## Prize:

One 1st prize and one 2nd prize may be awarded for this challenge*. Each member (up to 4 members) of the winning team will receive the prize individually if awarded.

1st prize: Huawei FreeClip Bluetooth earbuds or equivalent (~$260 CAD)

2nd prize: Huawei FreeArc Bluetooth earbuds or equivalent (~$160 CAD)

* Participation does not guarantee a prize (i.e. If only 1 or 2 teams took on a custom challenge, it doesn't mean they would each get one of the prizes.) The output must meet certain minimum threshold in order to be considered for the above prizes.
** Our judges will evaluate and decide whether to award a team the above prizes
*** If no team was able to win the above prizes, we may give out "Good effort" prizes ($50 Amazon gift cards) to teams that show good technical skills and novelty, on the judges' discretion.

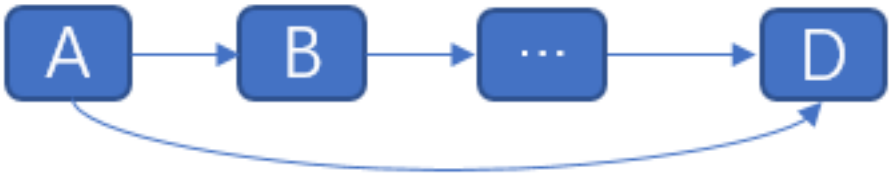
**We will invite the winning team(s) to our Burnaby office to receive the prizes and have a tour around the office.**

## Custom Challenge # 2: Automatic Decision for Re-computation

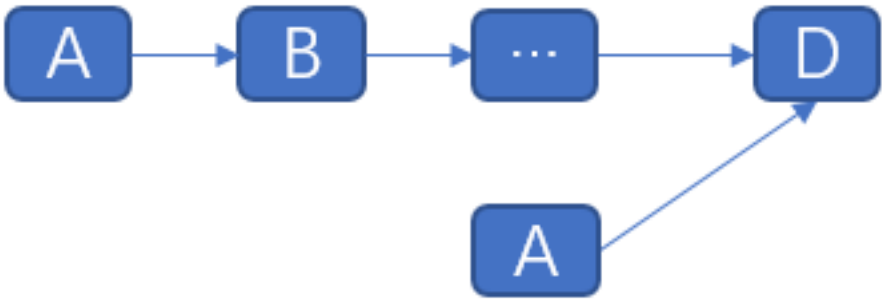
**Background information:**

In mainstream AI compilers, AI models can be represented by a Directed Acyclic Graph (DAG), where each node in the graph represents a computational operator. AI compilers have a memory reuse mechanism; when an operator completes its computation, its result remains in memory. This memory is only reclaimed once all other operators that require this memory (i.e., as input for other operators) have completed their computations, and it is then allocated for subsequent computations of other operators. Depending on the input-output relationships of the nodes within the graph, the results of certain operators may need to remain in memory for a long time before they can be reclaimed. The cumulative memory usage from layer upon layer of operator results can ultimately lead to excessively high memory peaks in AI models, causing insufficient memory on training devices and preventing training.

To address this issue, some machine learning frameworks employ recomputation techniques to mitigate it. As shown in the figure below, the memory occupied by the result of operator A might be substantial. However, due to the existence of the edge A-> D, the result of operator A must remain in memory until operator D completes its computation before it can be reclaimed.




After employing the re-computation technique, it can be transformed as shown in the figure below. Once the computation of operator B is completed, the memory occupied by the result of operator A can be reclaimed for computing other operators. Then, before the computation of operator D, operator A can be recomputed, thereby achieving the effect of trading computational performance for memory.




**Problem Description:**

Based on the above background, we propose a generalized problem. Given the input-output relationships of operator nodes within an AI model, the output operator node of the entire model, and the computation time, process, and memory size occupied by the results of each operator, how should the execution order of operators be arranged under a limited total memory size and the ability to recompute any one or more nodes, in order to minimize the time required to produce the final output? For simplicity, the problem is based on the following assumptions:

1. There is no memory fragmentation;

2. For each test case, the input operator names for each type of operator are fixed.

Data structure of the operator node:

```
class Node {

 private:

 std::string name_; // The name of the operator node, such as Add, Mul, etc.

std::vector<Node> inputs_; // All input nodes of the operator node

 int run_mem_; // Memory required for the operator's computation process

 int output_mem_; // Memory occupied by the operator's computation result

 int time_cost_; // Time taken for the operator's computation

}
```

**You are going to design a re-computation algorithm that can arrange the execution order of operators under a limited total memory size and has the ability to recompute any one or more nodes, in order to minimize the time required to produce the final output.**

More specifically, implement the following function to output the final operator execution sequence:

```
std::vector<Node> ExecuteOrder(const std::vector<Node> &all_nodes, const
std::string &output_name, long total_memory) {

...

}
```

In the rar file attached, there are 3 groups of test cases.

test_data.rar

Group 1: example1.txt and example2.txt
Group 2: example3.txt and example4.txt
Group 3: example5.txt, example 6.txt and example7.txt

They represent three levels complexity in terms of problem size. In each group, the input sequence is the same, except the first line used as “memory limit”.

You should write a simple parser to input those lines into the following or similar data structure.

```
class Node {

 private:

 std::string name_; // The name of the operator node, such as Add, Mul, etc.

 std::vector<Node> inputs_; // All input nodes of the operator node

 int run_mem_; // Memory required for the operator's computation process

 int output_mem_; // Memory occupied by the operator's computation result

 int time_cost_; // Time taken for the operator's computation

}
```

And use your designed algorithm to sort and produce the best (fastest) execution order within the constraints of the memory limit.

Please note:

1. Re-computation is allowed, as illustrate in the previous section, and it is a key for you to fit the problem in the memory budget.

2. Dependency is represented as number in the *.txt file, which is slightly different from the example in the previous section. A number as node name is actually easy to work with.

3. You need to pay attention that the input number of dependent nodes is variadic on each line.

4. Along the output of the execution sequence, please also output the “simulated time” from the accumulated “time_cost”. Sequential execution is assumed. Parallel execution is optional. It will be a plus, as it may save the total time reported. You should have debug flag to turn that off for verification purpose. However, parallel execution still needs to observe node dependency and it may not save memory. i.e. you need to account for memory cost for all the parallel operations in that parallel step.

5. AI assistant can be used by participating teams, however you will need to describe your design, e.g. how to define agent and create prompt, etc.

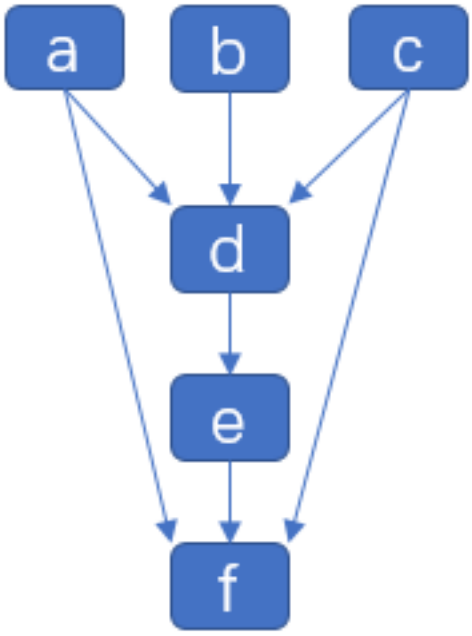
***Case description:***

***Example 1:***

Given the following operator nodes, operator output nodes, and total memory constraints, output the operator execution sequence that minimizes the total computation time as much as possible.

```
Node a{"A", {}, 100, 100, 10};
Node b{"B", {}, 100, 100, 5};
Node c{"C", {}, 100, 100, 5};
Node d{"D", {a, b, c}, 50, 100, 2};
Node e{"E", {d}, 100, 100, 2};
Node f{"F", {a, e, c}, 50, 100, 2};
std::vector<Node> all_nodes = {a, b, c, d, e, f};
std::string output_name = "F";
long total_memory = 350;
```

Based on the operator input relationships mentioned above, we can derive the following DAG:

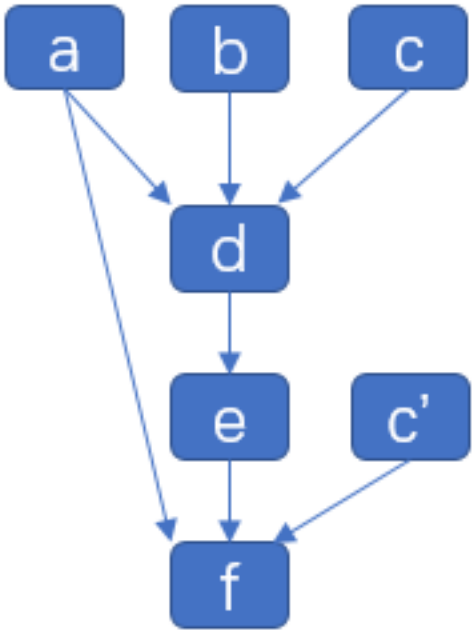



After analysis, we can output execution sequence 1:

```
Node a{"A", {}, 100, 100, 10};
Node b{"B", {}, 100, 100, 5};
```

```
Node c{"C", {}, 100, 100, 5};
Node d{"D", {a, b, c}, 50, 100, 2};
Node e{"E", {d}, 100, 100, 2};
Node c'{"C", {}, 100, 100, 5};
Node f{"F", {a, e, c'}, 50, 100, 2};
std::vector<Node> result = {a, b, c, d, e, c', f};
```

The corresponding DAG is as follows:




The execution process is as follows:

1. Current memory usage: 0. Execute operator a, resulting in a memory usage of 100. Total execution time: 10

2. Current memory usage: 100 (a). Execute operator b, resulting in a memory usage of 100. Total execution time: 15

3. Current memory usage: 200 (a, b). Execute operator c, resulting in a memory usage of 100. Total execution time: 20

4. Current memory usage is 300 (a, b, c). Execute operator d, which occupies 100 memory units for the calculation result. After execution, reclaim the memory occupied by b and c. Total execution time is 22.

5. Current memory usage is 200 (a, d). Execute operator e, which occupies 100 memory units for the calculation result. After execution, reclaim the memory occupied by d. Total execution time is 24.

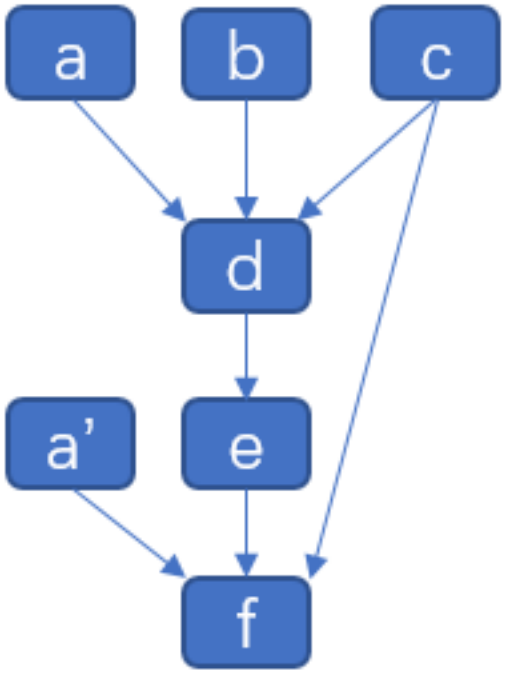
6. Current memory usage is 200 (a, e). Execute operator c', which occupies 100 memory units for the calculation result. Total execution time is 29.

7. Current memory usage is 300 (a, e, c'). Execute operator f, with a total execution time of 31. At this point, the DAG computation is completed.

Additionally, we can derive another execution sequence 2:

```
Node a{"A", {}, 100, 100, 10};
Node b{"B", {}, 100, 100, 5};
Node c{"C", {}, 100, 100, 5};
Node d{"D", {a, b, c}, 50, 100, 2};
Node e{"E", {d}, 100, 100, 2};
Node a'{"A", {}, 100, 100, 10};
Node f{"F", {a', e, c}, 50, 100, 2};
std::vector<Node> result = {a, b, c, d, e, a', f};
```

The corresponding DAG is:




The execution process is as follows:

1. Current memory usage is 0; execute operator a, and the calculation result occupies 100 memory. Total execution time is 10.

2. Current memory usage is 100 (a); execute operator b, and the calculation result occupies 100 memory. Total execution time is 15.

3. Current memory usage is 200 (a, b); execute operator c, and the calculation result occupies 100 memory. Total execution time is 20.

4. Current memory usage is 300 (a, b, c); execute operator d, and the calculation result occupies 100 memory. After execution, reclaim the memory occupied by a and b. Total execution time is 22.

5. Current memory usage is 200 (c, d); execute operator e, and the calculation result occupies 100 memory. After execution, reclaim the memory occupied by d. Total execution time is 24.

6. Current memory usage is 200 (c, e). Execute operator a', and the calculation result occupies 100 memory. The total execution time is 34.

7. Current memory usage is 300 (c, e, a'). Execute operator f, and the total execution time is 36. At this point, the DAG computation is completed.

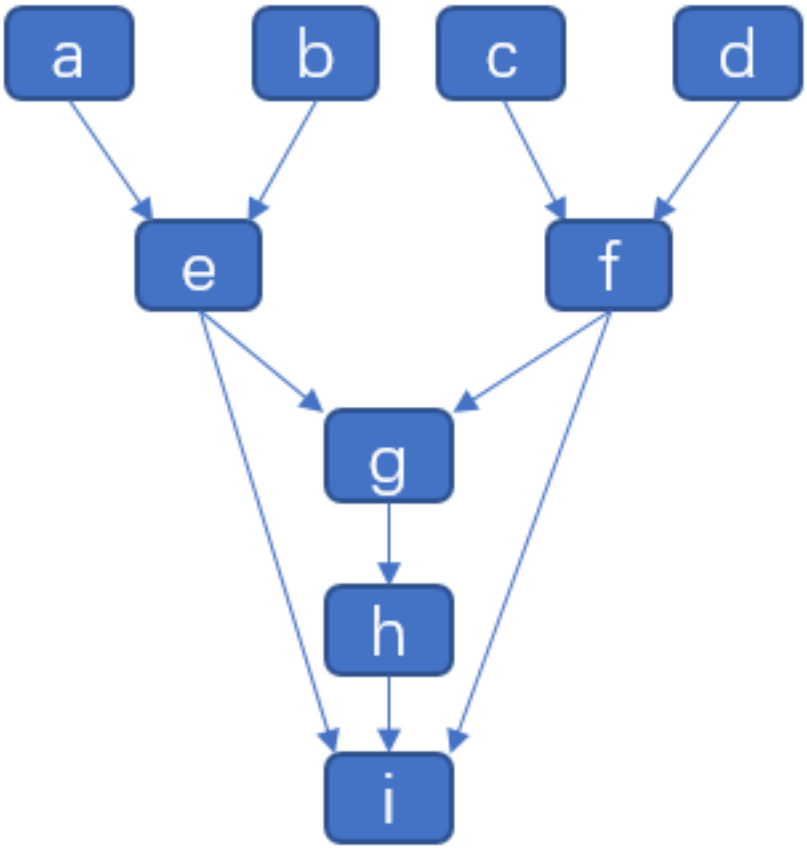
As can be seen, both operator execution sequences 1 and 2 satisfy the memory constraints, but the total execution time of sequence 1 is shorter than that of sequence 2. Therefore, the output of sequence 1 is more optimal.

***Example 2:***

Given the following operator nodes, operator output nodes, and total memory limit, output the operator execution sequence with the least total computation time.

```
Node a{"A", {}, 100, 100, 10};
Node b{"B", {}, 100, 100, 10};
Node c{"C", {}, 100, 25, 10};
Node d{"D", {}, 100, 25, 10};
Node e{"E", {a, b}, 100, 100, 5};
Node f{"F", {c, d}, 100, 100, 10};
Node g{"G", {e, f}, 100, 100, 10};
Node h{"H", {g}, 100, 100, 10};
Node i{"I", {e, f, h}, 50, 100, 10};
std::vector<Node> all_nodes = {a, b, c, d, e, f, g, h, i};
std::string output_name = "I";
long total_memory = 350;
```

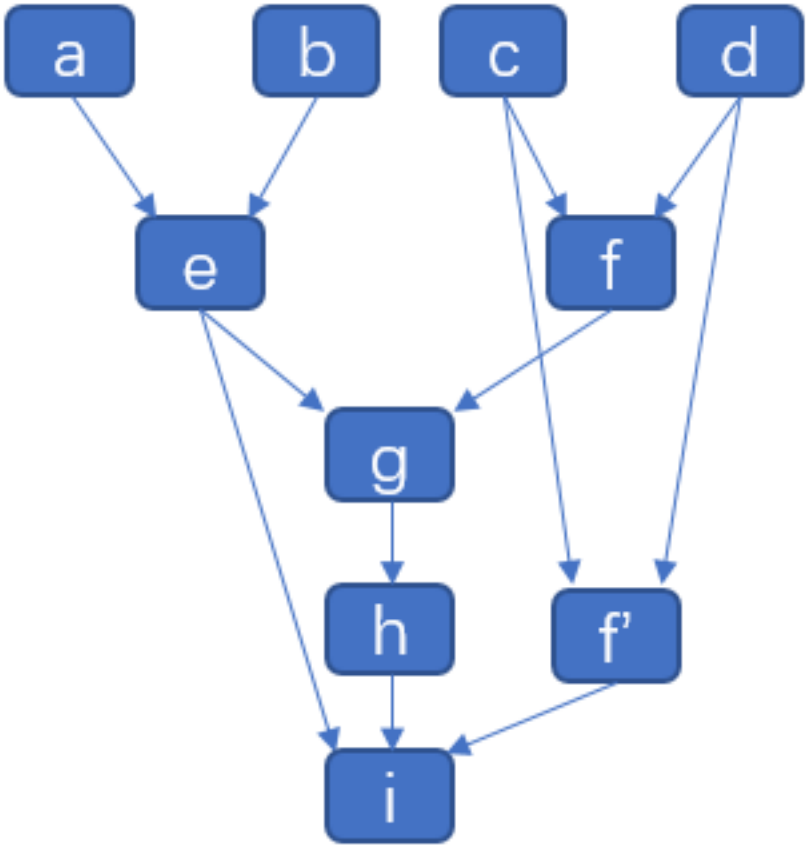
Based on the operator input relationship above, we can derive the following DAG:




After analysis, we can output the following execution order:

```
Node a{"A", {}, 100, 100, 10};
Node b{"B", {}, 100, 100, 10};
Node e{"E", {a, b}, 100, 100, 5};
Node c{"C", {}, 100, 25, 10};
Node d{"D", {}, 100, 25, 10};
Node f{"F", {c, d}, 100, 100, 10};
Node g{"G", {e, f}, 100, 100, 10};
Node h{"H", {g}, 100, 100, 10};
Node f'{"F", {c, d}, 100, 100, 10};
Node i{"I", {e, f, h}, 100, 100, 10};
std::vector<Node> result = {a, b, e, c, d, f, g, h, f', i};
```

The corresponding DAG is:




The execution process is as follows:

1. Current memory usage is 0, execute operator a, the calculation result occupies 100 memory. Total execution time is 10.

2. Current memory usage is 100 (a), execute operator b, the calculation result occupies 100 memory, total execution time is 20.

3. Current memory usage is 200 (a, b), execute operator e, the calculation result occupies 100 memory, after execution, the memory occupied by a and b is reclaimed, total execution time is 25.

4. Current memory usage is 100 (e), executing operator c, the result occupies 25 memory, total execution time is 35

5. Current memory usage is 125 (e, c), executing operator d, the result occupies 25 memory, total execution time is 45

6. Current memory usage is 150 (e, c, d), executing operator f, the result occupies 100 memory, total execution time is 55

7. Current memory usage is 250 (e, c, d, f), executing operator g, the result occupies 100 memory, after execution, memory occupied by f is reclaimed, total execution time is 65

8. Current memory usage is 250 (e, c, d, g), executing operator h, the result occupies 100 memory, after execution, memory occupied by g is reclaimed, total execution time is 75

9. Current memory usage is 250 (e, c, d, h). Execute operator f', and the calculation result occupies 100 memory. After execution, reclaim the memory occupied by c and d, with a total execution time of 85.

10. Current memory usage is 300 (e, h, f'). Execute operator i, with a total execution time of 95. At this point, the DAG computation is completed.

This execution sequence chooses to recompute operator F, resulting in the final memory peak meeting the memory limit requirements. However, if operators E or A, B, and E are chosen for re-computation, the final memory peaks do not meet the memory limit requirements.

**Evaluation Criteria:**

1. Under the constraint of total memory, the optimal solution is the one with the minimum total execution time. If multiple solutions have the same total execution time, the one with the smallest memory peak is preferred.

2. The functions are correct, the algorithm complexity is low, and the coding style is good, the final presentation is clear.