.6 Scheduling

In this section we present two problems concerning the optimal way to schedule jobs on a single machine. In the first, the problem is to minimize the average time that a job spends in the system. In the second, the jobs have deadlines, and a job brings in a certain profit only if it is completed by its deadline: our aim is to maximize profitability. Both these problems can be solved using greedy algorithms.

6.6.1 Minimizing time in the system

A single server, such as a processor, a petrol pump, or a cashier in a bank, has n customers to serve. The service time required by each customer is known in advance: customer i will take time t_i, $1 \le i \le n$. We want to minimize the average time that a customer spends in the system. Since n, the number of customers, is fixed, this is the same as minimizing the total time spent in the system by all the customers. In other words, we want to minimize

$$T = \sum_{i=1}^{n} (\text{time in system for customer } i).$$

Suppose for example we have three customers, with $t_1 = 5$, $t_2 = 10$ and $t_3 = 3$. There are six possible orders of service.

	Order	T	
✓	1 2 3:	$5 + (5 + 10) + (5 + 10 + 3) = 38$	
	1 3 2:	$5 + (5 + 3) + (5 + 3 + 10) = 31$	
	2 1 3:	$10 + (10 + 5) + (10 + 5 + 3) = 43$	
	2 3 1:	$10 + (10 + 3) + (10 + 3 + 5) = 41$	
	3 1 2:	$3 + (3 + 5) + (3 + 5 + 10) = 29$	← optimal
	3 2 1:	$3 + (3 + 10) + (3 + 10 + 5) = 34$	

In the first case, customer 1 is served immediately, customer 2 waits while customer 1 is served and then gets his turn, and customer 3 waits while both 1 and 2 are served and then is served last; the total time passed in the system by the three customers is 38. The calculations for the other cases are similar.

In this case, the optimal schedule is obtained when the three customers are served in order of increasing service time: customer 3, who needs the least time, is served first, while customer 2, who needs the most, is served last. Serving the customers in order of decreasing service time gives the worst schedule. However, one example is not a proof that this is always so.

To add plausibility to the idea that it may be optimal to schedule the customers in order of increasing service time, imagine a greedy algorithm that builds the optimal schedule item by item. Suppose that after scheduling service for customers $i_1, i_2, \ldots, i_m$ we add customer j. The increase in T at this stage is equal to the sum of the service times for customers i_1 to i_m (for this is how long customer j must wait before receiving service), plus t_j, the time needed to serve customer j. To minimize this, since a greedy algorithm never undoes its previous decisions, all we can do is to minimize t_j. Our greedy algorithm is therefore simple: at each step, add to the end of the schedule the customer requiring the least service among those who remain.

✓Theorem 6.6.1 *This greedy algorithm is optimal.*

Proof Let $P = p_1 p_2 \cdots p_n$ be any permutation of the integers from 1 to n, and let $s_i = t_{p_i}$. If customers are served in the order P, then the service time required by the i-th customer to be served is s_i, and the total time passed in the system by all the customers is

$$\begin{aligned} T(P) &= s_1 + (s_1 + s_2) + (s_1 + s_2 + s_3) + \cdots \\ &= ns_1 + (n-1)s_2 + (n-2)s_3 + \cdots \\ &= \sum_{k=1}^{n} (n-k+1)s_k. \end{aligned}$$

Suppose now that P does not arrange the customers in order of increasing service time. Then we can find two integers a and b with $a < b$ and $s_a > s_b$. In other words, the a-th customer is served before the b-th customer even though the former needs more service time than the latter; see Figure 6.7. If we exchange the positions of these two customers, we obtain a new order of service P', which is simply P with the integers p_a and p_b interchanged. The total time passed in the system by all the customers if schedule P' is used is

$$T(P') = (n-a+1)s_b + (n-b+1)s_a + \sum_{\substack{k=1 \\ k \neq a,b}}^{n} (n-k+1)s_k.$$

The new schedule is preferable to the old because

$$\begin{aligned} T(P) - T(P') &= (n-a+1)(s_a - s_b) + (n-b+1)(s_b - s_a) \\ &= (b-a)(s_a - s_b) > 0. \end{aligned}$$

✓

The same result can be obtained less formally from Figure 6.7. Comparing schedules P and P', we see that the first $a-1$ customers leave the system at exactly the same time in both schedules. The same is true of the last $n-b$ customers. Customer a now leaves when customer b used to, while customer b leaves earlier than customer a used to, because $s_b < s_a$. Finally those customers served in positions $a+1$ to $b-1$ also leave the system earlier, for the same reason. Overall, P' is therefore better than P.

✓Thus we can improve any schedule in which a customer is served before someone else who requires less service. The only schedules that remain are those obtained by putting the customers in order of nondecreasing service time. All such schedules are clearly equivalent, and therefore all optimal. ■

Implementing the algorithm is so straightforward ✓that we omit the details. In essence all that is necessary is to sort the customers into order of nondecreasing service time, which takes a time in $O(n \log n)$. The problem can be generalized to a system with s servers, as can the algorithm: see Problem 6.20.

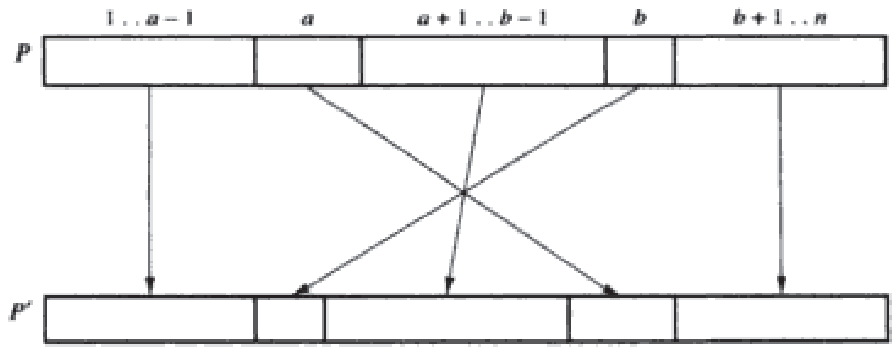
✓


Figure 6.7. Exchanging two customers ✓

6.6.2 Scheduling with deadlines

We have a set of n jobs to execute, each of which takes unit time. At any time $T = 1, 2, \ldots$ we can execute exactly one job. Job i earns us a profit $g_i > 0$ if and only if it is executed no later than time d_i.

For example, with $n = 4$ and the following values:

i	1	2	3	4
g_i	50	10	15	30
d_i	2	1	2	1

the schedules to consider and the corresponding profits are

Sequence	Profit	Sequence	Profit	
1	50	2, 1	60	
2	10	2, 3	25	
3	15	3, 1	65	
4	30	4, 1	80	← optimum
1, 3	65	4, 3	45	

The sequence 3,2 for instance is not considered because job 2 would be executed at time $t = 2$, after its deadline $d_2 = 1$. To maximize our profit in this example, we should execute the schedule 4,1.

A set of jobs is *feasible* if there exists at least one sequence (also called feasible) that allows all the jobs in the set to be executed no later than their respective deadlines. An obvious greedy algorithm consists of constructing the schedule step by step, adding at each step the job with the highest value of g_i among those not yet considered, provided that the chosen set of jobs remains feasible.

In the preceding example we first choose job 1. Next, we choose job 4; the set $\{1,4\}$ is feasible because it can be executed in the order 4,1. Next we try the set $\{1,3,4\}$, which turns out not to be feasible; job 3 is therefore rejected. Finally we try $\{1,2,4\}$, which is also infeasible, so job 2 is also rejected. Our solution—optimal in this case—is therefore to execute the set of jobs $\{1,4\}$, which can only be done in the order 4,1. It remains to be proved that this algorithm always finds an optimal schedule and to find an efficient way of implementing it.