The Combinatorial Mathematics of Scheduling

What is the best way to organize work so that it is finished in the shortest possible time? Discoveries in mathematics and computer science reveal the values and limits of various scheduling methods

by Ronald L. Graham

Scheduling problems arise in almost all areas of human activity. The Viking mission to Mars called for coordinating the activities of more than 20,000 people. Meeting the daily manufacturing quota in an automobile plant can depend on the precise allocation of manpower and tools. Even the preparation of a multicourse dinner can present a nontrivial scheduling problem. It might appear that there are natural algorithms, or step-by-step procedures, for constructing highly efficient schedules. That, however, is not the case. Apparently logical ways of constructing schedules cannot be counted on to perform equally well in different situations. For example, in some instances increasing the number of workers on a job can actually increase the time required to meet a schedule. Some of the commonest and most intuitive scheduling procedures can give rise to unexpected and even seemingly paradoxical results.

Over the past few years several mathematical models of scheduling processes have been devised and closely analyzed in an effort to find the causes of such anomalous results. It was necessary to determine how inefficient different scheduling procedures could be and how poor performance could be eliminated or at least minimized. The research has shown that in most cases perfect results cannot be expected of efficient scheduling algorithms. Nevertheless, general guidelines have emerged for avoiding the pitfalls in scheduling problems and for finding acceptable solutions to particular problems. Here I shall describe some recent results of this kind that apply to one of the most basic scheduling models. The model deals with the behavior of the finite sets of quantities that are involved in scheduling problems. The analysis of the model demonstrates the productive interaction that often occurs between computer science and mathematics. In this instance the theory of algorithms is paired with combinatorial theory (in particular the study of finite sets) to yield important results in scheduling theory.

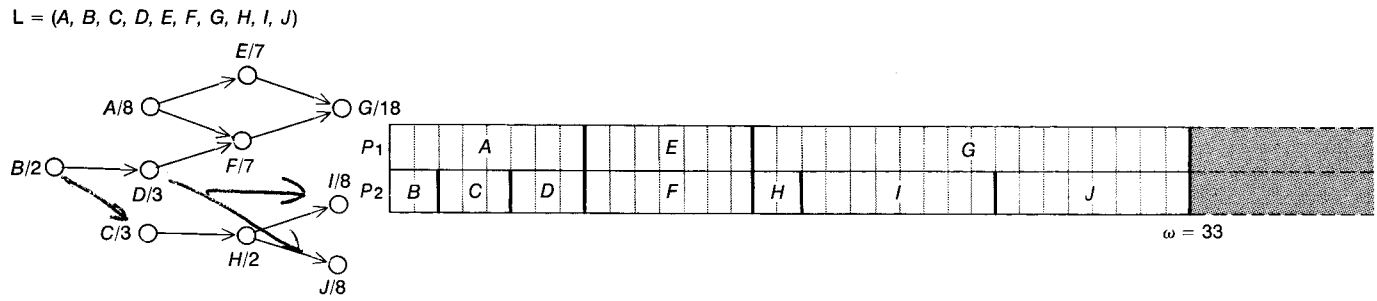
In order to provide insight into scheduling problems a model must isolate the essential parts of real scheduling situations. The basic scheduling model consists of a system of m identical processors $P_1,\ldots,P_m$ and a set of tasks $A, B,\ldots,T$ to be performed by the processors. (The processors could be workers on an assembly line or physical devices such as electronic microprocessors.) With each task T there is associated a positive number $\tau(T)$; it is the amount of time required by the processor to execute T, and so it is called the execution time of T. Although each processor is capable of executing each of the tasks, no processor can execute more than one task at a time. Moreover, when a processor begins to execute a task T, it must continue executing the task until T is completed.

A scheduling algorithm is the set of rules by which tasks are assigned to the individual processors; changing the rules changes the algorithm. In the model the order in which tasks are selected for execution depends on two factors: a collection of precedence constraints and a priority list. If task R must be completed before task S can be started, R is called a predecessor of S. This relation is called a precedence constraint and is written $R \rightarrow S$. The priority list L is an ordering of the tasks according to the preferences of the scheduler: $L = (B, C, R,\ldots)$. The list is not required to be consistent with any precedence constraints that may exist. This model is often called the priority-list scheduling model because changing the priority list is the only way to alter the schedules the model constructs.

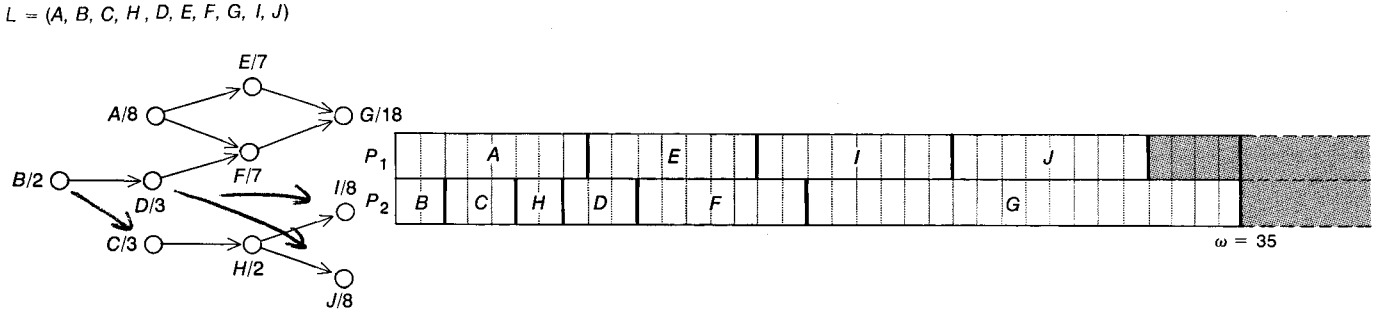
The operation of the model begins at time $t = 0$ with all the processors scanning the priority list of the set of tasks from the beginning in search of "ready tasks." A task is said to be ready for execution if all its predecessors have been completed and no processor has begun to execute it. Of course, at time $t = 0$ the ready tasks are those with no predecessors. If two or more processors are competing for a task, the task is assigned to the processor P_i that has the lowest index number i. When a processor cannot find a ready task, it stops scanning and becomes idle. The processor remains idle until a task is completed somewhere in the system. The task might be a predecessor of other tasks that would become ready on its completion. Therefore whenever a task is completed, all idle processors instantaneously begin to scan the priority list from the beginning. The schedule ends when the final task has been completed; the finishing time is denoted ω. (For the purpose of simplification the time spent in scanning the list is not included in the operating time of the model.)

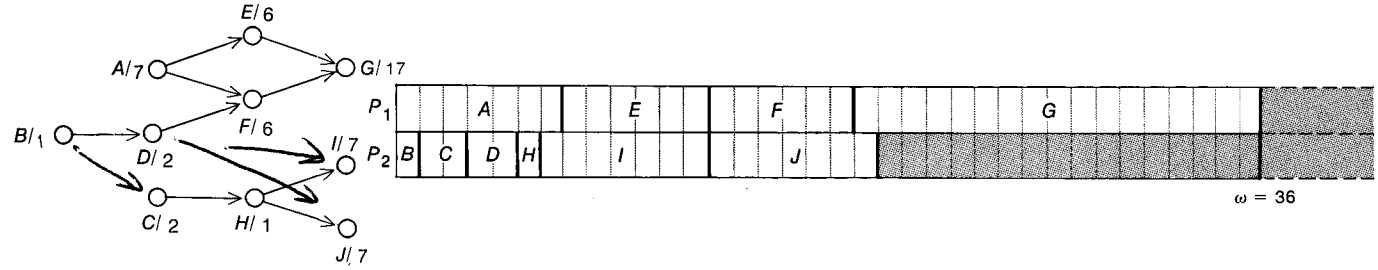
The model, which describes a typical scheduling situation, was studied to find answers to the following questions: How does the finishing time depend on the choice of the priority list? How does it depend on the precedence constraints? How does it depend on the execution times? How does it depend on the number of processors? In particular, how can the list be chosen in order to minimize the finishing time? In other words, the model serves to evaluate the scheduling algorithm by assessing its performance under different conditions, that is, with faster processors, fewer tasks and so on.

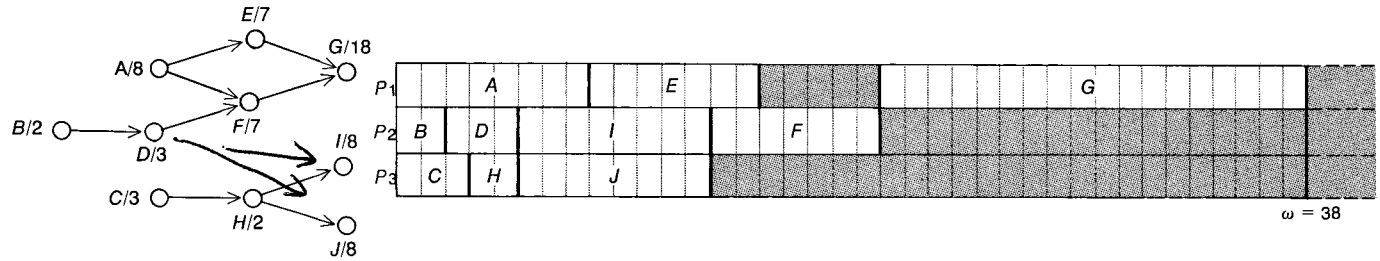
The model does not apply to all scheduling situations. For example, there is no consideration of the probabilistic aspects of scheduling in which, say, the execution times of tasks are not fixed but random, according to some distribution of probabilities. The model also regards finishing time as the only measure of algorithm performance, when actually there can be several others. On the other hand, the model does incorporate a number of features that are common to many real situations.



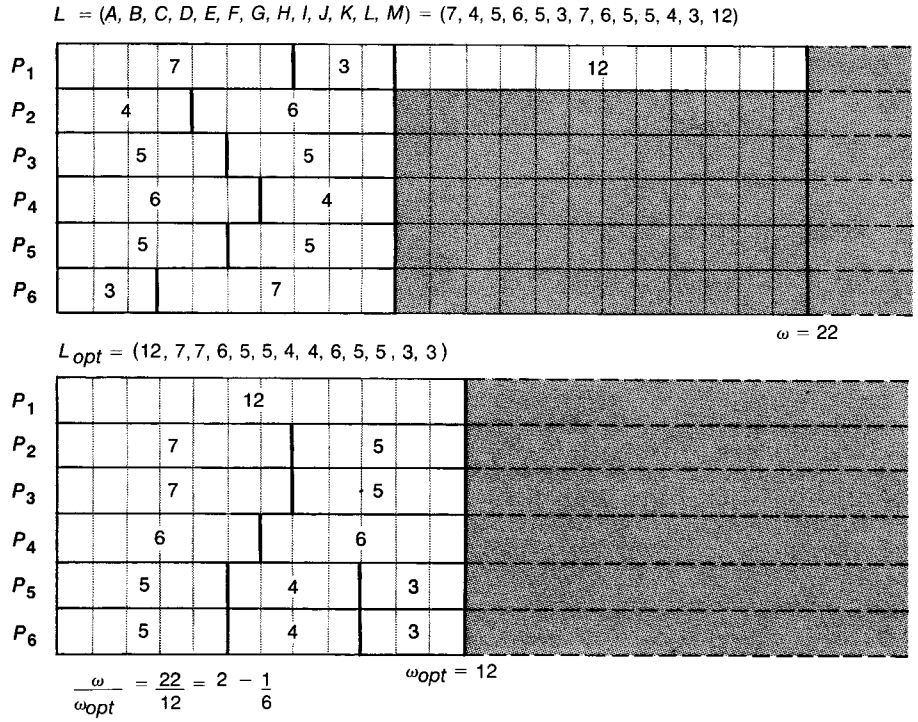
A SCHEDULING MODEL isolates the essential elements of real scheduling situations. It can be used to study the behavior of algorithms, or step-by-step procedures, for constructing efficient schedules. The basic scheduling model consists of a set of identical processors and a set of tasks to be performed by the processors according to certain rules. With each task T there is associated an execution time $\tau(T)$, equal to the amount of time required to execute T. In the example shown in this illustration there are 10 tasks, $A, B, \ldots, J$, and two processors, P_1 and P_2. The tasks to be executed are related by various precedence constraints. Task Q is said to be a predecessor of task S if Q must be completed before S can be started; this relation is called a precedence constraint and is written $Q \rightarrow S$. A scheduling algorithm is the set of rules by which tasks are assigned to processors. The assignment of tasks in the model is determined by the use of a priority list, an ordering of the tasks according to the scheduler's preferences. It is not required to be consistent with any precedence constraints that may exist. The priority list L for the example is shown at the top left; the graph at the bottom left displays the tasks (*open circles*) and the precedence constraints among the tasks (*arrows*). Each task T in the graph is labeled with its name and execution time: $T/\tau(T)$. The distribution of the tasks begins with all processors scanning the priority list from the beginning in search of ready tasks, that is, tasks whose precedence constraints have been satisfied. When a ready task is located, it is always assigned to the processor P_i with the lowest index i. In this case A is the first ready task in the list L and so it is assigned to the processor P_1. After a task has been assigned to a processor the unoccupied processors resume scanning the list. The timing diagram at the right shows the schedule, or allocation of processing time, determined by the rules of the model. The shaded areas in the diagram represent periods when a processor was idle. In the model a processor may become idle only if there is no ready task in the system, and whenever a task is completed, all idle processors begin scanning the list again. The schedule is completed when all the tasks in the set have been executed. In this example the finishing time ω is 33.



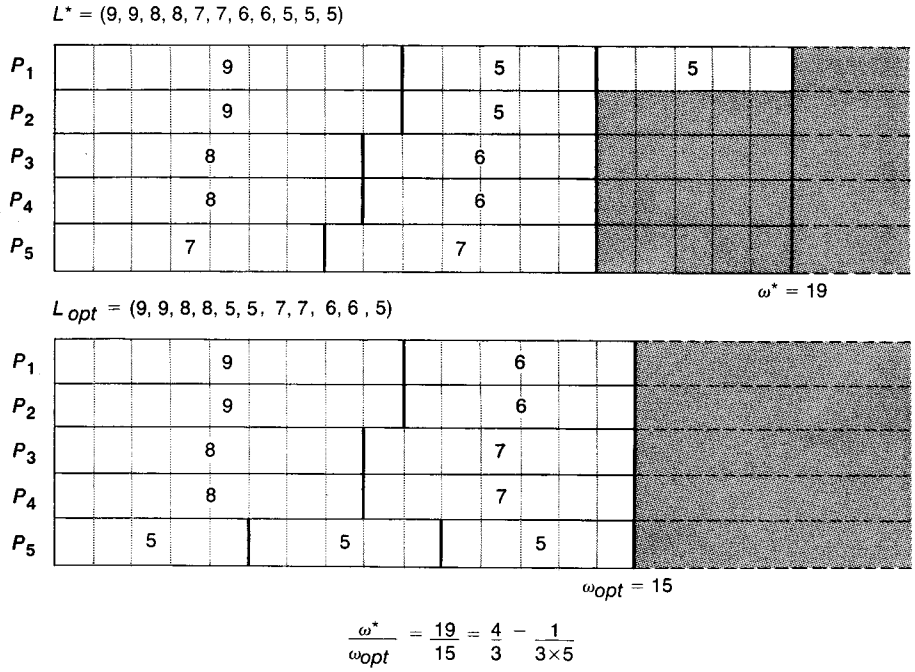




CHANGING THE PARAMETERS of the basic scheduling model can have unpredictable and often undesirable results. For example, when the priority list L of the scheduling problem in the illustration at the top of the page is rearranged (*top*), the finishing time increases to 35. A more surprising schedule results when all the execution times in the problem are decreased by 1 (*middle*): the finishing time increases to 36. Moreover, with the decreased execution times the finishing time is always at least 36, no matter what priority list is chosen. Even when third processor is added to original system (*bottom*), finishing time does not decrease but increases to 38 for any choice of list.



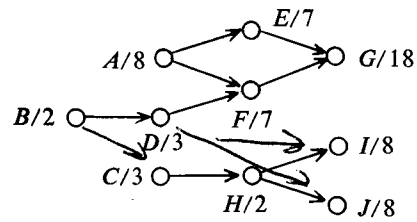
A SCHEDULE IS OPTIMUM for a particular set of tasks if its finishing time is the shortest that can be achieved by any rearrangement of the priority list for the tasks. In the scheduling problem shown here there are no precedence constraints among the tasks, and so the tasks are distributed to the processors in the order in which they appear in the priority list L. The finishing time of the resulting schedule (*top*) is $\omega = 22$; during the schedule many of the processors are idle for long periods of time. A different ordering of the tasks results in a schedule with a shorter finishing time (*bottom*). This schedule is clearly optimum because no processor is idle at any point before the finishing time $\omega_{opt} = 12$. The ratio of the two finishing times is the largest possible because it has been proved that ω/ω_{opt} is never greater than $2 - 1/m$, where m is the number of processors utilized. Tasks in illustration are identified by their execution times; since tasks are independent (there are no precedence constraints), no confusion results.



IN THE DECREASING-LIST ALGORITHM demonstrated here the more time-consuming tasks are executed as early as possible in the schedule so that there is less chance of some of the processors being idle while others are still operating near the end of the schedule. In the algorithm a priority list L^* is formed by arranging the tasks in decreasing order of execution times. When there are precedence constraints among the tasks, the algorithm can perform quite badly, but when the tasks are independent, the decreasing-list algorithm is guaranteed to give results that are well within the $2 - 1/m$ bound. In fact, for any list of tasks L, ω^*/ω_{opt} (ratio of finishing time of decreasing-list schedule for a set of tasks to optimum finishing time for the set) is never more than $4/3 - 1/3m$. Five-processor example shown here attains bound exactly.

The processors of the model could be typists working for a company and the tasks could be a set of reports to be typed. Although the company president's report might be at the head of the priority list, his report would probably depend on the results of subordinates' reports, that is, those reports would be predecessors of the president's. The processors of the model could also represent minicomputers in a multiprocessing computer system that is executing the various subroutines of a complex program. Although the model is rather simple, it does have sufficient structure to exhibit almost the full range of difficulties encountered in general combinatorial scheduling problems. An example will help to demonstrate this fact.

Assume that there are two processors, P_1 and P_2, and 10 tasks to be executed, arranged in a list: $L = (A, B, \ldots, J)$. The precedence constraints among the tasks and the execution times can be conveniently displayed in a diagram in which each open circle represents a task T, each arrow represents a precedence constraint and the tasks are labeled $T/\tau(T)$:



At time $t = 0$ the two processors begin to scan L and immediately arrive at task A, which, having no predecessors, is ready to be executed. According to the rules of the model, A is assigned to the processor P_1 because it has the lower index number. The next ready task is B, and so it is assigned to P_2. At time $t = 2$, B is completed and P_2 scans L again, finds that task C is ready and begins to execute it. The process continues until time $t = 33$, when all 10 tasks are completed [*see top illustration on preceding page*]. Since neither processor was idle at any time before $t = 33$, the list L is optimum in the sense that no matter how the tasks are arranged in a list it is not possible to finish all of them in a shorter amount of time.

The finishing time $\omega = 33$ cannot be improved by changing the list. What happens when the other parameters of the model—the execution times and the number of processors—are varied? The results are surprising [*see bottom illustration on preceding page*]. If all the execution times in the example are decreased by 1, the finishing time increases to 36. It is tempting to try to explain this odd result by postulating that the priority list L is simply an extremely poor choice for the revised set of execution times and that with a better priority list the finishing time would be reduced. A little

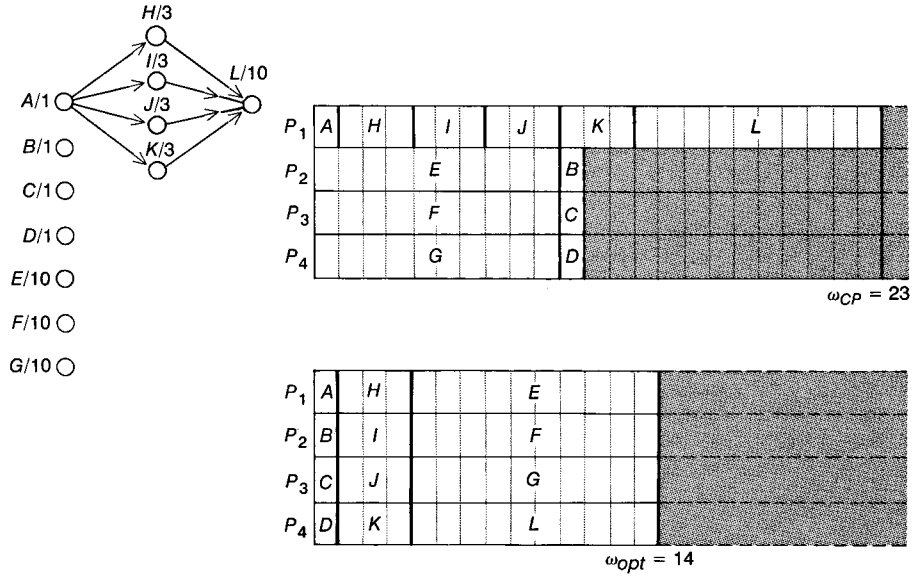
experimentation shows, however, that with the new execution times no matter how the tasks are arranged the finishing time is never less than 36.

Suppose another processor is added to the original system. With a 50 percent increase in processing capability it is not unreasonable to expect a comparable decrease in the finishing time. In this instance, however, the finishing time is 38, no matter what priority list is chosen.

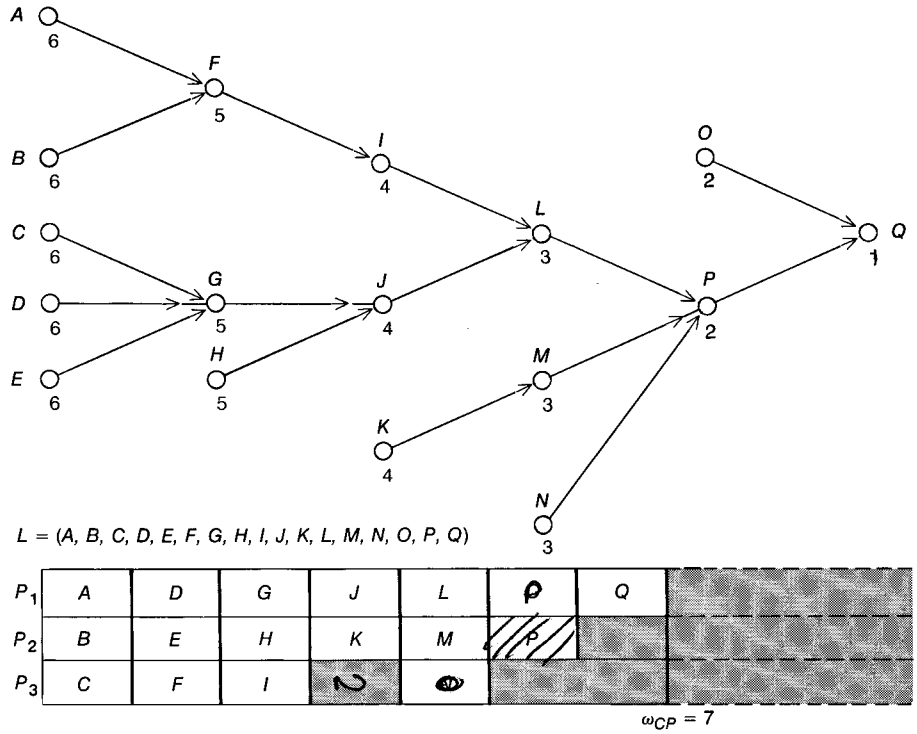
Two elements of the scheduling model are responsible for this unpredictable and undesirable behavior. First, no processor is allowed to be idle if there is a ready task in the system; second, once a processor begins to execute a task it cannot stop until the task is completed. As a result the processors are forced to begin the execution of relatively unimportant tasks (tasks that are short or that are involved in few precedence constraints), and once they have begun they cannot interrupt execution to begin more urgent tasks that subsequently become ready. Of course, the structure of the model could be altered, but the fact is that it reflects many real scheduling situations. Hence it is of interest to know just how much of an effect these aspects of the model can have on the schedules that are created. There is an unexpectedly simple answer to the question.

Imagine that a set of tasks is executed twice: on one occasion with m processors, a particular priority list, a set of precedence constraints and a set of execution times, and on another occasion with m' processors, a different list, a weaker set of precedence constraints and a set of reduced execution times. The finishing time is denoted by ω for the first schedule and by ω' for the second. The example with two processors and 10 tasks showed that ω'/ω, the ratio of the finishing times, can be greater than 1, that is, improving the values of the model parameters can cause an unavoidable increase in the finishing time. There is, however, a limit to the ratio and thus a limit on the adverse effects of the model structure. Some years ago I was able to show that ω'/ω is always less than or equal to $1 + (m - 1)/m'$, that is, $\omega'/\omega \leq 1 + (m - 1)/m'$. Moreover, there are examples where ω'/ω is equal to the upper bound $1 + (m - 1)/m'$, and so the bound cannot be improved by substituting a smaller quantity on the right-hand side of the inequality. (If a smaller bound were used, the examples would contradict the inequality.)

In instances where the schedules that are produced by different priority lists are compared, the number of processors remains the same. Therefore m is equal to m' and the inequality takes on its most elegant form: $\omega'/\omega \leq 2 - 1/m$. For example, if there are three processors, m equals 3, and so ω'/ω is never more than 5/3. This result means that for any set of tasks scheduled on three processors even utilizing the worst possible list



CRITICAL-PATH SCHEDULING is one of the commonest scheduling algorithms, although it does not always perform well. In a critical-path schedule such as the one shown at the top of the illustration tasks are distributed to processors according to the relative urgency of the tasks, that is, according to the length (the sum of the execution times) of the various processing chains each task heads in the unexecuted part of the precedence-constraint diagram. The longest chains in the diagram are called the critical paths because those chains are most likely to cause bottlenecks in the execution of the set of tasks. A task that heads a current critical path is always chosen as the next task to be executed in critical-path scheduling. At time $t = 0$ in this example there are four critical paths, each of length 14: $A \rightarrow H \rightarrow L$, $A \rightarrow I \rightarrow L$, $A \rightarrow J \rightarrow L$ and $A \rightarrow K \rightarrow L$. Therefore task A is the first task to be executed, and it is assigned to processor P_1 because P_1 has a lower index number than P_2, P_3 or P_4. In some instances critical-path schedules are extremely efficient, but for this particular problem ω_{CP}, finishing time of critical-path schedule, is almost twice ω_{opt}, finishing time of optimum schedule shown at bottom of illustration.



CRITICAL-PATH SCHEDULES ARE OPTIMUM for scheduling problems in which all the execution times are equal and the set of precedence constraints is treelike, that is, no task has more than one successor. Because all the execution times are equal (say 1) in this special case, critical-path scheduling can be accomplished by assigning to each task T a number $L(T)$ equal to the number of tasks in the longest chain headed by T. $L(T)$ is called the level of T, and the schedule is constructed by always choosing a ready task with the highest level as the next task to be executed. In the three-processor example shown here the levels are indicated in color. At time $t = 0$ the ready tasks are A, B, C, D, E, H, K, N and O. Tasks A, B, C, D and E have the highest level, 6, and A is chosen to be the first task executed. T. C. Hu of University of California at San Diego has shown that the level algorithm always creates optimum schedules in this special case, and so $\omega_{CP} = 7$ is the best possible finishing time for this particular problem.

can increase the finishing time obtained with the best possible list by no more than 66⅔ percent. The bound $2 - 1/m$ is a performance guarantee. It ensures that no matter how complicated or exotic a system of tasks, precedence constraints and execution times may be and no matter how cleverly or carelessly a list is chosen, the ratio of finishing times ω'/ω is still never greater than $2 - 1/m$.

It is obvious that within this bound a good or bad choice of a priority list can still make a great difference in the finishing time of a schedule. Common sense suggests that the best lists might be those in which the tasks with the longest execution times appear near the beginning of the list. In that case only relatively small additions to processor usage would be made at the end of the schedule and there would be less chance that some of the processors would be idle while others were still operating.

One of the commonest scheduling algorithms consists in forming a priority list L^* by placing the tasks in decreasing order of execution times and then executing the tasks according to the rules of the model. The finishing time for this decreasing-list algorithm is denoted ω^* [*see bottom illustration on page 126*]. How good is the algorithm? In other words, how close is ω^* to ω_{opt}, the optimum finishing time? When there are precedence constraints among the tasks, the algorithm can create the worst possible schedule, that is, the ratio of ω^* to ω_{opt} can attain the bound $2 - 1/m$.

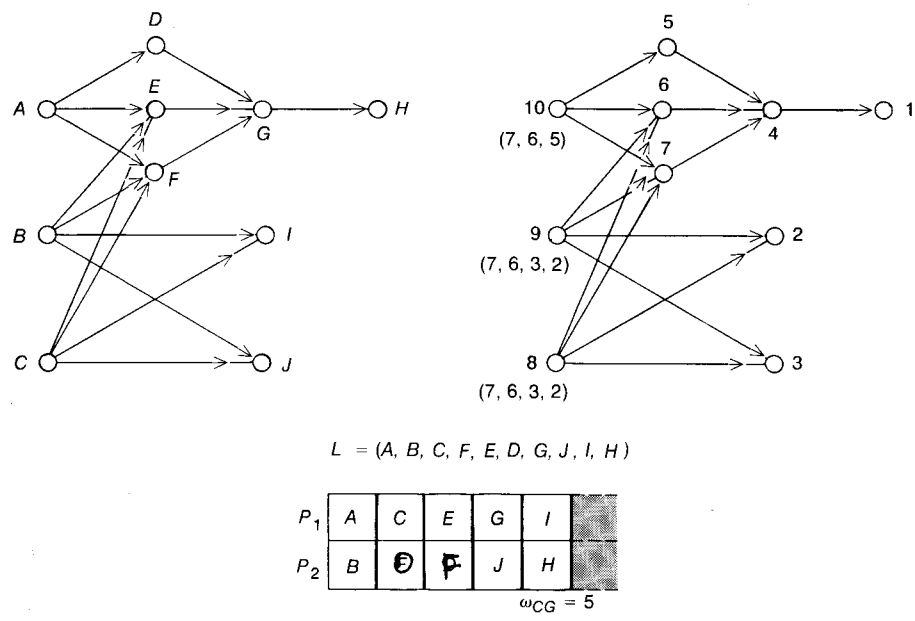
When there are no precedence constraints, the tasks are distributed to the processors in the order given in the priority list. In that case the tasks are said to be independent, and it has been shown that the decreasing-list algorithm will always give results that are well within the $2 - 1/m$ bound. In fact, for a set of independent tasks the ratio ω^*/ω_{opt} is never more than $4/3 - 1/3m$, which is substantially less than $2 - 1/m$, when m becomes large. Since there are instances where the ratio attains the bound, the bound cannot be improved. The inequality $\omega^*/\omega_{opt} \leq 4/3 - 1/3m$ guarantees that applying the decreasing-list algorithm to a set of independent tasks will never create a schedule whose finishing time is more than 33⅓ percent over the optimum.

At present there is no algorithm for scheduling sets of tasks with precedence constraints that compares with the decreasing-list algorithm for independent tasks. In fact, there is no known efficient procedure for constructing lists of non-independent tasks whose schedules for three or more processors are guaranteed to finish in $2 - \epsilon$ times the optimum finishing time for some positive number ϵ. The inequality $\omega'/\omega \leq 2 - 1/m$ shows, however, that the schedule for even the worst possible list has a finishing time that is less than twice the optimum. Hence there is clearly much progress to be made in this area.

The preceding discussion raises an obvious question: Why should so much effort be devoted to creating schedules that are good but not the best? Why not try to find the optimum schedule for a set of tasks? One way to do so would be to examine all the possible schedules for the tasks and then choose the one with the shortest finishing time. The trouble with this brute-force approach is that as the number of tasks in a set becomes large the number of possible priority lists (and thus the number of schedules) grows so explosively that there is no hope of examining even a small fraction of them. If there are n tasks in the set, the number of different lists is $n!$, or $n(n-1)(n-2)\cdots 1$, a very large number even for relatively small values of n. For example, when there are 20 tasks, even if a computer could check as many as a million schedules per second, it would take more than 70,000 years to check all 20! lists.

The number of possible lists $n!$ (and hence the number of operations and the computer time needed for checking the schedules) is an exponential function of the number of tasks n. Exponential functions increase rapidly as the value of the variable n increases. A polynomial function, however, say n^2, does not explode as rapidly as n increases. It would be practical to insist on optimum schedules if an algorithm for finding optimum schedules could be found in which the number of computational steps grows as a polynomial function of the number of tasks.

It seems highly unlikely that such an algorithm will be found. This gloomy prospect is the result of the fundamental work of Stephen A. Cook of the University of Toronto, who in 1971 introduced the concept of *NP*-complete, or nondeterministic-polynomial-time-complete, problems [see "The Efficiency of Algorithms," by Harry R. Lewis and Christos H. Papadimitriou; SCIENTIFIC AMERICAN, January]. Hundreds of problems notorious for their computational intractability are now known to belong to this class of problems. *NP*-complete problems have two important properties. First, all methods, or algorithms, currently known for finding general solutions to these problems require exponentially increasing amounts of time and thus are extremely inefficient. Sec-

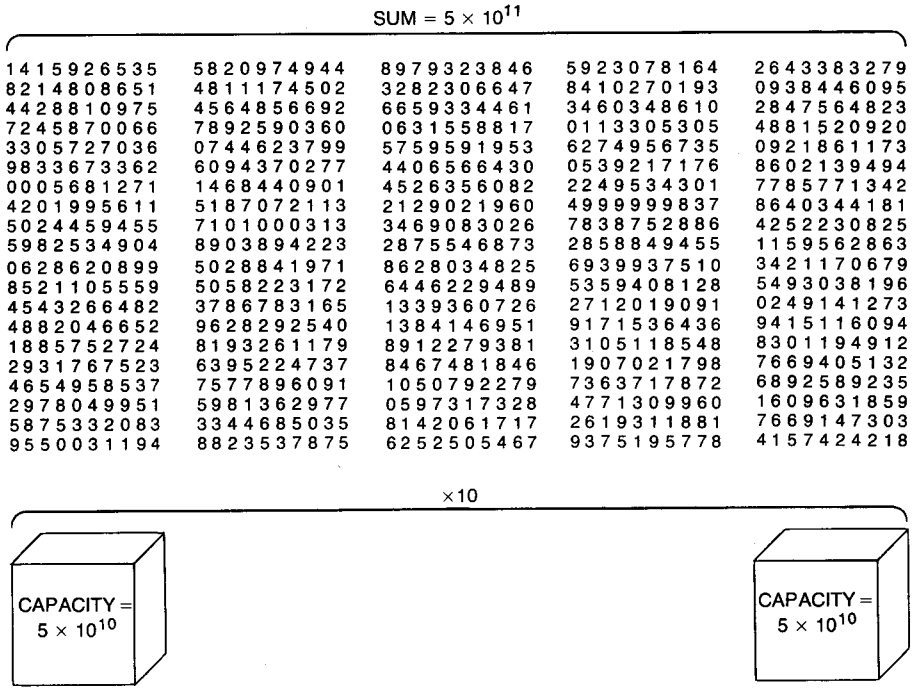


THE *CG* ALGORITHM creates optimum schedules for the special scheduling problem in which all the execution times are equal and only two processors execute the set of tasks. In this algorithm priority numbers (*color*) are assigned to each task so that tasks heading long processing chains or having many successors receive higher priority. Before the *CG* algorithm can be implemented all extraneous precedence constraints must be removed from the diagram of tasks to be executed. (For example, if $A \rightarrow B$ and $B \rightarrow C$, then $A \rightarrow C$ should be eliminated.) A reduced diagram is shown at the top left; the execution times have been omitted from the diagram because they are all equal, say, to 1. The *CG* algorithm begins by numbering the tasks as is shown at the top right. First, the number 1 is assigned to some task that has no successors, and if there are other tasks without successors, they are numbered 2, 3 and so on. In the illustration task *H*, which has no successors, was numbered 1 and tasks *I* and *J* were respectively numbered 2 and 3. Thereafter for each task for which all the successors have been numbered the decreasing sequence of all the successors' numbers is formed. The next task to be assigned a number is always the one whose sequence is first in the dictionary order (5, 3, 2 comes before 6, 1; 5, 4, 3; 5, 3, 2, 1, and so on) of the established sequences. For example, 7, 6, 3, 2, the sequence for tasks *B* and *C* in the illustration, comes before 7, 6, 5, the sequence for task *A*. Therefore the numbers 8 and 9 are assigned to *B* and *C*, and 10 is assigned to *A*. When all tasks have been numbered, priority list *L* is formed by arranging tasks in decreasing order of assigned numbers. Timing diagram at bottom of illustration shows that *CG* schedule for tasks *A, B, ..., H* is indeed optimum.

ond, if any one of the *NP*-complete problems had an efficient, or polynomial-time, solution, then all of them would. It seems highly probable, but it has not yet been proved, that the difficulty in finding efficient procedures for solving these problems is inherent in *NP*-complete problems; it appears that no such procedures can exist.

Most scheduling problems are *NP*-complete. In fact, even the comparatively simple situation in which there are no precedence constraints and only two processors presents an *NP*-complete problem. The discoveries about *NP*-completeness changed the direction of research on scheduling. Earlier efforts were directed at finding optimum, or exact, solutions to scheduling problems, but now most attention has been turned in the more fruitful direction of determining approximate solutions easily, of finding efficient methods that are guaranteed to give close to optimum results. The decreasing-list algorithm for scheduling independent tasks, guaranteed to finish within 33⅓ percent of the optimum finishing time, exemplifies this new approach.

The decreasing-list algorithm achieves its close to optimum finishing time by doing a certain amount of work. The n tasks to be executed must be sorted into a decreasing list before they are distributed to the processors. That sorting can be accomplished in an amount of time, or a number of computational steps, that is proportional to $n \log_2 n$. As n increases, $n \log_2 n$ increases only slightly faster, and so the algorithm is acceptably efficient.



BIN-PACKING, another type of scheduling problem, involves a set of items, or weights, and a collection of identical bins with a fixed weight capacity; the problem consists in packing all the weights of the set into a minimum number of bins. The difficulty in obtaining precise solutions to bin-packing problems is demonstrated in this example. The total of the 100 weights shown is 5×10^{11}. Can the weights be packed into 10 bins of capacity 5×10^{10}? The number of possible packings for this relatively small group of weights is so large that even if all the computing power in the world were applied, it is extremely unlikely that an answer to the question would be found. Most scheduling problems are similarly complex, and so many algorithms are designed to create packings or schedules guaranteed only to be reasonably close to optimum.

By doing more work it is possible to obtain schedules that are even closer to the optimum than those constructed with the decreasing-list algorithm. Consider a two-processor system. One way to obtain such superior schedules is to choose, for some integer k, the $2k$ longest tasks, construct the optimum schedule for these tasks and then schedule the remaining tasks arbitrarily. If the finishing time of this schedule is denoted ω_k, then it can be shown that $\omega_k/\omega_{opt} \leq 1 + 1/(2k + 2)$. For a set of n tasks the entire procedure can be accomplished in at most $2kn + 2^{2k}$ operations. (The term $2kn$ comes from choosing the $2k$ longest tasks and the term 2^{2k} from examining all possible schedules for the $2k$ tasks on the two processors.) Since k is a fixed number, the function $2kn + 2^{2k}$ is a polynomial function of n, not an exponential one, and it grows moderately as n increases. For example, if k is 3, then $\omega_3/\omega_{opt} \leq 9/8$ and the amount of work required is proportional to at most $6n + 64$. In general any degree of accuracy can be obtained with sufficient work. Of course, the amount of work required can rise rapidly; for example, obtaining a value of ω_k that is guaranteed to be within 2 percent of the optimum can take an amount of time proportional to $48n + 2^{48}$, which would be enough to exhaust more than a few computer budgets.

This type of behavior should not be too surprising. Since an exponentially increasing amount of time seems to be necessary for finding an optimum solution, it makes sense for the cost of approximate solutions to behave in the same way as the guaranteed accuracy of the solutions increases. What is surprising is that the exponential increase in time can be avoided; there is a method for constructing schedules for independent tasks that are guaranteed to be quite close to the optimum that requires polynomially increasing amounts of time. Oscar H. Ibarra of the University of Minnesota and Chul Kim of the University of Maryland have recently developed an efficient algorithm for constructing schedules for two processors that have a finishing time ω_k for which $\omega_k/\omega_{opt} \leq 1 + 1/k$. Implementing the algorithm requires an amount of time proportional to $n + k^4 \log n$. (When n and k are large, the value of $n + k^4 \log n$ is usually much smaller than 2^{2k}.) Sartaj K. Sahni of the University of Minnesota has extended the procedure to create efficient algorithms that can be applied to more than two processors. These procedures involve a clever combination of techniques that are beyond the scope of this discussion, but this kind of approximation may well be able to guarantee close to optimum results at a cost of reasonable amounts of time.

Although research into *NP*-completeness indicates that in general no efficient techniques will be found for constructing optimum schedules, there are many interesting special cases of scheduling problems that are not *NP*-complete and that it is possible to construct optimum schedules for in polynomial time. Much of the complexity in scheduling problems is derived from the intricate structure of precedence constraints and from the complicated relations among the execution times. Limiting one or both of these factors can result in the kinds of special cases for which optimum schedules can be found efficiently.

For example, assume that there is a scheduling situation with an arbitrary number of processors where all the processing times are equal and the set of precedence constraints is treelike, that is, every task has at most one successor. In this instance "critical path" scheduling, one of the commonest scheduling methods, will always create optimum schedules [*see top illustration on page 127*]. In critical-path scheduling the tasks are assigned to processors according to the length of the various precedence chains they head in the diagram of precedence constraints. The longest chains in the unexecuted part of the diagram are the ones that have the greatest sum of execution times; they are called

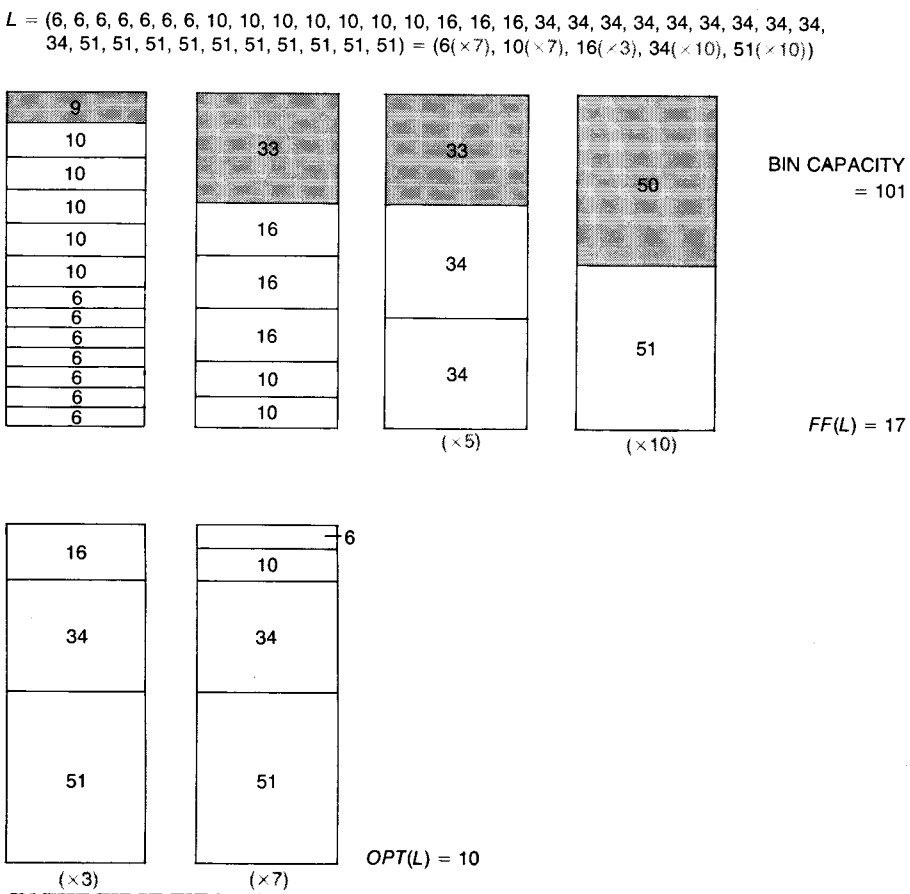
the critical paths because their tasks are the ones most likely to be the bottlenecks in the execution of the set of tasks. In critical-path scheduling a task that heads a current critical path is always chosen as the next task to be executed.

T. C. Hu of the University of California at San Diego proved in 1961 that critical-path schedules are optimum for the special case of treelike precedence constraints and equal execution times. Hu's result was one of the first in scheduling theory. Because in this special case all the execution times are equal, the critical-path scheduling consists in assigning to each task T a "level" $L(T)$ equal to the number of tasks in the longest chain headed by T [*see bottom illustration on page 127*]. Whenever a processor is available, a ready task with the highest level is assigned to it.

In another special case of scheduling no limit is placed on the structure of the precedence constraints but all the processing times must be equal and only two processors execute the set of tasks. There are now several methods for determining optimum schedules in this situation. One of them, sometimes called the *CG* algorithm, was developed in 1972 by Edward G. Coffman of the University of California at Santa Barbara and me. (*CG* stands for Coffman and Graham.) It is much in the spirit of the level algorithm for the case of treelike precedence constraints. In the *CG* algorithm, however, the order in which the tasks are executed depends on all the chains headed by each task rather than on a single longest chain, as is the case in the level algorithm.

Before the *CG* algorithm can be applied it is necessary to remove all extraneous precedence constraints from the diagram of the tasks to be executed. For example, if $A \rightarrow B$ and $B \rightarrow C$, then the precedence constraint $A \rightarrow C$ can be eliminated. This process can be accomplished in at most $n^{2.81}$ operations for a set of n tasks. The *CG* algorithm begins by numbering the tasks in the set [*see illustration on page 128*]. First the number 1 is assigned to some task that has no successors. Thereafter for each task for which all the successors have been numbered the decreasing sequence of all the successors' numbers is formed. The next task to be given a number is always the one whose sequence is first in the "dictionary order" of the established sequences. (In dictionary order 5, 3, 2 comes before 6, 1; 5, 4, 2; 5, 3, 2, 1, and so on.) After all the tasks have been numbered from 1 to n the priority list is constructed by placing the tasks in decreasing numerical order. It has been shown that the schedule constructed with this list is always optimum in the special case of two processors and equal execution times. Basically the *CG* algorithm works because it gives larger numbers and thus higher priority to tasks that either head long chains or have many successors. The numbering can be done in approximately n^2 operations, and so the algorithm is quite efficient.

The *CG* algorithm is only one of a variety of techniques that can be applied to special scheduling problems to get optimum results. Perhaps extensions of these techniques will lead to equally successful algorithms for similar problems, such as the problem of three processors with a set of tasks that have equal execution times. It should be noted, however, that even the slightly less special case of two processors with a set of tasks that have execution times of either one unit or two units has recently been shown to be *NP*-complete.

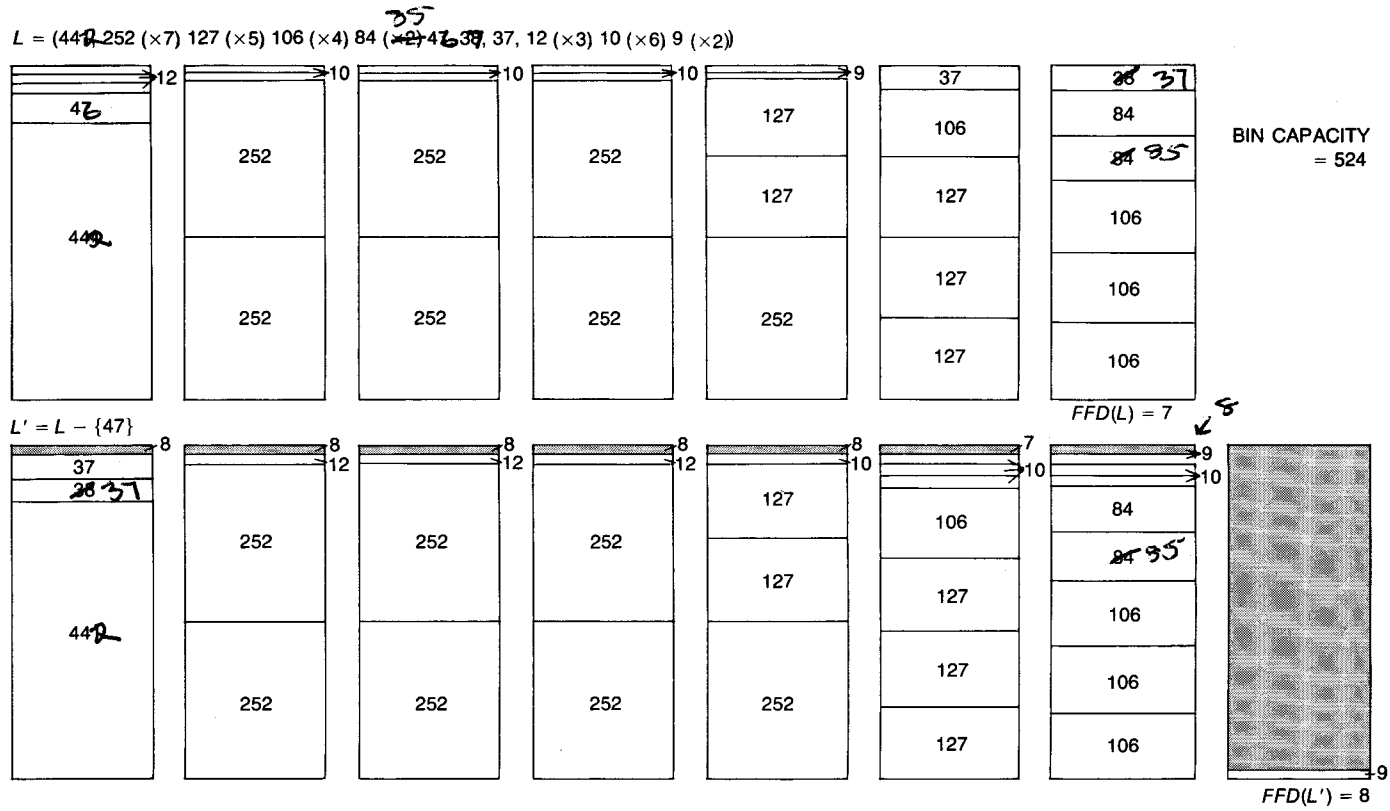


IN THE FIRST-FIT PACKING ALGORITHM weights are packed into bins $B_1, B_2, \ldots$ in the order in which the weights appear in their priority list. (If there is no priority list, the weights are arbitrarily arranged into one.) Each weight is placed in the first bin into which it fits. In other words, the weight is packed in the bin B_i with the smallest index i where the addition of the weight does not make the total of the weights in the bin exceed the fixed weight capacity. The first-fit packing of the list L shown at the top of the illustration is fairly efficient: $FF(L)$, the number of bins required, equals 17. A more efficient packing of L is shown at the bottom. This 10-bin packing is clearly optimum, since there is no wasted space in any of the bins. The example demonstrates worst possible performance of first-fit packing algorithm because it attains bound that has been established for algorithm: $FF(L) \leq (17/10)OPT(L)$, for any list L for which $OPT(L)$ is a multiple of 10. Numbers of multiple weights and bins are indicated in color.

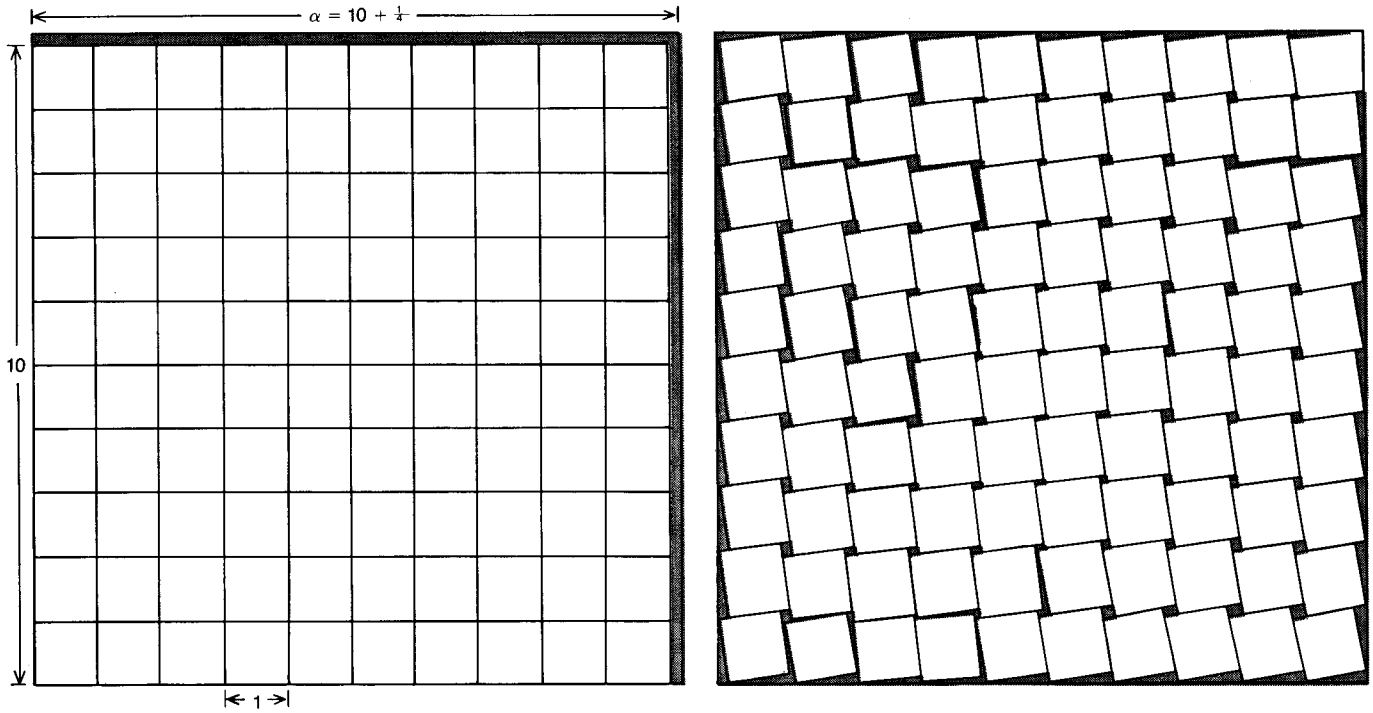
So far I have discussed only one type of scheduling problem, but scheduling problems arise in many places and in many different forms. One of the most interesting problems turns the basic scheduling model around; instead of fixing the number of processors and trying to minimize the finishing time, the problem is to try to complete the execution of a set of tasks by a fixed time with a minimum number of processors. In other words, the problem asks how few processors will suffice to execute a given set of tasks by a fixed deadline.

When the tasks are independent, this scheduling problem is stated in a different way and is called the bin-packing problem. In a model of the standard bin-packing problem there is a set of items $I_1, \ldots, I_r$; each item I_k has a weight w_k. The problem is to pack all the items into a minimum number of bins $B_1, B_2, \ldots$ so that the total weight of the items in each bin does not exceed some fixed weight W. (In the terminology of the basic scheduling model the items are tasks, the weights are execution times, the bins are processors and the fixed weight is a fixed finishing time.)

The bin-packing problem arises in a variety of guises in many practical situations. A plumber must cut a set of pipes of different lengths from a minimum number of standard-length pipes; a television network would like to schedule its commercials of varying lengths in a



FIRST-FIT DECREASING PACKING ALGORITHM improves on the first-fit packing algorithm by packing larger weights earlier. In the first-fit decreasing packing algorithm the list of weights to be packed is rearranged so that the weights are in decreasing order, and then the first-fit packing algorithm is applied to the altered list. In the first-fit decreasing packing of L (*top*) the number of bins required, $FFD(L) = 7$, is clearly optimum since each bin is filled to capacity. Like most scheduling algorithms, packing algorithms are subject to unpredictable behavior. When a weight is removed from L (*bottom*), additional bin is required for first-fit decreasing packing of smaller list.



IN TWO-DIMENSIONAL BIN-PACKING PROBLEM a list of planar regions, possibly of different sizes and shapes, must be placed without overlapping on a minimum number of identical regions. Placing sewing patterns on pieces of material is a common instance of this type of problem. Solutions to two-dimensional bin-packing problems are elusive, even when the shapes involved are highly regular. This fact is demonstrated by the following problem: How many squares with sides of unit length can be placed inside a larger square with sides of length α? If α is an integer, the problem is simple, but if α is not an integer (say α equals $N + 1/4$ for some integer N), the problem is more interesting. One obvious solution to the problem is to pack the $\alpha \times \alpha$ region by filling an $N \times N$ square with N^2 unit squares and sacrificing the uncovered area (*color*) of nearly $\alpha/2$ square units as unavoidable waste (*left*). After experimenting with other packings (*right*) it is tempting to conclude that no improvement can be made on the obvious packing, but surprisingly this is not the case. Paul Erdös, Hugh Montgomery and the author have recently proved that when α becomes large, there are packings for any $\alpha \times \alpha$ square that leave no more than $\alpha^{.634...}$ square units of uncovered area, significantly less than the $\alpha/2$ square units wasted in obvious packing. It has not yet been determined how small an area can be left uncovered when α becomes very large, although $\alpha^{.5}$ seems to be a likely possibility.

minimum number of program breaks; a paper manufacturer must furnish his customers with rolls of paper of different widths that he slices from a minimum number of standard rolls. In general bin-packing problems are extremely difficult to solve. At present the only known methods for producing optimum packings (those that require the minimum number of bins) involve examining essentially all the possible packings and then choosing the best. Like most scheduling problems, bin-packing is *NP*-complete, and so it is likely that any general algorithm for producing optimum packings will be similarly flawed. Therefore many bin-packing algorithms are designed to create packings that are reasonably close to the optimum.

In considering the bin-packing problem it is convenient to arbitrarily arrange the weights of the items into a list: $L = (w_1, w_2, \ldots, w_r)$. Since there are no precedence constraints, no confusion will arise from identifying an item with its weight, and L can be regarded as a list of the items to be packed. One obvious way to pack the weights of L is called the first-fit packing algorithm [*see illustration on page 130*]. Under the rules of this algorithm the weights are placed in bins in the order of their appearance in L: w_1 first, w_2 second and so on. When it is w_k's turn to be packed, it is put into the first bin in which it fits, that is, into the bin B_i, with the smallest index i that can accommodate the weight. (A weight w_k fits into a bin if the addition of w_k to the weights already in the bin does not make the total of the weights exceed W.)

How good a scheduling algorithm is first-fit packing? In other words, if $FF(L)$ denotes the number of bins required when the first-fit packing algorithm is applied to L and $OPT(L)$ denotes the number of bins required in an optimum packing of the weights of L, how much larger than $OPT(L)$ can $FF(L)$ be? In 1973 Jeffrey D. Ullman of Princeton University discovered that for any list L, $FF(L) \leq (17/10)OPT(L) + 2$. Ullman also showed that the coefficient 17/10 cannot be improved. If $OPT(L)$ is a multiple of 10, however, the constant 2 can be dropped from the inequality: $FF(L) \leq (17/10)OPT(L)$. It is conjectured that this simpler bound applies in all cases.

This bound shows that first-fit packing can perform rather poorly: it can require as much as 70 percent more than the optimum number of bins. Experimenting with the first-fit packing algorithm shows that the results are worse when large weights appear at the end of the list, requiring that new bins be started even though a great deal of space remains in partly filled bins. It makes sense to rearrange the list, putting all the large weights near the beginning so that the small weights at the end will be placed in odd gaps in nearly filled bins. This notion suggests a new packing procedure called the first-fit decreasing packing algorithm. The weights of L are arranged in a decreasing list and then the first-fit packing algorithm is applied. The new algorithm turns out to be quite effective [*see top illustration on preceding page*]. If $FFD(L)$ denotes the number of bins required for the first-fit decreasing packing of L, then it can be shown that $FFD(L) \leq (11/9)OPT(L) + 4$ for any list L. It has been shown that the coefficient 11/9 cannot be improved.

The expression $(11/9)OPT(L) + 4$ looks deceptively simple. Substantial difficulties are encountered in trying to prove that it is indeed the upper bound for $FFD(L)$. The only proof known at present is one devised by David S. Johnson of Bell Laboratories and it is more than 75 pages long.

When a large number of bins is required in a packing problem, the constant 4 in the inequality becomes relatively insignificant. In that case the first-fit decreasing packing algorithm is guaranteed to pack the weights of any list into no more than about 22 percent over the optimum number of bins. This result is certainly much better than the 70 percent increase over the optimum number of bins that can occur in the first-fit packing of a particularly unwieldy list.

Within their established bounds the first-fit and first-fit decreasing packing algorithms, like other scheduling procedures, are subject to unexpected behavior as the parameters of the model are varied. For example, removing a weight from a list for the first-fit decreasing packing algorithm can increase the number of bins needed. If the reduced list is denoted by L', it is still not known how large the ratio $FFD(L')/FFD(L)$ can be and whether $FFD(L)$ can increase when the largest element of L is removed. Once again it is the requirement that a weight be placed in the first available bin that is responsible for the unpredictable behavior of packing algorithms, but such behavior is not uncommon in the more complex scheduling situations of the real world.

The simple scheduling model I have described has provided a great deal of information about problems of realistic complexity. Many extensions of the basic scheduling model are possible. The model can be modified to allow interruption of tasks before completion or to allow unforced idleness. It can include the resources other than processors that are required for the execution of tasks, the probabilistic execution times of tasks, various measures of model performance and so on. By subjecting these extended models to the type of analysis I have discussed here investigators today are rapidly gaining insight into the difficult problems of scheduling.