# Шаг 1 – индексы

Определяемся с типом индексов. Индексы – это «имена» ребер и узлов. Пускай, в качестве индексов у нас будет *char*

Далее необходимо перегрузить 2 шаблона функций *DefaultIndex* и *NextIndex*.

*DefaultIndex* – это значение индекса по умолчанию, или же «пустой индекс». Пример:

template<>

char DefaultIndex() {

return '0';

}

*NextIndex* – это функция, генерирующая следующий индекс по переданному.

Пример:

template<>

char NextIndex(char i) {

return i + 1;

}

# Шаг 2 - ввод

Входные данные можно засунуть в любой контейнер, главное, чтобы он сохранял порядок элементов (list, array, vector и т.п.). А порядок элементов заключает в 1 простом правиле – работа может зависеть только от предыдущих работ.  
Для построения графа и нахождения критического пути достаточно названия работы, списка требований и максимального времени выполнения (см. объявление структуры work\_t).

Пример для *char*:

list<work\_t<char>> works{

{'1', {}, 4, 7 },

{'2', {}, 8, 11},

{'3', {}, 3, 5 },

{'4', {'1'}, 7, 10},

{'5', {'1', '2', '3'}, 1, 4 },

{'6', {'3'}, 9, 13},

{'7', {'3', '4', '5'}, 8, 12},

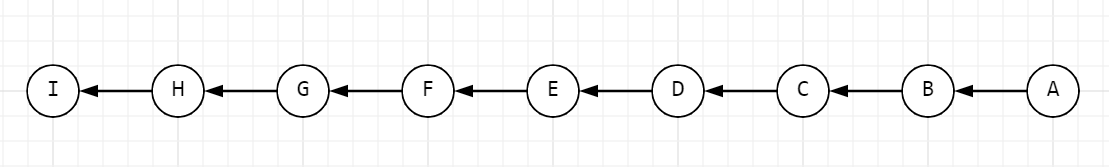
{'8', {'4'}, 5, 8 },

};

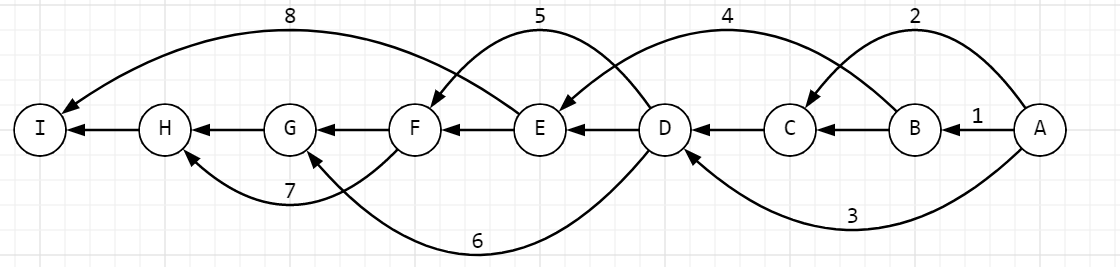
# Шаг 3 – строим граф

Построение графа происходит в 2 этапа: создание избыточного графа, оптимизация.  
Создание избыточного графа происходит так:

1. Создаем цепочку из событий и соединяем их попарно доп. работами



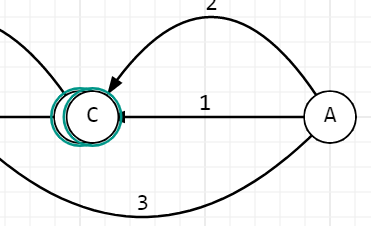
1. Проводим обязательные работ дугами. Обязательная работа выходит из i-го узла и входит в j-й, где i это порядковый номер самой последней требуемой работы, а j это порядковый номер проводимой работы. То, что все узлы попарно связаны «хордой», позволяет выполнять этот трюк.



Оптимизация:

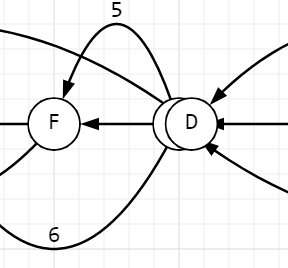
1. Представлю графическое объяснение. Задача состоит в том, чтобы поместить один узел на другой так, чтобы не произошло конфликтов связей. Под конфликтом понимается то, что между 2 узлами не может быть более 1 связи, но доп. работы можно перезаписывать и удалять при необходимости.

Попробуем поместить узел B на C.

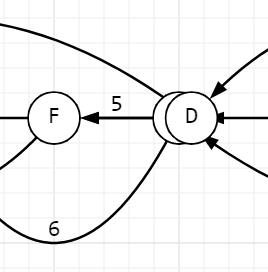


Связи 1 2 конфликтуют – произвести такую замену невозможно.

Попробуем поместить узел D на E.

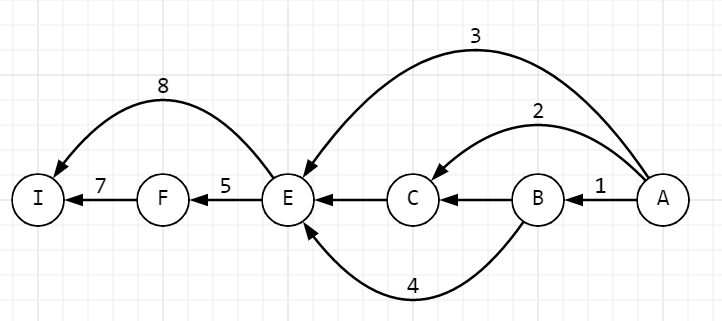


На первый взгляд кажется, что опять возникает конфликт между связями 5 и DF. Но связь DF – дополнительная, а значит ее можно просто заменить связью 5.



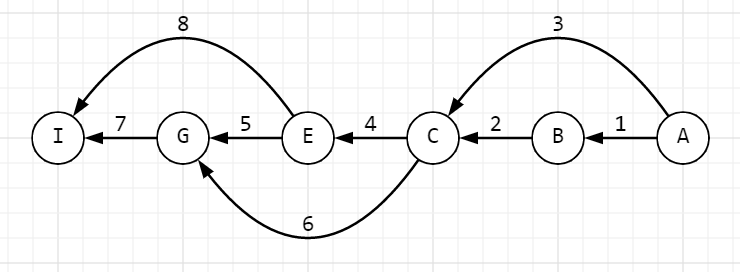
И вот уже никакого конфликта нет – узел D можно удалить, а все входящие\исходящие связи пустить в\из узла E.

1. Тоже представлю графическое объяснение. После первого шага получается такой граф



Обратим внимание на узлы E и C. В них входят доп. работы EС и CB, а еще они имеют общий родительский узел A. Это значит, что имеем право заменить доп работы EC и CB работами 3 и 2.

Если выполнять эти шаги для каждого узла (не 1-ый шаг для всех, потом 2-ой для всех, а 1-ый и 2-ой для узла A, B, C и т.д.), то результат будет немного отличаться. Но цель – убрать все лишние работы, успешно достигнута.



Graph<char> graph;

graph.assign\_nodes\_by\_works(works.begin(), works.end(), 'A');

graph.optimize();

# Шаг 4 – критический путь

По сравнению с построением графа, поиск критического пути намного проще. Все делается одной рекурсивной функцией, которая во время прохода по графу аккумулирует максимальное время выполнения и индексы каждой из работ.

list<char> critical\_path(graph.nodes.size());

graph.critical\_path(works.begin(), works.end(), critical\_path.begin());

# Шаг 5 – визуализация

Граф имеет оператор каста к строке. Это оператор вернет строковое представление списка связей. Для построения графа по списку связей можно использовать сайт <https://programforyou.ru/graph-redactor>

printf(string(graph).c\_str());