14

Combinatorial Problems

We now consider several algorithmic problems of a purely combinatorial nature. These include sorting and permutation generations, both of which were among the first non-numerical problems arising on electronic computers. Sorting can be viewed as identifying or imposing a total order on the keys, while searching and selection involve identifying specific keys based on their position in this total order.

The rest of this section deals with other combinatorial objects, such as permutations, partitions, subsets, calendars, and schedules. We are particularly interested in algorithms that *rank* and *unrank* combinatorial objects—i.e. , that map each distinct object to and from a unique integer. Rank and unrank operations make many other tasks simple, such as generating random objects (pick a random number and unrank) or listing all objects in order (iterate from 1 to n and unrank).

We conclude with the problem of generating graphs. Graph algorithms are more fully presented in subsequent sections of the catalog.

Books on general combinatorial algorithms, in this restricted sense, include:

- *Nijenhuis and Wilf* [NW78] – This book specializes in algorithms for constructing basic combinatorial objects such as permutations, subsets, and partitions. Such algorithms are often very short but hard to locate and usually are surprisingly subtle. Fortran programs for all of the algorithms are provided, as well as a discussion of the theory behind each of them. See Section 19.1.10 for details.

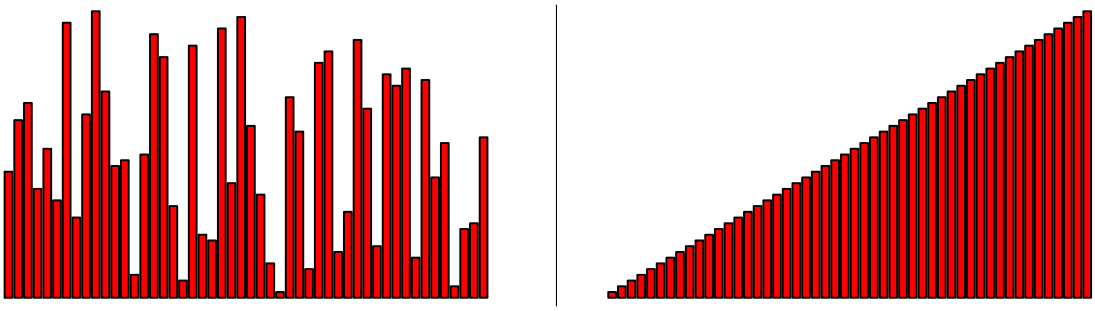
- *Kreher and Stinson* [KS99] – The most recent book on combinatorial generation algorithms, with additional particular focus on algebraic problems such as isomorphism and dealing with symmetry.

S.S. Skiena, *The Algorithm Design Manual*, 2nd ed., DOI: 10.1007/978-1-84800-070-4_14,

- *Knuth* [Knu97a, Knu98] – The standard reference on searching and sorting, with significant material on combinatorial objects such as permutations. New material on the generation of permutations [Knu05a], subsets and partitions [Knu05b], and trees [Knu06] has been released on "fasciles,"—short paperback chunks of what is slated to be the mythical Volume 4.

- *Stanton and White* [SW86a] – An undergraduate combinatorics text with algorithms for generating permutations, subsets, and set partitions. It contains relevant programs in Pascal.

- *Pemmaraju and Skiena* [PS03] – This description of *Combinatorica*, a library of over 400 Mathematica functions for generating combinatorial objects and graph theory (see Section 19.1.9), provides a distinctive view of how different algorithms can fit together. Its second author is considered qualified to write a manual on algorithm design.



INPUT OUTPUT

14.1 Sorting

Input description: A set of n items.

Problem description: Arrange the items in increasing (or decreasing) order.

Discussion: Sorting is the most fundamental algorithmic problem in computer science. Learning the different sorting algorithms is like learning scales for a musician. Sorting is the first step in solving a host of other algorithm problems, as shown in Section 4.2 (page 107). Indeed, *"when in doubt, sort"* is one of the first rules of algorithm design.

Sorting also illustrates all the standard paradigms of algorithm design. The result is that most programmers are familiar with many different sorting algorithms, which sows confusion as to which should be used for a given application. The following criteria can help you decide:

- *How many keys will you be sorting?* – For small amounts of data (say $n \leq 100$), it really doesn't matter much which of the quadratic-time algorithms you use. Insertion sort is faster, simpler, and less likely to be buggy than bubblesort. Shellsort is closely related to, but much faster than, insertion sort, but it involves looking up the right insert sequences in Knuth [Knu98].

  When you have more than 100 items to sort, it is important to use an $O(n \lg n)$-time algorithm like heapsort, quicksort, or mergesort. There are various partisans who favor one of these algorithms over the others, but since it can be hard to tell which is fastest, it usually doesn't matter.

  Once you get past (say) 5,000,000 items, it is important to start thinking about external-memory sorting algorithms that minimize disk access. Both types of algorithm are discussed below.

- *Will there be duplicate keys in the data?* – The sorted order is completely defined if all items have distinct keys. However, when two items share the same key, something else must determine which one comes first. In many applications it doesn't matter, so any sorting algorithm suffices. Ties are often broken by sorting on a secondary key, like the first name or initial when the family names collide.

  Occasionally, ties need to be broken by their initial position in the data set. Suppose the 5th and 27th items of the initial data set share the same key. This means the 5th item must appear before the 27th in the final order. A *stable* sorting algorithm preserves the original ordering in case of ties. Most of the quadratic-time sorting algorithms are stable, while many of the $O(n \lg n)$ algorithms are not. If it is important that your sort be stable, it is probably better to explicitly use the initial position as a secondary key in your comparison function rather than trust the stability of your implementation.

- *What do you know about your data?* – You can often exploit special knowledge about your data to get it sorted faster or more easily. Of course, general sorting is a fast $O(n \lg n)$ operation, so if the time spent sorting is really the bottleneck in your application, you are a fortunate person indeed.

  - *Has the data already been partially sorted?* If so, certain algorithms like insertion sort perform better than they otherwise would.
  - *Do you know the distribution of the keys?* If the keys are randomly or uniformly distributed, a *bucket* or *distribution sort* makes sense. Throw the keys into bins based on their first letter, and recur until each bin is small enough to sort by brute force. This is very efficient when the keys get evenly distributed into buckets. However, bucket sort would perform very badly sorting names on the membership roster of the "Smith Society."
  - *Are your keys very long or hard to compare?* If your keys are long text strings, it might pay to use a relatively short prefix (say ten characters) of each key for an initial sort, and then resolve ties using the full key. This is particularly important in external sorting (see below), since you don't want to waste your fast memory on the dead weight of irrelevant detail.

    Another idea might be to use radix sort. This always takes time linear in the number of characters in the file, instead of $O(n \lg n)$ times the cost of comparing two keys.
  - *Is the range of possible keys very small?* If you want to sort a subset of, say, $n/2$ distinct integers, each with a value from 1 to n, the fastest algorithm would be to initialize an n-element bit vector, turn on the bits corresponding to keys, then scan from left to right and report the positions with true bits.

- *Do I have to worry about disk accesses?* – In massive sorting problems, it may not be possible to keep all data in memory simultaneously. Such a problem is called *external sorting*, because one must use an external storage device. Traditionally, this meant tape drives, and Knuth [Knu98] describes a variety of intricate algorithms for efficiently merging data from different tapes. Today, it usually means virtual memory and swapping. Any sorting algorithm will run using virtual memory, but most will spend all their time swapping.

  The simplest approach to external sorting loads the data into a B-tree (see Section 12.1 (page 367)) and then does an in-order traversal of the tree to read the keys off in sorted order. Real high-performance sorting algorithms are based on multiway-mergesort. Files containing portions of the data are sorted into runs using a fast internal sort, and then files with these sorted runs are merged in stages using 2- or k-way merging. Complicated merging patterns and buffer management based on the properties of the external storage device can be used to optimize performance.

- *How much time do you have to write and debug your routine?* – If I had under an hour to deliver a working routine, I would probably just implement a simple selection sort. If I had an afternoon to build an efficient sort routine, I would probably use heapsort, for it delivers reliable performance without tuning.

The best general-purpose internal sorting algorithm is quicksort (see Section 4.2 (page 107)), although it requires tuning effort to achieve maximum performance. Indeed, you are much better off using a library function instead of coding it yourself. A poorly written quicksort will likely run more slowly than a poorly written heapsort.

If you are determined to implement your own quicksort, use the following heuristics, which make a big difference in practice:

- *Use randomization* – By randomly permuting (see Section 14.4 (page 448)) the keys before sorting, you can eliminate the potential embarrassment of quadratic-time behavior on nearly-sorted data.

- *Median of three* – For your pivot element, use the median of the first, last, and middle elements of the array to increase the likelihood of partitioning the array into roughly equal pieces. Some experiments suggest using a larger sample on big subarrays and a smaller sample on small ones.

- *Leave small subarrays for insertion sort* – Terminating the quicksort recursion and switching to insertion sort makes sense when the subarrays get small, say fewer than 20 elements. You should experiment to determine the best switchpoint for your implementation.

- *Do the smaller partition first* – Assuming that your compiler is smart enough to remove tail recursion, you can minimize run-time memory by processing

the smaller partition before the larger one. Since successive stored calls are at most half as large as the previous one, only $O(\lg n)$ stack space is needed.

Before you get started, see Bentley's article on building a faster quicksort [Ben92b].

Implementations: The best freely available sort program is presumably GNU sort, part of the GNU core utilities library. See *http://www.gnu.org/software/coreutils/*. Be aware that there are also commercial vendors of high-performance external sorting programs. These include Cosort (*www.cosort.com*), Syncsort (*www.syncsort.com*) and Ordinal Technology (*www.ordinal.com*).

Modern programming languages provide libraries offering complete and efficient container implementations. Thus, you should never need to implement your own sort routine. The C standard library contains `qsort`, a generic implementation of (presumably) quicksort. The C++ *Standard Template Library* (STL) provides both `sort` and `stable_sort` methods. It is available with documentation at *http://www.sgi.com/tech/stl/*. See Josuttis [Jos99], Meyers [Mey01] and Musser [MDS01] for more detailed guides to using STL and the C++ standard library.

Java Collections (JC), a small library of data structures, is included in the `java.util` package of Java standard edition (*http://java.sun.com/javase/*). In particular, `SortedMap` and `SortedSet` classes are provided.

For a C++ implementation of an cache-oblivious algorithm (*funnelsort*), check out *http://kristoffer.vinther.name/projects/funnelsort/*. Benchmarks attest to its excellent performance.

There are a variety of websites that provide applets/animations of all the basic sorting algorithms, including bubblesort, heapsort, mergesort, quicksort, radix sort, and shellsort. Many of these are quite interesting to watch. Indeed, sorting is the canonical problem for algorithm animation. Representative examples include Harrison (*http://www.cs.ubc.ca/spider/harrison/Java/sorting-demo.html*) and Bentley [Ben99] (*http://www.cs.bell-labs.com/cm/cs/pearls/sortanim.html*).

Notes: Knuth [Knu98] is the best book that has been written on sorting and indeed is the best book that will ever be written on sorting. It is now over thirty years old, but remains fascinating reading. One area that has developed since Knuth is sorting under presortedness measures, surveyed in [ECW92]. A noteworthy reference on sorting is [GBY91], which includes pointers to algorithms for partially sorted data and includes implementations in C and Pascal for all of the fundamental algorithms.

Expositions on the basic internal sorting algorithms appear in every algorithms text. Heapsort was first invented by Williams [Wil64]. Quicksort was invented by Hoare [Hoa62], with careful analysis and implementation by Sedgewick [Sed78]. Von Neumann is credited with having produced the first implementation of mergesort on the EDVAC in 1945. See Knuth for a full discussion of the history of sorting, dating back to the days of punch-card tabulating machines.

The primary competitive forum for high-performance sorting is an annual competition initiated by the late Jim Gray. See *http://research.microsoft.com/barc/SortBenchmark/* for current and previous results, which are are either inspiring or depressing depending

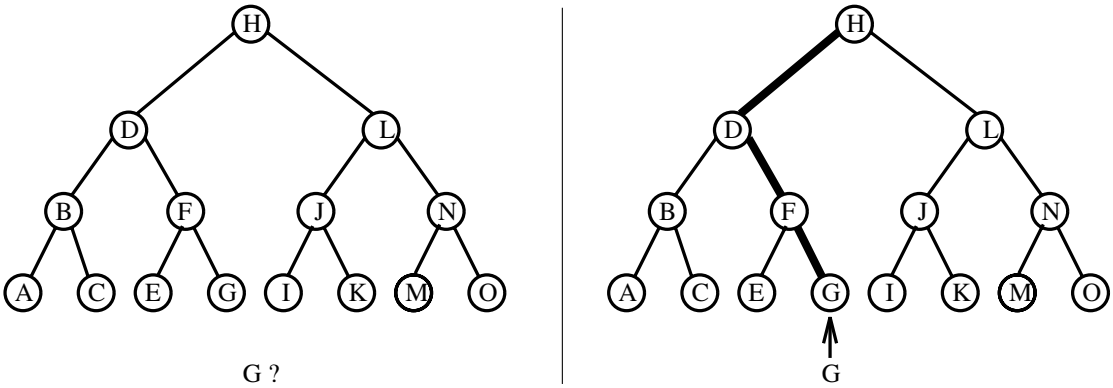
upon how you look at it. The magnitude of progress is inspiring (the million-record instances of the original benchmarks are now too small to bother with) but it is depressing (to me) the extent that systems/memory management issues thoroughly trump the combinatorial/algorithmic aspects of sorting.

Modern attempts to engineer high-performance sort programs include work on both cache-conscious [LL99] and cache-oblivious [BFV07] sorting.

Sorting has a well-known $\Omega(n \lg n)$ lower bound under the algebraic decision tree model [BO83]. Determining the exact number of comparisons required for sorting n elements, for small values of n, has generated considerable study. See [Aig88, Raw92] for expositions and Peczarski [Pec04, Pec07] for the latest results.

This lower-bound does not hold under different models of computation. Fredman and Willard [FW93] present an $O(n\sqrt{\lg n})$ algorithm for sorting under a model of computation that permits arithmetic operations on keys. See Andersson [And05] for a survey of algorithms for fast sorting on such nonstandard models of computation.

Related Problems: Dictionaries (see page 367), searching (see page 441), topological sorting (see page 481).



INPUT OUTPUT

14.2 Searching

Input description: A set of n keys S, and a query key q.

Problem description: Where is q in S?

Discussion: "Searching" is a word that means different things to different people. Searching for the global maximum or minimum of a function is the problem of *unconstrained optimization* and is discussed in Section 13.5 (page 407). Chess-playing programs select the best move to make via an exhaustive search of possible moves using a variation of backtracking (see Section 7.1 (page 231)).

Here we consider the task of searching for a key in a list, array, or tree. Dictionary data structures maintain efficient access to sets of keys under insertion and deletion and are discussed in Section 12.1 (page 367). Typical dictionaries include binary trees and hash tables.

We treat searching as a problem distinct from dictionaries because simpler and more efficient solutions emerge when our primary interest is static searching without insertion/deletion. These little data structures can yield large performance improvements when properly employed in an innermost loop. Also, ideas such as binary search and self-organization apply to other problems and well justify our attention.

Our two basic approaches are sequential search and binary search. Both are simple, yet have interesting and subtle variations. In *sequential search*, we start from the front of our list/array of keys and compare each successive item against the key until we find a match or reach the end. In *binary search*, we start with a sorted array of keys. To search for key q, we compare q to the middle key $S_{n/2}$. If q is before $S_{n/2}$, it must reside in the top half of our set; if not, it must reside in the

bottom half of our set. By repeating this process on the correct half, we find the key using $\lceil \lg n \rceil$ comparisons. This is a big win over the $n/2$ comparisons we expect with sequential search. See Section 4.9 (page 132) for more on binary search.

A sequential search is the simplest algorithm, and likely to be fastest on up to about 20 elements. Beyond (say) 100 elements, binary search will clearly be more efficient than sequential search, easily justifying the cost of sorting if there will be multiple queries. Other issues come into play, however, in identifying the proper variant of the algorithm:

- *How much time can you spend programming?* – A binary search is a notoriously tricky algorithm to program correctly. It took seventeen years after its invention until the first *correct* version of a binary search was published! Don't be afraid to start from one of the implementations described below. Test it completely by writing a driver that searches for every key in the set S as well as between the keys.

- *Are certain items accessed more often than other ones?* – Certain English words (such as "the") are much more likely to occur than others (such as "defenestrate"). We can reduce the number of comparisons in a sequential search by putting the most popular words on the top of the list and the least popular ones at the bottom. Nonuniform access is usually the rule, not the exception. Many real-world distributions are governed by *power laws*. A classic example is word use in English, which is fairly accurately modeled by *Zipf's law*. Under *Zipf's law*, the ith most frequently accessed key is selected with probability $(i-1)/i$ times the probability of the $(i-1)$st most popular key, for all $1 \leq i \leq n$.

  Knowledge of access frequencies is easy to exploit with sequential search. But the issue is more complicated with binary trees. We want popular keys close to the root (so we hit them quickly) but not at the expense of losing balance and degenerating into sequential search. The answer is to employ a dynamic programming algorithm to find the *optimal binary search tree*. The key observation is that each possible root node i partitions the space of keys into those to the left of i and those to the right; each of which should be represented by an optimal binary search tree on a smaller subrange of keys. The root of the optimal tree is selected to minimize the expected search costs of the resulting partition.

- *Might access frequencies change over time?* – Preordering a list or tree to exploit a skewed access pattern requires knowing the access pattern in advance. For many applications, it can be difficult to obtain such information. Better are *self-organizing lists*, where the order of the keys changes in response to the queries. The best self-organizing scheme is move-to-front; that is, we move the most recently searched-for key from its current position to the front of the list. Popular keys keep getting boosted to the front, while unsearched-for

keys drift towards the back of the list. There is no need to keep track of the frequency of access; just move the keys on demand. Self-organizing lists also exploit *locality of reference*, since accesses to a given key are likely to occur in clusters. A hot key will be maintained near the top of the list during a cluster of accesses, even if other keys have proven more popular in the past.

Self-organization can extend the useful size range of sequential search. However, you should switch to binary search beyond 100 elements. But consider using *splay trees*. which are self-organizing binary search trees that rotate each searched-for node to the root. They offer excellent amortized performance guarantees.

- *Is the key close by?* – Suppose we know that the target key is to the right of position p, and we think it is nearby. A sequential search is fast if we are correct, but we will be punished severely when we guess wrong. A better idea is to test repeatedly at larger intervals ($p+1$, $p+2$, $p+4$, $p+8$, $p+16$, ...) to the right until we find a key to the right of our target. Now we have a window containing the target and we can proceed with binary search.

  Such a *one-sided binary search* finds the target at position $p+l$ using at most $2\lceil \lg l \rceil$ comparisons, so it is faster than binary search when $l << n$, yet it can never be much worse. One-sided binary search is particularly useful in unbounded search problems, such as in numerical root finding.

- *Is my data structure sitting on external memory?* – Once the number of keys grows *too* large, a binary search loses its status as the best search technique. A binary search jumps wildly around the set of keys looking for midpoints to compare, and so each comparison requires reading a new page in from external memory. Much better are data structures such as B-trees (see Section 12.1 (page 367)) or Emde Boas trees (see notes below), which cluster the keys into pages to minimize the number of disk accesses per search.

- *Can I guess where the key should be?* – In *interpolation search*, we exploit our understanding of the distribution of keys to guess where to look next. An interpolation search is probably a more accurate description of how we use a telephone book than binary search. Suppose we are searching for *Washington, George* in a sorted telephone book. We would be safe making our first comparison three-fourths of the way down the list, essentially doing two comparisons for the price of one.

  Although an interpolation search is an appealing idea, we caution against it for three reasons: First, you have to work very hard to optimize your search algorithm before you can hope for a speedup over binary search. Second, even if you do beat a binary search, it is unlikely to be by enough to have justified the exercise. Finally, your program will be much less robust and efficient when the distribution changes, such as when your application gets ported to work on French words instead of English.

Implementations: The basic sequential and binary search algorithms are simple enough that you may consider implementing them yourself. That said, the C standard library contains `bsearch`, a generic implementation of (presumably) a binary search. The C++ *Standard Template Library* (STL) provides `find` (sequential search) and `binary_search` iterators. *Java Collections* (JC), provides `binarySearch` in the `java.util` package of Java standard edition (*http://java.sun.com/javase/*).

Many data structure textbooks provide extensive and illustrative implementations. Sedgewick (*http://www.cs.princeton.edu/∼rs/*) [Sed98] and Weiss (*http://www.cs.fiu.edu/∼weiss/*) [Wei06] provide implementation of splay trees and other search structures in both C++ and Java.

Notes: *The Handbook of Data Structures and Applications* [MS05] provides up-to-date surveys on all aspects of dictionary data structures. Other surveys include Mehlhorn and Tsakalidis [MT90b] and Gonnet and Baeza-Yates [GBY91]. Knuth [Knu97a] provides a detailed analysis and exposition on all fundamental search algorithms and dictionary data structures, but omits such modern data structures as red-black and splay trees.

The next position probed in linear interpolation search on an array of sorted numbers is given by

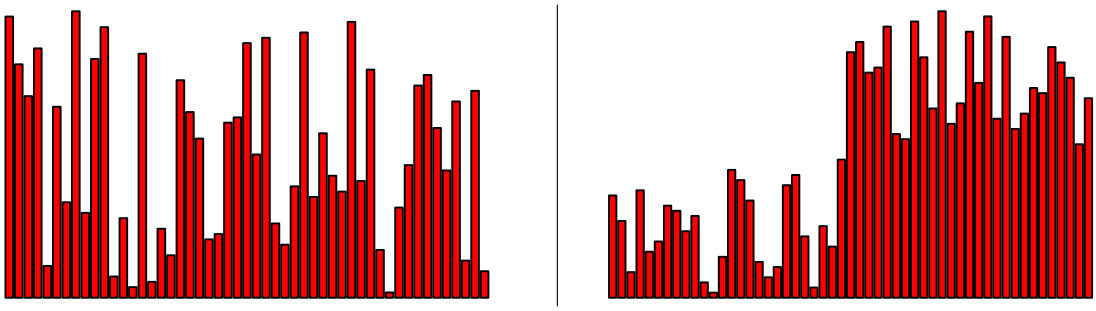
$$next = (low - 1) + \lceil \frac{q - S[low - 1]}{S[high + 1] - S[low - 1]} \times (high - low + 1) \rceil$$

where q is the query numerical key and S the sorted numerical array. If the keys are drawn independently from a uniform distribution, the expected search time is $O(\lg \lg n)$ [DJP04, PIA78].

Nonuniform access patterns can be exploited in binary search trees by structuring them so that popular keys are located near the root, thus minimizing search time. Dynamic programming can be used to construct such optimal search trees in $O(n \lg n)$ time [Knu98]. Stout and Warren [SW86b] provide a slick algorithm to efficiently transform a binary tree to a minimum height (optimally balanced) tree using rotations.

The Van Emde Boas layout of a binary tree (or sorted array) offers better external memory performance than conventional binary search, at a cost of greater implementation complexity. See the survey of Arge, et al. [ABF05] for more on this and other cache-oblivious data structures.

Related Problems: Dictionaries (see page 367), sorting (see page 436).



INPUT OUTPUT

14.3 Median and Selection

Input description: A set of n numbers or keys, and an integer k.

Problem description: Find the key smaller than exactly k of the n keys.

Discussion: Median finding is an essential problem in statistics, where it provides a more robust notion of average than the *mean*. The mean wealth of people who have published research papers on sorting is significantly affected by the presence of one William Gates [GP79], although his effect on the *median* wealth is merely to cancel out one starving graduate student.

Median finding is a special case of the more general *selection* problem, which asks for the kth element in sorted order. Selection arises in several applications:

- *Filtering outlying elements* – In dealing with noisy data, it is usually a good idea to throw out (say) the 10% largest and smallest values. Selection can be used to identify the items defining the 10^{th} and 90^{th} percentiles, and the outliers then filtered out by comparing each item to the two selected bounds.
- *Identifying the most promising candidates* – In a computer chess program, we might quickly evaluate all possible next moves, and then decide to study the top 25% more carefully. Selection followed by filtering is the way to go.
- *Deciles and related divisions* – A useful way to present income distribution in a population is to chart the salary of the people ranked at regular intervals, say exactly at the 10th percentile, 20th percentile, etc. Computing these values is simply selection on the appropriate position ranks.
- *Order statistics* – Particularly interesting special cases of selection include finding the smallest element ($k = 1$), the largest element ($k = n$), and the median element ($k = n/2$).

The mean of n numbers can be computed in linear time by summing the elements and dividing by n. However, finding the median is a more difficult problem. Algorithms that compute the median can readily be generalized to arbitrary selection. Issues in median finding and selection include:

- *How fast does it have to be?* – The most elementary median-finding algorithm sorts the items in $O(n \lg n)$ time and then returns the item occupying the $(n/2)$nd position. The good thing is that this gives much more information than just the median, enabling you to select the kth element (for any $1 \leq k \leq n$) in constant time after the sort. However, there are faster algorithms if all you want is the median.

  In particular, there is an $O(n)$ *expected*-time algorithm based on quicksort. Select a random element in the data set as a pivot, and use it to partition the data into sets of elements less than and greater than the pivot. From the sizes of these sets, we know the position of the pivot in the total order, and hence whether the median lies to the left or right of this point. Now we recur on the appropriate subset until it converges on the median. This takes (on average) $O(\lg n)$ iterations, with the cost of each iteration being roughly half that of the previous one. This defines a geometric series that converges to a linear-time algorithm, although if you are very unlucky it takes the same time as quicksort, $O(n^2)$.

  More complicated algorithms can find the median in worst-case linear time. However, the expected-time algorithm will likely win in practice. Just make sure to select random pivots to avoid the worst case.

- *What if you only get to see each element once?* – Selection and median finding become expensive on large datasets because they typically require several passes through external memory. In data-streaming applications, the volume of data is often too large to store, making repeated consideration (and thus exact median finding) impossible. Much better is computing a small summary of the data for future analysis, say approximate deciles or frequency moments (where the kth moment of stream x is defined as $F_k = \sum_i x_i^k$).

  One solution to such a problem is random sampling. Flip a coin for each value to decide whether to store it, with the probability of heads set low enough that you won't overflow your buffer. Likely the median of your samples will be close to that of the underlying data set. Alternately, you can devote some fraction of memory to retaining (say) decile values of large blocks, and then combine these decile distributions to yield more refined decile bounds.

- *How fast can you find the mode?* – Beyond mean and median lies a third notion of average. The *mode* is defined to be the element that occurs the greatest number of times in the data set. The best way to compute the mode sorts the set in $O(n \lg n)$ time, which places all identical elements next to each other. By doing a linear sweep from left to right on this sorted set, we can

count the length of the longest run of identical elements and hence compute the mode in a total of $O(n \lg n)$ time.

In fact, there is no faster worst-case algorithm possible to compute the mode, since the problem of testing whether there exist two identical elements in a set (called element uniqueness) can be shown to have an $\Omega(n \log n)$ lower bound. Element uniqueness is equivalent to asking whether the mode occurs more than once. Possibilities exist, at least theoretically, for improvements when the mode is large by using fast median computations.

Implementations: The C++ *Standard Template Library* (STL) provides a general selection method (`nth_element`) implemented using the linear expected-time algorithm. It is available with documentation at *http://www.sgi.com/tech/stl/*. See Josuttis [Jos99], Meyers [Mey01] and Musser [MDS01] for more detailed guides to using STL and the C++ standard library.

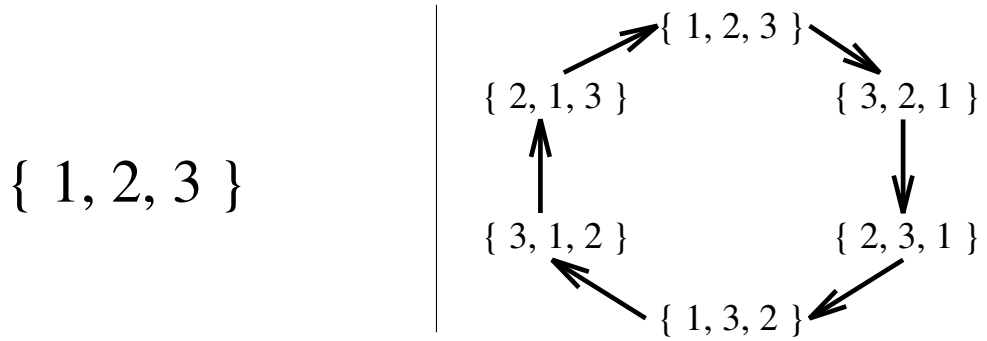
Notes: The linear expected-time algorithm for median and selection is due to Hoare [Hoa61]. Floyd and Rivest [FR75] provide an algorithm that uses fewer comparisons on average. Good expositions on linear-time selection include [AHU74, BvG99, CLRS01, Raw92], with [Raw92] being particularly enlightening.

Streaming algorithms have extensive applications to large data sets, and are well surveyed by Muthukrishnan [Mut05].

A sport of considerable theoretical interest is determining *exactly* how many comparisons are sufficient to find the median of n items. The linear-time algorithm of Blum et al. [BFP+72] proves that $c \cdot n$ suffice, but we want to know what c is. Dor and Zwick [DZ99] proved that $2.95n$ comparisons suffice to find the median. These algorithms attempt to minimize the number of element comparisons but not the total number of operations, and hence do not lead to faster algorithms in practice. They also hold the current best lower bower bound of $(2 + \epsilon)$ comparisons for median finding [DZ01].

Tight combinatorial bounds for selection problems are presented in [Aig88]. An optimal algorithm for computing the mode is given by [DM80].

Related Problems: Priority queues (see page 373), sorting (see page 436).



INPUT OUTPUT

14.4 Generating Permutations

Input description: An integer n.

Problem description: Generate (1) all, or (2) a random, or (3) the next permutation of length n.

Discussion: A permutation describes an arrangement or ordering of items. Many algorithmic problems in this catalog seek the best way to order a set of objects, including *traveling salesman* (the least-cost order to visit n cities), *bandwidth* (order the vertices of a graph on a line so as to minimize the length of the longest edge), and *graph isomorphism* (order the vertices of one graph so that it is identical to another). Any algorithm for solving such problems exactly must construct a series of permutations along the way.

There are $n!$ permutations of n items. This grows so quickly that you can't really expect to generate all permutations for $n > 12$, since $12! = 479{,}001{,}600$. Numbers like these should cool the ardor of anyone interested in exhaustive search and help explain the importance of generating random permutations.

Fundamental to any permutation-generation algorithm is a notion of order, the sequence in which the permutations are constructed from first to last. The most natural generation order is *lexicographic*, the sequence they would appear if they were sorted numerically. Lexicographic order for $n = 3$ is $\{1, 2, 3\}$, $\{1, 3, 2\}$, $\{2, 1, 3\}$, $\{2, 3, 1\}$, $\{3, 1, 2\}$, and finally $\{3, 2, 1\}$. Although lexicographic order is aesthetically pleasing, there is often no particular advantage to using it. For example, if you are searching through a collection of files, it does not matter whether the filenames are encountered in sorted order, so long as you eventually search through all of them. Indeed, nonlexicographic orders lead to faster and simpler permutation generation algorithms.

There are two different paradigms for constructing permutations: ranking/unranking and incremental change methods. The latter are more efficient, but ranking and unranking can be applied to solve a much wider class of problems. The key is to define the functions *rank* and *unrank* on all permutations p and integers n, m, where $|p| = n$ and $0 \leq m \leq n!$.

- *Rank(p)* – What is the position of p in the given generation order? A typical ranking function is recursive, such as basis case $Rank(\{1\}) = 0$ with

  $$Rank(p) = (p_1 - 1) \cdot (|p| - 1)! + Rank(p_2, \ldots, p_{|p|})$$

  Getting this right means relabeling the elements of the smaller permutation to reflect the deleted first element. Thus

  $$Rank(\{2, 1, 3\}) = 1 \cdot 2! + Rank(\{1, 2\}) = 2 + 0 \cdot 1! + Rank(\{1\}) = 2$$

- *Unrank(m,n)* – Which permutation is in position m of the $n!$ permutations of n items? A typical unranking function finds the number of times $(n-1)!$ goes into m and proceeds recursively. $Unrank(2, 3)$ tells us that the first element of the permutation must be '2', since $(2-1) \cdot (3-1)! \leq 2$ but $(3-1) \cdot (3-1)! > 2$. Deleting $(2-1) \cdot (3-1)!$ from m leaves the smaller problem $Unrank(0, 2)$. The ranking of 0 corresponds to the total order. The total order on the two remaining elements (since 2 has been used) is $\{1, 3\}$, so $Unrank(2, 3) = \{2, 1, 3\}$.

What the actual rank and unrank functions are does not matter as much as the fact that they must be inverses of each other. In other words, $p = Unrank(Rank(p), n)$ for all permutations p. Once you define ranking and unranking functions for permutations, you can solve a host of related problems:

- *Sequencing permutations* – To determine the *next* permutation that occurs in order after p, we can $Rank(p)$, add 1, and then $Unrank(p)$. Similarly, the permutation right before p in order is $Unrank(Rank(p) - 1, |p|)$. Counting through the integers from 0 to $n! - 1$ and unranking them is equivalent to generating all permutations.

- *Generating random permutations* – Selecting a random integer from 0 to $n!-1$ and then unranking it yields a truly random permutation.

- *Keep track of a set of permutations* – Suppose we want to construct random permutations and act only when we encounter one we have not seen before. We can set up a bit vector (see Section 12.5 (page 385)) with $n!$ bits, and set bit i to 1 if permutation *Unrank(i,n)* has been seen. A similar technique was employed with k-subsets in the Lotto application of Section 1.6 (page 23).

This rank/unrank method is best suited for small values of n, since $n!$ quickly exceeds the capacity of machine integers unless arbitrary-precision arithmetic is available (see Section 13.9 (page 423)). The incremental change methods work by defining the *next* and *previous* operations to transform one permutation into another, typically by swapping two elements. The tricky part is to schedule the swaps so that permutations do not repeat until all of them have been generated. The output picture above gives an ordering of the six permutations of $\{1, 2, 3\}$ using a single swap between successive permutations.

Incremental change algorithms for sequencing permutations are tricky, but they are so concise that they can be expressed in a dozen-line program. See the implementation section for pointers to code. Because the incremental change is only a single swap, these algorithms can be extremely fast—on average, constant time—which is independent of the size of the permutation! The secret is to represent the permutation using an n-element array to facilitate the swap. In certain applications, only the change between permutations is important. For example, in a brute-force program to search for the optimal TSP tour, the cost of the tour associated with the new permutation will be that of the previous permutation, with the addition and deletion of four edges.

Throughout this discussion, we have assumed that the items we are permuting are all distinguishable. However, if there are duplicates (meaning our set is a *multi-set*), you can save considerable time and effort by avoiding identical permutations. For example, there are only ten distinct permutations of $\{1, 1, 2, 2, 2\}$, instead of 120. To avoid duplicates, use backtracking and generate the permutations in lexicographic order.

Generating random permutations is an important little problem that people often stumble across, and often botch up. The right way is to use the following two-line, linear-time algorithm. We assume that *Random[i,n]* generates a random integer between i and n, inclusive.

for $i = 1$ to n do $a[i] = i$;
for $i = 1$ to $n - 1$ do $swap[a[i], a[Random[i, n]]$;

That this algorithm generates all permutations uniformly at random is not obvious. If you think so, convincingly explain why the following algorithm *does not* generate permutations uniformly:

for $i = 1$ to n do $a[i] = i$;
for $i = 1$ to $n - 1$ do $swap[a[i], a[Random[1, n]]$;

Such subtleties demonstrate why you must be very careful with random generation algorithms. Indeed, we recommend that you try some reasonably extensive experiments with *any* random generator before really believing it. For example, generate 10,000 random permutations of length 4 and see whether all 24 distinct permutations occur approximately the same number of times. If you know how to measure statistical significance, you are in even better shape.

Implementations: The C++ *Standard Template Library* (STL) provides two functions (`next_permutation` and `prev_permutation`) for sequencing permutations in lexicographic order. Kreher and Stinson [KS99] provide implementations of minimum change and lexicographic permutation generation in C at *http://www.math.mtu.edu/~kreher/cages/Src.html*.

The *Combinatorial Object Server* (*http://theory.cs.uvic.ca/*) developed by Frank Ruskey of the University of Victoria is a unique resource for generating permutations, subsets, partitions, graphs, and other objects. An interactive interface enables you to specify which objects you would like returned to you. Implementations in C, Pascal, and Java are available for certain types of objects.

C++ routines for generating an astonishing variety of combinatorial objects, including permutations and cyclic permutations, are available at *http://www.jjj.de/fxt/*.

Nijenhuis and Wilf [NW78] is a venerable but still excellent source on generating combinatorial objects. They provide efficient Fortran implementations of algorithms to construct random permutations and to sequence permutations in minimum-change order. Also included are routines to extract the cycle structure of a permutation. See Section 19.1.10 (page 661) for details.

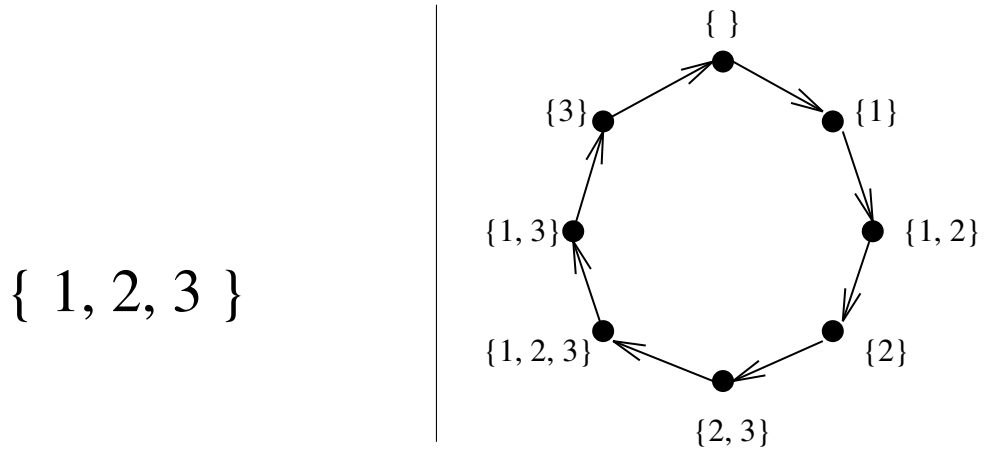
Combinatorica [PS03] provides Mathematica implementations of algorithms that construct random permutations and sequence permutations in minimum change and lexicographic orders. It also provides a backtracking routine to construct all distinct permutations of a multiset, and it supports various permutation group operations. See Section 19.1.9 (page 661).

Notes: The best recent reference on permutation generation is Knuth [Knu05a]. Sedgewick's excellent survey on the topic is older [Sed77], but this is not a fast moving area. Good expositions include [KS99, NW78, Rus03].

Fast permutation generation methods make only a single swap between successive permutations. The Johnson-Trotter algorithm [Joh63, Tro62] satisfies an even stronger condition, namely that the two elements being swapped are always adjacent. Simple linear-time ranking and unranking functions for permutations are given by Myrvold and Ruskey [MR01].

In the days before ready access to computers, books with tables of random permutations [MO63] were used instead of algorithms. The swap-based random permutation algorithm presented above was first described in [MO63].

Related Problems: Random-number generation (see page 415), generating subsets (see page 452), generating partitions (see page 456).



INPUT OUTPUT

14.5 Generating Subsets

Input description: An integer n.

Problem description: Generate (1) all, or (2) a random, or (3) the next subset of the integers $\{1, \ldots, n\}$.

Discussion: A subset describes a selection of objects, where the order among them does not matter. Many important algorithmic problems seek the best subset of a group of things: *vertex cover* seeks the smallest subset of vertices to touch each edge in a graph; *knapsack* seeks the most profitable subset of items of bounded total size; while *set packing* seeks the smallest subset of subsets that together cover each item exactly once.

There are 2^n distinct subsets of an n-element set, including the empty set as well as the set itself. This grows exponentially, but at a considerably slower rate than the $n!$ permutations of n items. Indeed, since $2^{20} = 1{,}048{,}576$, a brute-force search through all subsets of 20 elements is easily manageable. Since $2^{30} = 1{,}073{,}741{,}824$, you will certainly hit limits for slightly larger values of n.

By definition, the relative order among the elements does not distinguish different subsets. Thus, $\{1, 2, 5\}$ is the same as $\{2, 1, 5\}$. However, it is a very good idea to maintain your subsets in a sorted or *canonical* order to speed up such operations as testing whether or not two subsets are identical.

As with permutations (see Section 14.4 (page 448)), the key to subset generation problems is establishing a numerical sequence among all 2^n subsets. There are three primary alternatives:

- *Lexicographic order* – Lexicographic order means sorted order, and is often the most natural way to generate combinatorial objects. The eight subsets of $\{1,2,3\}$ in lexicographic order are $\{\}$, $\{1\}$, $\{1,2\}$, $\{1,2,3\}$, $\{1,3\}$, $\{2\}$, $\{2,3\}$, and $\{3\}$. But it is surprisingly difficult to generate subsets in lexicographic order. Unless you have a compelling reason to do so, don't bother.

- *Gray Code* – A particularly interesting and useful subset sequence is the minimum change order, wherein adjacent subsets differ by the insertion or deletion of exactly one element. Such an ordering, called a *Gray code*, appears in the output picture above.

  Generating subsets in Gray code order can be very fast, because there is a nice recursive construction. Construct a Gray code of $n-1$ elements G_{n-1}. Reverse a second copy of G_{n-1} and add n to each subset in this copy. Then concatenate them together to create G_n. Study the output example for clarification.

  Further, since only one element changes between subsets, exhaustive search algorithms built on Gray codes can be quite efficient. A set cover program would only have to update the change in coverage by the addition or deletion of one subset. See the implementation section below for Gray code subset-generation programs.

- *Binary counting* – The simplest approach to subset-generation problems is based on the observation that any subset S' is defined by the items of that S are in S'. We can represent S' by a binary string of n bits, where bit i is 1 iff the ith element of S is in S'. This defines a bijection between the 2^n binary strings of length n, and the 2^n subsets of n items. For $n = 3$, binary counting generates subsets in the following order: {}, {3}, {2}, {2,3}, {1}, {1,3}, {1,2}, {1,2,3}.

  This binary representation is the key to solving all subset generation problems. To generate all subsets in order, simply count from 0 to $2^n - 1$. For each integer, successively mask off each of the bits and compose a subset of exactly the items corresponding to 1 bits. To generate the *next* or *previous* subset, increment or decrement the integer by one. *Unranking* a subset is exactly the masking procedure, while *ranking* constructs a binary number with 1's corresponding to items in S and then converts this binary number to an integer.

  To generate a random subset, you might generate a random integer from 0 to $2^n - 1$ and unrank, although how your random number generator rounds things off might mean that certain subsets can never occur. Much better is to flip a coin n times, with the ith flip deciding whether to include element i in the subset. A coin flip can be robustly simulated by generating a random real or large integer and testing whether it is bigger or smaller than half its range. A Boolean array of n items can thus be used to represent subsets as a sort of premasked integer.

Generation problems for two closely related problems arise often in practice:

- *K-subsets* – Instead of constructing all subsets, we may only be interested in the subsets containing exactly k elements. There are $\binom{n}{k}$ such subsets, which is substantially less than 2^n, particularly for small values of k.

  The best way to construct all k-subsets is in lexicographic order. The ranking function is based on the observation that there are $\binom{n-f}{k-1}$ k-subsets whose smallest element is f. Using this, it is possible to determine the smallest element in the mth k-subset of n items. We then proceed recursively for subsequent elements of the subset. See the implementations below for details.

- *Strings* – Generating all subsets is equivalent to generating all 2^n strings of true and false. The same basic techniques apply to generate all or random strings on alphabets of size α, except there will be α^n strings in total.

Implementations: Kreher and Stinson [KS99] provide generators for both subsets and k-subsets, including lexicographic and Gray code orders. These implementations in C are available at *http://www.math.mtu.edu/~kreher/cages/Src.html*.

The *Combinatorial Object Server* (*http://theory.cs.uvic.ca/*), developed by Frank Ruskey of the University of Victoria, is a unique resource for generating permutations, subsets, partitions, graphs, and other objects. An interactive interface enables you to specify which objects you would like returned to you. Implementations in C, Pascal, and Java are available for certain types of objects.

C++ routines for generating an astonishing variety of combinatorial objects, including subsets and k-subsets (combinations), are available in the combinatorics package at *http://www.jjj.de/fxt/*.

Nijenhuis and Wilf [NW78] is a venerable but still excellent source on generating combinatorial objects. They provide efficient Fortran implementations of algorithms to construct random subsets and to sequence subsets in Gray code and lexicographic order. They also provide routines to construct random k-subsets and sequence them in lexicographic order. See Section 19.1.10 (page 661) for details on ftp-ing these programs. Algorithm 515 [BL77] of the *Collected Algorithms of the ACM* is another Fortran implementation of lexicographic k-subsets, available from Netlib (see Section 19.1.5 (page 659)).

Combinatorica [PS03] provides Mathematica implementations of algorithms to construct random subsets and sequence subsets in Gray code, binary, and lexicographic order. They also provide routines to construct random k-subsets and strings, and sequence them lexicographically. See Section 19.1.9 (page 661) for further information on Combinatorica.

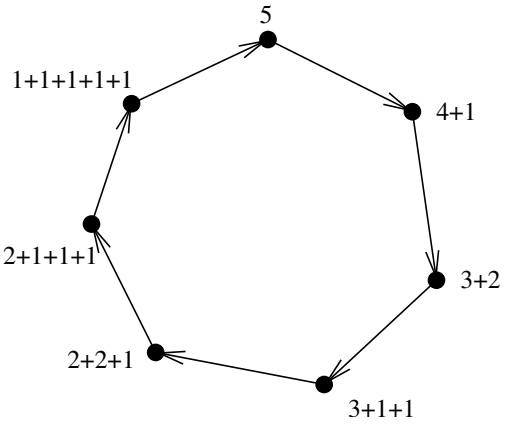
Notes: The best reference on subset generation is Knuth [Knu05b]. Good expositions include [KS99, NW78, Rus03]. Wilf [Wil89] provides an update of [NW78], including a thorough discussion of modern Gray code generation problems.

Gray codes were first developed [Gra53] to transmit digital information in a robust manner over an analog channel. By assigning the code words in Gray code order, the ith word differs only slightly from the $(i+1)$st, so minor fluctuations in analog signal strength corrupts only a few bits. Gray codes have a particularly nice correspondence to Hamiltonian cycles on the hypercube. Savage [Sav97] gives an excellent survey of Gray codes (minimum change orderings) for a large class of combinatorial objects, including subsets.

The popular puzzle *Spinout*, manufactured by ThinkFun (formerly Binary Arts Corporation), is solved using ideas from Gray codes.

Related Problems: Generating permutations (see page 448), generating partitions (see page 456).

N = 5



INPUT OUTPUT

14.6 Generating Partitions

Input description: An integer n.

Problem description: Generate (1) all, or (2) a random, or (3) the next integer or set partitions of length n.

Discussion: There are two different types of combinatorial objects denoted by the word "partition," namely integer partitions and set partitions. They are quite different beasts, but it is a good idea to make both a part of your vocabulary:

- *Integer partitions* are multisets of nonzero integers that add up exactly to n. For example, the seven distinct integer partitions of 5 are {5}, {4,1}, {3,2}, {3,1,1}, {2,2,1}, {2,1,1,1}, and {1,1,1,1,1}. An interesting application I encountered that required generating integer partitions was in a simulation of nuclear fission. When an atom is smashed, the nucleus of protons and neutrons is broken into a set of smaller clusters. The sum of the particles in the set of clusters must equal the original size of the nucleus. As such, the integer partitions of this original size represent all the possible ways to smash an atom.

- *Set partitions* divide the elements $1, \ldots, n$ into nonempty subsets. There are 15 distinct set partitions of $n = 4$: {1234}, {123,4}, {124,3}, {12,34}, {12,3,4}, {134,2}, {13,24}, {13,2,4}, {14,23}, {1,234}, {1,23,4}, {14,2,3}, {1,24,3}, {1,2,34}, and {1,2,3,4}. Several algorithm problems return set partitions as results, including *vertex/edge coloring* and *connected components*.

Although the number of integer partitions grows exponentially with n, they do so at a refreshingly slow rate. There are only 627 partitions of $n = 20$. It is even possible to enumerate all integer partitions of $n = 100$, since there there are only 190,569,292 of them.

The easiest way to generate integer partitions is to construct them in lexicographically decreasing order. The first partition is $\{n\}$ itself. The general rule is to subtract 1 from the smallest part that is > 1 and then collect all the 1's so as to match the new smallest part > 1. For example, the partition following $\{4, 3, 3, 3, 1, 1, 1, 1\}$ is $\{4, 3, 3, 2, 2, 2, 1\}$, since the five 1's left after $3 - 1 = 2$ becomes the smallest part are best packaged as 2,2,1. When the partition is all 1's, we have completed one pass through all the partitions.

This algorithm is sufficiently intricate to program that you should consider using one of the implementations below. In either case, test it to make sure that you get exactly 627 distinct partitions for $n = 20$.

Generating integer partitions uniformly at random is a trickier business than generating random permutations or subsets. This is because selecting the first (i.e. largest) element of the partition has a dramatic effect on the number of possible partitions that can be generated. Observe that no matter how large n is, there is only one partition of n whose largest part is 1. The number of partitions of n with largest part at most k is given by the recurrence

$$P_{n,k} = P_{n-k,k} + P_{n,k-1}$$

with the two boundary conditions $P_{x-y,x} = P_{x-y,x-y}$ and $P_{n,1} = 1$. This function can be used to select the largest part of your random partition with the correct probabilities and, by proceeding recursively, to eventually construct the entire random partition. Implementations are cited below.

Random partitions tend to have large numbers of fairly small parts, best visualized by a Ferrers diagram as in Figure 14.1. Each row of the diagram corresponds to one part of the partition, with the size of each part represented by that many dots.

Set partitions can be generated using techniques akin to integer partitions. Each set partition is encoded as a *restricted growth function*, $a_1, \ldots, a_n$, where $a_1 = 0$ and $a_i \leq 1 + \max(a_1, \ldots, a_i)$, for $i = 2, \ldots, n$. Each distinct digit identifies a subset, or *block*, of the partition, while the growth condition ensures that the blocks are sorted into a canonical order based on the smallest element in each block. For example, the restricted growth function $0, 1, 1, 2, 0, 3, 1$ defines the set partition $\{\{1, 5\}, \{2, 3, 7\}, \{4\}, \{6\}\}$.

Since there is a one-to-one equivalence between set partitions and restricted growth functions, we can use lexicographic order on the restricted growth functions to order the set partitions. Indeed, the fifteen partitions of $\{1, 2, 3, 4\}$ listed above are sequenced according to the lexicographic order of their restricted growth function (check it out).

We can use a similar counting strategy to generate random set partitions as we did with integer partitions. The Stirling numbers of the second kind $\{{n \atop k}\}$ count the

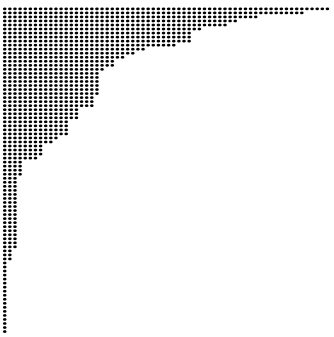


Figure 14.1: The Ferrers diagram of a random partition of $n = 1000$

number of partitions of $\{1, \ldots, n\}$ with exactly k blocks. They are computed using the recurrence

$$\left\{ {n \atop k} \right\} = \left\{ {n-1 \atop k-1} \right\} + k \left\{ {n-1 \atop k} \right\}$$

with the boundary conditions $\left\{ {n \atop n} \right\} = \left\{ {n \atop 1} \right\} = 1$. The reader is referred to the Notes and Implementations section for more details.

Implementations: Kreher and Stinson [KS99] generate both integer and set partitions in lexicographic order, including ranking/unranking functions. These implementations in C are available at *http://www.math.mtu.edu/~kreher/cages/Src.html*.

The *Combinatorial Object Server* (*http://theory.cs.uvic.ca/*) developed by Frank Ruskey of the University of Victoria is a unique resource for generating permutations, subsets, partitions, graphs, and other objects. An interactive interface enables you to specify which objects you would like returned. Implementations in C, Pascal, and Java are available for certain types of objects.

C++ routines for generating an astonishing variety of combinatorial objects, including integer partitions and compositions, are available in the combinatorics package at *http://www.jjj.de/fxt/*.

Nijenhuis and Wilf [NW78] is a venerable but still excellent source on generating combinatorial objects. They provide efficient Fortran implementations of algorithms to construct random and sequential integer partitions, set partitions, compositions, and Young tableaux. See Section 19.1.10 (page 661) for details.

Combinatorica [PS03] provides Mathematica implementations of algorithms to construct random and sequential integer partitions, compositions, strings, and

Young tableaux, as well as to count and manipulate these objects. See Section 19.1.9 (page 661).

Notes: The best reference on algorithms for generating both integer and set partitions is Knuth [Knu05b]. Good expositions include [KS99, NW78, Rus03, PS03]. Andrews [And98] is the primary reference on integer partitions and related topics, with [AE04] his more accessible introduction.

Integer and set partitions are both special cases of *multiset partitions*, or set partitions of nonnecessarily distinct numbers. In particular, the distinct set partitions on the multiset $\{1, 1, 1, \ldots, 1\}$ correspond exactly to integer partitions. Multiset partitions are discussed in Knuth [Knu05b].

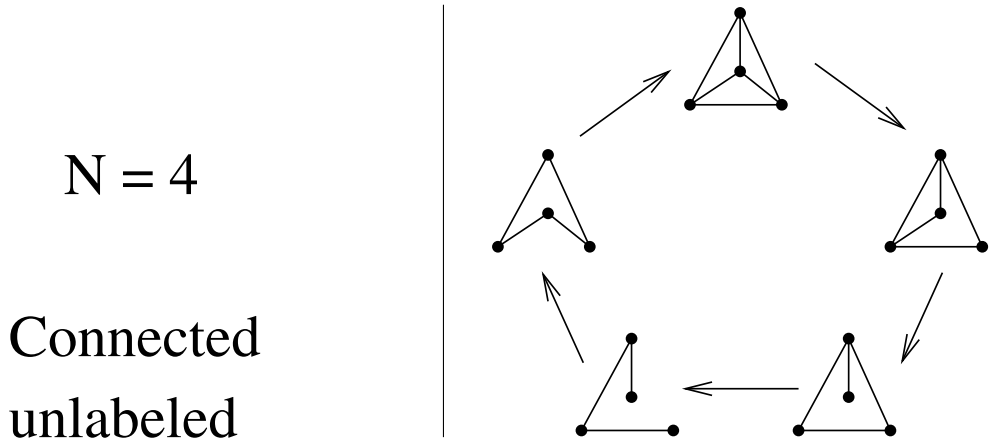
The (long) history of combinatorial object generation is detailed by Knuth [Knu06]. Particularly interesting are connections between set partitions and a Japanese incense burning game, and the naming of all 52 set partitions for $n = 5$ with distinct chapters from the oldest novel known, *The Tale of Genji*.

Two related combinatorial objects are Young tableaux and integer compositions, although they are less likely to emerge in applications. Generation algorithms for both are presented in [NW78, Rus03, PS03].

Young tableaux are two-dimensional configurations of integers $\{1, \ldots, n\}$ where the number of elements in each row is defined by an integer partition of n. Further, the elements of each row and column are sorted in increasing order, and the rows are left-justified. This notion of shape captures a wide variety of structures as special cases. They have many interesting properties, including the existence of a bijection between pairs of tableaux and permutations.

Compositions represent the possible assignments of a set of n indistinguishable balls to k distinguishable boxes. For example, we can place three balls into two boxes as {3,0}, {2,1}, {1,2}, or {0,3}. Compositions are most easily constructed sequentially in lexicographic order. To construct them randomly, pick a random $(k-1)$-subset of $n+k-1$ items using the algorithm of Section 14.5 (page 452), and count the number of unselected items between the selected ones. For example, if $k = 5$ and $n = 10$, the $(5-1)$ subset {1,3,7,14} of $1, \ldots, (n+k-1) = 14$ defines the composition {0,1,3,6,0}, since there are no items to the left of element 1 nor right of element 14.

Related Problems: Generating permutations (see page 448), generating subsets (see page 452).



INPUT OUTPUT

14.7 Generating Graphs

Input description: Parameters describing the desired graph, including the number of vertices n, and the number of edges m or edge probability p.

Problem description: Generate (1) all, or (2) a random, or (3) the next graph satisfying the parameters.

Discussion: Graph generation typically arises in constructing test data for programs. Perhaps you have two different programs that solve the same problem, and you want to see which one is faster or make sure that they always give the same answer. Another application is experimental graph theory, verifying whether a particular property is true for all graphs or how often it is true. It is much easier to believe the four-color theorem after you have demonstrated four colorings for all planar graphs on 15 vertices.

A different application of graph generation arises in network design. Suppose you need to design a network linking ten machines using as few cables as possible, such that this network can survive up to two vertex failures. One approach is to test all the networks with a given number of edges until you find one that will work. For larger graphs, heuristic approaches like simulated annealing will likely be necessary.

Many factors complicate the problem of generating graphs. First, make sure you know exactly what types of graphs you want to generate. Figure 5.2 on page 147 illustrates several important properties of graphs. For purposes of generation, the most important questions are:

- *Do I want labeled or unlabeled graphs?* – The issue here is whether the names of the vertices matter in deciding whether two graphs are the same. In generating *labeled graphs*, we seek to construct all possible labelings of all possible graph topologies. In generating *unlabeled graphs*, we seek only one representative for each topology and ignore labelings. For example, there are only two connected unlabeled graphs on three vertices—a triangle and a simple path. However, there are four connected labeled graphs on three vertices—one triangle and three 3-vertex paths, each distinguished by the name of their central vertex. In general, labeled graphs are much easier to generate. However, there are so many more of them that you quickly get swamped with isomorphic copies of the same few graphs.

- *Do I want directed or undirected graphs?* – Most natural generation algorithms generate undirected graphs. These can be turned into directed graphs by flipping coins to orient the edges. Any graph can be oriented to be directed and acyclic (i.e. , a DAG) by randomly permuting the vertices on a line and aiming each edge from left to right. With all such ideas, careful thought must be given to decide whether you are generating all graphs uniformly at random, and how much this matters to you.

You also must define what you mean by random. There are three primary models of random graphs, all of which generate graphs according to different probability distributions:

- *Random edge generation* – The first model is parameterized by a given edge probability p. Typically, $p = 0.5$, although smaller values can be used to construct sparser random graphs. In this model, a coin is flipped for each pair of vertices x and y to decide whether to add an edge (x, y). All *labeled* graphs will be generated with equal probability when $p = 1/2$.

- *Random edge selection* – The second model is parameterized by the desired number of edges m. It selects m distinct edges uniformly at random. One way to do this is by drawing random (x, y)-pairs and creating an edge if that pair is not already in the graph. An alternative approach to computing the same things constructs the set of $\binom{n}{2}$ possible edges and selects a random m-subset of them, as discussed in Section 14.5 (page 452).

- *Preferential attachment* – Under a rich-get-richer model, newly created edges are likely to point to high-degree vertices than low-degree ones. Consider new links (edges) being added to the graph of webpages. Under any realistic web generation model, it is much more likely the next link will be to Google than *http://www.cs.sunysb.edu/∼algorith.*[1] Selecting the next neighbor with

[1]Please link to us from your homepage to correct for this travesty.

probability proportional to its degree yields graphs with *power law* properties encountered in many real networks.

Which of these options best models your application? Probably none of them. Random graphs have very little structure by definition. But graphs are used to model relationships, which are often highly structured. Experiments conducted on random graphs, although interesting and easy to perform, often fail to capture what you are looking for.

An alternative to random graphs is "organic" graphs—graphs that reflect the relationships among real-world objects. The Stanford GraphBase, discussed below, is an outstanding source of organic graphs. Many raw sources of relationships are available on the web that can be turned into interesting organic graphs with a little programming and imagination. Consider the graph defined by a set of web-pages, with any hyperlink between two pages defining an edge. Or, what about the graph implicit in railroad, subway, or airline networks, with vertices being stations and edges between two stations connected by direct service? As a final example, every large computer program defines a call graph, where the vertices represent subroutines, and there is an edge (x, y) if x calls y.

Two classes of graphs have particularly interesting generation algorithms:

- *Trees* – Prüfer codes provide a simple way to rank and unrank *labeled* trees and thus solve all standard generation problems (see Section 14.4 (page 448)). There are exactly n^{n-2} labeled trees on n vertices, and exactly that many strings of length $n-2$ on the alphabet $\{1, 2, \ldots, n\}$.

  The key to Prüfer's bijection is the observation that every tree has at least two vertices of degree 1. Thus, in any labeled tree the vertex v incident on the leaf with lowest label is well defined. We take v to be S_1, the first character in the code. We then delete the associated leaf and repeat the procedure until only two vertices are left. This defines a unique code S for any given labeled tree that can be used to rank the tree. To go from code to tree, observe that the degree of vertex v in the tree is one more than the number of times v occurs in S. The lowest-labeled leaf will be the smallest integer missing from S, which when paired with S_1 determines the first edge of the tree. The entire tree follows by induction.

  Algorithms for efficiently generating unlabeled rooted trees are discussed in the Implementation section.

- *Fixed degree sequence graphs* – The *degree sequence* of a graph G is an integer partition $p = (p_1, \ldots, p_n)$, where p_i is the degree of the ith highest-degree vertex of G. Since each edge contributes to the degree of two vertices, p is an integer partition of $2m$, where m is the number of edges in G.

  Not all partitions correspond to degree sequences of graphs. However, there is a recursive construction that constructs a graph with a given degree sequence if one exists. If a partition is realizable, the highest-degree vertex v_1 can be

connected to the next p_1 highest-degree vertices in G, or the vertices corresponding to parts $p_2, \ldots, p_{p_1+1}$. Deleting p_1 and decrementing $p_2, \ldots, p_{p_1+1}$ yields a smaller partition, which we recur on. If we terminate without ever creating negative numbers, the partition was realizable. Since we always connect the highest-degree vertex to other high-degree vertices, it is important to reorder the parts of the partition by size after each iteration.

Although this construction is deterministic, a semi-random collection of graphs realizing this degree sequence can be generated from G using *edge-flipping* operations. Suppose edges (x, y) and (w, z) are in G, but (x, w) and (y, z) are not. Exchanging these pairs of edges creates a different (not necessarily connected) graph without changing the degrees of any vertex.

Implementations: The Stanford GraphBase [Knu94] is perhaps most useful as an instance generator for constructing graphs to serve as test data for other programs. It incorporates graphs derived from interactions of characters in famous novels, Roget's Thesaurus, the Mona Lisa, expander graphs, and the economy of the United States. It also contains routines for generating binary trees, graph products, line graphs, and other operations on basic graphs. Finally, because of its machine-independent, random-number generators, it provides a way to construct random graphs such that they can be reconstructed elsewhere, thus making them perfect for experimental comparisons of algorithms. See Section 19.1.8 (page 660) for additional information.

Combinatorica [PS03] provides Mathematica generators for such graphs as stars, wheels, complete graphs, random graphs and trees, and graphs with a given degree sequence. Further, it includes operations to construct more interesting graphs from these, including join, product, and line graph.

The *Combinatorial Object Server* (*http://theory.cs.uvic.ca/*) developed by Frank Ruskey of the University of Victoria provides routines for generating both free and rooted trees.

Viger [VL05] has made available a C++ implementation of his algorithm to generate simple connected graphs with a prescribed degree sequence. See *http://www.liafa.jussieu.fr/~fabien/generation/*.

The graph isomorphism testing program Nauty (see Section 16.9 (page 550)) includes a suite of programs for generating nonisomorphic graphs, plus special generators for bipartite graphs, digraphs, and multigraphs. They are available at *http://cs.anu.edu.au/~bdm/nauty/*.

The mathematicians Brendan McKay (*http://cs.anu.edu.au/~bdm/data/*) and Gordon Royle (*http://people.csse.uwa.edu.au/gordon/data.html*) provide exhaustive catalogs of several families of graphs and trees up to the largest reasonable number of vertices.

Nijenhuis and Wilf [NW78] provide efficient Fortran routines to enumerate all labeled trees via Prüfer codes and to construct random unlabeled rooted trees. See Section 19.1.10 (page 661). Kreher and Stinson [KS99] generate labeled trees in C,

with implementations available at *http://www.math.mtu.edu/~kreher/cages/Src.html*.

Notes: Extensive literature exists on generating graphs uniformly at random. Surveys include [Gol93, Tin90]. Closely related to the problem of generating classes of graphs is counting them. Harary and Palmer [HP73] survey results in graphical enumeration.

Knuth [Knu06] is the best recent reference on generating trees. The bijection between $n-2$ strings and labeled trees is due to Prüfer [Prü18].

Random graph theory is concerned with the properties of random graphs. Threshold laws in random graph theory define the edge density at which properties such as connectedness become highly likely to occur. Expositions on random graph theory include [Bol01, JLR00].

The preferential attachment model of graphical evolution has emerged relatively recently in the study of networks. See [Bar03, Wat04] for introductions to this exciting field.

An integer partition is *graphic* if there exists a simple graph with that degree sequence. Erdős and Gallai [EG60] proved that a degree sequence is graphic if and only if the sequence observes the following condition for each integer $r < n$:

$$\sum_{i=1}^{r} d_i \leq r(r-1) + \sum_{i=r+1}^{n} \min(r, d_i)$$

Related Problems: Generating permutations (see page 448), graph isomorphism (see page 550).

INPUT	OUTPUT
December 21, 2012 ? (Gregorian)	5773 Teveth 8 (Hebrew) 1434 Safar 7 (Islamic) 1934 Agrahayana 30 (Indian Civil) 13.0.0.0 (Mayan Long Count)

14.8 Calendrical Calculations

Input description: A particular calendar date d, specified by month, day, and year.

Problem description: Which day of the week did d fall on according to the given calendar system?

Discussion: Many business applications need to perform calendrical calculations. Perhaps we want to display a calendar of a specified month and year. Maybe we need to compute what day of the week or year some event occurs, as in figuring out the date on which a 180-day futures contract comes due. The importance of correct calendrical calculations was perhaps best revealed by the furor over the "Millennium bug"—the year 2000 crisis in legacy programs that allocated only two digits for storing the year.

More complicated questions arise in international applications, because different nations and ethnic groups use different calendar systems. Some of these, like the Gregorian calendar used in most of the world, are based on the Sun, while others, like the Hebrew calendar, are lunar calendars. How would you tell today's date according to the Chinese or Islamic calendar?

Calendrical calculations differ from other problems in this book because calendars are historical objects, not mathematical ones. The algorithmic issues revolve around specifying the rules of the calendrical system and implementing them correctly, rather than designing efficient shortcuts for the computation.

The basic approach underlying calendar systems is to start with a particular reference date (called the *epoch*) and count up from there. The particular rules for wrapping the count around into months and years is what distinguishes one system from another. Implementing a calendar requires two functions, one that, given a date, returns the integer number of days that have elapsed since the epoch, the other of which takes an integer n and returns the calendar date exactly n days

from epoch. These are exactly analogous to the ranking and unranking rules for combinatorial objects such as permutations (see Section 14.4 (page 448)).

That the solar year is not an integer number of days long is the major source of complications in calendar systems. To keep a calendar's annual dates in sync with the seasons, leap days must be added at both regular and irregular intervals. Since a solar year is 365 days and 5:49:12 hours long, adding a leap day every four years leaves an extra 10 minutes and 48 seconds unaccounted for each year.

The original Julian calendar (from Julius Caesar) ignored these extra minutes, which had accumulated to ten days by 1582. Pope Gregory XIII then proposed the Gregorian calendar used today, by deleting the ten days and eliminating leap days in years that are multiples of 100 but not 400. Supposedly, riots ensued because the masses feared their lives were being shortened by ten days. Outside the Catholic church, resistance to change slowed the reforms. The deletion of days did not occur in England and America until September 1752, and not until 1927 in Turkey.

The rules for most calendrical systems are sufficiently complicated and pointless that you should lift code from a reliable place rather than attempt to write your own. We identify suitable implementations next.

There are a variety of "impress your friends" algorithms that enable you to compute in your head what day of the week a particular date occurred. Such algorithms often fail to work reliably outside the given century and certainly should be avoided for computer implementation.

Implementations: Readily available calendar libraries exist in both C++ and Java. The Boost time-data library provides a reliable implementation of the Gregorian calendar in C++. See *http://www.boost.org/doc/html/date_time.html*. The Calendar class in `java.util.Calendar` implements the Gregorian calendar in Java. Either of these will likely suffice for most applications.

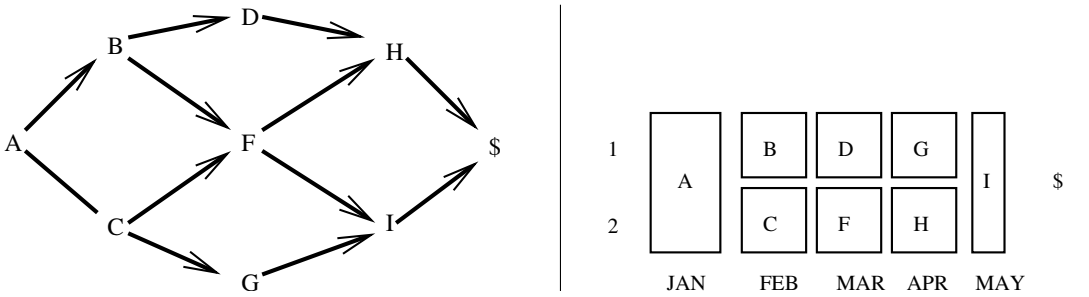
Dershowitz and Reingold provide a uniform algorithmic presentation [RD01] for a variety of different calendar systems, including the Gregorian, ISO, Chinese, Hindu, Islamic, and Hebrew calendars, as well as other calendars of historical interest. *Calendrical* is an implementation of these calendars in Common Lisp, Java, and Mathematica, with routines to convert dates between calendars, day of the week computations, and the determination of secular and religious holidays. *Calendrical* is likely to be the most comprehensive and reliable calendrical routines available. See their website at *http://calendarists.com*.

C and Java implementations of international calendars of unknown reliability are readily available at SourceForge (*http://sourceforge.net*). Search for "Gregorian calendar" to avoid the mass of datebook implementations.

Notes: A comprehensive discussion of calendrical computation algorithms appear in the papers of Dershowitz and Reingold [DR90, RDC93], which have been superseded by their book [RD01] that outlines algorithms for no less than 25 international and historical calendars Three hundred years of calendars representing tabulations for all dates from 1900 to 2200 appear in [DR02].

Concern exists in certain quarters whether December 21, 2012 represents the end of the world. The argument rests on it being the date the Mayan calendar spins over to 13.0.0.0.0 after a 5,125 year cycle. The reader should rest assured, since I would never have devoted so much effort to writing this book were the world to be ending so imminently. The Mayan calendar is authoritatively described in [RD01].

Related Problems: Arbitrary-precision arithmetic (see page 423), generating permutations (see page 448).



INPUT OUTPUT

14.9 Job Scheduling

Input description: A directed acyclic graph $G = (V, E)$, where vertices represent jobs and edge (u, v) implies that task u must be completed before task v.

Problem description: What schedule of tasks completes the job using the minimum amount of time or processors?

Discussion: Devising a proper schedule to satisfy a set of constraints is fundamental to many applications. Mapping tasks to processors is a critical aspect of any parallel-processing system. Poor scheduling can leave all other machines sitting idle while one bottleneck task is performed. Assigning people-to-jobs, meetings-to-rooms, or courses-to-exam periods are all different examples of scheduling problems.

Scheduling problems differ widely in the nature of the constraints that must be satisfied and the type of schedule desired. Several other catalog problems have application to various kinds of scheduling:

- Topological sorting can construct a schedule consistent with the precedence constraints. See Section 15.2 (page 481).
- Bipartite matching can assign a set of jobs to people who have the appropriate skills for them. See Section 15.6 (page 498).
- Vertex and edge coloring can assign a set of jobs to time slots such that no two interfering jobs are assigned the same time slot. See Sections 16.7 and 16.8.
- Traveling salesman can schedule select the most efficient route for a delivery person to visit a given set of locations. See Section 16.4 (page 533).

- Eulerian cycle can construct the most efficient route for a snowplow or mailman to completely traverse a given set of edges. See Section 15.7 (page 502).

Here we focus on precedence-constrained scheduling problems for directed acyclic graphs. Suppose you have broken a big job into a large number of smaller tasks. For each task, you know how long it should take (or perhaps an upper bound on how long it might take). Further, for each pair of tasks you know whether it is essential that one task be performed before the other. The fewer constraints we have to enforce, the better our schedule can be. These constraints must define a directed acyclic graph—acyclic because a cycle in the precedence constraints represents a Catch-22 situation that can never be resolved.

We are interested in several problems on these networks:

- *Critical path* – The longest path from the start vertex to the completion vertex defines the *critical path*. This can be important to know, for the only way to shorten the minimum total completion time is to reduce the time of one of the tasks on each critical path. The tasks on the critical paths can be determined in $O(n + m)$ time using the dynamic programming presented in Section 15.4 (page 489).

- *Minimum completion time* – What is the fastest we can get this job completed while respecting precedence constraints, assuming that we have an unlimited number of workers. If there were *no* precedence constraints, each task could be worked on by its own worker, and the total time would be that of the longest single task. If there are such strict precedence constraints that each task must follow the completion of its immediate predecessor, the minimum completion time would be obtained by summing up the times for each task.

  The minimum completion time for a DAG can be computed in $O(n+m)$ time. The completion time is defined by the critical path. To get such a schedule, consider the jobs in topological order, and start each job on a new processor the moment its latest prerequisite completes.

- *What is the tradeoff between the number of workers and completion time?* – What we really are interested in is how best to complete the schedule with a given number of workers. Unfortunately, this and most similar problems are NP-complete.

Any real scheduling application is likely to present combinations of constraints that will be difficult or impossible to model using these techniques. There are two reasonable ways to deal with such problems. First, we can ignore constraints until the problem reduces to one of the types that we have described here, solve it, and then see how bad it is with respect to the other constraints. Perhaps the schedule can be easily modified by hand to satisfy other esoteric constraints, like keeping Joe and Bob apart so they won't kill each other. Another approach is to formulate

your scheduling problem via linear-integer programming (see Section 13.6 (page 411)) in all its complexity. I always recommend starting out with something simple and see how it works before trying something complicated.

Another fundamental scheduling problem takes a set of jobs without precedence constraints and assigns them to identical machines to minimize the total elapsed time. Consider a copy shop with k Xerox machines and a stack of jobs to finish by the end of the day. Such tasks are called *job-shop scheduling*. They can be modeled as bin-packing (see Section 17.9 (page 595)), where each job is assigned a number equal to the number of hours it will take to complete, and each machine is represented by a bin with space equal to the number of hours in a day.

More sophisticated variations of job-shop scheduling provide each task with allowable start and required finishing times. Effective heuristics are known, based on sorting the tasks by size and finishing time. We refer the reader to the references for more information. Note that these scheduling problems become hard only when the tasks cannot be broken up onto multiple machines or interrupted (preempted) and then rescheduled. If your application allows these degrees of freedom, you should exploit them.

Implementations: JOBSHOP is a collection of C programs for job-shop scheduling created in the course of a computational study by Applegate and Cook [AC91]. They are available at *http://www2.isye.gatech.edu/∼wcook/jobshop/*.

Tablix (*http://tablix.org*) is an open source program for solving timetabling problems faced by real schools. Support for parallel/cluster computation is provided.

LEKIN is a flexible job-shop scheduling system designed for educational use [Pin02]. It supports single machine, parallel machines, flow-shop, flexible flow-shop, job-shop, and flexible job-shop scheduling, and is available at *http://www.stern.nyu.edu/om/software/lekin*.

For commercial scheduling applications, ILOG CP is reflective of the state-of-the-art. For more, see *http://www.ilog.com/products/cp/*.

Algorithm 520 [WBCS77] of the *Collected Algorithms of the ACM* is a Fortran code for multiple-resource network scheduling. It is available from Netlib (see Section 19.1.5 (page 659)).

Notes: The literature on scheduling algorithms is a vast one. Brucker [Bru07] and Pinedo [Pin02] provide comprehensive overviews of the field. The *Handbook of Scheduling* [LA04] provides an up-to-date collection of surveys on all aspects of scheduling.

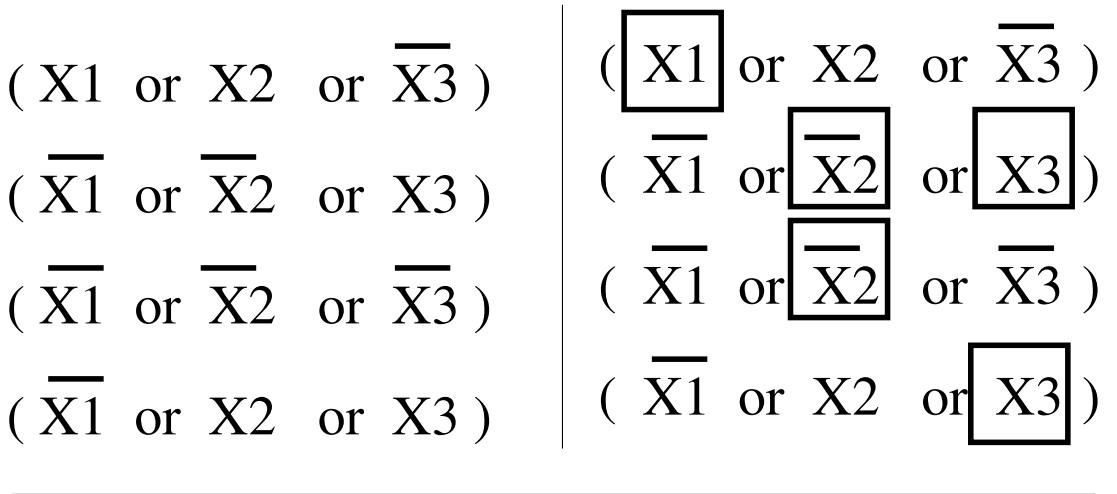
A well-defined taxonomy exists covering thousands of job-shop scheduling variants, which classifies each problem $\alpha|\beta|\gamma$ according to (α) the machine environment, (β) details of processing characteristics and constraints, and (γ) the objectives to be minimized. Surveys of results include [Bru07, CPW98, LLK83, Pin02].

Gantt charts provide visual representations of job-shop scheduling solutions, where the x-axis represents time and rows represent distinct machines. The output figure above illustrates a Gantt chart. Each scheduled job is represented as a horizontal block, thus identifying its start-time, duration, and server. Project precedence-constrained scheduling

techniques are often called PERT/CPM, for *Program Evaluation and Review Technique/Critical Path Method*. Both Gantt charts and PERT/CPM appear in most textbooks on operations research, including [Pin02].

Timetabling is a term often used in discussion of classroom and related scheduling problems. PATAT (for *Practice and Theory of Automated Timetabling*) is a biannual conference reporting new results in the field. Its proceedings are available from *http://www.asap.cs.nott.ac.uk/patat/patat-index.shtml*.

Related Problems: Topological sorting (see page 481), matching (see page 498), vertex coloring (see page 544), edge coloring (see page 548), bin packing (see page 595).



14.10 Satisfiability

Input description: A set of clauses in conjunctive normal form.

Problem description: Is there a truth assignment to the Boolean variables such that every clause is simultaneously satisfied?

Discussion: Satisfiability arises whenever we seek a configuration or object that must be consistent with (i.e. , satisfy) a set of logical constraints. A primary application is in verifying that a hardware or software system design works correctly on all inputs. Suppose a given logical formula $S(\bar{X})$ denotes the specified result on input variables $\bar{X} = X_1, \ldots, X_n$, while a different formula $C(\bar{X})$ denotes the Boolean logic of a proposed circuit for computing $S(\bar{X})$. This circuit is correct unless there exists an $\bar{X}$ such that $S(\bar{X}) \neq C(\bar{X})$.

Satisfiability (or SAT) is *the* original NP-complete problem. Despite its applications to constraint satisfaction, logic, and automatic theorem proving, it is perhaps most important theoretically as the root problem from which all other NP-completeness proofs originate. So much engineering has gone into today's best SAT solvers that they represent a reasonable starting point if one really needs to solve an NP-complete problem *exactly*. That said, employing heuristics that give good but nonoptimal solutions is usually the better approach in practice.

Issues in satisfiability testing include:

- *Is your formula the AND of ORs (CNF) or the OR of ANDs (DNF)?* – In satisfiability, constraints are specified as a logical formula. There are two

primary ways of expressing logical formulas—conjunctive normal form (CNF) and disjunctive normal form (DNF). In CNF formulas, we must satisfy all clauses, where each clause is constructed by and-ing or's of literals together, such as

$$(v_1 \text{ or } \bar{v_2}) \text{ and } (v_2 \text{ or } v_3)$$

In DNF formulas, we must satisfy any one clause, where each clause is constructed by or-ing ands of literals together. The formula above can be written in DNF as

$$(\bar{v_1} \text{ and } \bar{v_2} \text{ and } v_3) \text{ or } (\bar{v_1} \text{ and } v_2 \text{ and } \bar{v_3}) \text{ or } (\bar{v_1} \text{ and } v_2 \text{ and } v_3) \text{ or } (v_1 \text{ and } \bar{v_2} \text{ and } v_3)$$

Solving DNF-satisfiability is trivial, since any DNF formula can be satisfied unless *every* clause contains both a literal and its complement (negation). However, CNF-satisfiability is NP-complete. This seems paradoxical, since we can use De Morgan's laws to convert CNF-formulae into equivalent DNF-formulae, and vice versa. The catch is that an exponential number of terms might be constructed in the course of translation, so that the translation itself does not run in polynomial time.

- *How big are your clauses?* – k-SAT is a special case of satisfiability when each clause contains at most k literals. The problem of 1-SAT is trivial, since we must set true any literal appearing in any clause. The problem of 2-SAT is not trivial, but can still be solved in linear time. This is interesting, because certain problems can be modeled as 2-SAT using a little cleverness. The good times end as soon as clauses contain three literals each (i.e. , 3-SAT) for 3-SAT is NP-complete.

- *Does it suffice to satisfy* most *of the clauses?* – If you must solve it exactly, there is not much you can do to solve satisfiability except by backtracking algorithms such as the Davis-Putnam procedure. In the worst case, there are 2^m truth assignments to be tested, but fortunately there are many ways to prune the search. Although satisfiability is NP-complete, how hard it is in practice depends on how the instances are generated. Naturally defined "random" instances are often surprisingly easy to solve, and in fact it is nontrivial to generate instances of the problem that are truly hard.

  Still, we can benefit by relaxing the problem so that the goal becomes satisfying as many clauses as possible. Optimization techniques such as simulated annealing can then be put to work to refine random or heuristic solutions. Indeed, any random truth assignment to the variables will satisfy each k-SAT clause with probability $1 - (1/2)^k$, so our first attempt is likely to satisfy most of the clauses. Finishing off the job is the hard part. Finding an assignment that satisfies the maximum number of clauses is NP-complete even for nonsatisfiable instances.

When faced with a problem of unknown complexity, proving it NP-complete can be an important first step. If you think your problem might be hard, skim through Garey and Johnson [GJ79] looking for your problem. If you don't find it, I recommend that you put the book away and try to prove hardness from first principles, using the basic problems of 3-SAT, vertex cover, independent set, integer partition, clique, and Hamiltonian cycle. I find it much easier to start from these than some complicated problem in the book, and more insightful too, since the reason for the hardness is not obscured by the hidden hardness proof for the complicated problem. Chapter 9 focuses on strategies for proving hardness.

Implementations: Recent years have seen tremendous progress in the performance of SAT solvers. An annual SAT competition identifies the top performing solvers in each of three categories of instances (drawn from industrial, handmade, and random problem instances, respectively).

The top three programs of the SAT 2007 industrial competition were Rsat (*http://reasoning.cs.ucla.edu/rsat/*), PicoSAT (*http://fmv.jku.at/picosat/*) and MiniSAT (*http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/*). The source code for all these solvers and more are available from the competition webpage (*http://www.satcompetition.org/*).

SAT *Live!* (*http://www.satlive.org/*) is the most up-to-date source for papers, programs, and test sets for satisfiability and related logic optimization problems.

Notes: The most comprehensive overview of satisfiability testing is Kautz, et al. [KSBD07]. The Davis-Putnam-Logemann-Loveland (DPLL) algorithm is a backtracking algorithm introduced in 1962 for solving satisfiability problems. Local search techniques work better on certain classes of problems that are difficult for DPLL solvers. *Chaff* [MMZ$^+$01] is a particularly influential solver, available at *http://www.princeton.edu/∼chaff/*. See [KS07] for a survey of recent progress in the field of satisfiability testing.

An algorithm for solving 3-SAT in worst-case $O^*(1.4802^n)$ appears in [DGH$^+$02]. Efficient (but nonpolynomial) algorithms for NP-complete problems are surveyed in [Woe03].

The primary reference on NP-completeness is [GJ79], featuring a list of roughly 400 NP-complete problems. The book remains an extremely useful reference; it is perhaps the book I reach for most often. An occasional column by David Johnson in the *Journal of Algorithms* and (now) *ACM Transactions on Algorithms* provides updates.

Good expositions of Cook's theorem [Coo71], where satisfiability is proven hard, include [CLRS01, GJ79, KT06]. The importance of Cook's result became clear in Karp's paper [Kar72], showing the hardness of over 20 different combinatorial problems.

A linear-time algorithm for 2-SAT appears in [APT79]. See [WW95] for an interesting application of 2-SAT to map labeling. The best heuristic known approximates maximum 2-SAT to within a factor of 1.0741 [FG95].

Related Problems: Constrained optimization (see page 407), traveling salesman problem (see page 533).