10. Greedy Algorithms

Optimization, the process of finding the best solution from a set of feasible alternatives, is a key component of nearly every technical discipline. To give a few broad examples, most fields of science attempt to build simple theoretical models that accurately describe the behavior of physical systems; optimization (in this context, known as *regression*) is used to fit the parameters of such a model so it agrees with experimental data. In fields like physics and computational chemistry, the evolution of a system over time (e.g., a protein molecule as it folds, or a planet orbiting the sun) is described by a minimum-energy solution to a set of equations, and hence can be predicted using optimization. In most areas of engineering, we solve optimization problems to design systems that are as efficient, robust, and cost-effective as possible. In the field of machine learning, classification algorithms can optimize their parameters to "learn" from a set of known training data, in order to make better predictions against unknown test data (also another example of regression). If you study the field of operations research, you may spend a majority of your time studying optimization problems that have useful applications in many areas of practice, such as transportation, logistics, scheduling, routing, and many more.

This chapter is the first of five that focus on general algorithmic techniques for solving optimization problems. The first three study greedy algorithms, dynamic programming, and linear programming, all powerful techniques with broad applicability. We then study various "heuristics" for dealing with computationally hard problems, and finally techniques for continuous optimization problems.

Greedy Algorithms. Contrary to what you might have learned as a child, being greedy is sometimes good. Computer scientists are very fond of greedy algorithms because they are typically fast, simple to describe and implement, and widely applicable. Greedy algorithms give optimal or nearly-optimal solutions for a broad range of optimization problems in practice.

How do greedy algorithms operate? As a trivial example, suppose we know the skill levels of n prospective players in some sport, from which we want to build a team of k players having the largest possible total skill. A prototypical greedy solution would choose players one by one until the team is full, at each step picking the most skillful player still available. Greedy algorithms are the simplest examples of the "incremental construction" design principle applied to optimization problems. They build up a solution one element at a time, making each successive choice "greedily", giving the best short-term improvement to our partial solution. Most greedy algorithms never go back and revise old decisions, building a solution very quickly in just a single pass. In our example, once someone is added to the team, that decision is never reconsidered.

Although greedy algorithms are perhaps the simplest methods for solving many optimization problems, their inappropriate use when they do not apply is a very common error among novice algorithm designers. Many problems may at first glance appear greedily-solvable, but actually require more sophisticated techniques like dynamic programming (the subject of the next chapter). A common mistake among beginning students is to apply a greedy algorithm to a few simple inputs, to observe that it computes an optimal solution in each case, and then to conclude that it must give an optimal solution for every instance. Of course, it is unwise to draw such a conclusion for any algorithm without a more careful analysis; however, the sheer simplicity of a greedy algorithm can often tempt us to forgo the careful analysis and instead trust our intuition. In the pages that follow, we will study several techniques one can use to mathematically justify the correctness of a greedy algorithm. Such a formal justification can be very important in developing appropriate confidence that a given greedy algorithm works, particularly for more difficult problems in which our intuition may lead us astray.

As with many algorithmic concepts, the best way to develop a knack for recognizing greedily-solvable problems and analyzing greedy algorithms is by working through examples. The concepts in this chapter are therefore illustrated using progressively more complex examples, each one drawn from a domain of problems to which greedy algorithms have notable applicability. Along the way, we also highlight the use of greedy methods in online and approximation algorithms, two subfields where greedy techniques are often found. An extensive collection of practice problems is included at the end of the chapter.

10.1 Example: Scheduling on a Single Machine

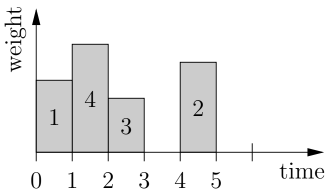
Scheduling problems involve computing an assignment of jobs to one or more machines over time. Greedy methods have been successfully applied to many of these, including the following simple example. Suppose we need to schedule n jobs on a single machine, subject to the following constraints:

- Each job requires exactly one unit of time to process,
- Each job j has an associated integer *release time*, r_j, known in advance, specifying the time at which it is released into the machine and made available for processing. We cannot schedule job j at any point in time earlier than r_j,
- The machine can process at most a single job at once, and
- The machine is *non-preemptive*; that is, once it starts a job, it continues working on that job until completion. It cannot temporarily interrupt a job, work on something else for a bit, and then return to finish the job later.

Our goal is to schedule the jobs so as to minimize the sum of their weighted completion times, $\sum_j w_j C_j$, where w_j is the *weight* of job j (higher weight meaning higher

job (j)	release time (r_j)	weight (w_j)
1	0	9
2	4	12
3	0	6
4	1	14

(a)



(b)

FIGURE 10.1: Graphical depiction of (a) an example instance for our scheduling problem, and (b) an optimal schedule for this instance. Each job j is represented by a unit-width box whose height corresponds to the weight, w_j, of the job. The objective value for schedule in (b) is $\sum_j w_j C_j = 9 \cdot 1 + 14 \cdot 2 + 6 \cdot 3 + 12 \cdot 5 = 115$.

priority) and C_j is the time it completes[1]. Figure 10.1 shows an example instance with 4 jobs and their optimal schedule. Sometimes the objective of this problem is written as minimizing the *average* weighted completion time of the jobs, $\frac{1}{n} \sum_j w_j C_j$, and sometimes it is written as minimizing the total or average weighted *flow time* over all the jobs, $\sum_j w_j F_j$, where the flow time $F_j = C_j - r_j$ of job j indicates the total time j spends in the system. All of these are essentially equivalent in that a schedule minimizing one of them clearly minimizes the others as well.

To build an optimal schedule, observe that a job j of high weight should intuitively be scheduled as early as possible, or else the term $w_j C_j$ will contribute a substantial penalty to the objective. This suggests a greedy algorithm where we simulate the arrival of jobs over time, always choosing from the set of available jobs the one having maximum weight to schedule next. This *greedy choice rule* is easy to implement in $O(n \log n)$ time by first sorting on release time and then using a binary heap keyed on weight to hold the set of active jobs as we simulate their arrival and schedule them one by one. Our algorithm is quite simple and intuitive, and indeed for most problems that allow greedy solutions, the right algorithm is usually one that is simple and feels "natural".

10.1.1 Proof by Contradiction: The Exchange Argument

Even for simple greedy algorithms, it is still important to justify correctness in a mathematically convincing fashion. One standard way of doing this is with a so-called *exchange argument*: we hypothetically suppose, for purposes of contradiction, that our greedy solution is not optimal on some input. It follows that any optimal solution deviates from our greedy solution somewhere, at a point where it used a non-greedy decision. In our scheduling example, this might be a point in time where the greedy solution chose job j, but an optimal schedule chose some other job j' with $w_{j'} < w_j$. We then consider making an exchange in the optimal solution so it looks more like the greedy solution; for example, what if the optimal solution had selected job j instead of j' at this point in time? If we can argue that such a modification does not negatively impact the objective value of our optimal solution,

[1]In case you wonder why C_j is a capital letter and w_j is lowercase, this is a convention used in the scheduling literature to indicate that C_j is an output while w_j is an input.

we can typically establish a contradiction — either that the greedy solution was just as good as the optimal solution, or that the supposedly "optimal" solution couldn't have been optimal in the first place. [Detailed exchange argument for our scheduling example]

10.1.2 Proof by Induction

A second popular approach for proving optimality of greedy algorithms uses induction on the size of our problem instance, arguing that our greedy algorithm optimally solves every instance of size n under the assumption (by induction) that it optimally solves every instance of size strictly smaller than n. To use this approach, it helps to phrase our algorithm in the following recursive form:

1. The algorithm initially makes a greedy choice and commits to some small piece of the solution.
2. Next, the algorithm applies itself recursively to the "left-over" part of the problem in order to complete the solution.

Two key properties must now hold for our algorithm to be optimal:

- **Greedy Choice.** Our algorithm must satisfy the *greedy choice property*, which states that the partial solution built in step (1) can always be extended to reach an optimal solution, so in some sense the algorithm is heading "in the right direction". This is often shown using an exchange argument. [Proof that our example algorithm satisfies the greedy choice property]
- **Optimal Substructure.** Our problem must satisfy the *optimal substructure property*, which states that completing any partial solution is equivalent to optimally solving a smaller "left-over" instance of the same problem[2]. In our scheduling example, once the algorithm schedules its initial job at time t, the best way to complete the schedule is to optimally schedule the remaining $n-1$ jobs so as to minimize their sum of weighted completion times (where the release times of all jobs are now considered to be at least $t+1$, so they cannot overlap the job we just scheduled).

Optimality follows as a natural consequence of these two properties. Greedy choice tells us an optimal solution is still reachable after step (1), and the optimal substructure property tells us that to reach this optimal solution, we need to optimally solve the smaller "left-over" problem in step (2). Since this left-over problem has the same form as the original, our algorithm will indeed optimally solve it due to induction on problem size.

The following problems offer a good opportunity to practice applying and analyzing greedy algorithms on relatives of our basic scheduling problem.

[2]Optimal substructure is a key property in optimization, since it tells us that optimal solutions to a problem contain within them optimal solutions to smaller subproblems. This has important algorithmic implications. For example, we will see in the next chapter how optimal substructure enables the use of *dynamic programming* to solve large problem by combining solutions to smaller subproblems, even when the greedy choice property fails to hold.

Problem 158 (Another Greedy Scheduling Algorithm). Another natural greedy approach to our scheduling problem is the following: sort the jobs in decreasing order by weight, then for each job in sequence, greedily try to schedule it as early as possible (depending on its release time and the jobs that have already been scheduled). Does this approach also compute an optimal schedule? Can you implement this algorithm in only $O(n \log n)$ time? [Solution]

Problem 159 (Non-Unit Processing Times). Assume that all jobs are available at the beginning of time (i.e., all jobs have $r_j = 0$), and let us associate an integer *processing time* p_j for each job j, specifying the time the job needs to run. Show that our original greedy algorithm no longer satisfies the greedy choice property, but that if we schedule our jobs in decreasing order of w_j/p_j, then we once again minimize the sum of weighted completion times. [Solution]

Problem 160 (An Alternate Objective). The problem above asks us to schedule n jobs, each with an associated weight w_j and processing time p_j, on a single machine, so as to minimize the weighted sum of completion times $\sum w_j C_j$. Another popular objective in this setting is to minimize $\sum_j w_j(1 - \alpha^{-C_j})$, where $\alpha > 1$ is some specified constant. Observe that jobs completing early contribute very little to this objective, while jobs completing late contribute almost their entire weight w_j. Please give an $O(n \log n)$ greedy algorithm for optimally ordering the jobs according to this new objective. [Solution]

Problem 161 (Scheduling with Deadlines). Let's assume that all jobs are available at the beginning of time (i.e., all jobs have $r_j = 0$), and that each job j has an associated processing time p_j as well as a *deadline* d_j (note that there are no job weights here). We would ideally like to produce a schedule where $C_j \leq d_j$ for all jobs j, but this may not be possible. Devise an $O(n \log n)$ greedy algorithm which minimizes the maximum *lateness* over all jobs. The lateness of job j is naturally defined as $L_j = C_j - d_j$; note that this can be a negative number if job j completes before its deadline. [Solution]

Problem 162 (Precedence Constraints). Let us generalize the objective for the preceding problem slightly. Associate with every job j a monotonically increasing *penalty* function $f_j(C_j)$ that tells us the amount of penalty we incur if job j completes at time C_j. We would like to order our jobs so as to minimize the largest penalty among all jobs (the preceding problem is a special case where the penalty of a job corresponds to its lateness). As in the preceding problem, each job j also has a processing time p_j and a release time $r_j = 0$. Show how to construct an optimal solution in $O(n^2)$ time using a simple greedy algorithm. To make things a bit more interesting, extend your solution to allow for *precedence constraints* among our jobs. We can model these using a directed acyclic graph (DAG), where the n nodes represent jobs and a directed path from node i to node j means that job i must be completed before job j starts. Your algorithm should run in $O(m + n \log n)$ time, where m is the number of edges in precedence graph. [Solution]

Problem 163 (Special Cases of Deadline-Constrained Scheduling). Suppose we are given n jobs, where each job j has a processing time p_j, a deadline $d_j \geq p_j$ by which it must be completed, and a value v_j. We would like to schedule a maximum-value subset of jobs on a single machine in a non-preemptive fashion (i.e., jobs must be processed in their entirety in a single contiguous block of time). This problem is NP-hard since if all deadlines are equal to a common value C, it is nothing more than the famous NP-hard "knapsack problem" (e.g., see Section 11.2).

(a) In the special case where all jobs have unit value we can solve the deadline-constrained scheduling problem optimally in $O(n \log n)$ time using a simple greedy algorithm. Try to devise such an algorithm. As a hint, consider the jobs in increasing order by their deadlines, and you may on occasion wish to unschedule a previously scheduled job (this is somewhat uncharacteristic of greedy algorithms, since they typically do not revise past decisions). [Solution]

(b) In this special case where all jobs have unit processing times, show that we can also construct an optimal solution using a simple $O(n \log n)$ greedy algorithm. [Solution]

Problem 164 (Two-Stage Flow Shop Scheduling). A *flow shop* scheduling problem requires that we send n jobs through a series of k sequential stages on an assembly line. Each job must be processed by each of stages $1, 2, \ldots, k$ in order, but the ordering of the jobs on each stage can be arbitrary (e.g., we can process job 1 followed by job 2 on the first stage, then job 2 followed by job 1 on the second stage). The stages operate in parallel, so they can both process (different) jobs at the same time. In this problem we consider the simple case of $k = 2$ stages, since the problem is NP-hard for $k \geq 3$. Suppose for each job i we know the time a_i required to process it on stage 1 and the time b_i required to process it on stage 2. Please describe an optimal greedy algorithm for scheduling jobs that minimizes the *makespan* of our schedule (the time the final job completes in stage 2). As a hint, you may want to consider first consider the simpler case where $a_i \leq b_i$ for all jobs i, and also the case where $a_i \geq b_i$ for all jobs i. [Solution]

Problem 165 (Two-Stage Scheduling with Non-Uniform Machines). You need to process n jobs, all of unit size, on m machines, each running at a different (known) speed. Please show how to schedule the jobs so as to minimize the makespan (latest completion time) of the entire schedule. Next, suppose we have two "stages" of machines: m_1 machines in the first stage and m_2 in the second. Again, all machines run at different speeds. Each job needs to be processed by some machine in stage 1 (any will do) and then later by some machine in stage 2 (again, any will do); the job can wait as long as necessary between these two steps. For a challenge, please devise a fast algorithm for computing a schedule of minimum makespan. [Solution]

The scheduling problems above are just the tip of the iceberg, as vast numbers of different scheduling problems have been studied in the literature. For those interested in more detail, the endnotes contain a table of all the scheduling problems found throughout this book, as well as a discussion of how scheduling problems are classified based on objective, machine environment, and other constraints.

10.1.3 Greedy Methods and Online Algorithms

Recall from Section 1.7.3 that an online algorithm does not see the entire input to a problem right away, but rather must make decisions as input gradually arrives over time. Greedy algorithms are rather common in such online settings, since they only consider short-term consequences of their decisions, and in an online setting, short-term information is all we have available.

As an example, suppose that for our example scheduling problem that we do not know the release times of jobs in advance, and that instead we only learn about jobs when they haphazardly arrive at our machine as time progresses. The machine does not know what jobs, if any, will be arriving in the future. It is not difficult to see that our original greedy algorithm operates correctly in this online setting. It still produces an optimal schedule, doing just as well as we could have done if we had the advantage of knowing what jobs would be arriving in the future.

Problem 166 (Greedy Online Scheduling). Suppose we are in charge of scheduling jobs (all of unit duration) on a single machine. In each time step, a set of jobs may arrive, each one having an associated value and deadline. After its deadline elapses, we can no longer schedule a job. In each time step, we must decide which of our currently-available jobs we wish to process (we can process exactly one job per time step). It may not be possible to schedule every job, so our goal is to schedule a set of jobs of maximum total value. Unfortunately, it is not always possible to compute an optimal solution due to the online nature of the problem, since we do not necessarily know which jobs (if any) will be arriving in the future. Nonetheless, please argue that the simple greedy strategy that always schedules the pending job of maximum value gives us at least half the value of an optimal schedule. That is, argue that this greedy strategy is 2-competitive. [Solution]

Problem 167 (Self-Organizing Linked Lists). We want maintain a linked list on n elements with associated access probability $p_1 \ldots p_n$. The linked list is subjected to a long sequence of queries, in which the elements are each accessed at random with frequencies according to the p_i's. To access the k^{th} element in this list requires walking down the list from its beginning, and hence requires $\Theta(k)$ time.

(a) Suppose we know $p_1 \ldots p_n$. How do we arrange the elements in the list so as to minimize the expected access time for a single generic element? [Solution]

(b) In an online setting we may not know the access probabilities. Using a technique somewhat reminiscent of splay trees (Section 6.2.7) let us employ the greedy *move-to-front* heuristic: whenever an element is accessed, we move it to the front of the list, hoping that frequently-accessed elements will therefore tend to stay near the front of the list. Argue that this approach is 2-competitive — i.e., that the expected access time per element is at most twice that of part (a). Please assume that sufficiently many access have occurred to bring us into "steady state", where the initial ordering of elements in the list no longer has significant influence. [Solution]

Problem 168 (Cache Page Replacement). Most computer systems employ a small, fast "cache" memory capable of holding several pages of data from main memory at once; typical page sizes tend to be in the 1K through 4K range. Memory access is very fast if it happens to be from a page residing in the cache. Otherwise, it causes a *cache miss*, where we bring the relevant page into the cache. When this happens, we must decide which page currently in the cache to evict. Common "greedy" strategies are to evict the *least recently used* (LRU) page in the cache, or to impose a *first in first out* (FIFO) discipline on the cache pages. In the long run, neither strategy may turn out to be optimal, since the best policy (if we magically knew all future accesses yet to come) would be to evict the page that will be accessed farthest ahead in the future — the so-called *ideal* cache policy. Although the LRU and FIFO policies can cause substantially more page faults than an ideal cache, they are reasonably competitive against an ideal cache with less memory. Please show that both the LRU and FIFO policies with a cache size of $2n$ blocks are $\frac{2}{1+1/n} \approx 2$-competitive against an ideal cache with a size of just n blocks, in terms of number of page faults on any memory access sequence. Assume both caches start out empty. As a hint, any sequence involving at most $2n$ faults for LRU and FIFO cannot fault on the same page twice, so consider consecutive fault sequences of this nature and argue about how many faults they cause on the ideal cache. [Solution]

10.2 Example: Compression with Huffman Codes

In the previous chapter, we studied several approaches for "lossless" data compression, allowing for decompression with no loss in data integrity. Given a string of symbols with non-uniform frequencies of occurrence, another popular method of lossless compression is to assign variable-length binary codes to the symbols, so that short codes correspond to symbols occurring more frequently. A convenient way to represent such a code is with a binary trie, shown in Figure 10.2(b). Each leaf represents a symbol, encoded with the binary string representing the path from the root down to the leaf; left edges are 0s and right edges are 1s. The resulting code is *prefix-free*, with no symbol's code being a prefix of any other. This allows for unambiguous decoding by simply walking down the tree as directed by the successive bits of an encoded string. Whenever we reach a leaf node, we emit its corresponding symbol and start over from the root. The process is quite efficient.

For each symbol i, let p_i denote the fraction of all locations in our string in which it appears, and let d_i denote its depth in the tree (i.e., the number of bits in its binary code). We would like to design a tree that minimizes $\sum_i p_i d_i$, the average number of bits per symbol used in our encoding[3]. It turns out this is easy to do with a simple greedy algorithm, generating what is known as a *Huffman code*. As seen in Figure 10.2(a), we start with each symbol being its own individual node, and we then repeatedly select two nodes having minimum frequency of occurrence and merge them together into a single subtree (whose frequency is their sum), repeating until we have built the entire tree. By storing all outstanding nodes in a binary heap keyed on frequency, construction takes only $O(n \log n)$ time, since each step involves two deletions and one insertion in the heap. An exchange argument shows that the resulting tree does indeed minimize $\sum_i p_i d_i$. [Details]

10.2.1 Brief Aside: Entropy Bounds and Arithmetic Coding

Take a random string of symbols, each independently generated according to a probability distribution p (symbol i being generated with probability p_i). Claude Shannon's famous *source coding theorem* relates the compressibility of this string to the entropy of p, $H(p) = -\sum_i p_i \log p_i$. It says that on average, $H(p)$ bits are necessary to represent each symbol or else data loss is likely. Huffman codes nearly achieve this bound, using $\sum_i p_i d_i \leq H(p) + 1$ bits per symbol on average [Proof]. For example, if p is the length-26 vector of letter frequencies in English text, then $H(p) \approx 4.165$, versus $\sum_i p_i d_i \approx 4.194$ for a Huffman tree[4].

To store a long n-digit base-10 number on a digital computer, it first needs to be converted to binary. The commonly-used *binary-coded decimal* format does this in perhaps the most straightforward fashion, using $\lceil \log_2 10 \rceil = 4$ bits to represent each digit separately. This takes $4n$ bits, even though writing the entire number in binary only takes $\lceil n \log_2 10 \rceil \approx 3.322n$ bits[5]. The slight loss comes from our insistence on using an integer number of bits for each symbol, the same reason we see the "+1" term in the Huffman coding bound above. Accumulated over an entire length-n string generated by a random source, Huffman coding uses at most

[3] This ignores the small amount of overhead involved in transmitting the encoding tree along with the encoded string, so the decoder knows the binary code for each symbol.

[4] English text isn't a string of randomly-chosen symbols though, so $H(p)$ here isn't a true lower bound on compressibility. Knowledge of, say, the preceding few characters lowers the entropy of the next symbol quite a bit. For example, the missing character in entrop_ is almost certainly 'y', and if we build a Huffman code not on single characters but on *bigrams* (pairs of adjacent characters) using their frequencies found in English text, this uses only about 3.73 bits per symbol on average. For simplicity, we are also ignoring the need to encode spaces and punctuation.

[5] On the other hand, if we write the entire number in binary, we lose the ability to easily examine individual digits in its base-10 representation, which the binary-coded decimal approach allows. Researchers have studied the problem of efficient encoding while still allowing $O(1)$-time inspection of individual digits; see the endnotes for further detail.

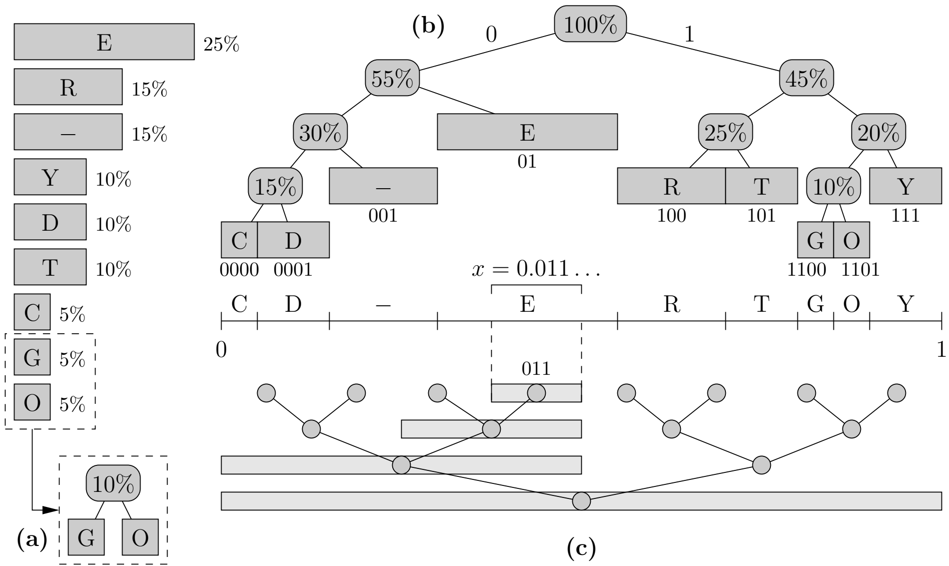


FIGURE 10.2: Building a variable-length binary code from the string 'TRY GREEDY TREE CODE'. In (a), we merge two nodes with minimum frequencies into a single subtree, the first step in building the Huffman tree shown in (b). This tree encodes 'TREE' by the binary string '1011000101'. In (c), we depict our probability distribution over symbols as a partition of the unit interval $[0, 1]$ (we've used the same order as the leaves of the Huffman tree for convenience in the figure, but the order here can be arbitrary). Here, any binary value $x = 0.011\dots$ codes for 'E', since this unambiguously belongs to E's interval. We can describe the same encoding with a binary radix tree built atop $[0, 1]$.

$nH(p) + n$ bits in expectation, compared to the theoretical lower bound of $nH(p)$. We can do slightly better using *arithmetic coding*, an alternative method that is sufficiently interesting and elegant that it is worth a brief diversion to explain.

As seen in Figure 10.2(c), imagine subdividing the unit interval $[0, 1]$ into subintervals representing our symbols, in arbitrary order, with lengths proportional to their frequencies. We can now encode a symbol with a point $x \in [0, 1]$, providing just enough digits of accuracy in x's binary representation so there is no ambiguity about the identity of the subinterval containing x. Having a random source is equivalent to generating $x \in [0, 1]$ uniformly at random[6]. Just as with Huffman coding, this encoding can be viewed in terms of a binary radix trie, where the digits of x are generated by walking down an implicit binary tree built on top of $[0, 1]$.

The true power of arithmetic coding lies not in coding individual symbols, but an entire length-n string, which we can abstractly view as a single "symbol" coming from a much more complicated probability distribution — imagine further subdividing each interval in Figure 10.2(c) to represent the second symbol, and so on. Despite its exponential size, the implicit structure of the resulting subdivision still allows

[6]The reader may want to revisit problem 26, which is equivalent to having a random source and just two subintervals, corresponding to heads and tails in a biased coin flip.

efficient encoding and decoding, using only $nH(p) + O(1)$ expected bits[7], essentially matching the lower bound above. This same approach also gives a nice method for base conversion of a number $x \in [0, 1]$. For example, to convert $x = 0.0110101$ in base 2 into base 7, we would set $p = [\frac{1}{7}\frac{1}{7} \ldots \frac{1}{7}]$ (recursively dividing the unit interval evenly by 7s, since our target is base 7), then we would begin decoding x, stopping at whatever precision is desired. [Further discussion]

Problem 169 (Adding Two Numbers in a Streaming Environment). Suppose two n-digit numbers in base $b = O(1)$ stream by, *most significant digit first*, and we want to output their sum, also in base b. Using similar tricks as above, show how to do this in a low-memory environment, using only $O(\log n)$ bits of storage. [Solution]

10.3 Example: Minimum Spanning Trees

Our next example comes from the domain of algorithmic graph theory. It is the well-known problem of computing a *minimum spanning tree* (MST) in a graph, a problem that demonstrates well the simplicity and elegance of greedy methods. Although there are many interesting things one could say about the MST problem, we'll focus our discussion at the moment on only those aspects which involve greedy techniques. In Chapter **??**, we cover this problem in much greater detail.

The input to the MST problem is a connected graph with n nodes and m edges, each edge having an associated cost. A spanning tree is a subset of edges of our graph that connects together (i.e., *spans*) all n nodes of our graph, and that contains no cycles. Our goal is to compute a spanning tree for which the sum of its edge costs is minimized. Figure 10.3 shows an example graph and its MST. A stereotypical application for the MST problem might be for a power company to connect a set of cities into a single circuit as cheaply as possible (here the cost of an edge between two cities represents the cost of building an electrical connection between the two). MSTs have many other applications in practice, and are often also used as building blocks for more sophisticated algorithms.

10.3.1 Kruskal's Algorithm

Perhaps the simplest and best-known algorithm for computing a minimum spanning tree is a greedy algorithm known as *Kruskal's algorithm*. The algorithm is very simple: add edges one by one in increasing order of cost, skipping over any edge that creates a cycle with edges we have added previously. The running time turns out to be dominated by the time required to sort the edges by cost, $O(m \log m)$, which is typically written as $O(m \log n)$ since $m \leq n^2$ (so $\log m \leq 2 \log n$) We won't provide further details into the implementation of the algorithm here, in particular how to answer "does this edge create a cycle?" queries in an efficient fashion, since this is discussed in detail in Chapter **??**. For now, we focus on proving that Kruskal's algorithm generates an optimal spanning tree, which we can show with a simple exchange argument. [Simple exchange argument]

[7]We are ignoring here the communication required to inform the other party about the frequencies of the symbols in our string (so the other party knows how $[0, 1]$ should be partitioned).

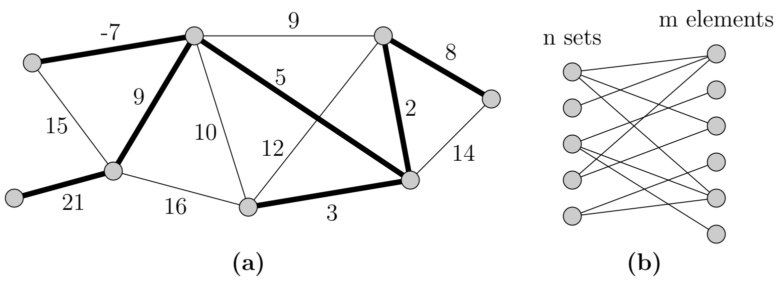


FIGURE 10.3: An example of (a) a graph and its minimum spanning tree (in bold), and (b) a set system drawn as a bipartite graph.

Problem 170 (Unique MST if Edge Costs are Distinct). Use an exchange argument to show that if all edges in a graph have distinct costs, then the graph has a unique minimum spanning tree. [Solution]

Problem 171 (Kruskal's Algorithm in Reverse). Consider the following "backwards" variant of Kruskal's algorithm: process edges in decreasing order of cost, deleting each edge in sequence unless this would cause the graph to become disconnected. Does this process yield an MST? [Solution]

10.3.2 Brief Aside: Matroids

Matroids are mathematical structures that capture and generalize many problems solvable by greedy algorithms. For example, Kruskal's algorithm and its correctness follow directly from the fact that the minimum spanning tree problem can be modeled by a *graphic* matroid.

More precisely, a matroid is a special type of *set system*, collection of n sets over a universe of m elements. The author likes to visualize set systems in terms of a bipartite graph, as shown in Figure 10.3(b). In order for a set system to be a matroid, it must be *hereditary* (if set A belongs to the system, then so must all of its subsets), and it must satisfy the *augmentation property*: if A and B are sets in the system and $|A| < |B|$, then there must exist an element in $B - A$ we can add to A to obtain another set in the system. To give a concrete example, the elements in a *graphic matroid* are the m edges in a graph, and a set of edges belongs to our system if it contains no cycles (i.e., if the set is a forest). Here, the augmentation property tells us that if we take two acyclic sets of edges A and B with $|A| < |B|$, then we can always find some edge $e \in B - A$ such that $A \cup \{e\}$ also contains no cycles. [Simple proof of this property]

If we associate a value with every element in a set system, then we can locate a set of maximum value using a simple incremental greedy algorithm: start with an empty set, process the elements in decreasing order of value, adding each element in sequence to our current set unless this would create a set that doesn't belong to the system. The astute reader will notice that this algorithm is exactly the same as Kruskal's algorithm, only applied in the somewhat more general setting of a matroid. Its analysis is identical to that of Kruskal's algorithm, except phrased more generally in terms of sets and elements instead of spanning trees and edges[8] [Details]. Kruskal's algorithm is just a special case of this more general algorithm run on a graphic matroid[9].

Problem 172 (Minimum Spanning Pseudotree). A *pseudotree* is either a tree or a tree plus one additional edge that forms a unique cycle. A *pseudoforest* is a collection of pseudotrees, just like a forest is a collection of trees. In the same way the graphic matroid is built from all the forests in a graph, the pseudoforests in a graph also form a matroid, known as a *bicircular* matroid. Please prove that this is indeed a matroid, and show how we can use this to build a greedy algorithm — similar in flavor to Kruskal's algorithm — that computes a minimum-cost spanning pseudotree of any connected graph. Do not worry about the running time of this algorithm; we will see later in problem **??** that we can surprisingly solve this problem faster than we currently know how to compute an MST! [Solution]

Matroids are an important class of objects in the study of combinatorial optimization, since they provide an abstract general framework that captures many prominent problems. They can help you prove certain results in a single step that would have otherwise taken substantially more work. Problem **??** in Chapter **??** gives another glimpse into their utility.

10.4 Example: Matching Problems

Matching problems involve pairing up objects in some optimal fashion. They come in many flavors, and we will later spend all of Chapter **??** studying them in depth. Since only a few matching problems can be optimally solved by a greedy algorithm, the main point of this section is to highlight the use of greedy methods as approximation algorithms. As a warm-up, let's start with some simple matching problems that *do* have optimal greedy solutions:

Problem 173 (Ballroom Matching). You are a dance instructor for a class of n boys with heights $b_1 \ldots b_n$ and n girls with heights $g_1 \ldots g_n$. You need to decide on a matching of the boys and girls (i.e., a one-to-one pairing up between the boys and girls), in which boys and girls of similar heights end up together.

(a) Give a greedy algorithm which finds a matching that minimizes the maximum height difference over all pairs of boys and girls. Justify its correctness. [Solution]

(b) Give a greedy algorithm which finds a matching that minimizes the sum of height differences over all pairs of boys and girls[10]. Justify its correctness. [Solution]

[8]It is interesting to note that this is a bi-directional result, as one can show that if this greedy algorithm works, then our underlying set system must be a matroid.

[9]Technically, the matroid approach corresponds to the variant of Kruskal's algorithm that computes a *maximum* spanning tree by choosing edges from highest cost to lowest cost. However, the maximum spanning tree problem is essentially the same as the minimum spanning tree problem; the two can be transformed into each-other by negating the costs of all edges.

[10]An interesting and surprisingly more challenging variant of this problem occurs when there are fewer boys than girls, and every boy must be matched (so a subset of girls will end up unmatched). For an $O(n \log n)$ solution to this more complicated variant, see problem **??**.

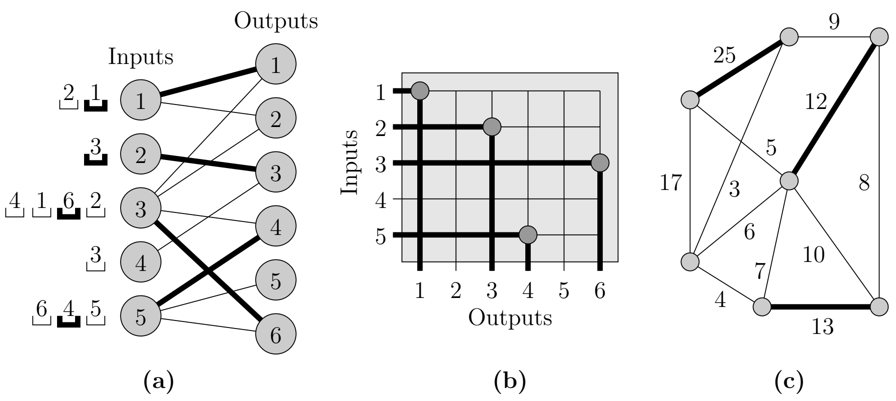


FIGURE 10.4: Examples of problem instances for (a-b) maximum-cardinality matching in a bipartite graph and (c) maximum-value matching in a general graph. In (a) and (b), we show how bipartite matching is used inside a "crossbar switch" in a communication network: (a) depicts the problem as a bipartite graph, and (b) shows the physical wiring pattern in the switch. Packets of information arrive and queue up at the input ports of the switch, each destined for a specific output port. In each time step, each input port can accept one packet from anywhere in its queue, and each output port can send at most one outgoing packet. To maximize the throughput, we therefore want to route as large a matching as possible in each time step.

Problem 174 (Matching Students). You are a science teacher with $2n$ students. Through working with the students over time, you've been able to determine the intelligence level of each student (i.e., each student has an associated number which represents his/her intelligence). For their next project, the students must be matched together into n teams of two students each. You'd like to make this matching as fair as possible, so no team is exceptionally smart or exceptionally "nonsmart". Describe (and analyze) a greedy algorithm which computes a matching that minimizes the combined intelligence of the smartest team. If you've picked the right algorithm, you should find that it also simultaneously maximizes the combined intelligence of the weakest team. [Solution]

10.4.1 Matchings in Graphs

Matching problems are usually described in the context of a graph: nodes indicate objects to be matched, and edges represent valid pairings between these objects, possibly with associated values. If we are pairing up elements between two different sets, the graph is bipartite, as shown in Figure 10.4(a). Since each node can be matched with at most one other node, a matching is just a subset of edges, no two of which share a common endpoint.

Two common objectives are to compute either *maximum-cardinality* or *maximum-value* matchings, where the goals are respectively to include as many edges or as much total value as possible. Examples are shown in Figure 10.4. Both problems can be optimally solved in polynomial time, as we shall see in Chapter **??**, although the algorithms in question are somewhat complicated and not quite fast enough for some applications, such as the example of crossbar switches in communication networks shown in Figure 10.4. Using greedy methods, we can compute approximately-optimal solutions to these problems much more efficiently.

10.4.2 Greedy Approximation Algorithms

Since greedy algorithms are so simple and fast, they are often applied in practice even when it is understood that they might not always compute an optimal solution. They still tend to generate solutions that are reasonably good in many cases. As we will see in Chapter 13, many heuristics for speeding up exhaustive searches (e.g., branch and bound) employ greediness to improve performance.

Even if a greedy algorithm seems to perform well in practice, we might still like to have a rigorous guarantee on how badly it might perform (relative to an optimal solution) in the worst case. As an example, consider the following greedy algorithm for maximum-cardinality matching: start with an empty matching, and as long as there exists an edge whose endpoints are not already matched, pick any such edge and add it to our matching. This algorithm is extremely simple and fast, running in $\Theta(m)$ time, and we can show that it is a 1/2-approximation algorithm — it always computes a matching that contains at least half the number of edges of an optimal matching. We can prove this fact by performing an exchange argument, just as with many of our preceding greedy algorithms. In this case, however, we will lose up to a factor of two during the process of modifying an optimal solution so it coincides with the greedy solution [Proof]. In Chapter **??**, we will see in problem **??** how to generalize this approach to build a $(1 - \varepsilon)$-approximation algorithm that runs in $\Theta(m/\varepsilon)$ time, which is $\Theta(m)$ time as long as ε is constant.

Problem 175 (Minimum Node Cover). The well-known minimum node cover problem (also commonly called the minimum vertex cover problem) is the following: given a graph, select a subset of as few nodes as possible which collectively cover all of the edges of the graph (i.e., every edge must have at least one of its endpoints included in the cover). The problem is NP-hard, but we can approximate its solution very efficiently. Suppose we run the linear-time greedy algorithm above for approximate maximum-cardinality matching, and select both endpoints of every edge in its output. Argue that (i) this set is a feasible cover, and that (ii) it contains no more than twice as many nodes as an optimal cover. [Solution]

Problem 176 (Approximate Maximum-Value Matching). Consider the problem of finding a maximum-value matching in a graph. Describe a natural greedy algorithm that computes a matching with value at least half that of an optimal matching. Your algorithm should run in $O(m \log n)$ time. [Solution]

10.5 Example: Packing and Covering Problems

Suppose you arrive at your first day of orientation at a new university, and are presented with a variety of activities in which you may participate. Each of these n activities occupies some interval of time during the day. Since some of them overlap, you may not be able to attend them all; however, you would like to take part in as many as possible. In other words, we would like to select a subset

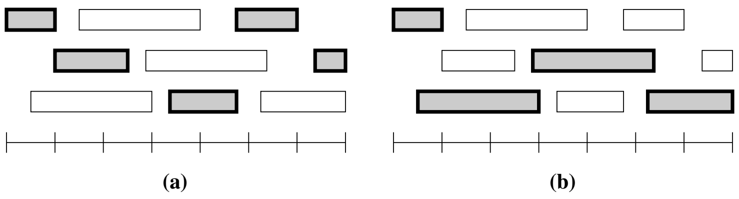


FIGURE 10.5: Examples of problem instances for (a) the maximum-cardinality interval packing problem (otherwise known as the activity scheduling problem), and (b) the minimum-cardinality interval covering problem. An optimal solution for each instance is shaded. Note that these intervals are only one-dimensional, even though we have arranged them with a certain amount of vertical spacing for visual clarity.

of pairwise-disjoint intervals having maximum cardinality. This is known as the *activity selection problem*, and it is commonly used as an example to illustrate the process of designing a greedy algorithm. Figure 10.5(a) shows an instance of this problem along with its optimal solution.

Activity selection is a *packing* problem, involving packing an optimal set of elements into our solution subject to a restriction that elements in the solution must not conflict with each-other. More formally, the activity scheduling problem could be called a *maximum-cardinality interval packing problem*. The matching problems we just studied are also packing problems, involving packing disjoint edges in a graph.

Problem 177 (Optimally Solving the Activity Scheduling Problem). Consider a greedy algorithm which repeatedly selects, from all remaining intervals (those still disjoint from our solution so far), an interval ending the earliest. Prove that this algorithm gives an optimal solution to the activity selection problem, and show how to implement it in $O(n \log n)$ time. [Solution]

Problem 178 (Approximating the Activity Scheduling Problem). Consider a greedy algorithm which repeatedly selects, from all remaining intervals (those still disjoint from our solution so far), an interval of smallest duration. Show that this may not give an optimal solution to the activity scheduling problem, but that it gives a solution containing at least half the number of intervals in an optimal solution. [Solution]

Problem 179 (Overlap Counts). Define the *overlap count* of an interval to be the number of other intervals it intersects (two intervals touching only at an endpoint do not overlap). For the activity-scheduling problem, consider the greedy algorithm which repeatedly selects, from all remaining intervals (those still disjoint from our solution so far), an interval of minimum overlap count. Does this algorithm give an optimal solution? Note that there could be several intervals that tie for the minimum overlap count — in this case, is it important how we break the tie? [Solution]

A related and equally important class of problems is that of *covering* problems. Suppose, for example, that you own a swimming pool and need to hire lifeguards to watch the pool all day long. This problem has the same input as the activity selection problem: a set of n intervals, each representing the duration of time covered by a lifeguard that you could potentially hire. Since lifeguards are expensive, you want to hire the minimum number of lifeguards while still ensuring that every point in time is covered by at least one lifeguard. Formally, we would call this a *minimum-cardinality interval cover problem.*

Whereas packing problems are maximization problems, (e.g., packing as many disjoint elements as possible), covering problems involve minimization (e.g., covering something with as few elements as possible). Both types are frequent targets for greedy methods. When solving a packing problem, we want to repeatedly select "good" items (perhaps of small size, or perhaps of high value with respect to their size) that are disjoint from those already selected. When solving a covering problem, we want to repeatedly select items that cover most efficiently whatever it is that remains to be covered (e.g., maximizing the amount of additional coverage, or covering additional elements at minimum cost per element).

Problem 180 (Minimum-Cardinality Interval Cover). Develop a greedy algorithm for finding a minimum-cardinality interval cover. [Solution]

Problem 181 (Minimum-Area Interval Cover). Suppose that instead of paying a fixed amount of money per lifeguard, you pay each lifeguard an amount of money proportional to the duration of the interval of time when he/she is on duty. Therefore, you would like to solve the *minimum-area interval covering problem*: select a subset of intervals which covers the entire day, but which minimizes the sum of their lengths (the *area* of the cover). Although this problem can be optimally solved by dynamic programming, we can use our solution to the minimum-cardinality interval covering problem to obtain a good approximation. Show that a minimum-cardinality cover has an area at most twice that of the minimum-area cover for the same problem instance. [Solution]

10.5.1 Brief Aside: Duality

In the context of optimization, *duality* is a min-max relationship between two intrinsically-related "dual" problems, one involving minimization and the other maximization. Usually, the two problems are related via an inequality of the form

$$\text{Maximum solution of problem A} \leq \text{Minimum solution of problem B},$$

where sometimes we even have equality between the two. Duality relationships are a useful tool for certifying optimality or near-optimality of solutions. If we can find solutions for A and B that are equal in value, then the inequality above implies that both must be optimal for their respective problems. If we can find solutions differing in value by only Δ, then we know each solution is at most Δ units away from being optimal for its respective problem. For a nice concrete example, consider the following problem:

Problem 182 (Minimum Multicut on a Path). The input to this problem a set of n intervals on a number line, each representing a communication session between its two endpoints taking place along a shared communication line. Your task is cut the line in a minimum number of locations so as to disconnect all n sessions. Each session must be cut at a point strictly between its endpoints. Devise a greedy algorithm which optimally solves this problem. [Solution]

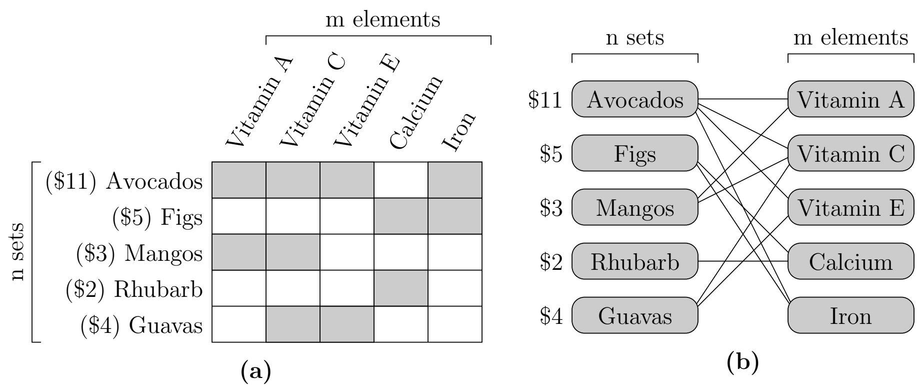


FIGURE 10.6: Example instance of a set system for the minimum-cost set cover problem, shown (a) in tabular form and (b) as a bipartite graph. The optimal solution $\{\text{Figs}, \text{Mangos}, \text{Guavas}\}$ covers all nutrients at minimum cost.

This problem is dual to the activity scheduling problem, enabling an elegant alternative means of proving optimality of our greedy algorithms for both problems. Given a set of input intervals, let I be the set of intervals in any feasible solution to the activity scheduling problem, and let C be the set of cuts in any feasible solution to the minimum multicut problem above. It is easy to see that $|I| \leq |C|$, since the intervals in I are disjoint from each-other, so each one requires its own separate cut if we are to separate them all. If we can come up with solutions for which $|I| = |C|$, then it follows that both must be optimal for their respective problems. It is no accident that both problems have very similar greedy algorithms. By examining these closely, we see that they do indeed generate such a pair of equal solutions, so both algorithms are therefore optimal. [Further elaboration]

Duality between two problems can often highlight useful structural relationships between the two, and often helps lend insight into the development of algorithms as well. This will be particularly true when we study linear programming duality in Chapter 12, where we shall see that the linear programming formulations of packing and covering problems are indeed duals of each-other.

10.5.2 General Covering and Packing Problems

Covering and packing problems abound in practice. We have seen several simple example above, and we will introduce many more throughout this book, particularly in Chapter **??** when we survey common problems on graphs. Here, we discuss the *set cover* and *set packing* problems[11], high-level problems that capture most other covering and packing problems because their input is a generic set system, with n sets over a universe U containing m elements. Due to their generality, both prob-

[11]A helpful hint on deciphering problem names: a "minimum X cover" problem asks us to compute a cover composed of Xs. For example, a minimum interval cover is a minimum cover comprised of intervals, a minimum node cover is a minimum cover comprised of nodes, and a minimum set cover is a minimum cover comprised of sets. The same convention holds for packing problems as well.

lems are NP-hard. They can both be approximated via simple greedy techniques, although neither problem allows for very strong approximation guarantees.

Set Cover. The minimum-cardinality set cover problem asks us to choose a minimum collection of sets whose union is U (i.e., a collection of sets that covers the universe of all elements). If our sets have associated costs, we can also ask for a set cover of minimum total cost. Figure 10.6 shows an example of the minimum-cost set cover problem involving the selection of a minimum-cost group of foods that covers a universe of essential nutrients.

Set cover contains all of our previous covering problems as special cases. For example, in the node cover problem, U is the set of all edges of a graph, and there is one set for every node containing the edges incident to that node. For the interval cover problem, we can take U to be the set of all interval endpoints, and each interval corresponds to the set of endpoints covered by that interval.

A simple greedy algorithm can approximate the set cover problem reasonably well. Let's define the *per-element cost* of each input set as its cost divided by the number of elements it covers (if we've already built a partial solution, we only count those elements not already covered by the partial solution). A natural greedy algorithm is to repeatedly select a set of minimum per-element cost, until all elements in U are covered. This always yields a solution whose cost is at most $1 + \ln m$ times the cost of an optimal solution [Short proof]. It turns out that this is essentially the best one can hope to accomplish, since it is actually NP-hard to achieve an approximation factor better than $c \log m$ for some constant $c > 0$.

Set Packing. The maximum-cardinality set packing problem asks for a collection of as many sets as possible that are all pairwise disjoint. A more general variant is the maximum-value set packing problem, where we assign values to our sets and seek a maximum-value collection of pairwise disjoint sets. Both of these problems are not only NP-hard, but also hard to approximate to within a factor of $n^{1/2-\varepsilon}$ for any positive ε as long as P $\neq$ ZPP (a conjecture about the complexity classes P and ZPP that is widely believed to be true, just like the more famous conjecture that P $\neq$ NP). On the positive side, however, we can match this approximation bound using simple greedy algorithms, an exercise we leave for the reader:

Problem 183 (Maximum Set Packing). In this problem, we analyze simple greedy approximation algorithms for set packing problems.

(a) For the maximum-cardinality set packing problem, consider the greedy algorithm that repeatedly selects a minimum-cardinality set disjoint from those already selected. Show that this is a $\frac{1}{\sqrt{m}}$-approximation algorithm, where $m = |U|$. [Solution]

(b) Please try to find a simple $\frac{1}{2\sqrt{m}}$-approximation algorithm for the maximum-value set packing problem. If you are feeling ambitious, see if you can improve the performance guarantee to $\frac{1}{\sqrt{m}}$. [Solution]

Problem 184 (The Minimum Test Collection Problem). Suppose you have n different diseases and m different tests, where each test returns "positive" for some subset of the diseases, and "negative" for the other diseases. Given which diseases produce positive and negative test results for each test, we would like to find the smallest possible subset of the tests that suffices to identify all of the diseases. In other words, for each two diseases x and y, there should be at least one test in our subset that gives different results for x and y — otherwise we could not tell x and y apart. See if you can design a greedy $O(\log n)$-approximation algorithm for this NP-hard problem. [Solution]

Problem 185 (The Maximum Coverage Problem). Another NP-hard problem very closely related to the set cover problem is the *maximum-value coverage problem*. The input to the problem consists of a collection of n sets over a size-m universe, where each element in the universe has an associated value $v_1 \ldots v_m$. We are only allowed to select k of these sets, however, and we would like to select the sets so that the value of the elements in their union is as large as possible. In other words, we want to cover the greatest amount of value using only k sets.

(a) Show that the natural greedy algorithm for this problem gives a $(1-1/e)$-approximate solution. [Solution]

(b) The minimum set cover problem with a size-m universe is NP-hard even to approximate to within a factor of $c \log m$, for some constant $c > 0$. Please show that this implies that the maximum coverage problem is NP-hard to approximate to within a factor of $1 - 1/e + \varepsilon$ for any $\varepsilon > 0$, by showing how one can develop an approximation algorithm for minimum set cover based on an approximation algorithm for maximum coverage. [Solution]

10.6 Additional Problems

After reading an entire chapter containing nothing but problems solvable by greedy algorithms, one may be tempted to go about trying to apply greedy algorithms to every problem in sight. Unfortunately, while many problems certainly have greedy solutions, many others do not. The reader is now faced with the challenge of differentiating between these types of problems. Our hope is that the intuition and collection of techniques discussed in this chapter will simplify this task. This task will also be easier after reading the next chapter, highlighting problems on which greedy techniques fall short.

Problem 186 (Minimum Node Multicut on a Tree). A nice generalization of the preceding problem involves a communication network shaped like a tree rather than a path. Suppose you are given a set of k communication sessions that are taking place in an n-node tree. Each session is described simply by a pair of nodes (i, j), since in any tree there is a unique path between any two nodes.

(a) It is NP-hard problem to find a minimum set of edges whose removal disconnects all k communication sessions. Just for fun, prove this fact by showing that if you could solve this problem in polynomial time, even on a star-shaped tree (a tree with one central node connected to n leaves), then you could easily compute a minimum node cover (problem 175) in polynomial time. [Solution]

(b) Fortunately, the problem of finding a minimum set of *nodes* in a tree whose removal disconnects all k sessions is much easier. Can you solve this problem in $O(n + k)$ time using a greedy algorithm? [Solution]

Problem 187 (Maximum Multicommodity Flow on a Path). As in problem 182, suppose we have n communication sessions (described by intervals) that we would like to route through a one-dimensional network (think of it as a number line) with C units of capacity everywhere on the line. Session i involves the transmission of d_i units of data per unit time from point x_i to point y_i. For each session i, we can choose to disable it, enable it fully, or to transmit data at only a fraction of the maximum rate d_i. If we enable session i at a rate rd_i ($0 \leq r \leq 1$), this will consume rd_i units of capacity only on the segment between x_i and y_i, leaving the rest of the network unaffected. Our goal is to maximize the total sum of transmission rates of all sessions. Please show how to solve this problem in $O(n \log n)$ time with a simple greedy algorithm. [Solution]

Problem 188 (The Quiz Problem). You are taking an oral quiz with n questions, where each question j has an associated point value v_j. It is up to you to decide the order in which you will attempt the problems. Each problem in sequence that you answer correctly contributes to the total number of points you earn for the quiz, and the quiz terminates upon your first wrong answer (and you receive no points for this wrong answer). Suppose you have estimated the probabilities $p_1 \ldots p_n$ with which you can correctly solve each of the n problems. How do you choose an ordering of the questions that maximizes the expected amount of value when you take the quiz? What if you receive points for final question answered incorrectly as well? As a hint (which applies to several greedily-solvable problems), you can sometimes find the right greedy choice rule by "reverse engineering" an exchange argument — that is, go through the motions of making an exchange argument with an unspecified greedy choice rule, allowing us to ultimately determine the initial rule that would make the argument work. [Solution]

Problem 189 (Boxes of Donuts). You are the proud owner of a donut shop. At the end of the day, you are left with n total donuts, a_i donuts of type i, for $i = 1 \ldots k$, where $\sum a_i = n$. Right before closing, you receive an unusual order: as many boxes of donuts as possible, where each box must contain T donuts all of different types. Show how to optimally fill this order using a greedy algorithm. [Solution]

Problem 190 (Shortest Non-Subsequence). Given a length-n string S, please show how to compute in $\Theta(n)$ expected time the shortest string T using only characters from S that does not appear as a subsequence of S. [Solution]

Problem 191 (The Shopping Bag Problem). Suppose we wish to place n items in a shopping bag, each with an associated *weight* and *strength*. The items in the bag are stacked vertically on top of each-other, and we would like to minimize the risk of squashing any item. In particular, we define the *risk* of squashing an item to be the combined weight of all items stacked above it minus its strength. Devise a greedy algorithm which stacks the items so as to minimize the maximum risk. Does this problem remind you of any other greedy problem we've seen already? [Solution]

Problem 192 (Ballroom Matching Revisited). Recall the Ballroom Matching problem from section 10.4, in which we are given the heights of n boys and n girls as input, and we would like to construct a matching between the boys and girls so each pair has similar height.

(a) Suppose we are not allowed to pair up a boy and a girl if their heights differ by more than H. It may no longer then be possible to match all the boys and girls. Describe a greedy algorithm that constructs a matching of maximum cardinality. [Solution]

(b) Suppose the boys and girls are picky about the speed of music to which they will dance. Specifically, we are given for each boy i a range $[b_i, B_i]$ of speeds to which he is willing to dance, and for each girl j we are given a range $[g_j, G_j]$ of speeds to which she is willing to dance. Given that one may only pair up a boy and a girl if their "tempo intervals" overlap, devise a greedy algorithm that computes a maximum-cardinality matching. [Solution]

(c) Suppose, for a difficult dance involving lifts, that each girl must be matched to a boy that is (i) at least as tall as her, and (ii) at least as heavy as her. Given the heights and weights of the boys and girls, please show how to match as many couples as possible, and also show how to implement your algorithm in only $O(n \log n)$ time. [Solution]

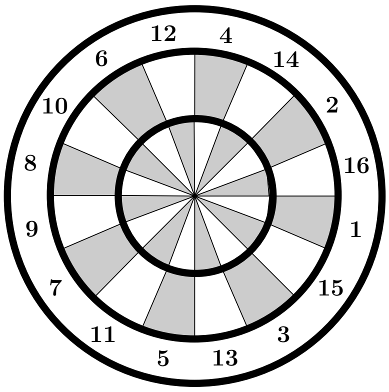


FIGURE 10.7: An interesting way to number the sectors of a dartboard.

Problem 193 (Interval Packing and Covering Revisited). Here are a few more variants of the interval packing and covering problems introduced in Section 10.5.

(a) Consider again the maximum-cardinality interval packing problem (a.k.a. the activity scheduling problem) and the minimum-cardinality interval covering problem. For both of these problems, suppose our intervals are part of a circle rather than a line segment. How efficiently can you solve these problems now? [Solution]

(b) Suppose we wish to solve a generalization of the activity-scheduling problem for k individuals, so at most k activities can be attended at any one point in time. Show how to select the maximum number of activities in this setting. Also, consider the analogous generalization of the minimum-cardinality interval covering problem, when every point must be covered not once, but by at least k different intervals. Give an efficient greedy solution for this problem. [Solution]

(c) Suppose we need to reserve rooms for a conference. There will be n meetings during the conference, provided to us as a set of time intervals $[a_1, b_1], \ldots, [a_n, b_n]$. Two meetings that overlap in time cannot be scheduled in the same room. Give a simple greedy algorithm that determines an assignment of meetings to rooms that uses the minimum possible total number of rooms. Viewed differently, this problem is the same as that of packing 2-dimensional axially-aligned rectangles in a disjoint fashion. Each rectangle has fixed starting and ending x coordinates as well as unit height (in the y dimension), and we can slide the rectangles around in the y direction so as to pack them into a strip having minimum y extent. [Solution]

Problem 194 (Designing a Dartboard). A fun problem is that of assigning scores to the sectors of a dartboard. For obvious reasons, a particular arrangement of scores is more "interesting" if there tends to be a large difference between the scores of adjacent sectors. More formally, let's say we wish to maximize the sum of squares of the score differences over all pairs of adjacent sectors. Given a set of n scores $s_1 \ldots s_n$ to allocate to the n sectors of the dartboard, show how to optimally arrange these scores with a greedy algorithm. [Solution]

Problem 195 (The Fleet Range Problem). Suppose you have a fleet of n vehicles with fuel capacities $c_1 \ldots c_n$ and fuel consumption rates $r_1 \ldots r_n$. All vehicles travel at the same speed and start at a common location. You would like to send the vehicles out in a convoy such that all of them eventually return back to the starting point. Any vehicles leaving the convoy early can donate excess fuel to the remaining vehicles before turning back (and this is the only time it can exchange fuel with the other vehicles). Devise a greedy algorithm that determines the greatest possible distance such a convoy can reach. [Solution]

Problem 196 (Making Change). This is a classic greedy problem. Given a set of n coin denominations, we wish to make change for an amount of money C using as few coins as possible. All coin denominations and the value of C are integers, and you may assume that the smallest denomination has value 1 (thus ensuring that a solution always exists). The problem is NP-hard, but a greedy algorithm gives an optimal solution for several common special cases. Show that a greedy algorithm optimally solves the problem if each denomination evenly divides the next-highest denomination. [Solution]

Problem 197 (The Uncapacitated Exact Transportation Problem with Only Two Sources). Suppose we have 2 factories and n warehouses. Our factories supply exactly s_1 and s_2 units of goods, respectively, and warehouses $1 \ldots n$ have demands of $d_1 \ldots d_n$, with $\sum d_i = s_1 + s_2$. There is a cost c_{ij} per unit of goods shipped from factory i to warehouse j. Our goal is to determine a minimum-cost assignment from factories to warehouses so that exactly s_i units depart from factory i and exactly d_j units arrive at warehouse j. This is a special case of a more general problem known as the *transportation* problem, which can be solved using a variety of methods (see problem 224 for more details). Please give an optimal $O(n \log n)$ greedy algorithm for this problem. [Solution]

Problem 198 (Two-Hub Hierarchical Routing). You are given as input n points in the 2D plane. Two of these points must be designated as *hubs*, and each point must be assigned to exactly one of these hubs, so as to minimize the total pairwise *routing cost* summed up over all pairs of points. If two points are connected to the same hub, their pairwise routing cost is the sum of their distances to that hub. If they are connected to different hubs, their routing cost is the sum of the distances from the points to their respective hubs plus the distance between the two hubs. Please show how to solve this problem optimally in $O(n^3 \log n)$ time; the solution of the preceding problem may be of some help. [Solution]

Problem 199 (The Knapsack Problem). This is another classic problem. We are given a knapsack of integer capacity C and n items, each with a positive size and a positive value (neither of which is necessarily an integer). Each item taken by itself is small enough to fit in the knapsack. We would like to select a maximum-value subset of the items that fits in the knapsack.

(a) Please show that the knapsack problem can be optimally solved in $O(n \log n)$ time with a simple greedy algorithm if either all items have unit size, or all items have unit value. Can you improve the running time to $\Theta(n)$? [Solution]

(b) Suppose that we can put fractions of items in the knapsack (this is known as the *fractional knapsack problem*). Of course, if we put only, say 30% of an item in the knapsack, we only get to count 30% of its value towards the objective value of our solution. Give an $O(n \log n)$ greedy algorithm which optimally solves this special case of the problem, and then try to reduce its running time to $\Theta(n)$. [Solution]

(c) Consider now the 0/1 *knapsack problem*, where we are only allowed to place zero or one copies of each item in the knapsack; that is, we cannot fractionally add items to the knapsack. Describe a simple greedy 1/2-approximation algorithm for this problem with running time $O(n \log n)$. Can you improve the running time to $\Theta(n)$? [Solution]

(d) Try to build an $O(n \log n)$ algorithm that obtains a 2/3-approximate solution for the 0/1 knapsack problem. As a hint, "guess" the single item in the solution having maximum value, and then apply the preceding algorithm as quickly as possible (using an appropriate data structure may help). [Solution]

(e) A common generalization of the 0/1 knapsack problem is known as the *multiple knap-*

sack problem, where as input we are given not one knapsack, but m different knapsacks with capacities $C_1 \ldots C_m$. The objective is now to maximize the value we can collectively pack into all of the knapsacks. Devise a simple greedy 1/4-approximation algorithm for the multiple knapsack problem. [Solution]

(f) The *knapsack cover problem* is in some sense the opposite of the 0/1 knapsack problem. We have a collection of n items each with a positive size and cost, and we would like to compute a minimum-cost subset of these items whose total size is at least as big as a some specified "capacity", C. For example, the items could be types of food where size corresponds to nutritional content. In this case we seek a minimum-cost subset of the foods that collectively provides sufficient nutritional value. Give a simple greedy algorithm that solves the natural fractional version of this problem, and then see if you can obtain a 2-approximation algorithm for this problem. [Solution]

(g) An interesting (also NP-hard) variant of the knapsack problem is one that allows us to overfill the knapsack by at most a single item. In other words, a solution is feasible if removing its largest item allows the remaining items to fit within capacity. This type of problem might occur in a situation where we are packing a container such as a railway car with an open top. Can you devise a simple 2-approximation algorithm for this variant? [Solution]

Problem 200 (SONET Ring Loading). A great deal of traffic on the Internet is currently carried by Synchronous Optical Networks (SONETs), which are often laid out in the shape of a circular ring. We would like to route n communication sessions through such a network. Each session is described by the location of its two endpoints on the circle, and the amount of demand (i.e., information) that must be routed between these points. We can route each session between its endpoints in one of two ways: clockwise or counter-clockwise, and we would like to route each connection so as to minimize the maximum *load* over the entire network. The load of a point on the network is defined to be the total demand being routed through that point (we don't count demand for sessions ending exactly at that point). This problem is NP-hard (although it can be solved in polynomial time if all demands are one). Show that we can approximate the optimal solution within a factor of 2 using a simple greedy algorithm. [Solution]

Problem 201 (Bin Packing). In the well-known *bin packing* problem, we are given a collection of n items and their associated sizes (all no larger than 1). Our goal is to pack the items into unit-sized bins while using as few bins as possible. The problem is NP-hard.

(a) Show how to approximate the optimal solution within a factor of 2 using a greedy algorithm. [Solution]

(b) Consider the following simple greedy algorithm: process the items in decreasing order by size, attempting to fit each one into any bin already in use. If an item doesn't fit into any of these, open a new bin for the item. This algorithm gives a solution that uses at most $(3/2)OPT + 1$ bins, where OPT denotes the number of bins required by an optimal solution. For a challenge, try to prove this. [Solution]

Problem 202 (Bin Covering). Given n items with varying sizes $s_1 \ldots s_n$, the *bin covering* problem asks us to place these items into a maximum number of unit-size bins so that every bin ends up filled at least to its capacity. For example, suppose the n items represent different pieces of candy, and we would like to give away a maximum number of bags of candy where each bag is at least of some specified fullness. This problem is NP-hard; however, show how to approximate its optimal solution within a factor of 2 using a greedy algorithm. [Solution]

Problem 203 (Minimum-Makespan Scheduling on Parallel Machines). Suppose we have n jobs, each with associated processing times $p_1 \ldots p_n$, and we wish to schedule them by assigning each to one of m available machines. Our objective is to minimize the *makespan* of the schedule, where the makespan is defined as the maximum completion time over all jobs (the time at which the most-heavily-loaded machine completes its processing). This well-known scheduling problem is NP-hard, but it can be approximated using greedy methods.

(a) Consider the greedy algorithm which processes jobs in arbitrary order, assigning each in sequence to the machine which is currently the least heavily loaded. Show that this algorithm produces a schedule having at most twice the optimal makespan. [Solution]

(b) A bit trickier: Show how the same algorithm delivers a schedule of makespan at most 4/3 times optimal if it schedules the jobs in decreasing order of processing time. It will help to distinguish *large* jobs (with processing time larger than one third of the optimal makespan) from the remaining *small* jobs. [Solution]

(c) In an *open shop* scheduling environment, every job must be processed by every machine. This might correspond, for example, to a factory in which each of the m machines performs some useful function in assembling a produce. A machine can still process only one job at a time, a job can only be processed by one machine at a time, and the jobs do not need to visit machines in any specified order (as is the case with more complicated "shop" scheduling environments). In a manner very similar to part (a), devise and analyze a greedy 2-approximation algorithm for this problem. [Solution]

Problem 204 (Successive Divisibility). For many NP-hard problems, greedy algorithms are known to optimally solve certain special cases. In this problem we investigate special cases of several NP-hard problems that involve input elements with successively divisible sizes (see also problem 196 on making change, for another example of this type of special case).

(a) For minimum-makespan scheduling (problem 203, suppose the processing time of each job evenly divides the processing time of the next-largest job. For the knapsack problem (problem 199), suppose each item size evenly divides the size of the next-largest item, and further suppose you can place multiple copies of the same item into the knapsack. For bin packing (problem 201) and bin covering (problem 202), suppose each item size evenly divides the size of the next-largest item. Please give optimal greedy algorithms for each of these special cases. [Solution]

(b) With each of the four proceeding problems, show how to obtain a 2-approximation algorithm for arbitrary problem instances by first rounding item sizes to nearby powers of 2, optimally solving the rounded instance, and then reverting item sizes back to their original values. [Solution]