Question 1 (a)

The counterexample: $A[1..n]=[90,89,-10,80]$
If the largest number 90 is chosen, the sum would be $90-10=80$ which is less than $89+80=169$.

Question 1 (b)

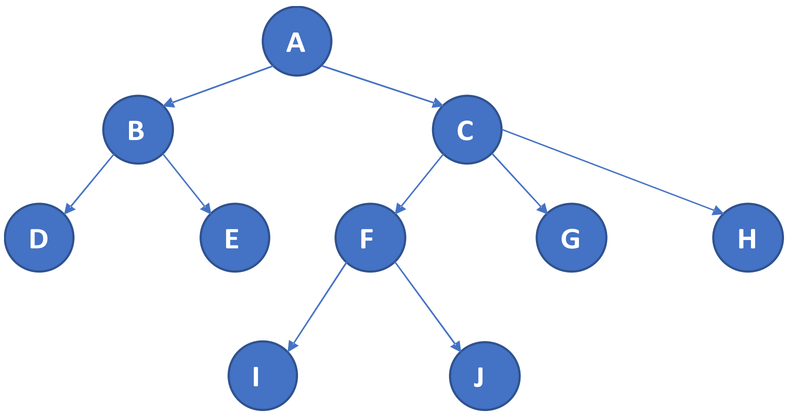
The counterexample: $A[1..n]=[80,0,90,1,78,2,3,77]$
The optimal solution is $80+90+78+77=325$ where index of 80, 90 and 78 are odd numbers while index of 77 is an even number (We assume index begins with 1, though we usually begin with 0 in computer programs).

Question 1 (c)

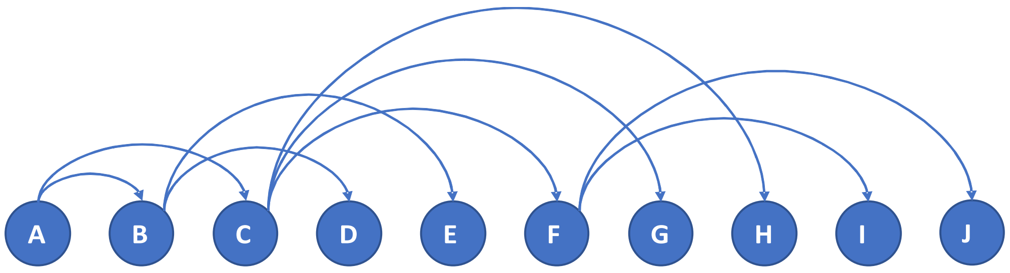
```cpp
#include <vector>
// This algorithm costs O(n) time
int get_max_sum_non_adjacent(const std::vector<int> & input)
{
    // Initialise the inclusive sum.
    // "Inclusive sum" means it is the "sum" of elements that
    // includes the left neighbour of current element.
    int incl_sum = input[0];

    // Initialise the exclusive sum.
    // "Exclusive sum" means it is the "sum" of elements that
    // does not include the left neighbour of current element.
    int excl_sum = 0;

    // Go through each element of this array.
    // We should start with the second element of this array.
    // Index of the second element is 1 (The first is 0).
    for (int i = 1; i < input.size(); ++i)
    {
        // Back up the inclusive sum
        int old_incl_sum = incl_sum;

        // The new inclusive sum is equal to the exclusive sum plus current element
        incl_sum = excl_sum + input[i];

        // Compare the old inclusive sum with the exclusive sum and pick up
        // the larger one as the new exclusive sum
        if (old_incl_sum > excl_sum)
        {
            excl_sum = old_incl_sum;
        }
    }

    // Return the larger one of incl_sum and excl_sum
    if (incl_sum > excl_sum)
    {
        return incl_sum;
    }
    else
    {
        return excl_sum;
    }
}
```

Question 2

Suppose hierarchy tree of this company is like this:



If we do Breadth-First search ($O(n)$), then we can construct a list of tree nodes like this:



We begin with the right most node J and find out cost of inviting J or not inviting J to the company retreat, then go through each node one by one until we reach the root node A . Time complexity is $O(n)$.

Each node will cache at least three members:

1. The cost of inviting myself;
2. Accumulative minimum cost if myself is invited;
3. Accumulative minimum cost if it is NOT mandatory that myself should be invited;
4. Whether myself is invited not (Optional, if all invited nodes are required)

The code would be like this:

```
#include <memory>
#include <vector>

class Employee;
typedef std::shared_ptr<Employee> Employee_ptr;

struct Employee
{
    int my_cost;
    int min_cost_if_invited;
    int min_cost;
    bool invited;
    std::vector<Employee_ptr> children;
};

int min_cost(std::vector<Employee_ptr> & node_list)
{
    // Start with the right most node in the list
    // Time complexity of this loop is O(n)
    for (int i = node_list.size() - 1; i >= 0; --i)
    {
        if (node_list[i]->children.empty())
        {
            // If current node has no children
            node_list[i]->min_cost_if_invited = node_list[i]->my_cost;
            node_list[i]->min_cost = 0;
            node_list[i]->invited = false;
        }
        else
        {
            // Min cost of children if inviting these children is not compulsory
            int min_cost_of_children = 0;

            // Cost of children if these children are invited
            int cost_of_inviting_children = 0;

            // Sum min_cost_of_children and cost_of_inviting_children
            for (auto child : node_list[i]->children)
            {
                min_cost_of_children += child->min_cost;
                cost_of_inviting_children += child->min_cost_if_invited;
            }

            // If current node is not invited, then we should consider
            // cost of inviting all children of this node.
            int min_cost_if_not_invited = cost_of_inviting_children;

            // If current node is invited, then we do not need to care too much
            // that we should invite children or not. Just include cost of myself
            // and minimum cost of all children.
            node_list[i]->min_cost_if_invited =
                node_list[i]->my_cost + min_cost_of_children;


            if (min_cost_if_not_invited > node_list[i]->min_cost_if_invited)
            {
                // There is less cost if current node is invited
                node_list[i]->invited = true;
                node_list[i]->min_cost = node_list[i]->min_cost_if_invited;
            }
            else // There is less cost if current node is not invited
            {
                node_list[i]->invited = false;
                node_list[i]->min_cost = min_cost_if_not_invited;

                // All children should be invited
                for (auto child : node_list[i]->children)
                {
                    child->invited = true;
                }
            }
        }
    }

    // Return min cost of the root node
    return node_list[0]->min_cost;
}
```

Question 3

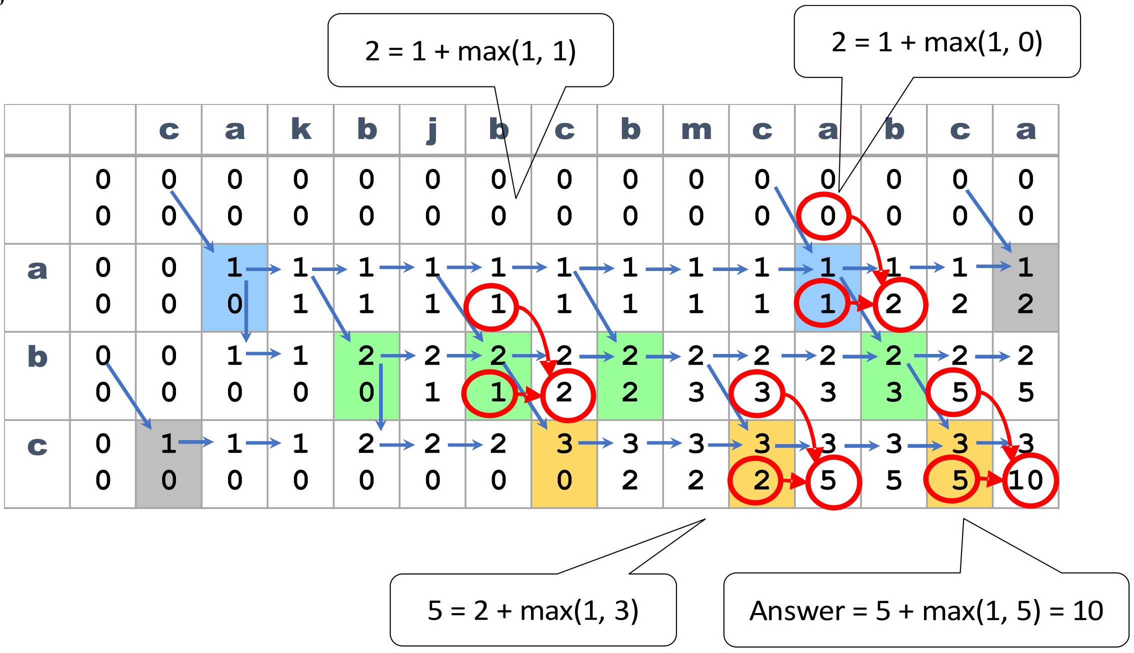
We can solve this problem via dynamic programming using a 2D table. Rows of this table is equal to length of sequence A + 1, and columns of this table is equal to length of sequence B + 1.

Each cell of this table stores 3 values:

1. Length of longest matched sequence (integer)
2. Number of all matches among left cells of current cell (integer)
3. Do letters from A and B match at this cell? (boolean)

All values in this table will be initialized to zero. We start from the top-left corner of this table and finish the algorithm at the bottom right corner, so the time complexity will be $O(mn)$ where m is length of sequence A and n is length of sequence B.

The following table is an example showing how many occurrences of “abc” are found from “cakbjbcbmcabca”.



		c	a	k	b	j	b	c	b	m	c	a	b	c	a
	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0
a	0 0	0 0	1 0	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 2	1 2	1 2
b	0 0	0 0	1 0	1 0	2 0	2 1	2 1	2 2	2 2	2 3	2 3	2 3	2 3	2 5	2 5
c	0 0	1 0	1 0	1 0	2 0	2 0	2 0	3 0	3 2	3 2	3 2	3 5	3 5	3 5	3 10

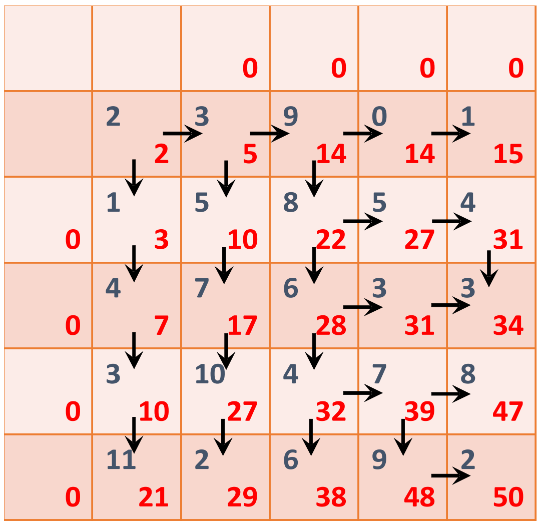
The source code clip:

```cpp
// We use the Dynamic Programming method to solve the match occurrence problem.
// Time complexity of this algorithm is O(m * n) where m is size of sequence A,
// n is size of sequence B.
int find_match_occurrences(const std::string & A, const std::string & B)
{
    // Create a 2D table to cache data of the algorithm
    // rows == A.size() + 1, cols == B.size() + 1
    // All values in this table are initialized as zero
    std::vector<std::vector<table_grid>> table;

    // Time complexity of initializtion of this table is O(m * n)
    for (int i = 0; i <= A.size(); ++i)
    {
        table.push_back(std::vector<table_grid>{});

        for (int j = 0; j <= B.size(); ++j)
        {
            table[i].push_back(table_grid{});
        }
    }

    // Traverse the table (except the first row and the first column).
    // Time complexity of this part is O(m * n)
    for (int i = 0; i < A.size(); ++i)
    {
        for (int j = 0; j < B.size(); ++j)
        {
            if (table[i + 1][j].matched && table[i + 1][j].val == i + 1)
            {
                // Update matches_from_left if the left neighbor is somewhere
                // a letter is matched.
                // Stay cautious that value of the left neighbor should be
                // no less than row index plus one.
                table[i + 1][j + 1].matches_from_left =
                    table[i + 1][j].matches_from_left +
                    std::max(1, table[i][j].matches_from_left);
            }
            else
            {
                // Just copy matches_from_left from the left neighbor
                table[i + 1][j + 1].matches_from_left = table[i + 1][j].matches_from_left;
            }

            if (A[i] == B[j])
            {
                // If a letter matches here, value of this grid is value of
                // the upper-left neighbor plus one.
                table[i + 1][j + 1].val = table[i][j].val + 1;
                // Set true flag of match
                table[i + 1][j + 1].matched = true;
            }
            else
            {
                // If there is no match here, choose the max value between
                // left neighbor and upper left neighbor
                table[i + 1][j + 1].val = std::max(table[i + 1][j].val, table[i][j + 1].val);
            }
        }
    }

    // Sum matches_from_up_left of the last row
    // Time complexity: O(n)
    for (int j = B.size(); j > 0; --j)
    {
        if (table[A.size()][j].matched && table[A.size()][j].val == A.size())
        {
            return table[A.size()][j].matches_from_left +
                std::max(1, table[A.size() - 1][j].matches_from_left);
        }
    }

    return 0;
}
```

Question 4 (a, b)

A $n \times n$ matrix can be used as a cache to store accumulated gain and direction of each movement in each cell. The accumulated gain is $max(gain(i-1, j), gain(i, j-1)) + A[i][j]$. We scan from top-left to bottom-right of the $n \times n$ grid.

Further more, the $n \times n$ cache can be extended to $(n+1) \times (n+1)$ so that it is more convenient for computer programming. The following is a 5×5 example.

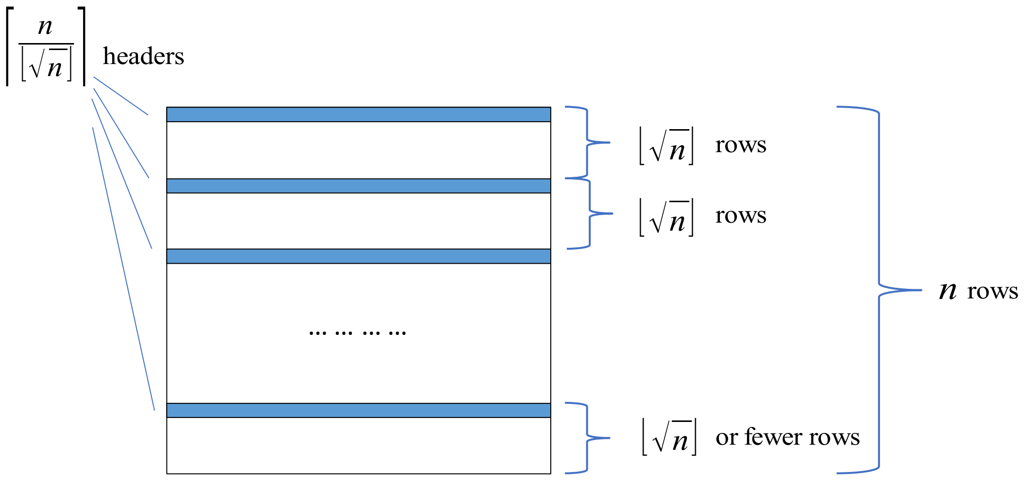


In this example the score is 50, and the optimal path is *RRDDDRDR*.

```
// Source code clip
// Time complexity of this algorithm is O(m * n)
// Space compleixty of this algorithm is O(m * n)
int route_of_maximum_gain(const std::vector<std::vector<int>> & grid, std::string & route)
{
    std::vector<std::vector<cell>> cache;
    std::vector<char> re_route;

    lint rows = grid.size();
    lint cols = grid[0].size();


    // Add the zero row
    cache.push_back(std::vector<cell>{ static_cast<size_t>(cols) + 1 });

    // Traverse the grid from top-left to bottom right.
    // Time complexity: O(m * n)
    for (lint i = 0; i < rows; ++i)
    {
        // Add a new row to the cache
        cache.push_back(std::vector<cell>{ static_cast<size_t>(cols) + 1 });

        for (lint j = 0; j < cols; ++j)
        {
            // Compare the accumulated value of the left neighbor and the upper neighbor.
            // Pick up the larger value, add the value to accumulated value of current cell.
            // By the way, determine the direction of movement, the direction of move is always
            // from the cell whose accumulated value is larger.
            if (cache[i][j + 1].acc_val > cache[i + 1][j].acc_val)
            {
                cache[i + 1][j + 1].acc_val = cache[i][j + 1].acc_val + grid[i][j];
                cache[i + 1][j + 1].move_from = 'D';
            }
            else
            {
                cache[i + 1][j + 1].acc_val = cache[i + 1][j].acc_val + grid[i][j];
                cache[i + 1][j + 1].move_from = 'R';
            }
        }
    }

    lint i = rows;
    lint j = cols;

    // Find out the optimal route.
    // Time complexity is O(m + n)
    while (i > 1 || j > 1)
    {
        re_route.push_back(cache[i][j].move_from);

        if (cache[i][j].move_from == 'D')
        {
            --i;
        }
        else
        {
            --j;
        }
    }

    // Reverse the sequence of the route
    // Time complexity: O(m + n)
    route.clear();
    for (auto itr = re_route.rbegin(); itr != re_route.rend(); ++itr)
    {
        route.push_back(*itr);
    }

    // Return the total gain
    return cache[rows][cols].acc_val;
}
```

Question 4 (c)

If it is required to use only $O(n\sqrt{n})$ additional memory, the grid can be virtually divided into $\left\lceil \frac{n}{\lfloor\sqrt{n}\rfloor} \right\rceil$ blocks and each block is of size $\lfloor\sqrt{n}\rfloor$.



$\left\lceil \frac{n}{\lfloor\sqrt{n}\rfloor} \right\rceil$ headers
$\lfloor\sqrt{n}\rfloor$ rows
$\lfloor\sqrt{n}\rfloor$ rows
… … … …
n rows
$\lfloor\sqrt{n}\rfloor$ or fewer rows

Then, we have to scan the entire grid to get headers caching data for the first row of each block. It requires $n\left\lceil \frac{n}{\lfloor\sqrt{n}\rfloor} \right\rceil = O(n\sqrt{n})$ memory to cache these headers. There is also a need to cache these blocks but we can only cache one block at each time. Size of each block would be $n\lfloor\sqrt{n}\rfloor = O(n\sqrt{n})$ (If n is not dividable by $\lfloor\sqrt{n}\rfloor$, size of the last block will be smaller than $n\lfloor\sqrt{n}\rfloor$.

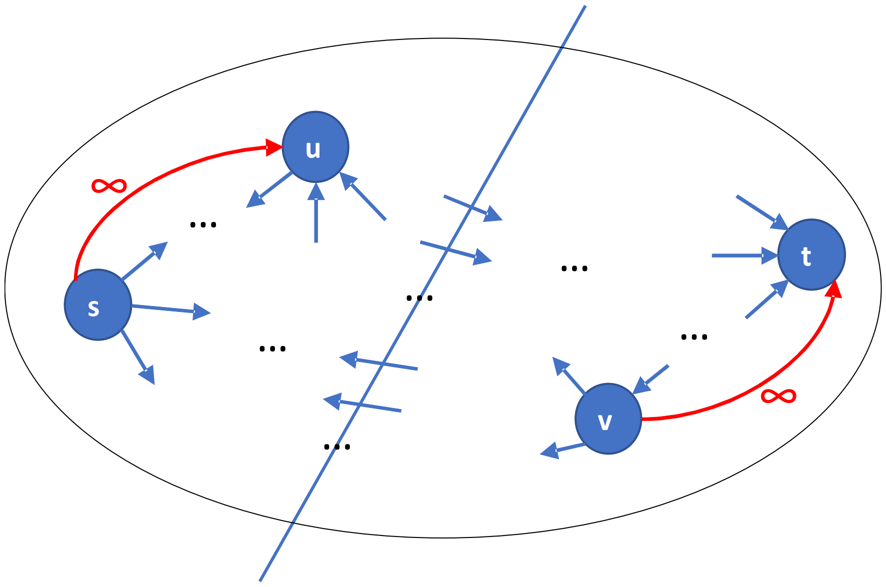
The next step is to get the **optimal path**. The solution is to scan each block using block cache whose size is $n\lfloor\sqrt{n}\rfloor$. This task starts with the last block and ends with the first block. Within each block, we scan from the first row that is header of this block to the last row.

Therefore, $2O(n\sqrt{n})=O(n\sqrt{n})$ additional memory is needed and time complexity is $2O(n^2)=O(n^2)$ because we have scanned the entire grid twice.

```
// Time complexity of this algorithm is O(m * n)
// Space compleixty of this algorithm is O(n * sqrt(m))
int route_of_maximum_gain(const std::vector<std::vector<int>> & grid, std::string & route)
{
    lint rows = grid.size();
    lint cols = grid[0].size();

    lint block_size = int_sqrt(rows);

    // This cache stores first row of each scanning block, there are
    // sqrt(m) scanning blocks, which means there are sqrt(m) block beginners.
    // Size of this cache is O(n * sqrt(m))
    std::vector<std::vector<cell>> block_beginners;

    // This cache stores accumulative values of one scanning block.
    // Each scanning block has sqrt(m) rows.
    // Size of this cache is O(n * sqrt(m))
    std::vector<std::vector<cell>> block_cache;

    // Cache of the optimal route, its size is O(m + n)
    std::vector<char> re_route;

    // There are 2 rows of scanners.
    // Scanner A represents the previous row.
    // Scanner B represents the current row.
    // We will swap A and B when finishing scanning each row.
    std::vector<cell> scanner_A{ static_cast<size_t>(cols) + 1 };
    std::vector<cell> scanner_B{ static_cast<size_t>(cols) + 1 };

    // Traverse the grid from top-left to bottom right.
    // Time complexity: O(m * n)
    for (lint i = 0; i < rows; ++i)
    {
        for (lint j = 0; j < cols; ++j)
        {
            // Compare the accumulated value of the left neighbor and the upper neighbor.
            // Pick up the larger value, add the value to accumulated value of current cell.
            // By the way, determine the direction of movement, the direction of move is always
            // from the cell whose accumulated value is larger.
            if (scanner_A[j + 1].acc_val > scanner_B[j].acc_val)
            {
                scanner_B[j + 1].acc_val = scanner_A[j + 1].acc_val + grid[i][j];
                scanner_B[j + 1].move_from = 'D';
            }
            else
            {
                scanner_B[j + 1].acc_val = scanner_B[j].acc_val + grid[i][j];
                scanner_B[j + 1].move_from = 'R';
            }
        }

        // Save first row of each scanning block.
        if (0 == i % block_size)
        {
            block_beginners.push_back(scanner_B);
        }

        // Swap scanner_A and scanner_B when finish scanning a block
        std::swap(scanner_A, scanner_B);
    }

    // Initialize the block cache whose size is sqrt(n)
    for (lint i = 0; i < block_size; ++i)
    {
        block_cache.push_back(std::vector<cell>{ static_cast<size_t>(cols) + 1 });
    }

    lint route_j_backup = cols;
    int max_gain = 0;
    // Scan each block from the last block to the first block figure out the
    // entire optimal route.
    // Time complexity of this part is O(m * n)
    for (lint i = block_beginners.size() - 1; i >= 0; --i)
    {
        block_cache[0] = block_beginners[i];

        // We should scan each block from the first row to the last row.
        // Time complexity of this part is O(n * sqrt(m))
        lint ii = 1;
        for (; ii < block_size; ++ii)
        {
            // We should be careful that rows of the last block may be fewer than sqrt(m),
            // which means we may exceed range of the original grid.
            // Check current position in the
            lint grid_row = i * block_size + ii;
            if (grid_row >= rows)
            {
                break;
            }
            // Compare the accumulated value of the left neighbor and the upper neighbor.
            // Pick up the larger value, add the value to accumulated value of current cell.
            // By the way, determine the direction of movement, the direction of move is always
            // from the cell whose accumulated value is larger.
            for (lint j = 0; j < cols; ++j)
            {
                if (block_cache[ii - 1][j + 1].acc_val > block_cache[ii][j].acc_val)
                {
                    block_cache[ii][j + 1].acc_val = block_cache[ii - 1][j + 1].acc_val
                                                     + grid[grid_row][j];
                    block_cache[ii][j + 1].move_from = 'D';
                }
                else
                {
                    block_cache[ii][j + 1].acc_val = block_cache[ii][j].acc_val + grid[grid_row][j];
                    block_cache[ii][j + 1].move_from = 'R';
                }
            }
        }

        // Do not forget to extract the final gain
        if (i == block_beginners.size() - 1)
        {
            max_gain = block_cache[ii - 1][cols].acc_val;
        }

        lint route_i = ii - 1;
        lint route_j = route_j_backup;

        // Find out part of the entire optimal route in this cached block.
        // Append the newly found route to the end of previously found route.
        while (route_i >= 0 && route_j > 0)
        {
            re_route.push_back(block_cache[route_i][route_j].move_from);

            if (block_cache[route_i][route_j].move_from == 'D')
            {
                --route_i;
            }
            else { --route_j; }
        }

        route_j_backup = route_j;
    }

    // Remove the first cell from the route
    re_route.pop_back();

    // Reverse the route
    route.clear();
    for (auto itr = re_route.rbegin(); itr != re_route.rend(); ++itr)
    {
        route.push_back(*itr);
    }

    return max_gain;
}
```

Question 5

To prevent the minimum cut from going between s and u or going between v and t , we can add two infinite capacity edges $E_{infinite}(s,u)$ connecting s and u , $E_{infinite}(v,t)$ connecting v and t as shortcuts. The following picture illustrates how the shortcuts are created.



Then we can apply Edmonds-Karp algorithm which is an improvement of Ford Fulkerson algorithm to find out the minimum cut in polynomial time that is $O(|V||E|^2)$ where V is number of vertices and E is number of edges.

Question 6 (a)

Question 6 (b)

We sort the list of tuple $\langle cost, deposit \rangle$, and then create a 2-dimensional table. In this table, columns represent which ride needs to be finished, while rows represent how much money has been spent.

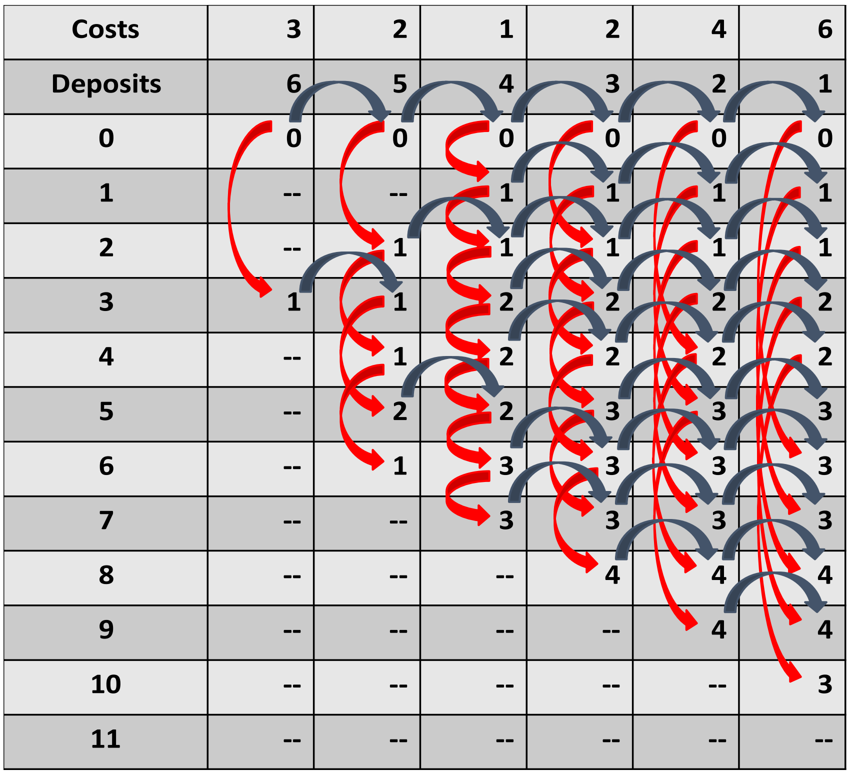
Each cell of this table caches 3 values:
1. Accumulative number of that how many different rides has been finished.
2. Column index of the previously attended ride.
3. Validity of this cell.

Why should a cell need a validity flag? For example, cost of ride 0 is 3$ then the row index cannot be 2 because you cannot spend totally 2 dollars after finishing ride 0. Thurs cell[2][0] is invalid.

The we begin with the top-left cell of this table and end with the bottom-right cell. If ride index of current cell is new (index of the previous ride is smaller), then the accumulative number of rides will increase one if the new ride is attended, otherwise, the value will stay the same.

In addition, at the same row of this table, if number of rides of the cell on the left is equal to or larger than the current cell, all members (number of rides, column index of previous ride and validity) in the current cell will be overwritten by its left neighbour.

Time complexity of this algorithm is $O(n\log n+nT)$.



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

```
#include <vector>
#include <algorithm>
#include <string>

typedef long long int lint;

// Definition of a cell in dynamic programming cache
struct cell
{
```

```cpp
    // Accumulative number of different rides
    int rides;
    // Flag showing of this cell is valid or not
    bool valid;
    // Showing id of previous ride
    // If previous ride is smaller than current ride, ++rides
    // If previous ride is equal to current ride, rides stays what it is
    int prev_ride_id;

    // Validity of this Cell is invalid by default
    // Prev_ride_id is -1 by default
    cell() : rides{ 0 }, valid{ false }, prev_ride_id{ -1 } {}

    cell(int r, bool v) : rides{ r }, valid{ v }, prev_ride_id{ -1 } {}
};

// A turple of <cost, deposit>
struct c_d
{
    int cost;
    int deposit;

    c_d() : cost{ 0 }, deposit{ 0 } {}

    c_d(int c, int d) : cost{ c }, deposit{ d } {}
};

// Compare two items via amounts of deposits
bool diff_more(const c_d & item1, const c_d & item2)
{
    if (item1.deposit > item2.deposit)
    {
        return true;
    }

    return false;
}

// Main entrance of this algorithm
// Time complexity of this algorithm is O(n * log(n) + n * T)
int max_different_rides(int T, const std::vector<int> & costs, const std::vector<int> & deposits)
{
    // Cache in 2D table (T * n) mode for dynamic programming
    std::vector<std::vector<cell>> DP;
    // List of <cost, deposit> turples
    std::vector<c_d> cd_list;
    // Number of all available rides
    lint all_rides = costs.size();

    // Initialisation of <cost, deposit> turples
    for (lint i = 0; i < all_rides; ++i)
    {
        cd_list.push_back(c_d{ costs[i], deposits[i] });
    }

    // Sort <cost, deposit> tuples in non-increase order of deposit
    // Time complexity: O(n * log(n))
    std::sort(cd_list.begin(), cd_list.end(), diff_more);

    // Initialisation of 2D cache (T * n)
    for (lint j = 0; j <= T; ++j)
    {
        DP.push_back(std::vector<cell>{costs.size()});
    }

    int max_ride = 0;

    // Begins with the top-left corner and finish with bottom-right corner of this table.
    // Row index i of this table represents how much money you have spent on rides.
    // For example, if i == 12, it means you have spent 12 on rides and you have T - 12 money left.
    // Time complexity is O(n * T)
    for (lint i = 0; i <= T; ++i)
    {
        for (lint j = 0; j < all_rides; ++j)
        {
            bool cell_valid = false;
            if (i == 0)
                // First row of this table are all of zero rides
            {
                DP[i][j].prev_ride_id = -1;
                DP[i][j].valid = true;
                cell_valid = true;
            }
            else if (DP[i][j].valid)
            {
                if (j > 0 && DP[i][j - 1].valid)
                    // If this cell is already valid, check whether the left neighbor is
                    // equal to or larger than the current one
                {
                    if (DP[i][j - 1].rides >= DP[i][j].rides)
                        // If the left one is larger, copy rides and prev_ride_id to
                        // current cell
                    {
                        DP[i][j].rides = DP[i][j - 1].rides;
                        DP[i][j].prev_ride_id = DP[i][j - 1].prev_ride_id;
                    }
                }

                cell_valid = true;
            }
            else if (j > 0 && DP[i][j - 1].valid)
                // If current cell is invalid but the left neighbour is valid
                // copy rides and prev_ride_id to current cell, and make current
                // cell valid.
            {
                DP[i][j].rides = DP[i][j - 1].rides;
                DP[i][j].prev_ride_id = DP[i][j - 1].prev_ride_id;
                DP[i][j].valid = true;

                cell_valid = true;
            }

            if (cell_valid)
            {
                if (DP[i][j].rides > max_ride)
                    // Always keep an eye on the max_ride
                {
                    max_ride = DP[i][j].rides;
                }

                if (i + cd_list[j].cost + cd_list[j].deposit <= T)
                    // (T - i - cost - deposit) shoule be >= 0.
                    // i is money you have spent.
                    // cd_list[j].cost is cost of the j-th ride.
                    // cd_list[j].deposit is deposit of the j-th ride.
                {
                    if (j > DP[i][j].prev_ride_id)
                        // Plus number of rides only when current ride is different
                        // from the previous ride.
                    {
                        DP[i + cd_list[j].cost][j].rides = DP[i][j].rides + 1;
                    }
                    else // Number of rides stays the same because we only count
                        // number of different rides.
                    {
                        DP[i + cd_list[j].cost][j].rides = DP[i][j].rides;
                    }

                    DP[i + cd_list[j].cost][j].prev_ride_id = j;
                    DP[i + cd_list[j].cost][j].valid = true;
                }
            }
        }
    }

    return max_ride;
}
```

Question 6 (c)

If T is an integer that is between n^{k-1} and n^k where $k \in \mathbb{N}$, then $O(n \log n + nT) = O(n \log n + n^{k+1}) = O(n^{k+1})$. Thus algorithm can run in polynomial time.

Question 6 (d)