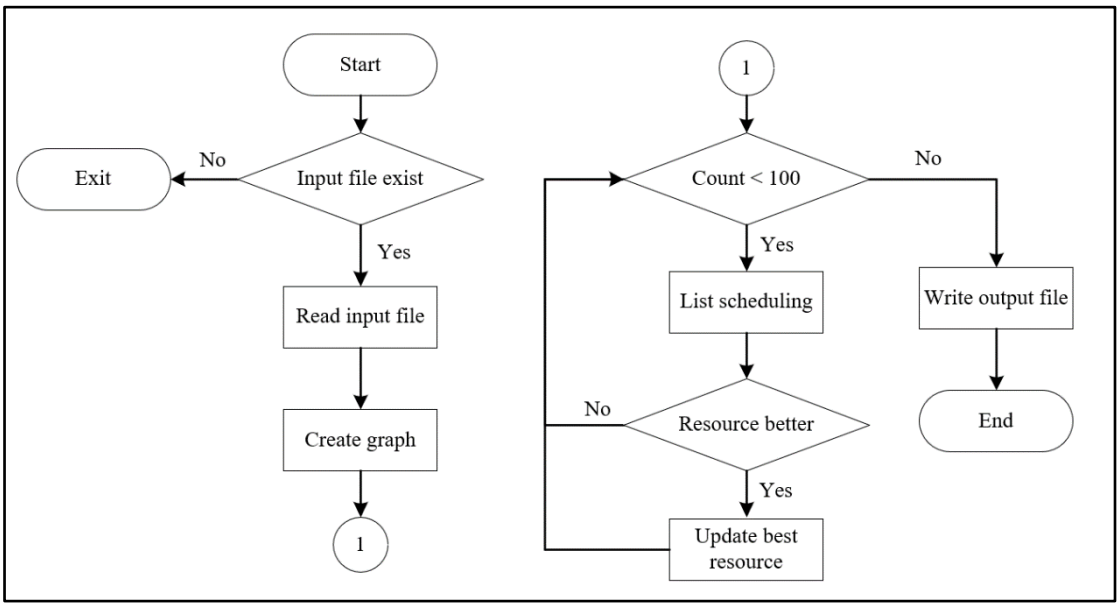
A. Readme

I. Flow Chart

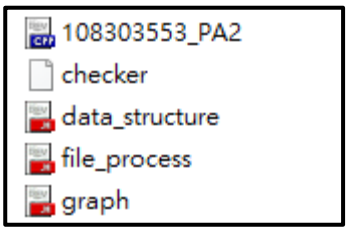


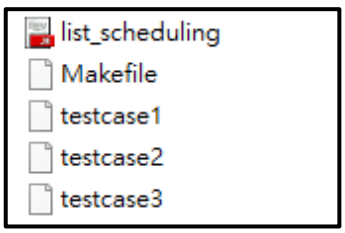
II. Compile and Execute

First, use "mkdir" command to create a directory named "PA2", then use "cd" command to enter this directory.

```
[s108303553@eda359_forclass ~]$ mkdir PA2
[s108303553@eda359_forclass ~]$ cd PA2
```

Next, put following files into this directory. Including cpp file, checker, four header files, Makefile and three testcases.





Then, use "chmod 700 checker" command to modify checker's permission so that we can use checker to check the answer. After that, use "make all" command to execute the cpp file automatically. We can find that "108303553_PA2.o" and "exe" are created after this command.

```
[s108303553@eda359_forclass ~/PA2]$ chmod 700 checker
[s108303553@eda359_forclass ~/PA2]$ make all
[s108303553@eda359_forclass ~/PA2]$ ls
108303553_PA2.cpp  checker           exe              graph.h           Makefile   testcase2
108303553_PA2.o    data_structure.h  file_process.h   list_scheduling.h  testcase1  testcase3
```

Now, we can use "make run Testcase=testcase" to generate out file. Later, use "make clean" command to remove "108303553_PA2.o" and "exe".

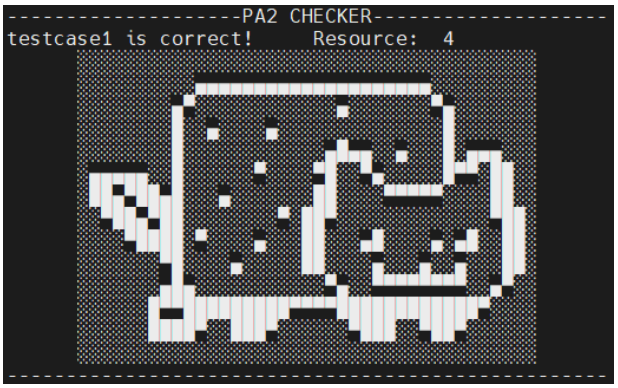
```
[s108303553@eda359_forclass ~/PA2]$ make run Testcase=testcase1
[s108303553@eda359_forclass ~/PA2]$ make run Testcase=testcase2
[s108303553@eda359_forclass ~/PA2]$ make run Testcase=testcase3
[s108303553@eda359_forclass ~/PA2]$ make clean
```

Finally, use "./checker testcase testcase.out" to check whether the output file is correct or not.

```
[s108303553@eda359_forclass ~/PA2]$ ./checker testcase1 testcase1.out
```

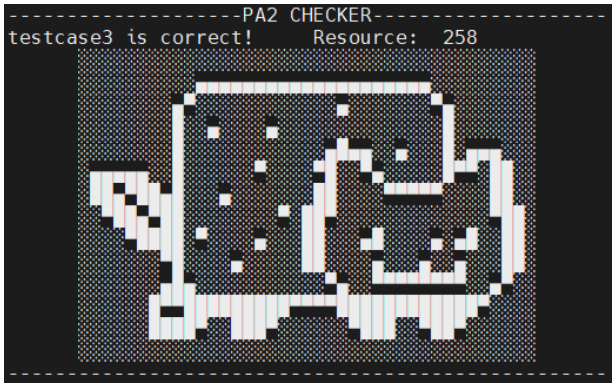
B. Completion

I. Testcase 1



II. Testcase 2

III. Testcase 3



We can see that all output files pass the check and the screen appears the "Nyan Cat". The following table shows the detail resource used in each testcase.

	adder	multiplier	total
testcase 1	3	1	4
testcase 2	3	8	11
testcase 3	69	189	258

C. Data Structure and Algorithm

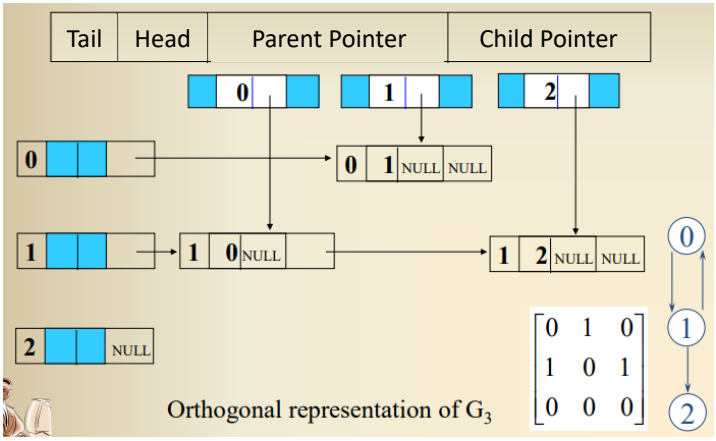
I. Data Structure

```
//------- define data structure of graph node -------
namespace data_structure
{
        struct Node
        {
                int tail; // tail id
                int head; // head id
                int tail_latency;
                int head_latency;
                int start_time;
                int alap_time; // as late as possible
                bool alap_state; // flag that show this node is teaversed or not in ALAP Calculating
                bool sche_state; // flag that show this node is teaversed or not in Scheduling
                string tail_type; // adder, multiplier, input, output
                string head_type; // adder, multiplier, input, output
                struct Node *parent;
                struct Node *child;
        };

        struct Slack_and_ID
        {
                int id; // the ID of this node
                int slack;      // the slack of this node
        };

        typedef Node* Nodeptr; // name Node* as Nodeptr
};
```

I use two kinds of data structure in PA2. One is "Node", which is used in graph. It includes tail's and head's ID, latency, start time (scheduled), type (input, output, adder or multiplier) and two pointers which point to parent (presuccessor) and child (successor). The data structure (graph) is shown below:



The other data structure is "Slack_and_ID", which will be used in list scheduling. My program records each unscheduled node's slack and their ID, then sort these slacks from small to large. If there are available resources to use, the nodes with smaller slacks will be scheduled even though their slacks are not zero.

The above two data structures are enclosed within the "data_structure" namespace, allowing me to use them by adding "using namespace data_structure;" in other header or cpp files. The use of a namespace makes the code more readable and convenient.

II. Main Module

```
int main(int argc, char *argv[])
{
        //---------------------------  data declaration  --------------------------
        vector < vector <string> > input; // use to store input data
        vector < Nodeptr > list_child, list_parent; // graph can be accessed by the two vectors
        vector < Nodeptr > best_ans; // the best result of list scheduling -> write to output file
        int best_resource[2] = {IntMax/2, IntMax/2}; // index_0 is #adder, while index_1 is #multiplier

        //-------  read iput file and store the data into a 2D string vector  --------
        File_Process file;
        file.in_file(input, argv[1]);

        //-----------------------  create graph by input data  ----------------------
        Graph graph;
        graph.create_graph(input, list_child, list_parent);

        //--------------------  use graph to do list scheduling  -------------------
        List_Scheduling list_sche;
        list_sche.best_answer(list_child, list_parent, best_ans, best_resource, input[1][2]);

        //---------------------------  write output file  --------------------------
        file.out_file(best_ans, best_resource, input[1][2], argv[2]);

        return 0;
}
```

In main module, I declared four variables:

1. input: A two-dimensional string vector that stores each word in the input file.

2. list_child, list_parent: Two vectors that store pointers to access the graph. list_child can be used to find a node's all successors, while list_parent can be used to find a node's all presuccessors.

3. best_ans: A copy of list_parent which is made when a better result occurs. After the iteration of list scheduling is complete, the data in this variable will be written to the output file.

4. best_resource: An array that stores the number of adders and multipliers used in each testcase. "best" refers to the fact that the program will execute list scheduling several times and output the best solution.

First, invoke the "in_file" class method from the "File_Process" class to read the input file and return its contents. Then, use the "create_graph" method from the "Graph" class to generate a graph. Next, call the "best_answer" method from the "List_Scheduling" class to perform list scheduling multiple times in order to find the best solution. Finally, write the optimal result to output file though "out_file" method.

III. File Process

```
//------- read input file and write output file -------
class File_Process
{
private:
    // transfer sentence to substrings
    vector <string> sentence_to_substrings(string input) // Ex: save "Hello World" as "Hello" and "World"
    {

    // transfer string to integer number
    int StoI(string num)
    {

public:
    // read input file
    void in_file(vector < vector <string> > &lvs, char *argv)
    {

    // write output file
    void out_file(vector <Nodeptr> list_parent, int resource[2], string limit_latency, char* argv)
    {

};
```

1.Private Functions

(1) sentence_to_substrings: It can split a sentence by spaces and store each word into a string vector, finally return the vector.

(2) StoI: This function can convert a string to an integer data type.

2.Public Functions

(1) in_file: The function can read input file and return the contents.

(2) out_file: It is capable of writing the results to an output file in a specified format.

IV. Graph

```
//------------------  create graph  ------------------
class Graph
{
private:
    // transfer string to integer number
    int StoI(string num)
    {

public:
    // create a new node in graph
    Nodeptr NewNode(int Tail, int Head, vector <string> NodeType)
    {

    // create a graph
    void create_graph(vector < vector <string> > input, vector <Nodeptr> &list_child,
                      vector <Nodeptr> &list_parent)
    {

};
```

1.Private Function

(1) StoI: The functionality is same as previous one in "File_Process" class.

2.Public Functions

(1) NewNode: It can allocate memory to create a new node in graph.

(2) create_graph: The function is responsible for creating a graph by DFG.

V. List Scheduling

```
//========= execute list scheduling by graph ==========
class List_Scheduling
{
private:
    // transfer string to integer number
    int StoI(string num)
    {
    // calculate ALAP time of each node
    void cal_alap(vector <Nodeptr> &list_child, vector <Nodeptr> &list_parent, string limit_latency)
    {
    // return true if all nodes are scheduled, otherwise return false
    bool sche_all(bool list_flag[], unsigned int size)
    {
    // initialize list_flag
    void list_flag_init(bool list_flag[], unsigned int size)
    {
    // remove each node's data, include: start_time, sche_state
    void remove_node(Nodeptr nptr)
    {
    // refresh graph to schedule again
    void refresh_graph(vector <Nodeptr> &list_child, vector <Nodeptr> &list_parent)
    {
    // copy resource data into resource_copy and update resource array
    void array_process(int resource[2])
    {
    // compare slack, return true if node_1's slack is smaller then node_2, otherwise return false
    static bool compare_slack(Slack_and_ID n1, Slack_and_ID n2)
    {
    // copy the data in list_parent, include its start_time, head_type and head_latency
    vector <Nodeptr> graph_copy(vector <Nodeptr> list_parent)
    {
    // schedual all nodes in graph
    bool list_scheduling(vector <Nodeptr> &list_child, vector <Nodeptr> &list_parent,
                         string limit_latency, int resource[2], bool &exe)
    {
public:
    // do list scheduling ten times to find best result
    void best_answer(vector <Nodeptr> list_child, vector <Nodeptr> list_parent,
                     vector <Nodeptr> &best_ans, int best_resource[2], string limit_latency)
    {
};
```

1.Private Functions

(1) StoI: The functionality is same as previous one in "File_Process" class.

(2) cal_alap: The function can calculate each node's ALAP time.

(3) sche_all: It is designed to determine whether all nodes have been scheduled.

(4) list_flag_init: It can initialize a bool flag array with all values set to false.

(5) remove_node: Reset a node's start_time and sche_state.

(6) refresh_graph: Reset the graph by using "remove_node".

(7) array_process: Change the number of resource for next round of list scheduling.

(8) compare_slack: Compare two nodes' slack and return a bool value.

(9) graph_copy: Copy the current graph and return it.

(10) list_scheduling: This function is responsible for scheduling all nodes in graph.

2.Public Functions

(1) best_answer: The function executes "list_scheduling" multiple times to identify the best solution for each testcase. The results are then saved in the "best_resource" and "best_ans" variables. The former stores the minimum amount of resources required, while the latter stores the graph when the optimal solution is found.

VI. Algorithm

First, I will create a graph that represents the DFG in adjacency list format. This is because the DFG is primarily a sparse graph (|Edge| << |Vertex|^2), and using adjacency lists is more efficient than using adjacency matrices.

After creating the graph, the program will start list scheduling process. During the first round of list scheduling, it will calculate the ALAP time for each node and set the initial number of adders and multipliers to 1. The list scheduling algorithm used is similar to the one described in the lecture notes. Therefore, I don't elaborate further.

Let's discuss my optimization method. After the first round of list scheduling, I will have obtained the number of adders and multipliers required. Then, use "array_process" to modify the initial number of resources to be used in the next round of list scheduling. This means that the initial number of adders and multipliers will no longer be set to 1. The code for "array_process" is as follows:

```
// copy resource data into resource_copy and update resource array
void array_process(int resource[2])
{
    // update data in resource
    if(resource[0] > 50) resource[0] = (resource[0] / 5)*4 + 1;
    else resource[0] = (resource[0] / 2) + 1;

    if(resource[1] > 50) resource[1] = (resource[1] / 5)*4 + 1;
    else resource[1] = (resource[1] / 2) + 1;
}
```

As can be seen from the code, when the current number of resources is greater than 50, 80% of the resources will be used as the initial number for the next scheduling, while if the number of resources is less than 50, 50% will be used instead. The reason why 50 is used as the threshold is that the data type of resources is an integer, so when performing division, only the quotient will be obtained. Therefore, when the number of resources is less than 50, dividing it by 5 does not result in a significant change in the data, so I instead divide it by 2.

The purpose of doing this is that once we know the number of resources used in the first round of scheduling, we can use it to set the initial value of resources for the next round of scheduling. In this way, there are many resources available for use in the early stages of scheduling, rather than waiting until the slack is 0 to schedule.

The program will execute the list scheduling algorithm 100 times. Since the most time-consuming part of my program is creating the graph and calculating the ALAP time for each node (approximately 6 seconds in testcase 3), but the graph only needs to be created once, so as ALAP time. So executing the list scheduling algorithm 100 times won't take a long time. Finally, the program will output the best result among these 100 rounds of scheduling to a file and end the entire program.