# 11

# Scheduling Problems and Bin Packing

***Author:*** Robert A. McGuigan, Department of Mathematics, Westfield State College.

***Prerequisites:*** The prerequisites for this chapter are basic concepts of graph theory. See, for example, Sections 9.1, 9.2, and 9.4 of *Discrete Mathematics and Its Applications*.

## Introduction

Imagine a job which is composed of a number of different tasks. To complete the job all the tasks must be completed. The tasks are performed by a number of identical *processors* (machines, humans, etc.) Each task has an associated length of time required for its completion. A **scheduling problem**, then, is to decide in what order and at what times the tasks should be assigned to processors; that is, how the tasks should be scheduled. These problems are found in many places. They can be involved in such activities as budgeting and resource allocation, making work assignments, manufacturing, and planning. Scheduling problems also arise in the efficient use of highly parallel computers.

The goal in scheduling may be to achieve the shortest possible time for completion of the job, to use the smallest possible number of processors to

complete the job within a specified time, or to optimize performance according to some other criterion.

## Scheduling Problems

In order to make our problem more specific and for simplicity, we need to make some assumptions. First we will assume that any processor can work on any task. In addition we will always assume the following two rules:

(i) No processor can be idle if there is some task it can be doing.

(ii) Once a processor has begun a task, it alone must continue to process that task until it is finished.

We make these rules to simplify our discussion. Rule (i) seems reasonable if our goal is efficiency. Rule (ii) seems more restrictive. After all, a lot of jobs are done by teams, and a processor might well do part of a task, leave it or turn it over to another processor, and work on some other task. In theory, though, we could consider very small, "atomic" tasks that could not be shared. An example might be punching a key on a keyboard. For such tasks rule (ii) does not seem unreasonable. We can also make a case for rule (ii) on the grounds of efficiency. If the processors are people, time may be lost in the change of processors in mid-task.

**Example 1** Suppose two people are planning a dinner party. Both people are skilled cooks and can do any of the tasks associated with this job, so rules (i) and (ii) are satisfied. Suppose the job of giving the dinner party is composed of the following tasks and completion times.

| Task | Time (in minutes) |
|---|---|
| Decide guest list (GL) | 15 |
| Plan the menu (M) | 30 |
| Clean house (CH) | 80 |
| Purchase food (PF) | 45 |
| Cook food (CF) | 125 |
| Set table (ST) | 15 |
| Serve food (SF) | 15 |

Some of these tasks must be done before others can be begun. For example, the food must be purchased before it is cooked or served. When this happens we say that purchasing the food is a **precedent** of cooking the food. Whenever task $A$ must be completed before task $B$ can be started we say $A$ is a **predecessor** or a **precedent** of $B$. Dependence of tasks on the completion of other tasks is common in complex projects. We can state the precedence relations for our cooking project in the following list.

| Task | Must be preceded by task(s) |
|---|---|
| Decide guest list (GL) | none |
| Plan the menu (M) | GL |
| Clean house (CH) | none |
| Purchase food (PF) | M |
| Cook food (CF) | CH, PF |
| Set table (ST) | CH, M |
| Serve food (SF) | ST, CF |

□

Another way to describe the precedence relationship between tasks is by a directed graph. Each task is represented by a vertex and there is a directed edge from $A$ to $B$ if $A$ must be completed before $B$ can be begun. The graph can be simplified (if desired) by eliminating an edge from $A$ to $B$ if there is a different directed path from $A$ to $B$. This simplification is not essential for the way we are going to use the graph, but it makes it easier to understand. We do not lose any precedence information this way since if we delete edge $(A, B)$ there must be an alternative directed path from $A$ to $B$. The fact that $A$ must be completed before $B$ is started is implied by the directed path.

The completion time for each task is written inside the circle for the vertex representing that task. A graph of this sort is called an **order requirements digraph**.

**Example 2** Draw the order requirements digraph for the dinner party of Example 1.

*Solution*: The digraph is drawn in Figure 1. □

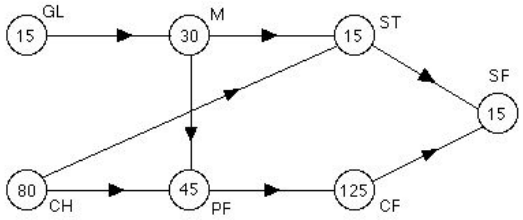



**Figure 1. An order requirements digraph.**

An important feature of an order requirements digraph is the *critical path*.

**Definition 1** A *critical path* in an order requirements digraph is a directed path such that the sum of the completion times for the vertices in the path is the largest possible. □

It is possible that there may be more than one critical path in an order requirements digraph. In that case all critical paths must have the same sum of times for their vertices.

Since the tasks on a critical path must be done in the order they occur on the path, the sum of the times for a critical path gives us a lower bound for the completion time for the whole project. If the number of processors available is unlimited, then the total time for the critical path is the shortest possible completion time for the project.

To see this, note that the earliest starting time (where the start of the project is time 0) for any task $T$ is the largest sum of times on a directed path to that task. Every precedent of $T$ will be on a directed path to $T$. Since we look for the longest such directed path, all precedents of $T$ will be finished before we start $T$.

If there are a limited number of processors, then this method of determining the starting time of a task does not apply since it might be necessary to wait until a processor is available.

**Example 3** Find the critical path for the party digraph in Figure 1.

*Solution*: The critical path for the party digraph is:

$$GL \to M \to PF \to CF \to SF$$

with a total time of 230 minutes. A schedule which is completed within this time limit is shown in Figure 2. Time is measured horizontally from left to right. The shaded areas indicate times when the processor is idle. □

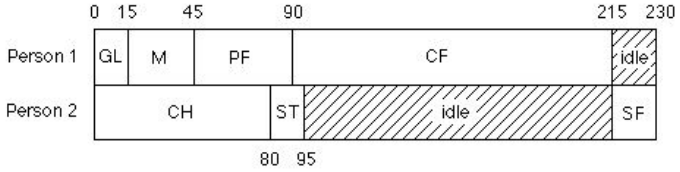



**Figure 2. A schedule for the dinner party preparation.**

If there is a limited, perhaps small, number of processors, then we could easily build up a backlog of tasks which have their predecessors completed but can't be begun because there is no free processor. When a processor does become free, which of these ready tasks should be assigned to it?

We amend our definition of the critical path method to state that the ready task which heads the directed path with the greatest sum of processing times should be assigned to the next free processor. If there is more than one such task, then any of them may be chosen at random. This criterion only applies when the number of processors is limited. In the unlimited case, every task

can be assigned to a processor immediately when its predecessors have been completed.

**Example 4** Our critical path method can yield a very bad schedule in some cases. The example in Figure 3, due to Graham [3], using four processors, shows the tasks ($A$–$L$), their times, and the order requirement digraph. Figure 4 shows the critical path schedule and an optimal schedule for this example. □

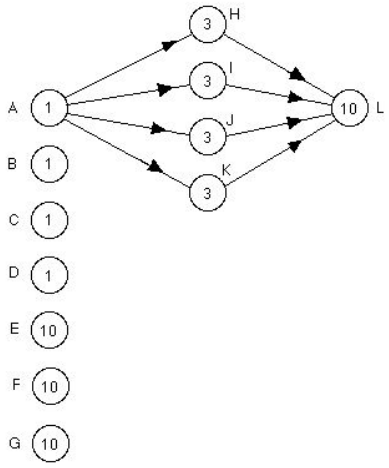



**Figure 3. A bad critical path scheduling problem.**

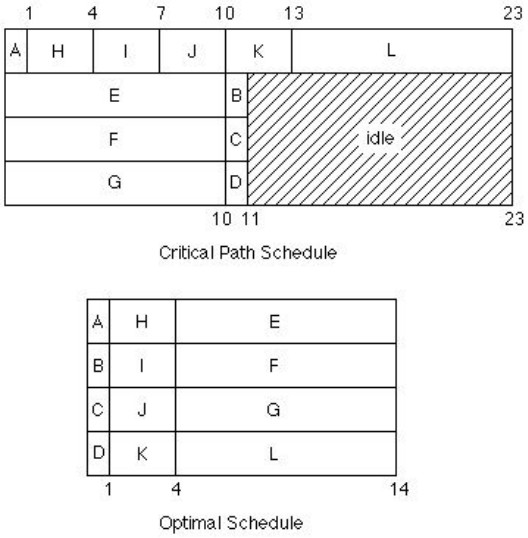



**Figure 4. Critical path and optimal schedules.**

If we carry out the critical path method for the order requirement digraph of Figure 3 assuming there are no limitations on the number of processors, then we would obtain the schedule shown in Figure 5.

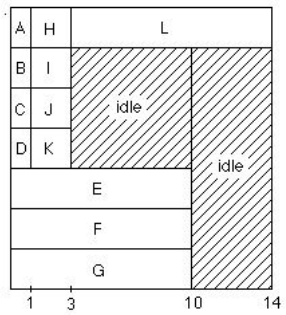



**Figure 5. An optimal schedule.**

In this case we would use seven processors and still obtain an optimal completion time of 14 time units.

## Optimal Algorithms

Since optimal schedules are by definition the best possible, it is natural to ask whether some efficient algorithm other than the critical path method would always yield the optimal schedule for a job. We ask for an efficient algorithm because it is always possible to find an optimal schedule by exhaustive search. That is, we simply examine all possible schedules and select one that takes the least time to complete. The problem with this approach is that if the number of tasks is very large, generating all possible schedules takes an unacceptable amount of time even using a computer.

Algorithm analysis is concerned with measuring how much time (or how many steps) is necessary for execution of an algorithm, as a function of the size of the problem. In the case of scheduling, the size might be measured by the number of tasks. An algorithm is generally considered efficient if the number of steps for its execution is bounded above by a polynomial function of the size of the problem. Then we would say that there is a *polynomial time* algorithm for the problem. For an exhaustive search to find an optimal schedule for $n$ tasks we would expect the number of steps to grow at a rate similar to that of $n!$. Exhaustive search is thus highly inefficient.

Mathematicians have developed a term which describes algorithmic problems for which no polynomial time algorithm has been found, but which have

a similar property. Specifically, a problem belongs to the **class NP** if having somehow (even by guessing) obtained the correct solution, it is possible to check it in polynomial time. Many difficult algorithmic problems have been shown to belong to class NP. In fact, many such problems are what is known as **NP complete**. This means that they belong to class NP and that if they have a polynomial time solution algorithm, then so do *all other problems in* NP. It is generally conjectured, but not yet proved, that NP-complete problems do not have polynomial time solution algorithms. It has been proved that the general problem of finding optimal schedules is NP-complete.

In view of this, we have little hope of finding efficient algorithms for the general optimal scheduling problem. Therefore, it becomes interesting and important to find efficient algorithms which, while not always yielding optimal schedules, do produce schedules which are close to optimal, or at least not too far from optimal.

It can be shown that the ratio of the completion time of a critical path schedule to that of an optimal schedule can never be greater than 2. That is, critical path schedules can never take more than twice the time of optimal schedules. In certain cases the critical path method does even better. If the tasks are independent (that is, there are no precedence relations among the tasks), then a critical path schedule never takes more than one third more time than an optimal schedule. (See reference [3].)

Critical path schedules could easily not be optimal, but as a practical matter we have seen that there is a limit to how bad they can be. Furthermore, it may be relatively easy to modify a critical path schedule to get an optimal one.

A special case in which good methods do exist for determining an optimal schedule occurs when there are just two processors and all the tasks have equal length. We describe two such methods here. Since all the tasks are assumed to take the same amount of time to process, we will not mention the time for each task explicitly.

**Example 5** The Fujii-Kasami-Ninomiya method, developed in 1969, makes use of the idea that two tasks $A$ and $B$ can be executed during the same time interval only if they are incomparable, i.e. there is no directed path from one to the other. Start with an order-requirement digraph. Create a second, non-directed graph called the *incomparability graph* of the tasks by having a vertex for each task and connecting two vertices by an edge if and only if the corresponding tasks are incomparable. Figure 6 shows an order-requirement digraph and its incomparability graph.

Now find in the incomparability graph the largest set of edges having no two with a vertex in common. In Figure 6 the edges $AB$, $CD$, $EF$, $GH$, and $IJ$ form such a set of edges. These pairs of vertices then show the tasks to execute simultaneously. The rest (if any) are executed one at a time.

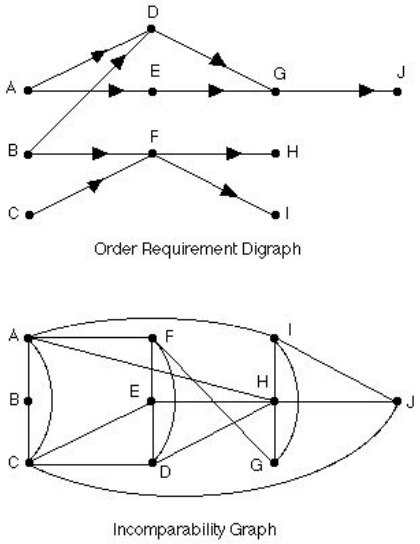



**Figure 6. An order requirement digraph and incomparability graph.**

Sometimes this method does not yield pairs of tasks which can be executed simultaneously. For example, instead of the edges we chose above suppose we had picked $AB$, $CG$, $EF$, $DH$, and $IJ$. Then we cannot execute $C$ and $G$ simultaneously and $D$ and $H$ simultaneously. In this and similar cases it is necessary to exchange some vertices to obtain simultaneously executable pairs. Here we would exchange $G$ and $D$ to get our original set. □

Good algorithms exist for finding the required maximal set of disjoint edges.

**Example 6** The second method we present is the Coffman-Graham algorithm. This algorithm works only with the order-requirement digraph. To begin, remove all unnecessary directed edges from the order-requirement digraph. That is, if there is a directed edge from $A$ to $B$ and some other directed path also from $A$ to $B$, then remove the directed edge. This must be done as much as possible.

Now choose a vertex with no outgoing directed edges, i.e. no successor vertices, and assign it label 1. Continue numbering tasks with no successors with 2, 3, etc., until all such tasks are numbered.

For each task which has all its successor vertices numbered, create a sequence of all its successors' numbers in decreasing order and assign that sequence as a temporary label to the vertex. Once all possible temporary sequence labels have been assigned, determine the vertex whose sequence label

comes first in dictionary order and assign that vertex the next available number as a permanent label. Recall that in dictionary order $(5, 6, 1)$ is smaller than $(6, 2, 1)$, $(5, 6, 1, 7)$, and $(7)$. Repeat this process until all vertices have been assigned permanent number labels. The optimal schedule is now obtained by assigning to any free processor the highest numbered task not yet processed. Figure 7 shows the digraph in Figure 6 with labels assigned.

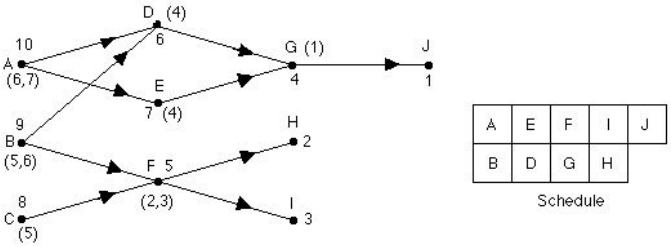



**Figure 7. Labeled order requirement digraph.**

The idea behind this algorithm is to place earlier in the schedule tasks which either head long directed paths or have many successors. This differs from the critical path method which only considers whether a task heads a long path. □

To describe the Coffman-Graham algorithm (Algorithm 1) in pseudocode, let $T(v)$ be the temporary label of $v$, $L(v)$ be the permanent label of $v$, and let $S(v)$ be the set of all successors of the vertex $v$, i.e. the set of all vertices $w$ such that there is a directed edge from $v$ to $w$.

## Bin Packing

Up to now we have been studying the problem of finding the shortest completion time for a job when the number of processors is fixed in advance. Now we ask, given a deadline time $d$, what is the minimum number of processors needed to complete the job within the deadline? This sort of problem is known as a **bin packing** problem. For bin packing we imagine a set of objects $O_1, \ldots, O_n$ with weights $w_1, \ldots, w_n$. The goal is to pack the objects into the minimum number of identical bins, each having the same capacity of holding at most a total weight of W. In the scheduling version the processors are the bins, the tasks are the objects, and the weights are the times of the tasks. We try to assign tasks to the fewest possible processors provided that no processor can be assigned tasks whose total time exceeds the deadline time.

Bin packing problems are very common. For example, consider the problem of determining the minimum number of standard length boards needed to

**ALGORITHM 1 Coffman-Graham Algorithm.**

**procedure** *Coffman-Graham*($G$: order requirement digraph)
{$G$ has vertices $v_1, \ldots, v_n$}
**for** all pairs of vertices $u, v$ of $G$
  **if** $(u, v)$ is a directed edge and there is a different directed path from $u$ to $v$ **then** delete $(u, v)$
$U :=$ set of all vertices of $G$
{$U$ is the set of unlabeled vertices}
**for** all $v \in U$, $L(r) := \infty$
$c := 1$
{$c$ is next available value for $L$}
**for** $i := 1$ **to** $n$
  **if** $outdegree(v_i) = 0$ **then**
  **begin**
    $L(v_i) := c$
    $c := c + 1$
  **end**
  {all vertices with no successors are labeled}
**while** $U \neq \emptyset$
  **for** all vertices $v \in U$
  **begin**
    **if** $L(w) \neq \infty$ for all $w \in S(v)$ **then** $T(v) :=$ the decreasing sequence of $L(w)$ for all $w \in S(v)$
    among all $v$ with $L(v) = \infty$ and $T(v)$ assigned, choose $v$ with minimal $T(v)$ in dictionary order
    $L(v) := c$
    {$v$ has a permanent label}
    $c := c + 1$
    $U := U - \{v\}$
  **end**
{assign the vertices to processors in decreasing order of $L(v)$}

produce a certain amount of boards of various non-standard lengths. In this case the objects are the shorter boards and the bins are the standard length boards. A more obvious example of bin packing is the problem of loading objects onto trucks for transportation. Each truck has a limit to the amount of weight it can carry and we wish to use as few trucks as possible. Consider also the problem of scheduling patients in an emergency room setting. Each doctor can only provide a limited amount of service, we want to have each patient taken care of as soon as possible with as few personnel as possible.

Analogous situations occur whenever some commodity is supplied in standard amounts and must be subdivided into given smaller amounts. In the context of scheduling theory we shall consider only the case where there are no precedence relations among the tasks, i.e., the tasks are independent. Even in this simplified situation no good algorithm is known which produces optimal solutions in every case. The only known algorithm which always produces optimal solutions is exhaustive search: examine all possible packings and choose the best one. This method is so inefficient that even moderate size problems would require more computing capacity than currently exists in the world.

Again we are forced to try for algorithms which will not produce packings that are too bad. One approach is known as **list processing**. The objects to be packed are arranged in a list according to one of many possible algorithms and then processed in the order of the list. For example, consider the **first-fit** algorithm. The weights are arranged arbitrarily in a list $(w_1, w_2, \ldots, w_n)$. We then pack the objects in the order of the list, putting an object in the first bin it will fit into and opening a new bin when the current weight will not fit in any previously opened bin. Let $FF(L)$ be the number of bins required for the list $L$ according to the first-fit algorithm and let $OPT(L)$ be the smallest possible number of bins for $L$. In 1973, Jeffrey Ullman showed that

$$FF(L) \leq \frac{17}{10} OPT(L) + 2$$

for any list of weights $L$.

Algorithm 2 gives the pseudocode description of the first-fit algorithm.

**ALGORITHM 2 First-fit Algorithm.**

**procedure** $FF(W$: an arbitrary list $w_1, \ldots, w_n)$
{$b_1, \ldots$ is the list of bins; object $O_i$ has weight $w_i$; $L(b_j)$ = total weight placed in $b_j$}
$k := 1$
{$k$ is number of bins opened}
**for** $i := 1$ **to** $n$
**begin**
{find first available bin}
  $j := 1$
  **while** $O_i$ not packed **and** $j \leq k$
  **begin**
    **if** $L(b_j) + w_i \leq d$ **then** pack $O_i$ in $b_j$
    $j := j + 1$
  **end**
  **if** j=k+1 **then** open $b_j$ and pack $O_i$ in $b_{k+1}$
**end**

One of the reasons FF(L) might be large is the arbitrary order of the weights in L. The large weights might come at the end of the list. Perhaps a more efficient packing could be accomplished if the weights were first sorted into decreasing order. This method is known as **first-fit-decreasing**, and the number of bins it requires is FFD(L). It has been shown that

$$FFD(L) \leq \frac{11}{9}OPT(L) + 4.$$

For lists requiring a large number of bins, the 4 is relatively insignificant and FFD is much more efficient than FF.

**Example 7** Use the first-fit algorithm to pack the following weights in bins of capacity 100:

7 7 7 7 7 12 12 12 12 12 15 15 15
36 36 36 36 36 52 52 52 52 52 52 52.

*Solution*: The first-fit algorithm yields the following bin packing:

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| bin 1: | 7 | 7 | 7 | 7 | 7 | 12 | 12 | 12 | 12 | 12 | | |
| bin 2: | 15 | 15 | 15 | 36 | | bin 3: | | 36 | 36 | | | |
| bin 4: | 36 | 36 | | | | bin 5: | | 52 | | | | |
| bin 6: | 52 | | | | | bin 7: | | 52 | | | | |
| bin 8: | 52 | | | | | bin 9: | | 52 | | | | |
| bin 10: | 52 | | | | | bin 11: | | 52. | | | | |

However, an optimal packing uses only 7 bins:

| | | | | | | |
|---|---|---|---|---|---|---|
| bin 1: | 52 | 36 | 12 | | | |
| bin 2: | 52 | 36 | 12 | | | |
| bin 3: | 52 | 36 | 12 | | | |
| bin 4: | 52 | 36 | 12 | | | |
| bin 5: | 52 | 36 | 12 | | | |
| bin 6: | 52 | 15 | 15 | 15 | | |
| bin 7: | 52 | 7 | 7 | 7 | 7 | 7. |

This packing is optimal because the total weight to be packed is 684, so it cannot be packed in 6 bins of capacity 100. □

Though FFD is more efficient, FF has at least one advantage. FF can be used when the list of weights is not completely available while the packing is being done. This could easily occur in a factory where trucks are loaded as objects are produced.

The scheduling and bin packing problems presented here are typical. Perhaps the reader can think of many more; indeed they are ubiquitous in modern life. Furthermore, as is the case with many difficult algorithmic problems, it

is most likely impossible to find efficient optimal algorithms. Thus research is focussed on finding algorithms which yield good solutions or ones that are not too far from optimal. This topic is the subject of much current research. The reader interested in additional results, more technical details, and additional technical references is urged to consult the references below.

### Suggested Readings


**1.** M. Garey and D. Johnson, *Computers and Intractability. A Guide to the Theory of NP-Completeness*, W. H. Freeman, New York, 1979.

**2.** M. Garey, R. Graham, and D. Johnson, "Performance guarantees for scheduling algorithms", *Operations Research*, Vol. 26, 1978, pp. 3–21.

**3.** R. Graham, "Combinatorial Scheduling Theory", *Mathematics Today*, ed. L. Steen, Springer-Verlag, 1978.


## Exercises

**1.** A researcher plans a survey consisting of the following tasks:

| Task | Precedences | Time (days) |
|---|---|---|
| Design of questionnaire ($A$) | none | 5 |
| Sample design ($B$) | none | 12 |
| Testing of questionnaire ($C$) | $A$ | 5 |
| Recruit interviewers ($D$) | $B$ | 3 |
| Train interviewers ($E$) | $D, A$ | 2 |
| Assign areas to interviewers ($F$) | $B$ | 5 |
| Conduct interviews ($G$) | $C, E, F$ | 14 |
| Evaluate results ($H$) | $G$ | 20 |

Construct the order requirement digraph for this project.

**2.** A construction company assembles prefabricated houses in its factory and transports them to the site where they are attached to the foundations. The job consists of the following tasks:

| Task | Precedences | Time (days) |
|---|---|---|
| Inspect site and prepare plans ($A$) | none | 3 |
| Level site and build foundation ($B$) | $A$ | 4 |
| Construct wall panels, floors roof, and assembly ($C$) | $A$ | 5 |
| Transportation to site and positioning on foundation ($D$) | $C$ | 1 |
| Attach to foundation and make final installation ($E$) | $B, D$ | 4 |

Construct the order requirement digraph for this project.

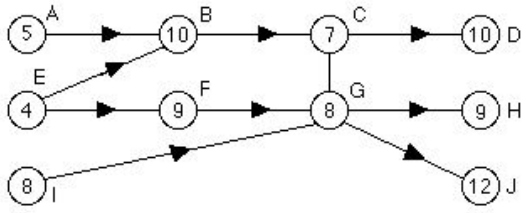
**3.** A publisher wishes to put out a new book. What is the earliest date that the book can be ready? The following tasks and times are involved. Assume unlimited processors. Find the optimal schedule.

| Task | Precedences | Time (weeks) |
|---|---|---|
| Appraisal of book by reviewers ($A$) | none | 8 |
| Initial pricing ($B$) | none | 2 |
| Market assessment ($C$) | $A, b$ | 2 |
| Revision by author ($D$) | $A$ | 6 |
| Editing of final draft ($E$) | $C, D$ | 4 |
| Typesetting ($F$) | $E$ | 3 |
| Preparation of plates ($G$) | $E$ | 4 |
| Design of jacket ($H$) | $C, D$ | 6 |
| Printing and binding ($I$) | $F, G$ | 8 |
| Inspection and final assembly ($J$) | $I, H$ | 1 |

**4.** Find the critical path schedule for the job with the following order requirement digraph.

a) With unlimited processors.

b) With only two processors.

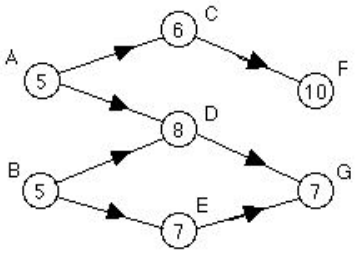



**5.** Consider the following order requirement digraph.

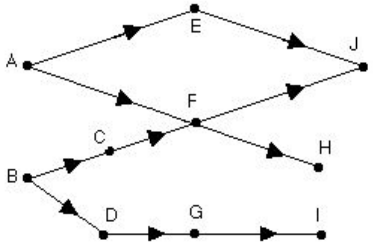
a) Assuming only two processors, find the critical path schedule.

b) What is the optimal schedule for two processors?

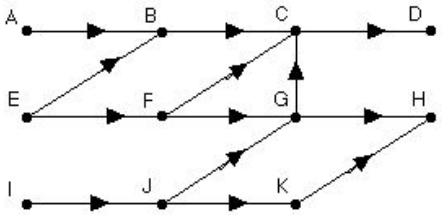
c) What about unlimited processors?




**6.** Apply the Fujii-Kasami-Ninomiya method to the following order requirement digraph.




**7.** Apply the Coffman-Graham algorithm to the following order requirement digraph.




**8.** A certain type of carton can hold a maximum of 140 pounds. Objects weighing 85, 95, 135, 55, 65, 25, 95, 35, 35, and 40 pounds are to be packed in these cartons.

a) Use first-fit on the list as given to determine how many cartons are required.

b) Use first-fit decreasing to determine how many cartons are required.

c) What is the optimal number of cartons for these weights?

**9.** A shelf system is to be constructed. It requires boards of lengths 7, 7, 6, 6, 6, 5, 5, 5, 5, 3, 3, 3, 3, 4, 4, 8, 8, 6, 6, 9, 9, 5, 5, and 6 feet. The lumber yard only sells boards with length 9 feet.

a) Determine how many boards must be bought using FF.

b) Determine how many boards must be bought using FFD.
c) What happens if the boards at the lumber yard are 10 feet long?

**10.** Make up a list processing method different from FF and FFD. Are there cases where your method might be preferable to FF and FFD?

## Computer Projects

**1.** Implement the FF algorithm for bin packing.

**2.** Implement the FFD algorithm for bin packing.

**3.** Carry out an empirical investigation of the relative efficiency of FF and FFD by generating lists of weights at random and bin sizes at random, running both algorithms on the same weights many times, and repeating this many times.

**4.** Implement the Coffman-Graham algorithm.