Chapter 4

Greedy Algorithms

In *Wall Street*, that iconic movie of the 1980s, Michael Douglas gets up in front of a room full of stockholders and proclaims, "Greed . . . is good. Greed is right. Greed works." In this chapter, we'll be taking a much more understated perspective as we investigate the pros and cons of short-sighted greed in the design of algorithms. Indeed, our aim is to approach a number of different computational problems with a recurring set of questions: Is greed good? Does greed work?

It is hard, if not impossible, to define precisely what is meant by a *greedy algorithm*. An algorithm is greedy if it builds up a solution in small steps, choosing a decision at each step myopically to optimize some underlying criterion. One can often design many different greedy algorithms for the same problem, each one locally, incrementally optimizing some different measure on its way to a solution.

When a greedy algorithm succeeds in solving a nontrivial problem optimally, it typically implies something interesting and useful about the structure of the problem itself; there is a local decision rule that one can use to construct optimal solutions. And as we'll see later, in Chapter 11, the same is true of problems in which a greedy algorithm can produce a solution that is guaranteed to be *close* to optimal, even if it does not achieve the precise optimum. These are the kinds of issues we'll be dealing with in this chapter. It's easy to invent greedy algorithms for almost any problem; finding cases in which they work well, and proving that they work well, is the interesting challenge.

The first two sections of this chapter will develop two basic methods for proving that a greedy algorithm produces an optimal solution to a problem. One can view the first approach as establishing that *the greedy algorithm stays ahead*. By this we mean that if one measures the greedy algorithm's progress in a step-by-step fashion, one sees that it does better than any other algorithm at each step; it then follows that it produces an optimal solution. The second approach is known as an *exchange argument*, and it is more general: one considers any possible solution to the problem and gradually transforms it into the solution found by the greedy algorithm without hurting its quality. Again, it will follow that the greedy algorithm must have found a solution that is at least as good as any other solution.

Following our introduction of these two styles of analysis, we focus on several of the most well-known applications of greedy algorithms: *shortest paths in a graph*, the *Minimum Spanning Tree Problem*, and the construction of *Huffman codes* for performing data compression. They each provide nice examples of our analysis techniques. We also explore an interesting relationship between minimum spanning trees and the long-studied problem of *clustering*. Finally, we consider a more complex application, the *Minimum-Cost Arborescence Problem*, which further extends our notion of what a greedy algorithm is.

4.1 Interval Scheduling: The Greedy Algorithm Stays Ahead

Let's recall the Interval Scheduling Problem, which was the first of the five representative problems we considered in Chapter 1. We have a set of requests $\{1, 2, \ldots, n\}$; the i^{th} request corresponds to an interval of time starting at $s(i)$ and finishing at $f(i)$. (Note that we are slightly changing the notation from Section 1.2, where we used s_i rather than $s(i)$ and f_i rather than $f(i)$. This change of notation will make things easier to talk about in the proofs.) We'll say that a subset of the requests is *compatible* if no two of them overlap in time, and our goal is to accept as large a compatible subset as possible. Compatible sets of maximum size will be called *optimal*.

Designing a Greedy Algorithm

Using the Interval Scheduling Problem, we can make our discussion of greedy algorithms much more concrete. The basic idea in a greedy algorithm for interval scheduling is to use a simple rule to select a first request i_1. Once a request i_1 is accepted, we reject all requests that are not compatible with i_1. We then select the next request i_2 to be accepted, and again reject all requests that are not compatible with i_2. We continue in this fashion until we run out of requests. The challenge in designing a good greedy algorithm is in deciding which simple rule to use for the selection—and there are many natural rules for this problem that do not give good solutions.

Let's try to think of some of the most natural rules and see how they work.

- The most obvious rule might be to always select the available request that starts earliest—that is, the one with minimal start time $s(i)$. This way our resource starts being used as quickly as possible.

  This method does not yield an optimal solution. If the earliest request i is for a very long interval, then by accepting request i we may have to reject a lot of requests for shorter time intervals. Since our goal is to satisfy as many requests as possible, we will end up with a suboptimal solution. In a really bad case—say, when the finish time $f(i)$ is the maximum among all requests—the accepted request i keeps our resource occupied for the whole time. In this case our greedy method would accept a single request, while the optimal solution could accept many. Such a situation is depicted in Figure 4.1(a).
- This might suggest that we should start out by accepting the request that requires the smallest interval of time—namely, the request for which $f(i) - s(i)$ is as small as possible. As it turns out, this is a somewhat better rule than the previous one, but it still can produce a suboptimal schedule. For example, in Figure 4.1(b), accepting the short interval in the middle would prevent us from accepting the other two, which form an optimal solution.

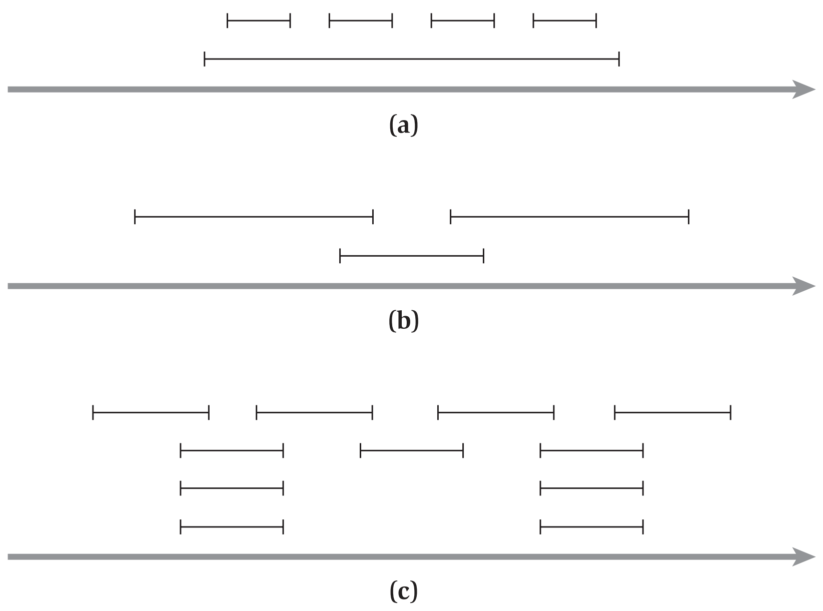

Figure 4.1 Some instances of the Interval Scheduling Problem on which natural greedy algorithms fail to find the optimal solution. In (a), it does not work to select the interval that starts earliest; in (b), it does not work to select the shortest interval; and in (c), it does not work to select the interval with the fewest conflicts.

- In the previous greedy rule, our problem was that the second request competes with both the first and the third—that is, accepting this request made us reject two other requests. We could design a greedy algorithm that is based on this idea: for each request, we count the number of other requests that are not compatible, and accept the request that has the fewest number of noncompatible requests. (In other words, we select the interval with the fewest "conflicts.") This greedy choice would lead to the optimum solution in the previous example. In fact, it is quite a bit harder to design a bad example for this rule; but it can be done, and we've drawn an example in Figure 4.1(c). The unique optimal solution in this example is to accept the four requests in the top row. The greedy method suggested here accepts the middle request in the second row and thereby ensures a solution of size no greater than three.

A greedy rule that does lead to the optimal solution is based on a fourth idea: we should accept first the request that finishes first, that is, the request i for which $f(i)$ is as small as possible. This is also quite a natural idea: we ensure that our resource becomes free as soon as possible while still satisfying one request. In this way we can maximize the time left to satisfy other requests.

Let us state the algorithm a bit more formally. We will use R to denote the set of requests that we have neither accepted nor rejected yet, and use A to denote the set of accepted requests. For an example of how the algorithm runs, see Figure 4.2.

```
Initially let R be the set of all requests, and let A be empty
While R is not yet empty
  Choose a request i ∈ R that has the smallest finishing time
  Add request i to A
  Delete all requests from R that are not compatible with request i
EndWhile
Return the set A as the set of accepted requests
```

Analyzing the Algorithm

While this greedy method is quite natural, it is certainly not obvious that it returns an optimal set of intervals. Indeed, it would only be sensible to reserve judgment on its optimality: the ideas that led to the previous nonoptimal versions of the greedy method also seemed promising at first.

As a start, we can immediately declare that the intervals in the set A returned by the algorithm are all compatible.

(4.1) *A is a compatible set of requests.*

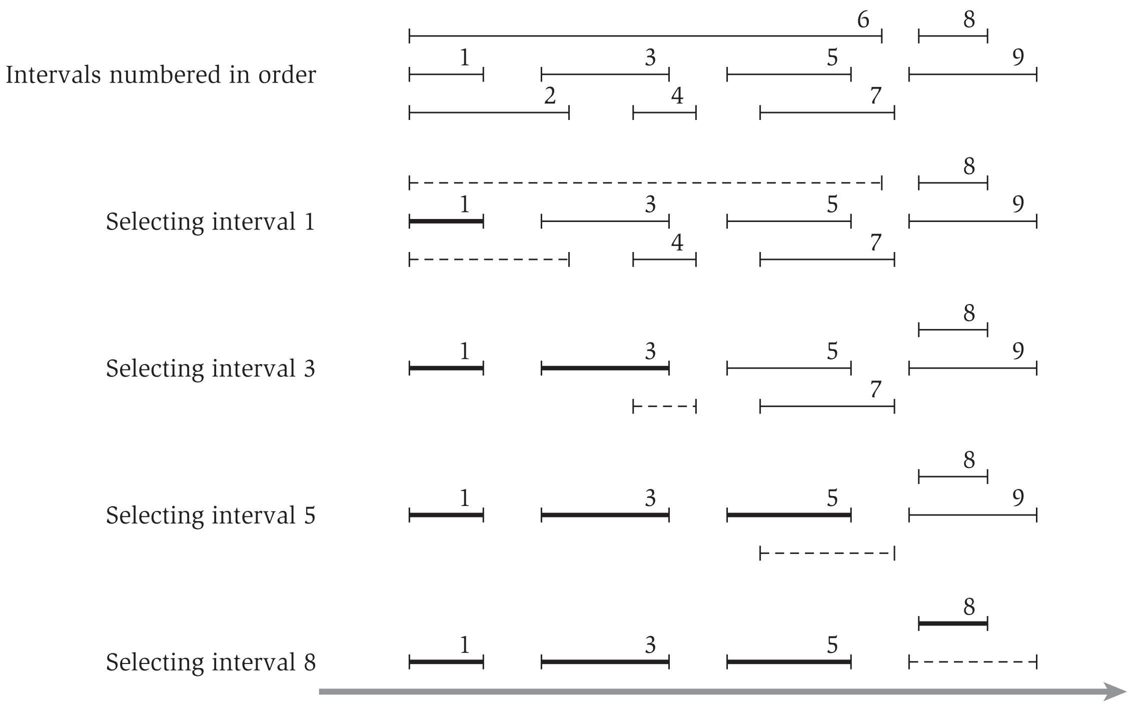

Figure 4.2 Sample run of the Interval Scheduling Algorithm. At each step the selected intervals are darker lines, and the intervals deleted at the corresponding step are indicated with dashed lines.

What we need to show is that this solution is optimal. So, for purposes of comparison, let $\mathcal{O}$ be an optimal set of intervals. Ideally one might want to show that $A = \mathcal{O}$, but this is too much to ask: there may be many optimal solutions, and at best A is equal to a single one of them. So instead we will simply show that $|A| = |\mathcal{O}|$, that is, that A contains the same number of intervals as $\mathcal{O}$ and hence is also an optimal solution.

The idea underlying the proof, as we suggested initially, will be to find a sense in which our greedy algorithm "stays ahead" of this solution $\mathcal{O}$. We will compare the partial solutions that the greedy algorithm constructs to initial segments of the solution $\mathcal{O}$, and show that the greedy algorithm is doing better in a step-by-step fashion.

We introduce some notation to help with this proof. Let $i_1, \ldots, i_k$ be the set of requests in A in the order they were added to A. Note that $|A| = k$. Similarly, let the set of requests in $\mathcal{O}$ be denoted by $j_1, \ldots, j_m$. Our goal is to prove that $k = m$. Assume that the requests in $\mathcal{O}$ are also ordered in the natural left-to-right order of the corresponding intervals, that is, in the order of the start and finish points. Note that the requests in $\mathcal{O}$ are compatible, which implies that the start points have the same order as the finish points.

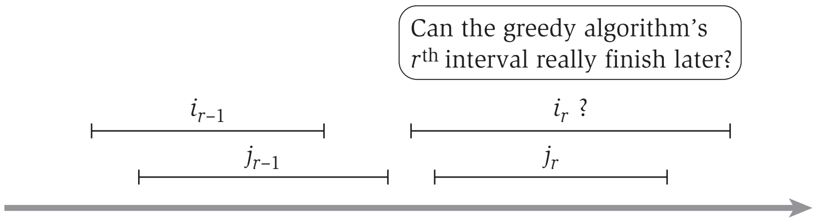
Can the greedy algorithm's r^{th} interval really finish later?
i_{r-1}
i_r ?
j_{r-1}
j_r

Figure 4.3 The inductive step in the proof that the greedy algorithm stays ahead.

Our intuition for the greedy method came from wanting our resource to become free again as soon as possible after satisfying the first request. And indeed, our greedy rule guarantees that $f(i_1) \leq f(j_1)$. This is the sense in which we want to show that our greedy rule "stays ahead"—that each of its intervals finishes at least as soon as the corresponding interval in the set $\mathcal{O}$. Thus we now prove that for each $r \geq 1$, the r^{th} accepted request in the algorithm's schedule finishes no later than the r^{th} request in the optimal schedule.

(4.2) *For all indices $r \leq k$ we have $f(i_r) \leq f(j_r)$.*

Proof. We will prove this statement by induction. For $r = 1$ the statement is clearly true: the algorithm starts by selecting the request i_1 with minimum finish time.

Now let $r > 1$. We will assume as our induction hypothesis that the statement is true for $r - 1$, and we will try to prove it for r. As shown in Figure 4.3, the induction hypothesis lets us assume that $f(i_{r-1}) \leq f(j_{r-1})$. In order for the algorithm's r^{th} interval not to finish earlier as well, it would need to "fall behind" as shown. But there's a simple reason why this could not happen: rather than choose a later-finishing interval, the greedy algorithm always has the option (at worst) of choosing j_r and thus fulfilling the induction step.

We can make this argument precise as follows. We know (since $\mathcal{O}$ consists of compatible intervals) that $f(j_{r-1}) \leq s(j_r)$. Combining this with the induction hypothesis $f(i_{r-1}) \leq f(j_{r-1})$, we get $f(i_{r-1}) \leq s(j_r)$. Thus the interval j_r is in the set R of available intervals at the time when the greedy algorithm selects i_r. The greedy algorithm selects the available interval with *smallest* finish time; since interval j_r is one of these available intervals, we have $f(i_r) \leq f(j_r)$. This completes the induction step. ■

Thus we have formalized the sense in which the greedy algorithm is remaining ahead of $\mathcal{O}$: for each r, the r^{th} interval it selects finishes at least as soon as the r^{th} interval in $\mathcal{O}$. We now see why this implies the optimality of the greedy algorithm's set A.

(4.3) *The greedy algorithm returns an optimal set A.*

Proof. We will prove the statement by contradiction. If A is not optimal, then an optimal set $\mathcal{O}$ must have more requests, that is, we must have $m > k$. Applying (4.2) with $r = k$, we get that $f(i_k) \leq f(j_k)$. Since $m > k$, there is a request j_{k+1} in $\mathcal{O}$. This request starts after request j_k ends, and hence after i_k ends. So after deleting all requests that are not compatible with requests $i_1, \ldots, i_k$, the set of possible requests R still contains j_{k+1}. But the greedy algorithm stops with request i_k, and it is only supposed to stop when R is empty—a contradiction. ■

Implementation and Running Time We can make our algorithm run in time $O(n \log n)$ as follows. We begin by sorting the n requests in order of finishing time and labeling them in this order; that is, we will assume that $f(i) \leq f(j)$ when $i < j$. This takes time $O(n \log n)$. In an additional $O(n)$ time, we construct an array $S[1 \ldots n]$ with the property that $S[i]$ contains the value $s(i)$.

We now select requests by processing the intervals in order of increasing $f(i)$. We always select the first interval; we then iterate through the intervals in order until reaching the first interval j for which $s(j) \geq f(1)$; we then select this one as well. More generally, if the most recent interval we've selected ends at time f, we continue iterating through subsequent intervals until we reach the first j for which $s(j) \geq f$. In this way, we implement the greedy algorithm analyzed above in one pass through the intervals, spending constant time per interval. Thus this part of the algorithm takes time $O(n)$.

Extensions

The Interval Scheduling Problem we considered here is a quite simple scheduling problem. There are many further complications that could arise in practical settings. The following point out issues that we will see later in the book in various forms.

- In defining the problem, we assumed that all requests were known to the scheduling algorithm when it was choosing the compatible subset. It would also be natural, of course, to think about the version of the problem in which the scheduler needs to make decisions about accepting or rejecting certain requests before knowing about the full set of requests. Customers (requestors) may well be impatient, and they may give up and leave if the scheduler waits too long to gather information about all other requests. An active area of research is concerned with such *on-line* algorithms, which must make decisions as time proceeds, without knowledge of future input.

- Our goal was to maximize the number of satisfied requests. But we could picture a situation in which each request has a different value to us. For example, each request i could also have a value v_i (the amount gained by satisfying request i), and the goal would be to maximize our income: the sum of the values of all satisfied requests. This leads to the *Weighted Interval Scheduling Problem*, the second of the representative problems we described in Chapter 1.

There are many other variants and combinations that can arise. We now discuss one of these further variants in more detail, since it forms another case in which a greedy algorithm can be used to produce an optimal solution.

A Related Problem: Scheduling All Intervals

The Problem In the Interval Scheduling Problem, there is a single resource and many requests in the form of time intervals, so we must choose which requests to accept and which to reject. A related problem arises if we have many identical resources available and we wish to schedule *all* the requests using as few resources as possible. Because the goal here is to partition all intervals across multiple resources, we will refer to this as the *Interval Partitioning* Problem.[1]

For example, suppose that each request corresponds to a lecture that needs to be scheduled in a classroom for a particular interval of time. We wish to satisfy all these requests, using as few classrooms as possible. The classrooms at our disposal are thus the multiple resources, and the basic constraint is that any two lectures that overlap in time must be scheduled in different classrooms. Equivalently, the interval requests could be jobs that need to be processed for a specific period of time, and the resources are machines capable of handling these jobs. Much later in the book, in Chapter 10, we will see a different application of this problem in which the intervals are routing requests that need to be allocated bandwidth on a fiber-optic cable.

As an illustration of the problem, consider the sample instance in Figure 4.4(a). The requests in this example can all be scheduled using three resources; this is indicated in Figure 4.4(b), where the requests are rearranged into three rows, each containing a set of nonoverlapping intervals. In general, one can imagine a solution using k resources as a rearrangement of the requests into k rows of nonoverlapping intervals: the first row contains all the intervals

[1] The problem is also referred to as the *Interval Coloring Problem*; the terminology arises from thinking of the different resources as having distinct colors—all the intervals assigned to a particular resource are given the corresponding color.

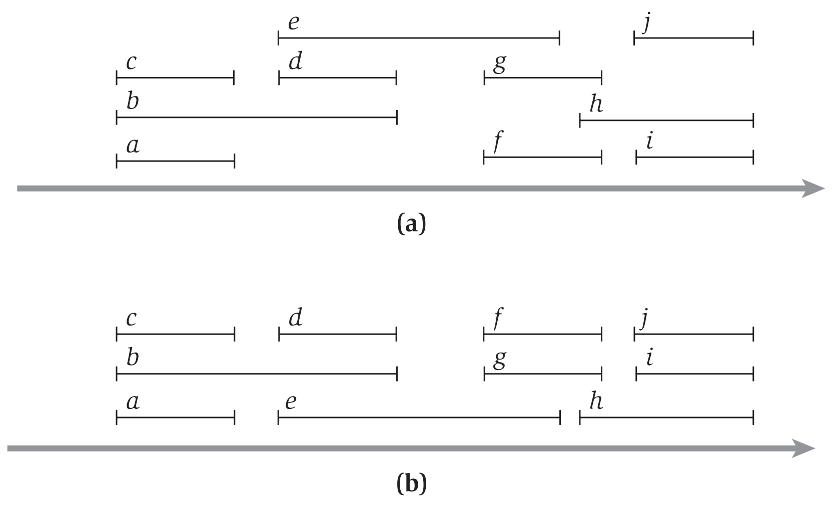

Figure 4.4 (a) An instance of the Interval Partitioning Problem with ten intervals (*a* through *j*). (b) A solution in which all intervals are scheduled using three resources: each row represents a set of intervals that can all be scheduled on a single resource.

assigned to the first resource, the second row contains all those assigned to the second resource, and so forth.

Now, is there any hope of using just two resources in this sample instance? Clearly the answer is no. We need at least three resources since, for example, intervals a, b, and c all pass over a common point on the time-line, and hence they all need to be scheduled on different resources. In fact, one can make this last argument in general for any instance of Interval Partitioning. Suppose we define the *depth* of a set of intervals to be the maximum number that pass over any single point on the time-line. Then we claim

(4.4) *In any instance of Interval Partitioning, the number of resources needed is at least the depth of the set of intervals.*

Proof. Suppose a set of intervals has depth d, and let $I_1, \ldots, I_d$ all pass over a common point on the time-line. Then each of these intervals must be scheduled on a different resource, so the whole instance needs at least d resources. ■

We now consider two questions, which turn out to be closely related. First, can we design an efficient algorithm that schedules all intervals using the minimum possible number of resources? Second, is there always a schedule using a number of resources that is *equal* to the depth? In effect, a positive answer to this second question would say that the only obstacles to partitioning intervals are purely local—a set of intervals all piled over the same point. It's not immediately clear that there couldn't exist other, “long-range” obstacles that push the number of required resources even higher.

We now design a simple greedy algorithm that schedules all intervals using a number of resources equal to the depth. This immediately implies the optimality of the algorithm: in view of (4.4), no solution could use a number of resources that is smaller than the depth. The analysis of our algorithm will therefore illustrate another general approach to proving optimality: one finds a simple, "structural" bound asserting that every possible solution must have at least a certain value, and then one shows that the algorithm under consideration always achieves this bound.

Designing the Algorithm Let d be the depth of the set of intervals; we show how to assign a *label* to each interval, where the labels come from the set of numbers $\{1, 2, \ldots, d\}$, and the assignment has the property that overlapping intervals are labeled with different numbers. This gives the desired solution, since we can interpret each number as the name of a resource, and the label of each interval as the name of the resource to which it is assigned.

The algorithm we use for this is a simple one-pass greedy strategy that orders intervals by their starting times. We go through the intervals in this order, and try to assign to each interval we encounter a label that hasn't already been assigned to any previous interval that overlaps it. Specifically, we have the following description.

```
Sort the intervals by their start times, breaking ties arbitrarily
Let $I_1, I_2, \ldots, I_n$ denote the intervals in this order
For $j = 1, 2, 3, \ldots, n$
  For each interval $I_i$ that precedes $I_j$ in sorted order and overlaps it
     Exclude the label of $I_i$ from consideration for $I_j$
  Endfor
  If there is any label from $\{1, 2, \ldots, d\}$ that has not been excluded then
    Assign a nonexcluded label to $I_j$
  Else
    Leave $I_j$ unlabeled
  Endif
Endfor
```

Analyzing the Algorithm We claim the following.

(4.5) *If we use the greedy algorithm above, every interval will be assigned a label, and no two overlapping intervals will receive the same label.*

Proof. First let's argue that no interval ends up unlabeled. Consider one of the intervals I_j, and suppose there are t intervals earlier in the sorted order that overlap it. These t intervals, together with I_j, form a set of $t + 1$ intervals that all pass over a common point on the time-line (namely, the start time of

I_j), and so $t + 1 \leq d$. Thus $t \leq d - 1$. It follows that at least one of the d labels is not excluded by this set of t intervals, and so there is a label that can be assigned to I_j.

Next we claim that no two overlapping intervals are assigned the same label. Indeed, consider any two intervals I and I' that overlap, and suppose I precedes I' in the sorted order. Then when I' is considered by the algorithm, I is in the set of intervals whose labels are excluded from consideration; consequently, the algorithm will not assign to I' the label that it used for I. ■

The algorithm and its analysis are very simple. Essentially, if you have d labels at your disposal, then as you sweep through the intervals from left to right, assigning an available label to each interval you encounter, you can never reach a point where all the labels are currently in use.

Since our algorithm is using d labels, we can use (4.4) to conclude that it is, in fact, always using the minimum possible number of labels. We sum this up as follows.

(4.6) *The greedy algorithm above schedules every interval on a resource, using a number of resources equal to the depth of the set of intervals. This is the optimal number of resources needed.*

4.2 Scheduling to Minimize Lateness: An Exchange Argument

We now discuss a scheduling problem related to the one with which we began the chapter. Despite the similarities in the problem formulation and in the greedy algorithm to solve it, the proof that this algorithm is optimal will require a more sophisticated kind of analysis.

The Problem

Consider again a situation in which we have a single resource and a set of n requests to use the resource for an interval of time. Assume that the resource is available starting at time s. In contrast to the previous problem, however, each request is now more flexible. Instead of a start time and finish time, the request i has a deadline d_i, and it requires a contiguous time interval of length t_i, but it is willing to be scheduled at any time before the deadline. Each accepted request must be assigned an interval of time of length t_i, and different requests must be assigned nonoverlapping intervals.

There are many objective functions we might seek to optimize when faced with this situation, and some are computationally much more difficult than

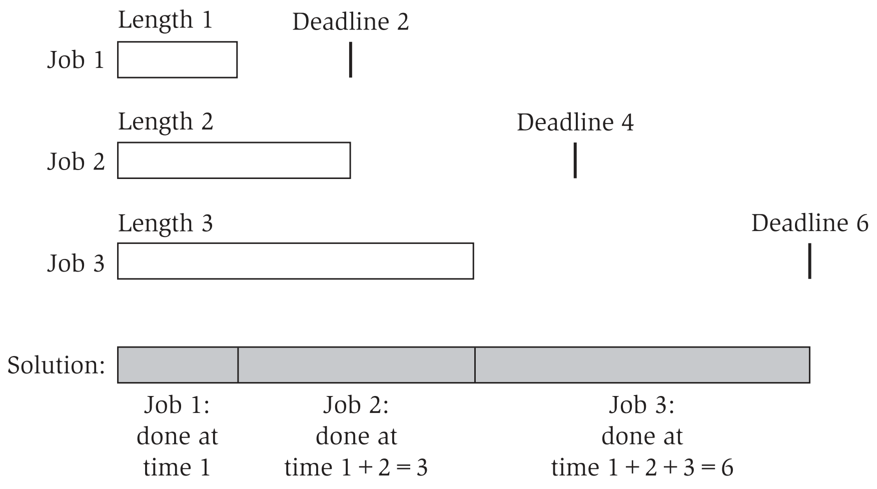

Figure 4.5 A sample instance of scheduling to minimize lateness.

others. Here we consider a very natural goal that can be optimized by a greedy algorithm. Suppose that we plan to satisfy each request, but we are allowed to let certain requests run late. Thus, beginning at our overall start time s, we will assign each request i an interval of time of length t_i; let us denote this interval by $[s(i), f(i)]$, with $f(i) = s(i) + t_i$. Unlike the previous problem, then, the algorithm must actually determine a start time (and hence a finish time) for each interval.

We say that a request i is *late* if it misses the deadline, that is, if $f(i) > d_i$. The *lateness* of such a request i is defined to be $l_i = f(i) - d_i$. We will say that $l_i = 0$ if request i is not late. The goal in our new optimization problem will be to schedule all requests, using nonoverlapping intervals, so as to minimize the *maximum lateness*, $L = \max_i l_i$. This problem arises naturally when scheduling jobs that need to use a single machine, and so we will refer to our requests as *jobs*.

Figure 4.5 shows a sample instance of this problem, consisting of three jobs: the first has length $t_1 = 1$ and deadline $d_1 = 2$; the second has $t_2 = 2$ and $d_2 = 4$; and the third has $t_3 = 3$ and $d_3 = 6$. It is not hard to check that scheduling the jobs in the order 1, 2, 3 incurs a maximum lateness of 0.

Designing the Algorithm

What would a greedy algorithm for this problem look like? There are several natural greedy approaches in which we look at the data (t_i, d_i) about the jobs and use this to order them according to some simple rule.

- One approach would be to schedule the jobs in order of increasing length t_i, so as to get the short jobs out of the way quickly. This immediately looks too simplistic, since it completely ignores the deadlines of the jobs. And indeed, consider a two-job instance where the first job has $t_1 = 1$ and $d_1 = 100$, while the second job has $t_2 = 10$ and $d_2 = 10$. Then the second job has to be started right away if we want to achieve lateness $L = 0$, and scheduling the second job first is indeed the optimal solution.

- The previous example suggests that we should be concerned about jobs whose available *slack time* $d_i - t_i$ is very small—they're the ones that need to be started with minimal delay. So a more natural greedy algorithm would be to sort jobs in order of increasing slack $d_i - t_i$.

  Unfortunately, this greedy rule fails as well. Consider a two-job instance where the first job has $t_1 = 1$ and $d_1 = 2$, while the second job has $t_2 = 10$ and $d_2 = 10$. Sorting by increasing slack would place the second job first in the schedule, and the first job would incur a lateness of 9. (It finishes at time 11, nine units beyond its deadline.) On the other hand, if we schedule the first job first, then it finishes on time and the second job incurs a lateness of only 1.

There is, however, an equally basic greedy algorithm that always produces an optimal solution. We simply sort the jobs in increasing order of their deadlines d_i, and schedule them in this order. (This rule is often called *Earliest Deadline First*.) There is an intuitive basis to this rule: we should make sure that jobs with earlier deadlines get completed earlier. At the same time, it's a little hard to believe that this algorithm always produces optimal solutions—specifically because it never looks at the lengths of the jobs. Earlier we were skeptical of the approach that sorted by length on the grounds that it threw away half the input data (i.e., the deadlines); but now we're considering a solution that throws away the other half of the data. Nevertheless, Earliest Deadline First does produce optimal solutions, and we will now prove this.

First we specify some notation that will be useful in talking about the algorithm. By renaming the jobs if necessary, we can assume that the jobs are labeled in the order of their deadlines, that is, we have

$$d_1 \leq \ldots \leq d_n.$$

We will simply schedule all jobs in this order. Again, let s be the start time for all jobs. Job 1 will start at time $s = s(1)$ and end at time $f(1) = s(1) + t_1$; Job 2 will start at time $s(2) = f(1)$ and end at time $f(2) = s(2) + t_2$; and so forth. We will use f to denote the finishing time of the last scheduled job. We write this algorithm here.

```
Order the jobs in order of their deadlines
Assume for simplicity of notation that $d_1 \leq \ldots \leq d_n$
Initially, $f = s$
```

```
Consider the jobs $i = 1, \ldots, n$ in this order
   Assign job $i$ to the time interval from $s(i) = f$ to $f(i) = f + t_i$
   Let $f = f + t_i$
End
Return the set of scheduled intervals $[s(i), f(i)]$ for $i = 1, \ldots, n$
```

Analyzing the Algorithm

To reason about the optimality of the algorithm, we first observe that the schedule it produces has no "gaps"—times when the machine is not working yet there are jobs left. The time that passes during a gap will be called *idle time:* there is work to be done, yet for some reason the machine is sitting idle. Not only does the schedule A produced by our algorithm have no idle time; it is also very easy to see that there is an optimal schedule with this property. We do not write down a proof for this.

(4.7) *There is an optimal schedule with no idle time.*

Now, how can we prove that our schedule A is optimal, that is, its maximum lateness L is as small as possible? As in previous analyses, we will start by considering an optimal schedule $\mathcal{O}$. Our plan here is to gradually modify $\mathcal{O}$, preserving its optimality at each step, but eventually transforming it into a schedule that is identical to the schedule A found by the greedy algorithm. We refer to this type of analysis as an *exchange argument*, and we will see that it is a powerful way to think about greedy algorithms in general.

We first try characterizing schedules in the following way. We say that a schedule A' has an *inversion* if a job i with deadline d_i is scheduled before another job j with earlier deadline $d_j < d_i$. Notice that, by definition, the schedule A produced by our algorithm has no inversions. If there are jobs with identical deadlines then there can be many different schedules with no inversions. However, we can show that all these schedules have the same maximum lateness L.

(4.8) *All schedules with no inversions and no idle time have the same maximum lateness.*

Proof. If two different schedules have neither inversions nor idle time, then they might not produce exactly the same order of jobs, but they can only differ in the order in which jobs with identical deadlines are scheduled. Consider such a deadline d. In both schedules, the jobs with deadline d are all scheduled consecutively (after all jobs with earlier deadlines and before all jobs with later deadlines). Among the jobs with deadline d, the last one has the greatest lateness, and this lateness does not depend on the order of the jobs. ■

The main step in showing the optimality of our algorithm is to establish that there is an optimal schedule that has no inversions and no idle time. To do this, we will start with any optimal schedule having no idle time; we will then convert it into a schedule with no inversions without increasing its maximum lateness. Thus the resulting scheduling after this conversion will be optimal as well.

(4.9) *There is an optimal schedule that has no inversions and no idle time.*

Proof. By (4.7), there is an optimal schedule $\mathcal{O}$ with no idle time. The proof will consist of a sequence of statements. The first of these is simple to establish.

(a) *If $\mathcal{O}$ has an inversion, then there is a pair of jobs i and j such that j is scheduled immediately after i and has $d_j < d_i$.*

Indeed, consider an inversion in which a job a is scheduled sometime before a job b, and $d_a > d_b$. If we advance in the scheduled order of jobs from a to b one at a time, there has to come a point at which the deadline we see decreases for the first time. This corresponds to a pair of consecutive jobs that form an inversion.

Now suppose $\mathcal{O}$ has at least one inversion, and by (a), let i and j be a pair of inverted requests that are consecutive in the scheduled order. We will decrease the number of inversions in $\mathcal{O}$ by swapping the requests i and j in the schedule $\mathcal{O}$. The pair (i, j) formed an inversion in $\mathcal{O}$, this inversion is eliminated by the swap, and no new inversions are created. Thus we have

(b) *After swapping i and j we get a schedule with one less inversion.*

The hardest part of this proof is to argue that the inverted schedule is also optimal.

(c) *The new swapped schedule has a maximum lateness no larger than that of $\mathcal{O}$.*

It is clear that if we can prove (c), then we are done. The initial schedule $\mathcal{O}$ can have at most $\binom{n}{2}$ inversions (if all pairs are inverted), and hence after at most $\binom{n}{2}$ swaps we get an optimal schedule with no inversions.

So we now conclude by proving (c), showing that by swapping a pair of consecutive, inverted jobs, we do not increase the maximum lateness L of the schedule. ∎

Proof of (c). We invent some notation to describe the schedule $\mathcal{O}$: assume that each request r is scheduled for the time interval $[s(r), f(r)]$ and has lateness l'_r. Let $L' = \max_r l'_r$ denote the maximum lateness of this schedule.

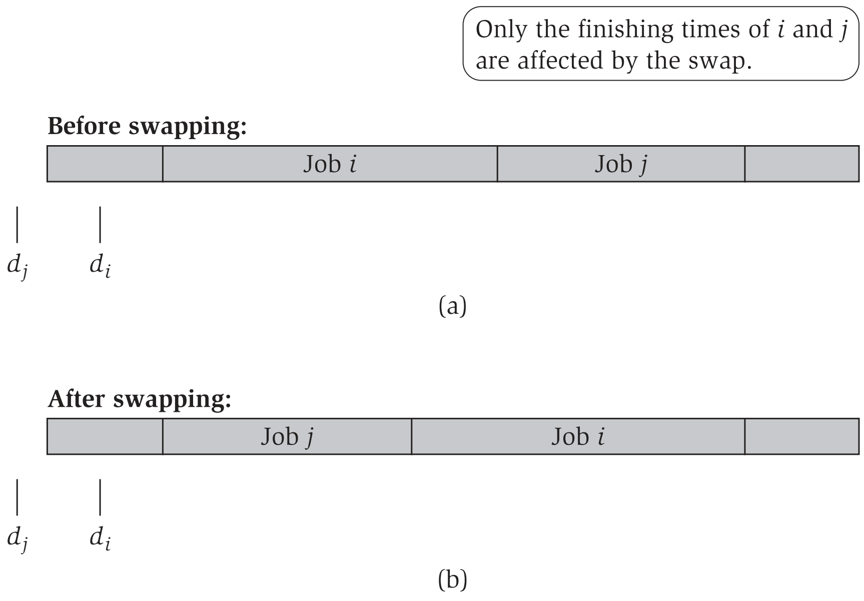

Figure 4.6 The effect of swapping two consecutive, inverted jobs.

Let $\overline{\mathcal{O}}$ denote the swapped schedule; we will use $\overline{s}(r)$, $\overline{f}(r)$, $\overline{l}_r$, and $\overline{L}$ to denote the corresponding quantities in the swapped schedule.

Now recall our two adjacent, inverted jobs i and j. The situation is roughly as pictured in Figure 4.6. The finishing time of j before the swap is exactly equal to the finishing time of i after the swap. Thus all jobs other than jobs i and j finish at the same time in the two schedules. Moreover, job j will get finished earlier in the new schedule, and hence the swap does not increase the lateness of job j.

Thus the only thing to worry about is job i: its lateness may have been increased, and what if this actually raises the maximum lateness of the whole schedule? After the swap, job i finishes at time $f(j)$, when job j was finished in the schedule $\mathcal{O}$. If job i is late in this new schedule, its lateness is $\overline{l}_i = \overline{f}(i) - d_i = f(j) - d_i$. But the crucial point is that i cannot be *more late* in the schedule $\overline{\mathcal{O}}$ than j was in the schedule $\mathcal{O}$. Specifically, our assumption $d_i > d_j$ implies that

$$\overline{l}_i = f(j) - d_i < f(j) - d_j = l'_j.$$

Since the lateness of the schedule $\mathcal{O}$ was $L' \geq l'_j > \overline{l}_i$, this shows that the swap does not increase the maximum lateness of the schedule. ∎

The optimality of our greedy algorithm now follows immediately.

(4.10) *The schedule A produced by the greedy algorithm has optimal maximum lateness L.*

Proof. Statement (4.9) proves that an optimal schedule with no inversions exists. Now by (4.8) all schedules with no inversions have the same maximum lateness, and so the schedule obtained by the greedy algorithm is optimal. ■

Extensions

There are many possible generalizations of this scheduling problem. For example, we assumed that all jobs were available to start at the common start time s. A natural, but harder, version of this problem would contain requests i that, in addition to the deadline d_i and the requested time t_i, would also have an earliest possible starting time r_i. This earliest possible starting time is usually referred to as the *release time*. Problems with release times arise naturally in scheduling problems where requests can take the form: Can I reserve the room for a two-hour lecture, sometime between 1 P.M. and 5 P.M.? Our proof that the greedy algorithm finds an optimal solution relied crucially on the fact that all jobs were available at the common start time s. (Do you see where?) Unfortunately, as we will see later in the book, in Chapter 8, this more general version of the problem is much more difficult to solve optimally.

4.3 Optimal Caching: A More Complex Exchange Argument

We now consider a problem that involves processing a sequence of requests of a different form, and we develop an algorithm whose analysis requires a more subtle use of the exchange argument. The problem is that of *cache maintenance*.

The Problem

To motivate caching, consider the following situation. You're working on a long research paper, and your draconian library will only allow you to have eight books checked out at once. You know that you'll probably need more than this over the course of working on the paper, but at any point in time, you'd like to have ready access to the eight books that are most relevant at that time. How should you decide which books to check out, and when should you return some in exchange for others, to minimize the number of times you have to exchange a book at the library?

This is precisely the problem that arises when dealing with a *memory hierarchy*: There is a small amount of data that can be accessed very quickly, and a large amount of data that requires more time to access; and you must decide which pieces of data to have close at hand.

Memory hierarchies have been a ubiquitous feature of computers since very early in their history. To begin with, data in the main memory of a processor can be accessed much more quickly than the data on its hard disk; but the disk has much more storage capacity. Thus, it is important to keep the most regularly used pieces of data in main memory, and go to disk as infrequently as possible. The same phenomenon, qualitatively, occurs with on-chip caches in modern processors. These can be accessed in a few cycles, and so data can be retrieved from cache much more quickly than it can be retrieved from main memory. This is another level of hierarchy: small caches have faster access time than main memory, which in turn is smaller and faster to access than disk. And one can see extensions of this hierarchy in many other settings. When one uses a Web browser, the disk often acts as a cache for frequently visited Web pages, since going to disk is still much faster than downloading something over the Internet.

Caching is a general term for the process of storing a small amount of data in a fast memory so as to reduce the amount of time spent interacting with a slow memory. In the previous examples, the on-chip cache reduces the need to fetch data from main memory, the main memory acts as a cache for the disk, and the disk acts as a cache for the Internet. (Much as your desk acts as a cache for the campus library, and the assorted facts you're able to remember without looking them up constitute a cache for the books on your desk.)

For caching to be as effective as possible, it should generally be the case that when you go to access a piece of data, it is already in the cache. To achieve this, a *cache maintenance* algorithm determines what to keep in the cache and what to evict from the cache when new data needs to be brought in.

Of course, as the caching problem arises in different settings, it involves various different considerations based on the underlying technology. For our purposes here, though, we take an abstract view of the problem that underlies most of these settings. We consider a set U of n pieces of data stored in *main memory*. We also have a faster memory, the *cache*, that can hold $k < n$ pieces of data at any one time. We will assume that the cache initially holds some set of k items. A sequence of data items $D = d_1, d_2, \ldots, d_m$ drawn from U is presented to us—this is the sequence of memory references we must process—and in processing them we must decide at all times which k items to keep in the cache. When item d_i is presented, we can access it very quickly if it is already in the cache; otherwise, we are required to bring it from main memory into the cache and, if the cache is full, to *evict* some other piece of data that is currently in the cache to make room for d_i. This is called a *cache miss*, and we want to have as few of these as possible.

Thus, on a particular sequence of memory references, a cache maintenance algorithm determines an *eviction schedule*—specifying which items should be evicted from the cache at which points in the sequence—and this determines the contents of the cache and the number of misses over time. Let's consider an example of this process.

- Suppose we have three items $\{a, b, c\}$, the cache size is $k = 2$, and we are presented with the sequence

  $$a, b, c, b, c, a, b.$$

  Suppose that the cache initially contains the items a and b. Then on the third item in the sequence, we could evict a so as to bring in c; and on the sixth item we could evict c so as to bring in a; we thereby incur two cache misses over the whole sequence. After thinking about it, one concludes that any eviction schedule for this sequence must include at least two cache misses.

Under real operating conditions, cache maintenance algorithms must process memory references $d_1, d_2, \ldots$ without knowledge of what's coming in the future; but for purposes of evaluating the quality of these algorithms, systems researchers very early on sought to understand the nature of the optimal solution to the caching problem. Given a full sequence S of memory references, what is the eviction schedule that incurs as few cache misses as possible?

Designing and Analyzing the Algorithm

In the 1960s, Les Belady showed that the following simple rule will always incur the minimum number of misses:

```
When $d_i$ needs to be brought into the cache,
  evict the item that is needed the farthest into the future
```

We will call this the *Farthest-in-Future Algorithm*. When it is time to evict something, we look at the next time that each item in the cache will be referenced, and choose the one for which this is as late as possible.

This is a very natural algorithm. At the same time, the fact that it is optimal on all sequences is somewhat more subtle than it first appears. Why evict the item that is needed farthest in the future, as opposed, for example, to the one that will be used least frequently in the future? Moreover, consider a sequence like

$$a, b, c, d, a, d, e, a, d, b, c$$

with $k = 3$ and items $\{a, b, c\}$ initially in the cache. The Farthest-in-Future rule will produce a schedule S that evicts c on the fourth step and b on the seventh step. But there are other eviction schedules that are just as good. Consider the schedule S' that evicts b on the fourth step and c on the seventh step, incurring the same number of misses. So in fact it's easy to find cases where schedules produced by rules other than Farthest-in-Future are also optimal; and given this flexibility, why might a deviation from Farthest-in-Future early on not yield an actual savings farther along in the sequence? For example, on the seventh step in our example, the schedule S' is actually evicting an item (c) that is needed *farther* into the future than the item evicted at this point by Farthest-in-Future, since Farthest-in-Future gave up c earlier on.

These are some of the kinds of things one should worry about before concluding that Farthest-in-Future really is optimal. In thinking about the example above, we quickly appreciate that it doesn't really matter whether b or c is evicted at the fourth step, since the other one should be evicted at the seventh step; so given a schedule where b is evicted first, we can swap the choices of b and c without changing the cost. This reasoning—swapping one decision for another—forms the first outline of an *exchange argument* that proves the optimality of Farthest-in-Future.

Before delving into this analysis, let's clear up one important issue. All the cache maintenance algorithms we've been considering so far produce schedules that only bring an item d into the cache in a step i if there is a request to d in step i, and d is not already in the cache. Let us call such a schedule *reduced*—it does the minimal amount of work necessary in a given step. But in general one could imagine an algorithm that produced schedules that are not reduced, by bringing in items in steps when they are not requested. We now show that for every nonreduced schedule, there is an equally good reduced schedule.

Let S be a schedule that may not be reduced. We define a new schedule $\overline{S}$—the *reduction* of S—as follows. In any step i where S brings in an item d that has not been requested, our construction of $\overline{S}$ "pretends" to do this but actually leaves d in main memory. It only really brings d into the cache in the next step j after this in which d is requested. In this way, the cache miss incurred by $\overline{S}$ in step j can be charged to the earlier cache operation performed by S in step i, when it brought in d. Hence we have the following fact.

(4.11) *$\overline{S}$ is a reduced schedule that brings in at most as many items as the schedule S.*

Note that for any reduced schedule, the number of items that are brought in is exactly the number of misses.

Proving the Optimalthy of Farthest-in-Future We now proceed with the exchange argument showing that Farthest-in-Future is optimal. Consider an arbitrary sequence D of memory references; let S_{FF} denote the schedule produced by Farthest-in-Future, and let S^* denote a schedule that incurs the minimum possible number of misses. We will now gradually "transform" the schedule S^* into the schedule S_{FF}, one eviction decision at a time, without increasing the number of misses.

Here is the basic fact we use to perform one step in the transformation.

(4.12) *Let S be a reduced schedule that makes the same eviction decisions as S_{FF} through the first j items in the sequence, for a number j. Then there is a reduced schedule S' that makes the same eviction decisions as S_{FF} through the first $j+1$ items, and incurs no more misses than S does.*

Proof. Consider the $(j+1)^{\text{st}}$ request, to item $d = d_{j+1}$. Since S and S_{FF} have agreed up to this point, they have the same cache contents. If d is in the cache for both, then no eviction decision is necessary (both schedules are reduced), and so S in fact agrees with S_{FF} through step $j+1$, and we can set $S' = S$. Similarly, if d needs to be brought into the cache, but S and S_{FF} both evict the same item to make room for d, then we can again set $S' = S$.

So the interesting case arises when d needs to be brought into the cache, and to do this S evicts item f while S_{FF} evicts item $e \neq f$. Here S and S_{FF} do not already agree through step $j+1$ since S has e in cache while S_{FF} has f in cache. Hence we must actually do something nontrivial to construct S'.

As a first step, we should have S' evict e rather than f. Now we need to further ensure that S' incurs no more misses than S. An easy way to do this would be to have S' agree with S for the remainder of the sequence; but this is no longer possible, since S and S' have slightly different caches from this point onward. So instead we'll have S' try to get its cache back to the same state as S as quickly as possible, while not incurring unnecessary misses. Once the caches are the same, we can finish the construction of S' by just having it behave like S.

Specifically, from request $j+2$ onward, S' behaves exactly like S until one of the following things happens for the first time.

(i) There is a request to an item $g \neq e, f$ that is not in the cache of S, and S evicts e to make room for it. Since S' and S only differ on e and f, it must be that g is not in the cache of S' either; so we can have S' evict f, and now the caches of S and S' are the same. We can then have S' behave exactly like S for the rest of the sequence.

(ii) There is a request to f, and S evicts an item e'. If $e' = e$, then we're all set: S' can simply access f from the cache, and after this step the caches of S and S' will be the same. If $e' \neq e$, then we have S' evict e' as well, and bring in e from main memory; this too results in S and S' having the same caches. However, we must be careful here, since S' is no longer a reduced schedule: it brought in e when it wasn't immediately needed. So to finish this part of the construction, we further transform S' to its reduction $\overline{S'}$ using (4.11); this doesn't increase the number of items brought in by S', and it still agrees with S_{FF} through step $j+1$.

Hence, in both these cases, we have a new reduced schedule S' that agrees with S_{FF} through the first $j+1$ items and incurs no more misses than S does. And crucially—here is where we use the defining property of the Farthest-in-Future Algorithm—one of these two cases will arise *before* there is a reference to e. This is because in step $j+1$, Farthest-in-Future evicted the item (e) that would be needed farthest in the future; so before there could be a request to e, there would have to be a request to f, and then case (ii) above would apply. ∎

Using this result, it is easy to complete the proof of optimality. We begin with an optimal schedule S^*, and use (4.12) to construct a schedule S_1 that agrees with S_{FF} through the first step. We continue applying (4.12) inductively for $j = 1, 2, 3, \ldots, m$, producing schedules S_j that agree with S_{FF} through the first j steps. Each schedule incurs no more misses than the previous one; and by definition $S_m = S_{FF}$, since it agrees with it through the whole sequence. Thus we have

(4.13) *S_{FF} incurs no more misses than any other schedule S^* and hence is optimal.*

Extensions: Caching under Real Operating Conditions

As mentioned in the previous subsection, Belady's optimal algorithm provides a benchmark for caching performance; but in applications, one generally must make eviction decisions on the fly without knowledge of future requests. Experimentally, the best caching algorithms under this requirement seem to be variants of the *Least-Recently-Used* (LRU) Principle, which proposes evicting the item from the cache that was referenced *longest ago*.

If one thinks about it, this is just Belady's Algorithm with the direction of time reversed—longest in the past rather than farthest in the future. It is effective because applications generally exhibit *locality of reference*: a running program will generally keep accessing the things it has just been accessing. (It is easy to invent pathological exceptions to this principle, but these are relatively rare in practice.) Thus one wants to keep the more recently referenced items in the cache.

Long after the adoption of LRU in practice, Sleator and Tarjan showed that one could actually provide some theoretical analysis of the performance of LRU, bounding the number of misses it incurs relative to Farthest-in-Future. We will discuss this analysis, as well as the analysis of a randomized variant on LRU, when we return to the caching problem in Chapter 13.