Math 203
Topics for the Chapter 7 quiz:
Scheduling

To do:

Construct an order-requirement digraph for a collection of tasks.

Find the earliest completion time from an o.r. digraph; find critical path(s).

Schedule a collection of tasks on a given number of processors using an o.r. digraph and priority list.

Construct a priority list using increasing time, decreasing time, and critical path scheduling.

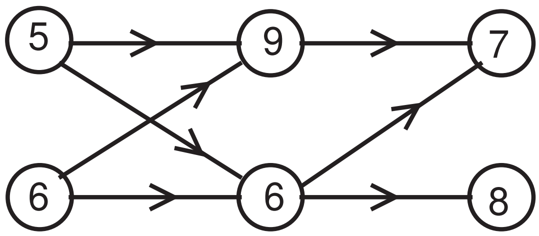
Basic idea: We have a project which involves several tasks, some of which must be completed before others can be begun (e.g., constructing a house, cooking dinner). What is the minimum amount of time required to complete all of the tasks?

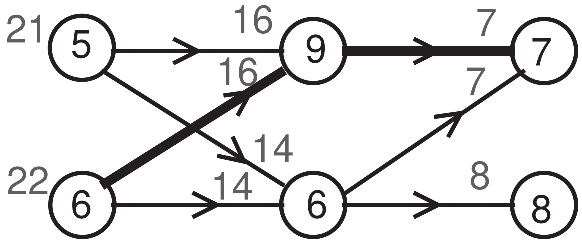
To start, we assume we have as many people as we need to assign each task to.

We model the project with a *directed graph.* Tasks = vertices; a directed edge points from one task to another if the second cannot be begun until the first is finished (e.g., build foundation → install refrigerator); the task at the head of the arrow requires the results of the one at the tail. This is the **order-requirement digraph**.

Finding the completion time: every directed path through the order-requirement digraph (**ORD**) represents a collection of tasks each of which must wait for the previous task to finish before starting. So the time to finish *just those tasks* is the sum of the times to do the tasks. The whole project cannot be completed in less than this time.

A **critical path** for a digraph is the longest (in terms of total completion time) directed path in the graph.
Fact: The shortest completion time for the collection of tasks is *equal* to the total time along the critical path(s).





Finding the completion time: set up the ORD so that every arrow goes left to right. Then, *working from right to left*, we determine the *forward completion time* (FCT) of each task, that is, the minimum time needed for the task *and all of the tasks that depend on it* to finish. First, find the *sinks*: tasks that no other task depends on (no out-arrows); their forward completion times are the times it takes to complete each task. Write this time just to the left of the task, as a label on all of the in-arrows for the task. Then, repeatedly, find a task T all of whose out-arrows have been so labelled (so the FCT of the tasks that depend on T have all been determined). The largest label on the out-arrows, plus the time for task T, is the FCT for the task T. Label all in-arrows to T with this time and continue. Repeat until every task has been dealt with. (Tasks that are *sources* - no in-arrows - should have their FCTs written to the left of each task.) The earliest completion time (ECT) of the project is the largest FCT that we wrote down (to the left of some source).

The *critical path(s)* for the project are the directed paths, starting from a source with FCT=ECT, which as you step forward traverse the out-arrows labelled with the largest FCT available at each vertex (=task). (That is, the times for the tasks along the path <u>force</u> the FCT of the source to equal the ECT.)

Tasks on the critical path(s) represent bottlenecks; places to focus efforts to save time. Shortening a task that does not lie on *all* of the critical paths will not decrease the completion time!

In the real world, we do not have an unlimited number of people (= processors) to assign to our tasks. How do things work if we have a set number of processors?

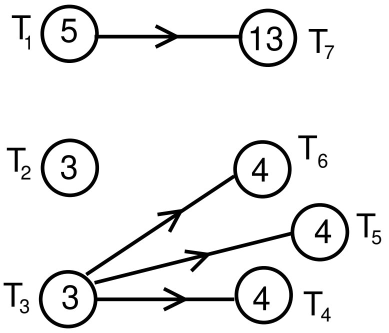
Simplifying assumptions:
– each processor can do any of the tasks (no specialists)
– once a task is started, it is completed without stopping
– when a processor finishes, it will immediately start a task if one is available (no coffee breaks)
– an order-requirement digraph tells us when a task is available (i.e., if it's prerequisite tasks have been completed)
– a **priority list** tells us which jobs we *want* to do first (e.g., shortest first, longest first, the one that gets us paid first...). Essentially, the priority list breaks ties when one processor has more than one available task to choose from.

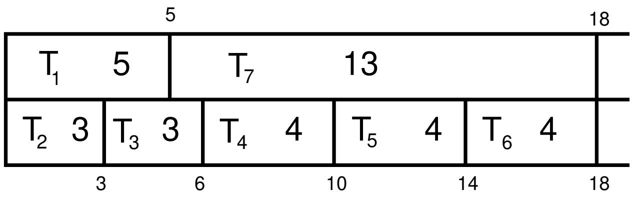
How do we assign tasks to the processors in the most efficient manner possible? The goal of our scheduling procedure might vary, depending on what we want to focus on:

1. Shortest completion time with available resources
2. Shortest amount of idle time for each processor
3. Determine the minimum number of processors needed to complete the project in a fixed amount of time.

A quick algorithm which will always find the best completion time has not been (will never be?) found. The next best thing is the **list processing algorithm**: As each processor finishes a task, assign it the highest priority task that the order requirement digraph says is available. If no task is available, stand idle until one is available. Example:

Priority list: $T_1, T_2, T_3, T_4, T_5, T_6, T_7$





total time: 18

Some strange behavior:

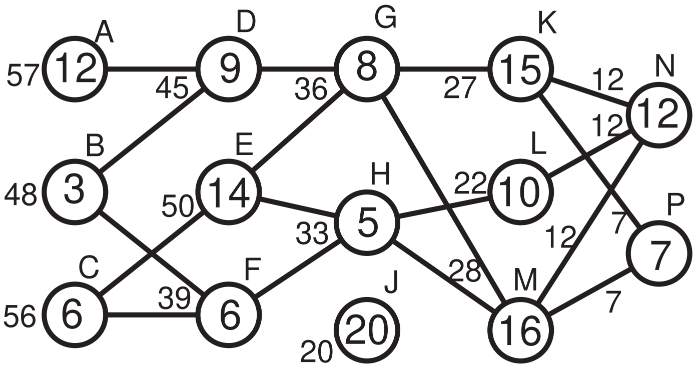
– shorten each task in the project above by 1: longer completion time!
– add a third processor: longer completion time!
– ignore the order requirement digraph: longer completion time!

Moral: We chose a very bad priority list?

Critical path analysis tells us that tasks on the critical path(s) are potential bottlenecks, slowing down overall completion time. The first task on a critical path should be given highest priority.

Critical path scheduling: Build the priority list using critical paths. The first task(s) on critical path(s) are given highest priority. Pretend that the first task is *completed*, and find the **new critical path(s)** for the remaining tasks. Give the first task on the new path the next highest priority. Continue through the ORD until every task has been placed on the priority list. Then schedule the tasks using the ORD and this priority list.

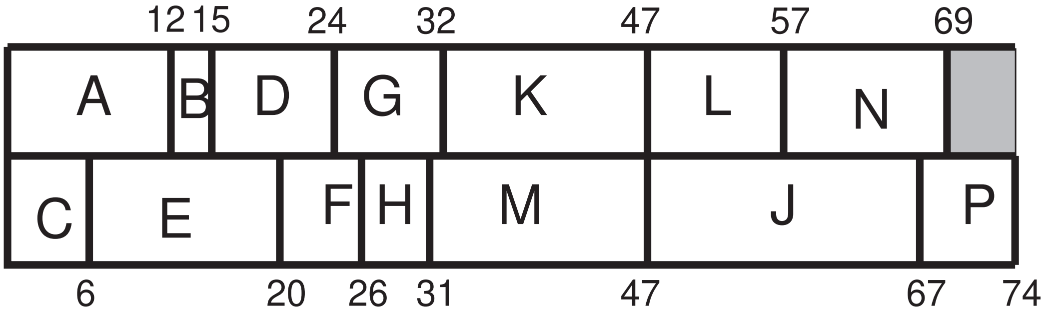
But by computing the FCTs of each task on our ORD, we have essentially <u>already</u> done the work of computing all of these critical paths. The first tasks on the critical paths are precisely the tasks with the largest FCT. Pretending that those tasks are completed essentially means to erase them from the ORD; then finding the first tasks on the new critical paths simply means to find the largest FCT that <u>remains</u>. That is, the critical path priority list is built simply by listing the tasks in order of their FCT, from largest to smallest.



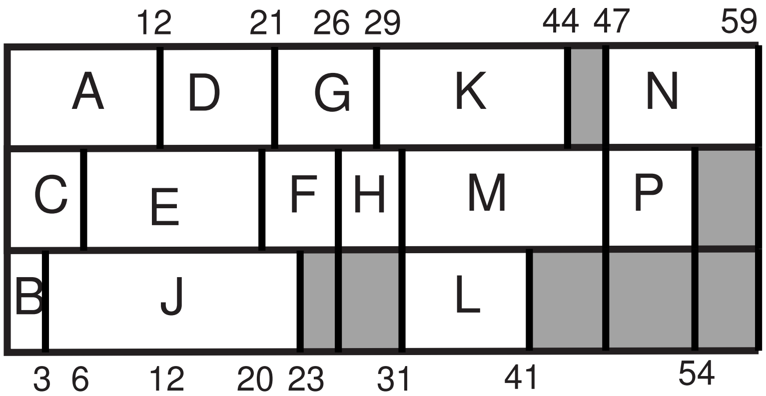
Critical path priority list:

A,C,E,B,D,F,G,H,M,K,L,J,N,P

Scheduling on 2 processors:



Scheduling on 3 processors:



What “rules” were broken to make this slightly more efficient schedule?