Proof. Our general definition of instability has four parts: This means that we have to make sure that none of the four bad things happens.

First, suppose there is an instability of type (i), consisting of pairs (m, w) and (m', w') in S with the property that $(m, w') \notin F$, m prefers w' to w, and w' prefers m to m'. It follows that m must have proposed to w'; so w' rejected m, and thus she prefers her final partner to m—a contradiction.

Next, suppose there is an instability of type (ii), consisting of a pair $(m, w) \in S$, and a man m', so that m' is not part of any pair in the matching, $(m', w) \notin F$, and w prefers m' to m. Then m' must have proposed to w and been rejected; again, it follows that w prefers her final partner to m'—a contradiction.

Third, suppose there is an instability of type (iii), consisting of a pair $(m, w) \in S$, and a woman w', so that w' is not part of any pair in the matching, $(m, w') \notin F$, and m prefers w' to w. Then no man proposed to w' at all; in particular, m never proposed to w', and so he must prefer w to w'—a contradiction.

Finally, suppose there is an instability of type (iv), consisting of a man m and a woman w, neither of which is part of any pair in the matching, so that $(m, w) \notin F$. But for m to be single, he must have proposed to every nonforbidden woman; in particular, he must have proposed to w, which means she would no longer be single—a contradiction. ■

Exercises

1. Decide whether you think the following statement is true or false. If it is true, give a short explanation. If it is false, give a counterexample.

   True or false? In every instance of the Stable Matching Problem, there is a stable matching containing a pair (m, w) such that m is ranked first on the preference list of w and w is ranked first on the preference list of m.

2. Decide whether you think the following statement is true or false. If it is true, give a short explanation. If it is false, give a counterexample.

   True or false? Consider an instance of the Stable Matching Problem in which there exists a man m and a woman w such that m is ranked first on the preference list of w and w is ranked first on the preference list of m. Then in every stable matching S for this instance, the pair (m, w) belongs to S.

3. There are many other settings in which we can ask questions related to some type of "stability" principle. Here's one, involving competition between two enterprises.

Suppose we have two television networks, whom we'll call $\mathcal{A}$ and $\mathcal{B}$. There are n prime-time programming slots, and each network has n TV shows. Each network wants to devise a *schedule*—an assignment of each show to a distinct slot—so as to attract as much market share as possible.

Here is the way we determine how well the two networks perform relative to each other, given their schedules. Each show has a fixed *rating*, which is based on the number of people who watched it last year; we'll assume that no two shows have exactly the same rating. A network *wins* a given time slot if the show that it schedules for the time slot has a larger rating than the show the other network schedules for that time slot. The goal of each network is to win as many time slots as possible.

Suppose in the opening week of the fall season, Network $\mathcal{A}$ reveals a schedule S and Network $\mathcal{B}$ reveals a schedule T. On the basis of this pair of schedules, each network wins certain time slots, according to the rule above. We'll say that the pair of schedules (S, T) is *stable* if neither network can unilaterally change its own schedule and win more time slots. That is, there is no schedule S' such that Network $\mathcal{A}$ wins more slots with the pair (S', T) than it did with the pair (S, T); and symmetrically, there is no schedule T' such that Network $\mathcal{B}$ wins more slots with the pair (S, T') than it did with the pair (S, T).

The analogue of Gale and Shapley's question for this kind of stability is the following: For every set of TV shows and ratings, is there always a stable pair of schedules? Resolve this question by doing one of the following two things:

(a) give an algorithm that, for any set of TV shows and associated ratings, produces a stable pair of schedules; or

(b) give an example of a set of TV shows and associated ratings for which there is no stable pair of schedules.

4. Gale and Shapley published their paper on the Stable Matching Problem in 1962; but a version of their algorithm had already been in use for ten years by the National Resident Matching Program, for the problem of assigning medical residents to hospitals.

   Basically, the situation was the following. There were m hospitals, each with a certain number of available positions for hiring residents. There were n medical students graduating in a given year, each interested in joining one of the hospitals. Each hospital had a ranking of the students in order of preference, and each student had a ranking of the hospitals in order of preference. We will assume that there were more students graduating than there were slots available in the m hospitals.

The interest, naturally, was in finding a way of assigning each student to at most one hospital, in such a way that all available positions in all hospitals were filled. (Since we are assuming a surplus of students, there would be some students who do not get assigned to any hospital.)

We say that an assignment of students to hospitals is *stable* if neither of the following situations arises.

- First type of instability: There are students s and s', and a hospital h, so that
  - s is assigned to h, and
  - s' is assigned to no hospital, and
  - h prefers s' to s.
- Second type of instability: There are students s and s', and hospitals h and h', so that
  - s is assigned to h, and
  - s' is assigned to h', and
  - h prefers s' to s, and
  - s' prefers h to h'.

So we basically have the Stable Matching Problem, except that (i) hospitals generally want more than one resident, and (ii) there is a surplus of medical students.

Show that there is always a stable assignment of students to hospitals, and give an algorithm to find one.

5. The Stable Matching Problem, as discussed in the text, assumes that all men and women have a fully ordered list of preferences. In this problem we will consider a version of the problem in which men and women can be *indifferent* between certain options. As before we have a set M of n men and a set W of n women. Assume each man and each woman ranks the members of the opposite gender, but now we allow ties in the ranking. For example (with $n = 4$), a woman could say that m_1 is ranked in first place; second place is a tie between m_2 and m_3 (she has no preference between them); and m_4 is in last place. We will say that w *prefers* m to m' if m is ranked higher than m' on her preference list (they are not tied).

With indifferences in the rankings, there could be two natural notions for stability. And for each, we can ask about the existence of stable matchings, as follows.

(a) A *strong instability* in a perfect matching S consists of a man m and a woman w, such that each of m and w prefers the other to their partner in S. Does there always exist a perfect matching with no strong instability? Either give an example of a set of men and women with preference lists for which every perfect matching has a strong instability; or give an algorithm that is guaranteed to find a perfect matching with no strong instability.

(b) A *weak instability* in a perfect matching S consists of a man m and a woman w, such that their partners in S are w' and m', respectively, and one of the following holds:

- m prefers w to w', and w either prefers m to m' or is indifferent between these two choices; or
- w prefers m to m', and m either prefers w to w' or is indifferent between these two choices.

In other words, the pairing between m and w is either preferred by both, or preferred by one while the other is indifferent. Does there always exist a perfect matching with no weak instability? Either give an example of a set of men and women with preference lists for which every perfect matching has a weak instability; or give an algorithm that is guaranteed to find a perfect matching with no weak instability.

6. Peripatetic Shipping Lines, Inc., is a shipping company that owns n ships and provides service to n ports. Each of its ships has a *schedule* that says, for each day of the month, which of the ports it's currently visiting, or whether it's out at sea. (You can assume the "month" here has m days, for some $m > n$.) Each ship visits each port for exactly one day during the month. For safety reasons, PSL Inc. has the following strict requirement:

(†) *No two ships can be in the same port on the same day.*

The company wants to perform maintenance on all the ships this month, via the following scheme. They want to *truncate* each ship's schedule: for each ship S_i, there will be some day when it arrives in its scheduled port and simply remains there for the rest of the month (for maintenance). This means that S_i will not visit the remaining ports on its schedule (if any) that month, but this is okay. So the *truncation* of S_i's schedule will simply consist of its original schedule up to a certain specified day on which it is in a port P; the remainder of the truncated schedule simply has it remain in port P.

Now the company's question to you is the following: Given the schedule for each ship, find a truncation of each so that condition (†) continues to hold: no two ships are ever in the same port on the same day.

Show that such a set of truncations can always be found, and give an algorithm to find them.

Example. Suppose we have two ships and two ports, and the "month" has four days. Suppose the first ship's schedule is

port P_1; at sea; port P_2; at sea

and the second ship's schedule is

at sea; port P_1; at sea; port P_2

Then the (only) way to choose truncations would be to have the first ship remain in port P_2 starting on day 3, and have the second ship remain in port P_1 starting on day 2.

7. Some of your friends are working for CluNet, a builder of large communication networks, and they are looking at algorithms for switching in a particular type of input/output crossbar.

Here is the setup. There are n *input wires* and n *output wires*, each directed from a *source* to a *terminus*. Each input wire meets each output wire in exactly one distinct point, at a special piece of hardware called a *junction box*. Points on the wire are naturally ordered in the direction from source to terminus; for two distinct points x and y on the same wire, we say that x is *upstream* from y if x is closer to the source than y, and otherwise we say x is *downstream* from y. The order in which one input wire meets the output wires is not necessarily the same as the order in which another input wire meets the output wires. (And similarly for the orders in which output wires meet input wires.) Figure 1.8 gives an example of such a collection of input and output wires.

Now, here's the switching component of this situation. Each input wire is carrying a distinct data stream, and this data stream must be *switched* onto one of the output wires. If the stream of Input i is switched onto Output j, at junction box B, then this stream passes through all junction boxes upstream from B on Input i, then through B, then through all junction boxes downstream from B on Output j. It does not matter which input data stream gets switched onto which output wire, but each input data stream must be switched onto a *different* output wire. Furthermore—and this is the tricky constraint—no two data streams can pass through the same junction box following the switching operation.

Finally, here's the problem. Show that for any specified pattern in which the input wires and output wires meet each other (each pair meeting exactly once), a valid switching of the data streams can always be found—one in which each input data stream is switched onto a different output, and no two of the resulting streams pass through the same junction box. Additionally, give an algorithm to find such a valid switching.

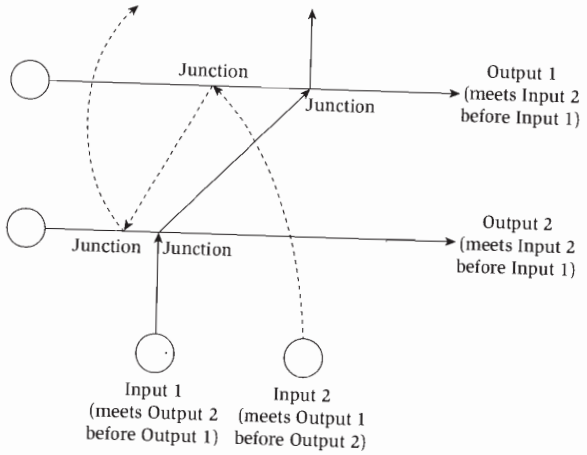

Figure 1.8 An example with two input wires and two output wires. Input 1 has its junction with Output 2 upstream from its junction with Output 1; Input 2 has its junction with Output 1 upstream from its junction with Output 2. A valid solution is to switch the data stream of Input 1 onto Output 2, and the data stream of Input 2 onto Output 1. On the other hand, if the stream of Input 1 were switched onto Output 1, and the stream of Input 2 were switched onto Output 2, then both streams would pass through the junction box at the meeting of Input 1 and Output 2—and this is not allowed.

8. For this problem, we will explore the issue of *truthfulness* in the Stable Matching Problem and specifically in the Gale-Shapley algorithm. The basic question is: Can a man or a woman end up better off by lying about his or her preferences? More concretely, we suppose each participant has a true preference order. Now consider a woman w. Suppose w prefers man m to m', but both m and m' are low on her list of preferences. Can it be the case that by switching the order of m and m' on her list of preferences (i.e., by falsely claiming that she prefers m' to m) and running the algorithm with this false preference list, w will end up with a man m'' that she truly prefers to both m and m'? (We can ask the same question for men, but will focus on the case of women for purposes of this question.)

Resolve this question by doing one of the following two things:

(a) Give a proof that, for any set of preference lists, switching the order of a pair on the list cannot improve a woman's partner in the Gale-Shapley algorithm; or

Now, the function f_3 isn't so hard to deal with. It starts out smaller than 10^n, but once $n \geq 10$, then clearly $10^n \leq n^n$. This is exactly what we need for the definition of $O(\cdot)$ notation: for all $n \geq 10$, we have $10^n \leq cn^n$, where in this case $c = 1$, and so $10^n = O(n^n)$.

Finally, we come to function f_5, which is admittedly kind of strange-looking. A useful rule of thumb in such situations is to try taking logarithms to see whether this makes things clearer. In this case, $\log_2 f_5(n) = \sqrt{\log_2 n} = (\log_2 n)^{1/2}$. What do the logarithms of the other functions look like? $\log f_4(n) = \log_2 \log_2 n$, while $\log f_2(n) = \frac{1}{3} \log_2 n$. All of these can be viewed as functions of $\log_2 n$, and so using the notation $z = \log_2 n$, we can write

$$\log f_2(n) = \frac{1}{3}z$$
$$\log f_4(n) = \log_2 z$$
$$\log f_5(n) = z^{1/2}$$

Now it's easier to see what's going on. First, for $z \geq 16$, we have $\log_2 z \leq z^{1/2}$. But the condition $z \geq 16$ is the same as $n \geq 2^{16} = 65,536$; thus once $n \geq 2^{16}$ we have $\log f_4(n) \leq \log f_5(n)$, and so $f_4(n) \leq f_5(n)$. Thus we can write $f_4(n) = O(f_5(n))$. Similarly we have $z^{1/2} \leq \frac{1}{3}z$ once $z \geq 9$—in other words, once $n \geq 2^9 = 512$. For n above this bound we have $\log f_5(n) \leq \log f_2(n)$ and hence $f_5(n) \leq f_2(n)$, and so we can write $f_5(n) = O(f_2(n))$. Essentially, we have discovered that $2^{\sqrt{\log_2 n}}$ is a function whose growth rate lies somewhere between that of logarithms and polynomials.

Since we have sandwiched f_5 between f_4 and f_2, this finishes the task of putting the functions in order.

Solved Exercise 2

Let f and g be two functions that take nonnegative values, and suppose that $f = O(g)$. Show that $g = \Omega(f)$.

Solution This exercise is a way to formalize the intuition that $O(\cdot)$ and $\Omega(\cdot)$ are in a sense opposites. It is, in fact, not difficult to prove; it is just a matter of unwinding the definitions.

We're given that, for some constants c and n_0, we have $f(n) \leq cg(n)$ for all $n \geq n_0$. Dividing both sides by c, we can conclude that $g(n) \geq \frac{1}{c}f(n)$ for all $n \geq n_0$. But this is exactly what is required to show that $g = \Omega(f)$: we have established that $g(n)$ is at least a constant multiple of $f(n)$ (where the constant is $\frac{1}{c}$), for all sufficiently large n (at least n_0).

Exercises

1. Suppose you have algorithms with the five running times listed below. (Assume these are the exact running times.) How much slower do each of these algorithms get when you (a) double the input size, or (b) increase the input size by one?
   (a) n^2
   (b) n^3
   (c) $100n^2$
   (d) $n \log n$
   (e) 2^n

2. Suppose you have algorithms with the six running times listed below. (Assume these are the exact number of operations performed as a function of the input size n.) Suppose you have a computer that can perform 10^{10} operations per second, and you need to compute a result in at most an hour of computation. For each of the algorithms, what is the largest input size n for which you would be able to get the result within an hour?
   (a) n^2
   (b) n^3
   (c) $100n^2$
   (d) $n \log n$
   (e) 2^n
   (f) 2^{2^n}

3. Take the following list of functions and arrange them in ascending order of growth rate. That is, if function $g(n)$ immediately follows function $f(n)$ in your list, then it should be the case that $f(n)$ is $O(g(n))$.
   $f_1(n) = n^{2.5}$
   $f_2(n) = \sqrt{2n}$
   $f_3(n) = n + 10$
   $f_4(n) = 10^n$
   $f_5(n) = 100^n$
   $f_6(n) = n^2 \log n$

4. Take the following list of functions and arrange them in ascending order of growth rate. That is, if function $g(n)$ immediately follows function $f(n)$ in your list, then it should be the case that $f(n)$ is $O(g(n))$.

$$g_1(n) = 2^{\sqrt{\log n}}$$
$$g_2(n) = 2^n$$
$$g_4(n) = n^{4/3}$$
$$g_3(n) = n(\log n)^3$$
$$g_5(n) = n^{\log n}$$
$$g_6(n) = 2^{2^n}$$
$$g_7(n) = 2^{n^2}$$

5. Assume you have functions f and g such that $f(n)$ is $O(g(n))$. For each of the following statements, decide whether you think it is true or false and give a proof or counterexample.

(a) $\log_2 f(n)$ is $O(\log_2 g(n))$.

(b) $2^{f(n)}$ is $O(2^{g(n)})$.

(c) $f(n)^2$ is $O(g(n)^2)$.

6. Consider the following basic problem. You're given an array A consisting of n integers $A[1], A[2], \ldots, A[n]$. You'd like to output a two-dimensional n-by-n array B in which $B[i, j]$ (for $i < j$) contains the sum of array entries $A[i]$ through $A[j]$—that is, the sum $A[i] + A[i+1] + \cdots + A[j]$. (The value of array entry $B[i, j]$ is left unspecified whenever $i \geq j$, so it doesn't matter what is output for these values.)

Here's a simple algorithm to solve this problem.

```
For i = 1, 2, ..., n
  For j = i+1, i+2, ..., n
    Add up array entries A[i] through A[j]
    Store the result in B[i, j]
  Endfor
Endfor
```

(a) For some function f that you should choose, give a bound of the form $O(f(n))$ on the running time of this algorithm on an input of size n (i.e., a bound on the number of operations performed by the algorithm).

(b) For this same function f, show that the running time of the algorithm on an input of size n is also $\Omega(f(n))$. (This shows an asymptotically tight bound of $\Theta(f(n))$ on the running time.)

(c) Although the algorithm you analyzed in parts (a) and (b) is the most natural way to solve the problem—after all, it just iterates through the relevant entries of the array B, filling in a value for each—it contains some highly unnecessary sources of inefficiency. Give a different algorithm to solve this problem, with an asymptotically better running time. In other words, you should design an algorithm with running time $O(g(n))$, where $\lim_{n\to\infty} g(n)/f(n) = 0$.

7. There's a class of folk songs and holiday songs in which each verse consists of the previous verse, with one extra line added on. "The Twelve Days of Christmas" has this property; for example, when you get to the fifth verse, you sing about the five golden rings and then, reprising the lines from the fourth verse, also cover the four calling birds, the three French hens, the two turtle doves, and of course the partridge in the pear tree. The Aramaic song "Had gadya" from the Passover Haggadah works like this as well, as do many other songs.

These songs tend to last a long time, despite having relatively short scripts. In particular, you can convey the words plus instructions for one of these songs by specifying just the new line that is added in each verse, without having to write out all the previous lines each time. (So the phrase "five golden rings" only has to be written once, even though it will appear in verses five and onward.)

There's something asymptotic that can be analyzed here. Suppose, for concreteness, that each line has a length that is bounded by a constant c, and suppose that the song, when sung out loud, runs for n words total. Show how to encode such a song using a script that has length $f(n)$, for a function $f(n)$ that grows as slowly as possible.

8. You're doing some stress-testing on various models of glass jars to determine the height from which they can be dropped and still not break. The setup for this experiment, on a particular type of jar, is as follows. You have a ladder with n rungs, and you want to find the highest rung from which you can drop a copy of the jar and not have it break. We call this the *highest safe rung*.

It might be natural to try binary search: drop a jar from the middle rung, see if it breaks, and then recursively try from rung $n/4$ or $3n/4$ depending on the outcome. But this has the drawback that you could break a lot of jars in finding the answer.

If your primary goal were to conserve jars, on the other hand, you could try the following strategy. Start by dropping a jar from the first rung, then the second rung, and so forth, climbing one higher each time until the jar breaks. In this way, you only need a single jar—at the moment

it breaks, you have the correct answer—but you may have to drop it n times (rather than $\log n$ as in the binary search solution).

So here is the trade-off: it seems you can perform fewer drops if you're willing to break more jars. To understand better how this trade-off works at a quantitative level, let's consider how to run this experiment given a fixed "budget" of $k \geq 1$ jars. In other words, you have to determine the correct answer—the highest safe rung—and can use at most k jars in doing so.

(a) Suppose you are given a budget of $k = 2$ jars. Describe a strategy for finding the highest safe rung that requires you to drop a jar at most $f(n)$ times, for some function $f(n)$ that grows slower than linearly. (In other words, it should be the case that $\lim_{n\to\infty} f(n)/n = 0$.)

(b) Now suppose you have a budget of $k > 2$ jars, for some given k. Describe a strategy for finding the highest safe rung using at most k jars. If $f_k(n)$ denotes the number of times you need to drop a jar according to your strategy, then the functions $f_1, f_2, f_3, \ldots$ should have the property that each grows asymptotically slower than the previous one: $\lim_{n\to\infty} f_k(n)/f_{k-1}(n) = 0$ for each k.

Notes and Further Reading

Polynomial-time solvability emerged as a formal notion of efficiency by a gradual process, motivated by the work of a number of researchers including Cobham, Rabin, Edmonds, Hartmanis, and Stearns. The survey by Sipser (1992) provides both a historical and technical perspective on these developments. Similarly, the use of asymptotic order of growth notation to bound the running time of algorithms—as opposed to working out exact formulas with leading coefficients and lower-order terms—is a modeling decision that was quite non-obvious at the time it was introduced; Tarjan's Turing Award lecture (1987) offers an interesting perspective on the early thinking of researchers including Hopcroft, Tarjan, and others on this issue. Further discussion of asymptotic notation and the growth of basic functions can be found in Knuth (1997a).

The implementation of priority queues using heaps, and the application to sorting, is generally credited to Williams (1964) and Floyd (1964). The priority queue is an example of a nontrivial data structure with many applications; in later chapters we will discuss other data structures as they become useful for the implementation of particular algorithms. We will consider the `Union-Find` data structure in Chapter 4 for implementing an algorithm to find minimum-cost spanning trees, and we will discuss randomized hashing in Chapter 13. A number of other data structures are discussed in the book by Tarjan (1983). The LEDA library (Library of Efficient Datatypes and Algorithms) of Mehlhorn and Näher (1999) offers an extensive library of data structures useful in combinatorial and geometric applications.

Notes on the Exercises Exercise 8 is based on a problem we learned from Sam Toueg.