5

Combinatorial Algorithms

Combinatorial algorithms is a vast field, so vast in fact that even Donald Knuth is unsure how long Vol. 4 on Combinatorial Algorithms in *The Art of Computer Programming* (TAOCP) will be. As we write this in 2024, Volumes 4A, 4B, and 4C (very soon) are available, with more pre-fascicles, fascicles, and sub-volumes projected. A good collection of links to these can be found at the end of TAOCP's Wikipedia entry (`https://en.wikipedia.org/wiki/The_Art_of_Computer_Programming`).

Most beginner texts start with permutations, combinations, sorting, and partitioning, moving onto undirected and directed graphs to focus on algorithms related to search, shortest paths, spanning trees, and optimization problems.

We've decided to follow a similar approach, but rather than be too explicit about graphs, that topic is introduced through a series of recreational search puzzles (Sec. 5.5) and games (Sec. 5.8).

5.1 Sorting

Sorting is a major topic in computer science education since it provides a straightforward way to introduce concepts such as best, worst, and average case analysis, time–space tradeoffs, big-Oh notation, recursion, and data structures such as heaps and binary trees. The textbooks by Cormen *et al.* [**CLRS22**] and Sedgewick and Wayne [**SW11**] cover all of these topics excellently, as does Vol. 3 of Knuth's classic work [**Knu97**].

Sorting is an active research area – the widely used Timsort dates from 2002 [`https://en.wikipedia.org/wiki/Timsort`), and library sort was released

in 2006 (`https://en.wikipedia.org/wiki/Library_sort`). Timsort is derived from merge sort and insertion sort (described below), and was Python's standard sorting algorithm from version 2.3 to version 3.10 (until 2022), when it was dethroned by powersort (`https://www.wild-inter.net/posts/powersort-in-python-3.11`).

Fig. 5.1 summarizes the seven algorithms presented in this section.

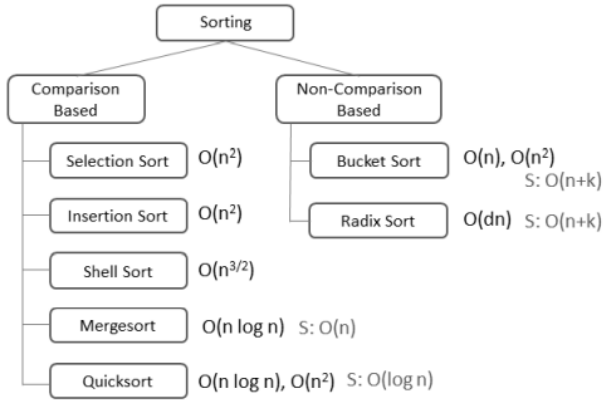


Figure 5.1. Seven sorting algorithms

Comparison sorts compare elements to determine which of them should be placed ahead of the other in the sorted structure. Usually a "less than or equal to" operation is utilized, but the only requirement is that the comparator forms a total preorder over the data, with:

- if $a \leq b$ and $b \leq c$ then $a \leq c$
- for all a and b, $a \leq b$ or $b \leq a$.

It's entirely possible that $a \leq b$ *and* $b \leq a$, and in a *stable* sort, the input order determines the sorted order in this case.

The fastest comparison sorts have an average-case lower bound of $O(n \log n)$ comparisons (n refers to the number of elements in the list). In this sense, mergesort and quicksort are optimal, but then other metrics, such as memory usage, become important.

Non-comparison sorts achieve $O(n)$ performance by using operations other than comparisons, allowing them to sidestep the $n \log n$ lower bound. However, they may lose out in other ways, such as by requiring more memory or by having to make assumptions about the type and range of the data.

Algorithms are usually analyzed in terms of their best, worst, and average running times, i.e. time complexity, but memory usage is also a common measure. Average and worst-case performance are most frequently used in algorithm

analysis, and we'll restrict ourselves to those two in what follows. Calculating an average running time can be quite difficult since we have to assume that the data was produced by a random process, typically one employing a uniform probability distribution.

Up until now, we've been using $O()$ (big-Oh notation) to describe the limiting behavior of functions, but there are several related notations, including $\Omega()$ and $\Theta()$ for other kinds of bounds on asymptotic growth rates. To summarize:

- $f(x) = O(g(x))$ means that the growth rate of the function $f(x)$ is asymptotically *less than or equal to* the growth rate of the function $g(x)$.
- $f(x) = \Omega(g(x))$ means that the growth rate of $f(x)$ is asymptotically *greater than or equal to* the growth rate of $g(x)$.
- $f(x) = \Theta(g(x))$ means that the growth rate of $f(x)$ is asymptotically *equal to* the growth rate of $g(x)$.

In short, O(n) represents an upper bound, $\Omega(n)$ a lower bound, and $\Theta(n)$ a tight (close) bound. We're going to keep on using $O()$ for simplicity's sake.

In Fig. 5.1, the big-Oh values preceded by 'S:' relate to space usage, and the algorithms without space big-Oh values should be understood to utilize constant amounts of memory (i.e. O(1)).

When there are two big-Oh values next to a sort name, the first is its average running time, the second its worst case time. The sorts with a single big-Oh have the same average and worst case running times.

An excellent way to gain an intuition about the seven algorithms in this section is to watch animations of their sorts at `https://m.youtube.com/watch?v=kPRA0W1kECg`. Altogether, fifteen sorting methods are covered by the video.

5.1.1 Selection Sort.

We proceed as follows: find the smallest element in the elems[0..n-1] list and exchange it with elems[0]. Then find the smallest element in elems[1..n-1] and exchange it with elems[1], progressing in this manner until the whole list is sorted. The code appears in Listing 5.1 (selectionsort.py).

```
def selectionSort(elems):
  n = len(elems)
  for idx in range(n):
    minIdx = idx
    for j in range(idx+1, n):
      # Find index of smallest number
      if elems[j] < elems[minIdx]:
        minIdx = j
    # swap elements
    elems[idx], elems[minIdx] = elems[minIdx], elems[idx]
```

Listing 5.1. Selection sort

The running time is $O(n^2)$ in both the worst and average cases. This can be confirmed by looking at the nested loops – the outer one iterates n times, while the inner one executes $n-1$ times on its first loop, $n-2$ times on its second, and so on. The total number of iteration is $(n-1)+(n-2)+...+1$, which is $\frac{(n-1)n}{2}$ or $O(n^2)$.

5.1.2 Insertion Sort. Insertion sort is almost as simple as selection sort, and may be familiar to card players as the way they sort their hands: look at the cards one at a time, inserting each in its proper place among those already sorted. The code appears in Listing 5.2 (insertionsort.py).

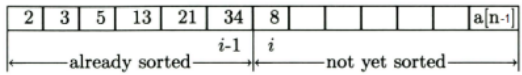


Figure 5.2. Partially sorted

```
def insertionSort(elems):
  n = len(elems)
  if n > 1:
    for i in range(1, n):
      val = elems[i]
      j = i-1
      while j >= 0 and val < elems[j]: # Move elems > val
        elems[j+1] = elems[j]  # Shift to right
        j -= 1
      elems[j+1] = val
```

Listing 5.2. Insertion sort

In common with selection sort, insertion sort's worst and average case running times are both $O(n^2)$. The n elements of the list can be viewed as $n(n-1)/2$ pairs. Each pair is out of order with a probability of 0.5, so the expected number of out of order pairs is $n(n-1)/4$. To bring them into order requires $n(n-1)/4$ exchanges, or $O(n^2)$ running time.

5.1.3 Shell Sort. Since insertion sort's running time is $O(n^2)$, it will take less time to sort h lists of n/h elements than to sort a single list of n elements. Shell sort takes advantage of this behavior.

For a given value of h it separately sorts sublists consisting of elements which are multiples of h apart. For large values of h, this will move elements long distances and, in general, rearrange the list to be much closer to its natural order. The algorithm incrementally switches to smaller h-values until finally $h = 1$, which is just insertion sort.

Some sequences of h-values seem to sort faster than others. Knuth recommends ..., 1093, 364, 121, 40, 13, 4, 1 [**Knu97**], with each h obtained from the next by the substitution $h_{i+1} = 3h_i + 1$. Rather foolishly, we've ignored this advice in Listing 5.3 (shellsort.py), and gone with the easier to code halving of h, starting at $n \mathbin{//} 2$. An interesting exercise is to measure how much speed gain comes from the use of $3h + 1$ instead.

```
def shellSort(elems):
  n = len(elems)
  h = n//2
  while h > 0:
    for i in range(h, n):
      val = elems[i]
      j = i
      while j >= h and val < elems[j-h]:
        elems[j] = elems[j-h]
        j -= h
      elems[j] = val
    h = h//2
```

Listing 5.3. Shell sort

Shell sort's worst case and average case time complexity depend very much on the h-sequence, so it's hard to calculate an accurate O() value. Pratt (1979) has shown that the worst case is proportional to $O(n^{3/2})$ for many sequences [**SW11**], but some researchers believe that shell sort may be able to run in an average time of $O(n \log n)$ for h-sequences that grow in proportion to the logarithm of the list size. In Fig. 5.1, we've conservatively assigned shell sort a running time of $O(n^{3/2})$, but it's a lot faster than that in practice, as indicated by the running time graph in Fig. 5.5b.

5.1.4 Mergesort. Mergesort is a divide-and-conquer algorithm invented by John von Neumann in 1945. Assuming there's an n-element list, elems, divide it into two sublists, each of half the size. To do so, compute the midpoint mid, and divide elems[left..right] into sublists elems[left..mid] and elems[(mid+1)..right]. Sort each sublist recursively using mergesort, only stopping when it reaches a sublist with 0 or 1 element.

The sublists are recombined by *merging* as the recursion levels created during the 'conquer' stage return. The merging can assume that adjacent sublists elems[left..mid] and elems[(mid +1)..right] are sorted so it only needs to compare the first elements in each sublist to decide which one is smaller and should be moved to the larger sublist elems[left..right]. It repeats this comparison with the next first elements of the sublists, continuing until one of the lists is empty, which causes the other to be copied over in its entirety. Merging requires the creation of a temporary list so that the sublists aren't overwritten while the merged

larger one is being built. Merging takes time roughly proportional to the length of the full list, so is $O(n)$.

The top-level code is given in Listing 5.4 (the complete program is in mergeSort.py). The recursive division of the list is managed by msort(), and merging the resulting sublists is handled by merge().

```
def mergeSort(elems):
  msort(elems, 0, len(elems)-1)

def msort(elems, left, right):
  if left < right:
    mid = left+(right-left)//2
    # Sort halves
    msort(elems, left, mid)
    msort(elems, mid+1, right)
    merge(elems, left, mid, right)
```

Listing 5.4. Mergesort

5.1.4.1 Mergesort Big-Oh Analysis. Let $T(n)$ be the maximum number of comparisons used while merge-sorting a list of n numbers. To simplify matters, let's assume that n is a power of 2, which ensures that the input can be divided in half at every stage of the recursion. If there's only one number in the list, then no comparisons are required, so $T(1) = 0$. More generally, the $T(n)$ time includes the time sorting the first half (at most $T(n/2)$), sorting the second half (also at most $T(n/2)$), and merging the two sorted halves (at most n). Therefore, the number of comparisons needed to mergesort n items becomes the recurrence:

$$\begin{aligned} T(1) &= 0 \\ T(n) &= 2T(n/2) + n \quad \text{for } n \geq 2 \end{aligned}$$

The easiest way to obtain a closed form for $T()$ is to expand the recurrence relation until we (hopefully) see a pattern:

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2(2T(n/4) + n/2) + n &&= 4T(n/4) + 2n \\ &= 4(2T(n/8) + n/4) + 2n &&= 8T(n/8) + 3n \\ &= 8(2T(n/16) + n/8) + 3n &&= 16T(n/16) + 4n \end{aligned}$$

A pattern emerges when we notice that $16 = 2^4$. Let's generalize the division on the right-hand side from 16 to 2^k, and also rewrite the $4n$ term as kn:

$$T(n) = 2^k T(n/(2^k)) + kn, \text{ for } k = 1, 2, ...$$

As k increases, the $n/2^k$ argument on the right-hand side keeps halving, reaching 1 when $n = 2^k$, or equivalently when $\log_2 n = k$. Substitute this value of k into

the equation:

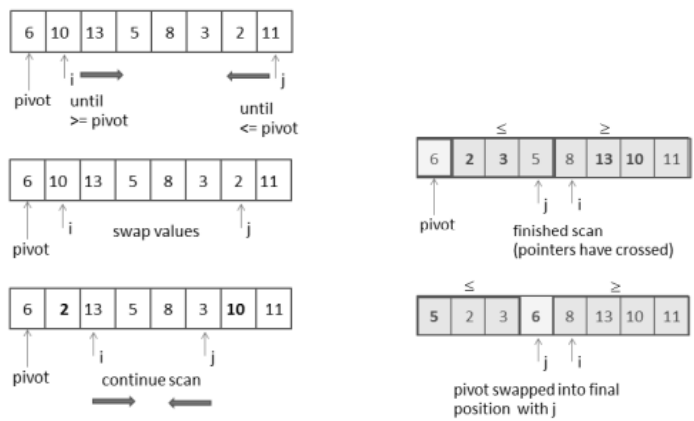
$$\begin{aligned} T(n) &= 2^k T(n/(2^k)) + kn \\ &= nT(n/n) + \log_2 n \cdot n \\ &= nT(1) + n \log_2 n \\ &= n \log_2 n \end{aligned}$$

Thus, $T(n)$ is $O(n \log_2 n)$, but a drawback is that it requires memory of size $O(n)$ for the temporary lists required by merge().

5.1.5 Quicksort. Quicksort, due to C.A.R. Hoare (1960), is a popular, fast, general sorting method. It employs divide-and-conquer like mergesort, but without the extra memory overheads.

The algorithm starts by partitioning a list into two, then sorts them by recursively calling itself on the two parts. The partitioning of list elems[left..right] moves all the elements smaller than a chosen pivot value to the left, and all the bigger elements to the right. The pivot can be any element in the list, so we'll choose the leftmost value for simplicity. However, quicksort is only 'quick' if the pivot divides the list into two roughly equal sized halves.

The movement of the smaller elements to the left and bigger elements to the right is done by scanning the list from the left until an element bigger (or equal) to the pivot is found, *and* from the right until an element smaller (or equal) is found. The two elements are swapped, and scanning and swapping continues until the two traversals meet. These steps are illustrated in Fig. 5.3a.



(a) Part 1 (b) Part 2

Figure 5.3. Partitioning a list

When the i and j pointers in Fig. 5.3a cross, the list can be considered to have three sections (which are shown in various shades of gray in Fig. 5.3b). The pivot is at the start of the list, the values $\leq$ the pivot are stored up to the j position, and the values $\geq$ the pivot complete the list.

The last step swaps the pivot value with the j value, causing it to finish in its correct sorted position. Note that the values on either side of the pivot are *not* yet sorted, but they are $\leq$ and $\geq$ than the pivot.

The partition() code is in quicksort.py.

Quicksort has and average case running time of $O(n \log n)$. Unfortunately, its worst case is $O(n^2)$, which occurs when the pivot divides the list into two very unequal parts, but that occurs rarely in practice.

Python's list comprehensions allows partitioning to be implemented very simply in Listing 5.5 (quicksort.py). However, this version is a little less time efficient than the original since two passes are made over the list to create the sublists. It's also less space efficient since the comprehensions create new sublists rather than carry out in-place repositioning.

```
def quickSortL(elems):
  if len(elems) <= 1:
    return elems
  else:
    pivot = elems[0]  # use left-most
    left = [x for x in elems[1:] if x < pivot]
    right = [x for x in elems[1:] if x >= pivot]
    return quickSortL(left) + [pivot] + quickSortL(right)
```

Listing 5.5. Slower but elegant quicksort

This version also needs to assign its result back to elems in order to change the original:

```
elems = quickSortL(elems)
```

5.1.5.1 Quicksort Average-Case Analysis. We assume that quicksort is sorting a list S of n elements. The partitioning function P, will return an index that partitions the list into S_L (left) and S_R (right). We have $P(n) = O(n)$, because this work can be done in one pass through S. Hence the running time, $T(n)$ is given by:

$$T(n) = T(i) + T(n - i - 1) + n, \text{ for } n > 1, \text{ and } T(0) = T(1) = 1, \tag{5.1}$$

where $i = |S_L|$ is the size of the left sublist.

We'll assume that every possible size of S_L (and consequently S_R) is equally likely (this is a probabilistic argument, which is needed when dealing with average case analyses). Possible sizes are $0, 1, ..., n - 1$, each having a probability $1/n$.

Hence, the average value of $T(n)$ in Equ. (5.1) is given by,

$$T(n) = \frac{1}{n}\left[\sum_{i=0}^{n-1} T(i) + T(n-i-1)\right] + n \tag{5.2}$$

Since $\sum_{i=0}^{n-1} T(i) = \sum_{i=0}^{n-1} T(n-i-1)$, $T(n)$ can be written as,

$$T(n) = \frac{2}{n}\left[\sum_{i=0}^{n-1} T(i)\right] + n \tag{5.3}$$

Multiplying by n, we get

$$nT(n) = 2\left[\sum_{i=0}^{n-1} T(i)\right] + n^2 \tag{5.4}$$

Replacing n by $n-1$:

$$(n-1)T(n-1) = 2\left[\sum_{i=0}^{n-2} T(i)\right] + (n-1)^2 \tag{5.5}$$

Subtracting Equ. (5.5) from Equ. (5.4), we obtain,

$$nT(n) - (n-1)T(n-1) = 2T(n-1) + 2n - 1$$

Rearranging and simplifying, produces

$$nT(n) = (n+1)T(n-1) + 2n$$

Dividing both sides by $n(n+1)$:

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2}{n+1} \tag{5.6}$$

We can now repeatedly expand the $T()$ expression on the right-hand side by applying Equ. (5.6) multiple times:

$$\begin{aligned}
\tfrac{T(n)}{n+1} &= \tfrac{T(n-2)}{n-1} + \tfrac{2}{n} + \tfrac{2}{n+1} \\
&= \tfrac{T(n-3)}{n-2} + \tfrac{2}{n-1} + \tfrac{2}{n} + \tfrac{2}{n+1} \\
&= \cdots \\
&= \tfrac{T(1)}{2} + 2\textstyle\sum_{i=3}^{n+1} \tfrac{1}{i} \\
&\approx \tfrac{1}{2} + 2[\ln(n+1) + \gamma - \tfrac{3}{2}]
\end{aligned}$$

In the last line, we've replaced the harmonic series summation by its approximation using Euler's constant. Therefore,

$$\frac{T(n)}{n+1} = O(\log n)$$

and hence,

$$T(n) = O(n \log n)$$

Exercises

(1) Suppose $n = 2^m$ equal numbers are sorted by quicksort. Show that $0.5n \log_2 n$ interchanges will be made.

(2) Write a function that sorts a list that is known to have at most three different values. (Edsgar Dijkstra called this the Dutch-national-flag problem because the result is three "stripes" of values like the three stripes in the Dutch flag.)

Hint: First partition the list into two parts with all elements having the smallest value in the first part and all other elements in the second part, then partition the second part.

5.1.6 Bucket Sort. If we have a list of data that is known to reside in a certain range; for example, floats between 0 and 1, or integers between 1900 and 2100, then we can employ a bucket sort to obtain close to linear running times for the sorting. This is possible because we don't need to do comparisons on the data (or at least very few comparisons), instead relying on the separation of the data into buckets. Each bucket will represent a subrange of the data, hopefully small enough so that only one data item will be assigned to each bucket. Sorting then becomes a matter of scanning the filled buckets in order.

Listing 5.6 (bucketsort2.py) can sort lists that contain floats between 0 and 1. If the list has n elements, then n buckets are created in the hope that each data element will be assigned to its own bucket. This best case may not occur, so after the data has been distributed, insertion sort is used to sort each bucket. Since there will probably only be a few elements per bucket this step will be very fast, with near linear running time. The final step is to combine the buckets into a single list.

```
def bucketSort(elems):
  n = len(elems)
  buckets = [[] for _ in range(n)]
  # Put elements in different buckets
  for num in elems:
    bi = int(n * num)
    buckets[bi].append(num)
  # Sort individual buckets using insertion sort
  for bucket in buckets:
    insertionSort(bucket)
  # Concatenate all buckets into elems[]
  index = 0
  for bucket in buckets:
    for num in bucket:
      elems[index] = num
      index += 1
```

Listing 5.6. Bucket sort

The only tricky aspect of Listing 5.6 is how a data item is mapped to a particular bucket. This will vary depending on the type of data and the number of buckets. In this case, we're assuming that the data is uniformly distributed, so its sufficient to multiply each value by n then cast it to an integer to get a bucket index. The underlying aim is to avoid placing a large amount of the data in a single bucket.

The worst-case running time occurs when all the elements are placed in a single bucket. The overall performance will then be dominated by the algorithm used to sort each bucket – $O(n^2)$ for insertion sort. The average case however will be as sketched above and so the three loops inside bucketSort() will each be linear, resulting in a running time of $O(n)$.

The speed-up by not using comparison tests is not cost free. In the case of bucket sort, extra memory is needed for the buckets. Typically, if the data has n elements, and there are k buckets, then the memory usage will be $O(n + k)$.

5.1.7 Radix Sort. Radix sort dates from the 1890 US census, developed by Herman Hollerith for his punched card-sorting machines which tallied the results. A sorter could be mechanically 'programmed' to examine a given column on each card in a deck and drop the card into a bin depending on which place had been punched. An operator would gather the cards bin by bin, and order them so that those with the first place punched were on top of the cards with the second place punched, and so on.

Hollerith's original (inefficient) idea was to sort by starting with the most significant digit, then move right. This generates lots of intermediate piles of cards (i.e. extra temporary memory) to keep track of the calculations. A surprisingly simple fix is to sort on the least-significant digit first, and move to the left. The pseudo-code is:

```
def radixSort(elems, digits):
  for i in range(digits):
    stableSort(elems) on digit i
```

The use of a 'stable' sort is necessary so that cards with the same digit in a particular column stay in their original relative order. If this ordering isn't maintained, then a sort of a later digit may position the cards incorrectly.

Fig. 5.4 shows the operation of radix sort on seven 3-digit positive integers. The right-most 1's digits are sorted first, then the 10's digits, and so on. After three iterations (the number of digits in each number), the list is sorted.

radixSort.py uses a variant of the bucket sort from the last section. The main change is to hardwire the use of ten buckets representing the ten possible values

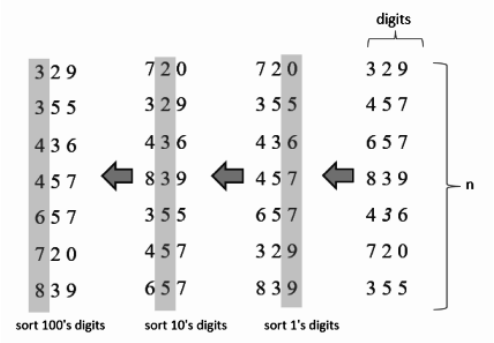


Figure 5.4. Radix sort of seven 3-digit numbers

for a decimal digit. This bucket sort version is stable because it no longer calls insertion sort.

Big-Oh calculations for radix sort typically involve three variables: n (the size of the list), k (the range of values for a single digit), and d (the maximum number of digits used by a value in the data set). In Fig. 5.4, $n = 7$, $k = 10$, and $d = 3$.

Each call to bucket sort will sort n numbers based on looking at a particular column of digits, so its running time will be $O(n)$. This work will be repeated by the outer loop which examines each digit position 0, 1, ... $d - 1$. Therefore its total running time is $O(dn)$. Since d is often a constant, we can simplify this to $O(n)$.

One downside of radix sort, as illustrated in the code, is the need to calculate the number of digits d, which requires a linear scan of the list before sorting can begin. Another drawback is the assumption that the data consists of positive base-10 integers. Radix sort can be used on other types of data, such as floats and binary data, but different coding tricks are needed, based on the range of values.

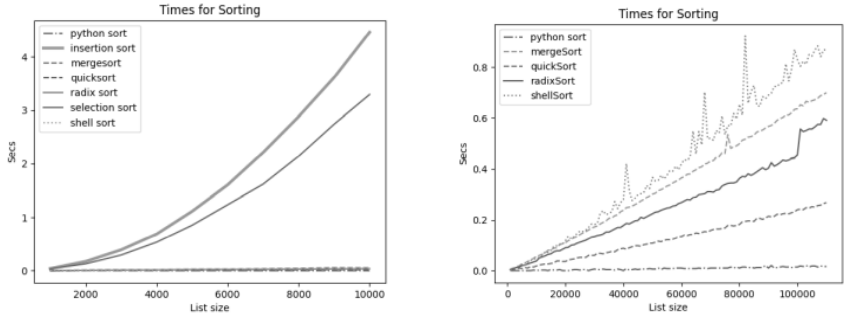
A less obvious weakness, compared to quicksort for example, is poor *locality of reference* which means that the algorithm may need to access values widely spread across the list when comparing digits. If the list is very big, such widely dispersed data may not be cached in RAM, requiring the OS to do a slow access to the hard disk to get it.

5.1.8 Which is Fastest? sortTimings.py imports six of the seven sorting functions previously described, excluding Listing 5.6 since it was designed to work with floats between 0 and 1. However, bucket sort *is* represented here since radix sort uses a version suitable for positive integers. `sortTimings.py` generates random integer lists of various sizes between 1000 and 10000 elements, and calls timer() in sortTimings.py to obtain an average time for each sort function. The timing data for each sort are plotted in Fig. 5.5a.

The incredibly fast 'python sort' is Python's built-in powersort, which is utilized by timer() calling:

```
def pSort(elems):
  elems.sort()
```

There are three groups of timings in Fig. 5.5a – sorts which have quadratic running time (selection sort, insertion sort), those with $n \log n$ running times (shell sort, mergesort, quicksort, Python's sort), and linear running time (radix sort).



(a) All the sorts (b) Faster sorts

Figure 5.5. Sort timings

The calls to the two quadratic sorts were commented out before generating Fig. 5.5b so that the slight differences between the others are easier to see. We also increased the random list sizes to 110000 elements.

Quicksort lives up to its name, beating even radix sort. Also, the timing shown in Fig. 5.5b is based on calling quicksortL() (Listing 5.5), which uses list comprehensions to partition the lists.

It may seem surprising that an $n \log n$ algorithm can 'beat' a linear running time, but this illustrates a problem with big-Oh calculations: constants are elided which may in practice be a significant component of the run time. Recall that big-Oh is only an upper bound, with no guarantee that it's particularly close to the actual running time. For a tighter bound we need $\Theta()$ which usually requires a much deeper analysis.

The erratic behavior of shell sort is a consequence of its susceptibility to h-interval size. Our code simply halves the h-interval on each iteration, and (as mentioned before) it's likely that better speeds could be gained by using Knuth's $3h + 1$ equation.

The final graph (Fig. 5.6) only measures Python's sort, with lists sizes between 1 and 11 million.

The documentation for powersort (https://github.com/python/cpython/blob/main/Objects/listsort.txt) refers to its "supernatural performance" on many kinds of partially ordered arrays (less than $\log(n!)$ comparisons needed, and as few as $n-1$), and this is confirmed by our plots.

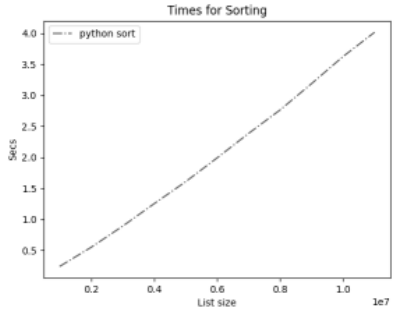


Figure 5.6. Timings for Python's sort()

The C source code for powersort is available at https://github.com/python/cpython/blob/main/Objects/listobject.c, starting with function islt() and proceeding for *many* lines. Sebastian Wild's blog post https://www.wild-inter.net/publications/munro-wild-2018 includes a link to an academic paper by Munro and Wild on the algorithm.

5.2 The kth Smallest Element of a List

We want to find the k-th smallest element of an n-element list. We could utilize selection sort's approach (Sec. 5.1.1) and find the first smallest, second smallest, ..., until we reach the k-th smallest, which is fine if k is small. We could also sort the entire list and then print elems[k].

Generally, we can reach our goal faster if we use quicksort's partition function (Sec. 5.1.5). It rearranges the list's values so that all of those less (or equal) are placed before the pivot, and those that are greater (or equal) are positioned after it. It finishes by returning the pivot's index, which is its final position when the list is fully sorted.

We can use this returned value to guide a search over the list. If partition() returns a pivot index greater than k then the element we are looking for must be in the left half of the list. Conversely, if the pivot index is less than k then we should examine the right half. We stop when the pivot index equals k. This is implemented in Listing 5.7 (kselect.py).

```
def kselectP(elems, k):
  left = 0
  right = len(elems)-1
  while left <= right:
    pivot = partition(elems, left, right)
    if pivot == k:
      return elems[pivot]
    elif pivot < k:
      left = pivot+1   # focus on right half
    else:  # pivot > k
```

```
      right = pivot-1  # focus on left half
  return -1
```

Listing 5.7. k-th selection using partition()

In the worst case, each call to partition() will half the sequence range being searched, so the running time $\approx n + n/2 + n/4 + ...1 \approx 2n$, or $O(n)$.

Currently, partition() simply chooses the left-most element in the subsequence. Listing 5.7 can be made faster if we can make the choice of pivot more relevant to the selection problem. Listing 5.8 (kselect.py) implements this idea by replacing the call to partition() in Listing 5.7 by partition()'s code. The pivot choice (`pivot = elems[left]`) can now be modified.

```
def kselect(elems, k):
  left = 0; right = len(elems)-1
  while left < right:
    pivot = elems[k] # use kth element
    i = left; j = right
    while (i <= k) and (k <= j):
      while elems[i] < pivot:
        i += 1   # move right
      while pivot < elems[j]:
        j -= 1   # move left
      elems[i], elems[j] = elems[j], elems[i]
      i += 1; j -= 1
    if k < i:
      right = j  # focus on left half
    if j < k:
      left = i   # focus on right half
  return elems[k]
```

Listing 5.8. k-th selection with kth element as pivot

The key line change (on line 4) is:

```
pivot = elems[k]
```

If the data is partially sorted then this pivot will probably be closer to the actual k-th smallest element, and so the algorithm will find it sooner, although in the worst case it will still take $O(n)$ time.

Chapter 15 of Bentley [**Ben88**] considers the k-th smallest problem in detail, including some revealing visualizations of how various different algorithms perform.

5.3 Binary Search

Listing 5.9 (binSearch2.py) is a recursive function that employs binary search on a sorted list to find the index position of a specified v data item. If the data cannot be found then the function returns -1.

```
def binSearch(elems, v):
  return binSrch(elems, v, 0, len(
      elems)-1)

def binSrch(elems,v,left,right):
  if left > right:
    return -1
  mid = (left + right)//2
  if v < elems[mid]:
    return binSrch(elems, v, left,
        mid-1)
  elif v == elems[mid]:
    return mid
  else:
    return binSrch(elems, v, mid
        +1, right)
```

Listing 5.9. Recursive binary search

Two examples:

```
>>> from binSearch2 import *
>>> elems = [2, 4, 5, 7, 8, 9, 12,
14, 17, 19, 22, 25, 27, 28, 33, 37]

>>> binSearch(elems, 22)
10
>>> binSearch(elems, 44)
-1
```

Fig. 5.7 shows the four stages in the search for 22 in the list.

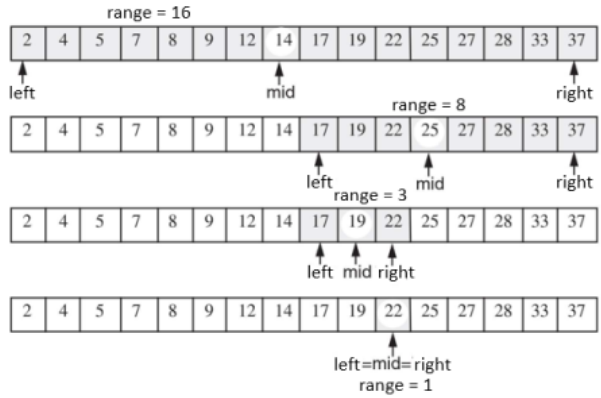


Figure 5.7. Binary search for 22

Binary search is sometimes known as 'binary chop' since it finds an element by *roughly* halving the search interval at each step. It compares the target value to the middle element of the list. If they are not equal, the half in which the target cannot lie is ignored and the search continues with the other half, comparing its middle element to the target, and repeating this until the value is found. If the search ends with the searchable half empty, then the target is not in the list.

Binary search runs in logarithmic time in the worst and average cases, using $O(\log_2 n)$ comparisons, where n is the number of elements. This means that it's faster than linear search (except for small lists), but the list must be sorted.

The worst case running time is fairly easy to determine. If T(n) represent the running time, then we can characterize it as:

$$T(n) \leq \begin{cases} a & \text{if } n < 2 \\ T(n/2) + b & \text{otherwise} \end{cases}$$

where a and b are constants (usually written as $O(1)$).

If we assume that n starts as a power of 2, 2^k for example, then we can expand $T(n)$ to look for a pattern:

$$\begin{aligned} T(2^k) &= b + T(2^{k-1}) \\ &= b + b + T(2^{k-2}) \\ &= b + b + b + T(2^{k-3}) \\ &= \cdots \\ &= kb + T(2^{k-k}) \\ &= bk + a \end{aligned}$$

We assumed that $2^k = n$, so $k = \log_2 n$. This means that:

$$T(n) = b \times \log_2 n + a$$

If we replace constants by Big-Oh notation:

$$\begin{aligned} T(n) &= O(\log_2 n) + O(1) \\ &= O(\log_2 n) \end{aligned}$$

5.3.1 Average-case Analysis for Binary Search. It helps if we adjust our view of the list, thinking of it as a *binary tree*, as in Fig. 5.8. We've also deleted the last node (37) to make the list 15 nodes long (i.e. $2^4 - 1$), so the tree is balanced. In general we'll assume that $n = 2^k - 1$.

The shaded nodes in Fig. 5.8 show the sequence of comparisons that were made by the binary search, matching those in Fig. 5.7.

One comparison was done to find the element that is at the root of the tree. Two comparisons were done to get to level 2, three comparisons for level 3, and four comparisons got us to level 4 and the answer. In general, we will need i comparisons to get to level i.

Another point to note is the number of nodes at each level: 1 node at level 1, 2 nodes at level 2, 4 nodes at level 3, and 8 nodes at level 4. In general there will be 2^{i-1} nodes at level i.

We can express the total number of comparisons made in every possible case by summing the product of the number of nodes on each level and the number

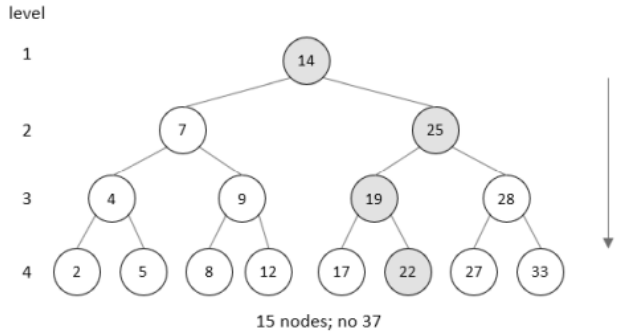


Figure 5.8. Binary search over a binary tree

of comparisons for that level. This gives an average case analysis of

$$\begin{aligned} T(n) &= \frac{1}{n}\sum_{i=1}^{k} i2^{i-1} \quad \text{for } n = 2^k - 1 \\ &= \frac{1}{n}\cdot\frac{1}{2}\sum_{i=1}^{k} i2^{i} \end{aligned}$$

The $1/n$ assumes that every possible case is equally likely, and introduces a probabilistic component into the calculation.

For the next step, we'll use a handy summation (which can be proved by induction, or by differentiation):

$$\sum_{i=1}^{n} i2^i = (n-1)2^{n+1} + 2$$

Continuing:

$$\begin{aligned} T(n) &= \frac{1}{n}\cdot\frac{1}{2}[(k-1)2^{k+1} + 2] \\ &= \frac{1}{n}[(k-1)2^k + 1] \\ &= \frac{1}{n}[k2^k - 2^k + 1] = \frac{[k2^k - (2^k - 1])}{n} \\ &= \frac{[k2^k - n]}{n} = \frac{k2^k}{n} - 1 \end{aligned}$$

Since $n = 2^k - 1$, $2^k = n + 1$, so:

$$T(n) = \frac{k(n+1)}{n} - 1 = \frac{kn + k}{n} - 1$$

As n gets larger, k/n approaches 0, giving

$$\begin{aligned} T(n) &\approx k-1 \\ &= \log_2(n+1)-1 \\ &= O(\log_2 n) \end{aligned}$$

This is unfortunately not quite enough since we've only consider the n cases where a value is found; we should also include the $n+1$ cases when a value is *not* found. The number is $n+1$ since the target can be smaller than the first element, larger than the first element but smaller than the second, larger than the second but smaller than the third, and so on, through to the possibility that the target is larger than the last n-th element. In each of these cases, it takes i comparisons to decide that the target is not in the list.

We must redo our calculation with $2n+1$ possibilities. This gives

$$T(n) = \frac{1}{2n+1}\left[\left(\sum_{i=1}^{k} i2^{i-1}\right) + (n+1)k\right] \quad \text{for } n = 2^k - 1$$

A similar series of substitutions eventually leads to:

$$\begin{aligned} T(n) &\approx k-\tfrac{1}{2} \\ &= \log_2(n+1)-\tfrac{1}{2} \\ &= O(\log_2 n) \end{aligned}$$

This is a little larger than the average case when the target is in the list, but still logarithmic.

5.3.2 An Iterative Binary Search. The recursion in Listing 5.9 occurs at the end of the algorithm. This makes it fairly easy to translate the function into the iterative version in Listing 5.10 (binSearch2.py).

```
def binSearchIter(elems, v):
  left = 0
  right = len(elems) - 1
  while left <= right:
    mid = (left + right)//2
    if v < elems[mid]:
      right = mid-1
    elif v == elems[mid]:
      return mid
    else:
      left = mid+1
  return -1
```

Listing 5.10. Iterative binary search

Two examples:

```
>>> from binSearch2 import *
>>> elems = [2, 4, 5, 7, 8, 9, 12,
14, 17, 19, 22, 25, 27, 28, 33, 37]
>>> binSearchIter(elems, 22)
10
>>> binSearchIter(elems, 44)
-1
```

Exercises

(1) Modify binSearch() so that it also finds the longest:
a) strictly increasing interval; b) interval of constancy.

If there is more than one such interval it should give the leftmost such interval (and the number of such intervals).

(2) Modify binSearch() so that it starts searching for a given v at index position i = int(t*i), where $t = (\sqrt{5}+1)/2$.

(3) A list is *bitonic* if it consists of an increasing sequence of values followed immediately by a decreasing sequence of values. Given a bitonic list, design a logarithmic algorithm to find the index of a maximum value. An efficient solution uses a modified binary search.

5.4 Binary Questions

Carl thinks of a number x ∈ [1..MAX] which Python will guess with a minimum of questions (see binguess.py) to which Carl need only answer 'yes' or 'no'.

Since MAX = 100, it is hardly surprising that Python will always get to the answer after six questions (MAX $< 2^7$).

Exercises

(1) If MAX $= 2^{10}$ and Carl thinks of the number $x = 777$ then the sequence of questions posed by binguess.py will be answered as 0011110111 in terms of 'no' = 0 and 'yes' = 1. How is this sequence related to the binary expansion of 777? Generally, what is the relationship between a number x that Carl thinks up and the (0, 1)-sequence of answers which leads to Python finding x?

(2) *Random Guessing*. Carl thinks of a number G ∈ 1..n. Abby tries to guess G as follows: she chooses a random subset R ⊂ 1..n and Carl replies 'yes' or 'no' if G ∈ R or G ∉ R, respectively.

Let E(n) be the expected number of guesses to find G. Estimate E(n) by simulation for different values of n. You can check your implementation against the following: $E(2) = 2$, $E(3) = 8/3$, $E(4) = 22/7$, $E(5) = 368/105$, $E(16) = 5.287$. The most surprising thing about this approach is how surprisingly rarely you lose compared to deterministic guessing.

The coding is easier if you play the game with a coin several times first, to get a 'feel' for how it works.

How is this problem related to Ex. 7 of Sec. 4.13?

5.5 Recreational Search

Many recreational math puzzles can be treated as search problems: we start from an initial state, use the puzzle conditions to guide a traversal of the problem space towards a suitable goal, with the resulting path becoming the solution. The search should produce an answer within a reasonable amount of time, and the ever-increasing speed of hardware, combined with the small size of most puzzles, means that the relatively simple breadth-first search (BFS) and depth-first search (DFS) algorithms are viable choices.

BFS and DFS are 'uninformed' algorithms in that they don't utilize any problem-specific information to guide their search. The early years of AI research [**RND10**] saw these techniques augmented with 'heuristics' to improve their results, and we'll briefly look at one of the most popular: the A* (A-Star) algorithm.

One direction to take with these problems is towards graph theory, converting the search spaces into graphs in order to access a considerable collection of algorithms for their manipulation. We've decided *not* to do that, but any good algorithms textbook spends a considerably amount of time on that topic (e.g. Cormen *et al.* [**CLRS22**], Sedgewick and Wayne [**SW11**]).

The puzzles considered here are classics: the water jugs problem, the missionaries and cannibals, maze search, the 8-puzzle (a more tractable version of the 15-puzzle), and eight queens. A good source for details on these, and many others, is the magnificent text by Rouse Ball and Coxeter [**BC87**].

5.5.1 The Water Jugs Problem. Two unmarked jugs can hold m and n liters of water respectively ($0 < m < n$). The permitted operations are: 1) empty a jug, 2) fill it to the top, and 3) pour water from one jug into the other until the jug is empty or the other is full. The goal is to measure out d liters of water ($d < n$) in a minimum number of operations.

The problem can be modeled as a Diophantine equation $mx + ny = d$ which is solvable if and only if $gcd(m, n)$ divides d, which means that the solution x, y can be obtained using the extended GCD algorithm. However, we'll utilize BFS without relying on these attributes. We'll also set $m = 3$, $n = 5$, and $d = 4$ in honor of this problem's appearance in *Die Hard with a Vengeance* (1995). Of course, the puzzle is a bit older, occurring in Bachet's 17th-century textbook, *Problèmes plaisans et délectables qui se font par les nombres*. It's also closely related to the *three* jugs problem devised by the sixteenth-century Italian mathematician, Niccolo Fontana, better known by his nickname, Tartaglia, "the Stutterer".

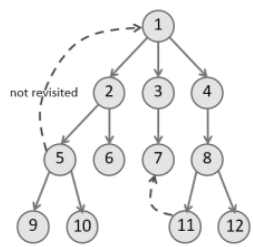


Figure 5.9. Nodes examined in BFS order

BFS treats the search space as a tree (see Fig. 5.9) with the initial state as the root (the node at the *top*). The algorithm explores all the nodes at the current depth before moving onto those at the next level. This means that when BFS finds the goal state that the path from the root will be the shortest.

Most search spaces aren't trees but graphs with cycles (e.g. it's perfectly acceptable to repeatedly fill and empty a jug forever). This is avoided by ignoring nodes that have already been visited, which is why the edges from nodes 5 to 1 and 11 to 7 aren't part of the search.

A simple BFS implementation stores each level in a queue whose size is proportional to b^d, where b is a node's branching factor (its number of 'children'), and d is the tree depth.

The bfs() function in Listing 5.11 is located in searchUtils.py.

```
def bfs(startState):
  queue = [ node(startState, None)]   # no parent
  visited = []
  while queue != []:
    currNode = queue.pop(0)   # remove from start
    visited.append(currNode.state)
    if isGoal(currNode.state):
      return currNode.getPath(), len(visited)
    for s in nextStates(currNode.state):
      if not (s in queue) and not (s in visited):
        queue.append( node(s, currNode))
  return None, len(visited)
```

Listing 5.11. Breadth-first search

bfs() manipulates a `state` whose details will vary from problem to problem. The state is managed by a `node` class (searchUtils.py). Aside from the state, node also maintains a link to its parent, which is utilized to generate the path of states when the search has finished. There's also a cost variable, but that's only used by the A* algorithm described later.

bfs() utilizes two state functions, **isGoal()** and **nextStates()**, which will be redefined for each problem. For this puzzle, they're implemented in Listing 5.12 (jugs.py).

```
def isGoal(state):
  jug1, jug2 = state
  return ((jug1 == GOAL) and (jug2 == 0)) or \
         ((jug1 == 0) and (jug2 == GOAL))

def nextStates(state):
  jug1, jug2 = state
  sts = []
  # fill a jug
  sts.append((MAX_JUG1, jug2)) # Fill jug1
  sts.append((jug1, MAX_JUG2)) # Fill jug2
```

```
  # empty a jug
  sts.append((0, jug2)) # empty jug1
  sts.append((jug1, 0)) # empty jug2
  # pour from jug1 to jug2
  filljug2 = MAX_JUG2 - jug2
  if filljug2 > jug1:
    sts.append((0, jug2+jug1)) # Fill jug2 until jug1 empty
  else:
    sts.append((jug1-filljug2, MAX_JUG2)) # Fill jug2
  # pour from jug2 to jug1
  filljug1 = MAX_JUG1 - jug1
  if filljug1 > jug2:
    sts.append((jug1+jug2, 0)) # Fill jug1 until jug2 empty
  else:
    sts.append((MAX_JUG1, jug2-filljug1)) # Fill jug1
  return sts
```

Listing 5.12. jugs: isGoal() and nextStates()

A look inside isGoal() or nextStates() in Listing 5.12 reveals that the 'jugs' state is a tuple containing the amounts of water currently in the two jugs.

isGoal() returns true if either of the jugs holds the required amount (in this case the GOAL amount = 4) and the other is empty. nextStates() generates all the possible moves from the current jugs state – all combinations of filling and emptying the jugs, and pouring water from one to the other.

The search is initiated in Listing 5.13 (jugs.py).

```
path, numVisited = searchUtils.bfs((0,0))
print("No. of states visited:", numVisited)
if path == None:
  print("No path found");
  sys.exit()
```

Listing 5.13. Die Hard with Jugs

The result:

```
Max of jug1: 5
Max of jug2: 3
Goal: 4
No. of states visited: 16
Path:
(0, 0)
Increase jug1 by 5
(5, 0)
Decrease jug1 by 3
Increase jug2 by 3
(2, 3)
Decrease jug2 by 3
(2, 0)
Decrease jug1 by 2
Increase jug2 by 2
(0, 2)
Increase jug1 by 5
(5, 2)
Decrease jug1 by 1
Increase jug2 by 1
(4, 3)
Decrease jug2 by 3
(4, 0)
```

The number of visited states is printed as a measure of the search's memory usage. Also, a Cartesian grid is plotted (Fig. 5.10) showing the transitions between states, starting in (0,0) and finishing at (4,0).

Grids of this type are sometimes termed "mathematical billiards" due to the way that the trajectories between the states "bounce" off the edges of the space. Incidentally, Tartaglia's three jugs antecedent of this puzzle can be drawn as lines bouncing around a *triangular* pool table [**Oli93**].

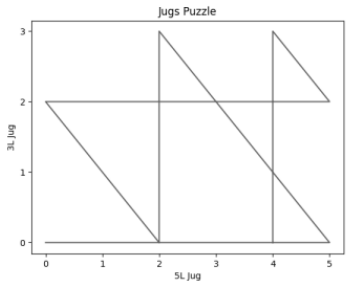


Figure 5.10. Cartesian grid for the jugs problem

bfs() terminates upon finding a solution even though alternative answers are possible. For example: (0,0) - (0,3) - (3,0) - (3,3) - (5,1) - (0,1) - (1,0) - (1,3) - (4,0), which is a longer path, so not reached by bfs().

5.5.2 The Missionaries and Cannibals Problem. Three missionaries and three cannibals are standing on the west bank of a river, along with a boat which can hold up to two people. The group want to reach the east bank but with the proviso that the cannibals should never outnumber the missionaries on either side or the missionaries will be eaten.

This puzzle can be traced back to the medieval text *Propositiones ad Acuendos Juvenes* by Alcuin (as Problem 17). However, his participants were brothers and sisters, and no woman could be in the company of another man unless her brother was present. There are many other variants involving the number of people in the two groups, the size of the boat, and sometimes even an island to act a staging post [**PS89**].

We'll use bfs() again, but with a more complex state tuple: (mWest, cWest, mEast, cEast, boatPos), where mWest = the number of missionaries on the west bank, cWest = the number of cannibals on the west bank, mEast = the number of east bank missionaries, cEast = the number of east bank cannibals, and boatPos = "west" or "east".

The isGoal() and nextStates() are implemented as in Listing 5.14 (missionaries.py).

```
def isGoal(state):
  # are all the missionaries and cannibals on the east bank?
  mWest, cWest, mEast, cEast, boatPos = state
  return (mEast == MAX_NUM) and (cEast == MAX_NUM)

def nextStates(state):
  mWest, cWest, mEast, cEast, boatPos = state
  sts = []
```

```
  if boatPos == "west":  # boat on west bank
    nboat = "east"
    if mWest > 1:
      sts.append((mWest-2, cWest, mEast+2, cEast, nboat))
    if mWest > 0:
      sts.append((mWest-1, cWest, mEast+1, cEast, nboat))
    if cWest > 1:
      sts.append((mWest, cWest-2, mEast, cEast+2, nboat))
    if cWest > 0:
      sts.append((mWest, cWest-1, mEast, cEast+1, nboat))
    if (cWest > 0) and (mWest > 0):
      sts.append((mWest-1, cWest-1, mEast+1, cEast+1, nboat))
  else: # boat on east bank
    nboat = "west"
    if mEast > 1:
      sts.append((mWest+2, cWest, mEast-2, cEast, nboat))
    if mEast > 0:
      sts.append((mWest+1, cWest, mEast-1, cEast, nboat))
    if cEast > 1:
      sts.append((mWest, cWest+2, mEast, cEast-2, nboat))
    if cEast > 0:
      sts.append((mWest, cWest+1, mEast, cEast-1, nboat))
    if (cEast > 0) and (mEast > 0):
      sts.append((mWest+1, cWest+1, mEast-1, cEast-1, nboat))
  return [s for s in sts if isValid(s)]

def isValid(state):
  mWest, cWest, mEast, cEast, boatPos = state
  if (mWest < cWest) and (mWest > 0):
    return False
  if (mEast < cEast) and (mEast > 0):
    return False
  return True
```

Listing 5.14. missionaries: isGoal() and nextStates()

The search is initiated by Listing 5.15.

```
path, numVisited = searchUtils.bfs((MAX_NUM, MAX_NUM, 0, 0, "west"))
print("No. of states visited:", numVisited)
if path == None:
  print("No path found");
  sys.exit()
```

Listing 5.15. Crossing the river safely

The result:

```
No. of states visited: 25
Path length: 12
West: 3m 3c; East: 0m 0c | Boat: west
  West -> East: 0m 2c
West: 3m 1c; East: 0m 2c | Boat: east
  East -> West: 0m 1c
West: 3m 2c; East: 0m 1c | Boat: west
  West -> East: 0m 2c
```

```
West: 3m 0c; East: 0m 3c | Boat: east
  East -> West: 0m 1c
West: 3m 1c; East: 0m 2c | Boat: west
  West -> East: 2m 0c
West: 1m 1c; East: 2m 2c | Boat: east
  East -> West: 1m 1c
West: 2m 2c; East: 1m 1c | Boat: west
  West -> East: 2m 0c
West: 0m 2c; East: 3m 1c | Boat: east
  East -> West: 0m 1c
West: 0m 3c; East: 3m 0c | Boat: west
  West -> East: 0m 2c
West: 0m 1c; East: 3m 2c | Boat: east
  East -> West: 1m 0c
West: 1m 1c; East: 2m 2c | Boat: west
  West -> East: 1m 1c
West: 0m 0c; East: 3m 3c | Boat: east
```

The output is also plotted as a 3D grid shown in Fig. 5.11.

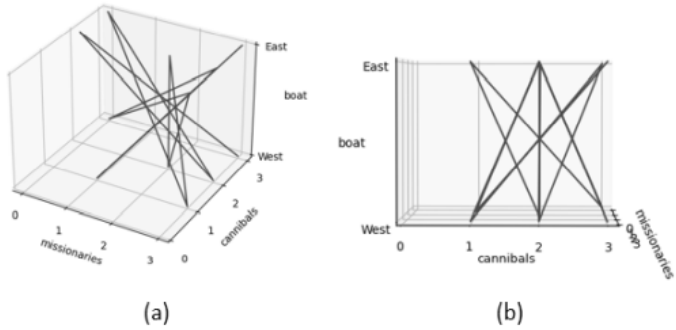


Figure 5.11. 3D plot for the missionaries problem

Matplotlib 3D plots can be rotated, which reveals an interesting symmetry for the movements of the cannibals (Fig. 5.11b) which is absent for the missionaries.

A more graphical representation of the movements of the groups is shown in Fig. 5.12, which also shows other solutions to the problem.

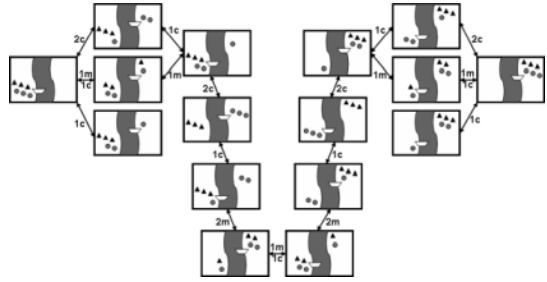


Figure 5.12. Search space for the missionaries (m) and cannibals (c)

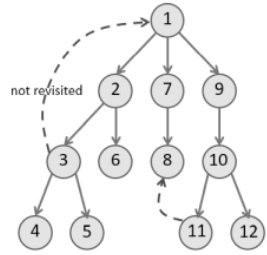


Figure 5.13. Nodes examined in DFS order

5.5.3 Searching a Maze. We'll use maze searching to discuss depth-first search (DFS) which, given the same state space as BFS, focuses on investigating as far along a branch as possible before backtracking to an earlier choice point and trying another branch. It's worth comparing the node numbering in Fig. 5.13 with Fig. 5.9 to see how the two algorithms differ. As with BFS, DFS must maintain a list of visited nodes to prevent it falling into cycles.

DFS generally produces a longer path than BFS, but one of its advantages is obscured by Fig. 5.13. A recursive implementation of DFS only investigates one branch at a time, and so doesn't need to store a large part of the tree in memory. Whereas BFS uses space proportional to $O(b^d)$, DFS is closer to $O(bm)$ where m is the maximum depth of the tree. However, this generally means that Python's limit for recursion depth needs to be increased.

Another reason for using a recursive DFS is that its code can be easily modified to improve its speed. This is illustrated in the dfs() function in Listing 5.16 (searchUtils.py).

```
def dfs(state, max=MAX):
    path = dfsDepth( node(state, None), 1, max)
    numVisited = len(visited)
    return path, numVisited

def dfsDepth(currNode, depth, max):
  if depth > max:  # constrain search depth
    return None
  visited.append(currNode.state)
  if isGoal(currNode.state):
    return currNode.getPath()
  for s in nextStates(currNode.state):
    if not (s in visited):  # no stack to examine
      path = dfsDepth( node(s, currNode), depth+1, max)
      if path != None:
        return path
  return None
```

Listing 5.16. Depth-first search

dfsDepth() shows that DFS can be coded succinctly as a recursive call inside a loop. The recursion implements the search down the tree and the backtracking, while the loop handles the traversal through the options at a choice point. Also, note that the state-specific functions, isGoal() and nextStates(), are still being used.

dfs() adds a depth limit to the basic algorithm which causes the recursion to backtrack upon reaching a specified call depth. This idea is at the heart of *iterative deepening* which repeatedly tries DFS searches with increasing depth limits until an answer is (hopefully) found.

Another important variant of DFS replaces recursion with a stack (see `searchUtils.py`). The coding is very similar to the BFS implementation except that BFS employs a queue. This approach may be preferable if recursion is limited (as in Python) or unavailable (as was the case in older languages such as Fortran 77).

The maze state is encoded as a 2D list. For example:

```
maze = [ [' ', '*', '*', '*', ' ', ' ', ' '],
         [' ', ' ', ' ', ' ', ' ', '*', ' '],
         ['*', ' ', '*', '*', '*', ' ', ' '],
         ['*', ' ', '*', '*', ' ', ' ', ' '],
         ['*', ' ', '*', '*', ' ', '*', ' '],
         ['*', ' ', '*', '*', ' ', '*', ' '],
         ['*', ' ', '*', '*', ' ', '*', ' '],
         ['*', ' ', '*', '*', ' ', '*', ' '],
         [' ', ' ', ' ', ' ', ' ', '*', ' '] ]
```

The start point is the top left corner, the goal is to reach the bottom right, and the '*'s are walls.

maze.py allows the user to choose between using bfs(), dfs() or dfsIter() to traverse the maze, with the following results:

```
BFS
No. of states visited: 40
Path length: 17
-----------------
| . * * * . . . |
| . . . . . * . |
| *   * * *   . |
| *   * *     . |
| *   * *   * . |
| *   * *   * . |
| *   * *   * . |
| *   * *   * . |
|           * . |
-----------------

DFS Recursive
No. of states visited: 25
Path length: 25
-----------------
| . * * *       |
| . .       *   |
| * . * * *     |
| * . * * . . . |
| * . * * . * . |
| * . * * . * . |
| * . * * . * . |
| * . * * . * . |
|   . . . . * . |
-----------------

DFS Iterative
No. of states visited: 25
Path length: 25
-----------------
| . * * *       |
| . .       *   |
| * . * * *     |
| * . * * . . . |
| * . * * . * . |
| * . * * . * . |
| * . * * . * . |
| * . * * . * . |
|   . . . . * . |
-----------------
```

As expected, the two DFS versions produce the same path, which is longer than the one found by BFS. Note though that the number of states was greater for BFS than DFS (40 vs. 25) which reflects the amount of memory used.

In dfs(), the maximum depth (MAX) was set to 30 since the maze grid is 10-by-10, and so even the shortest path will use 19 states (a player can only move horizontally or vertically).

isGoal() and nextStates() deal with the maze state (Listing 5.17; maze.py).

```
def isGoal(state):
  (r, c) = state    # at bottom right corner?
  return (r == len(maze)-1) and (c == len(maze[0])-1)

def nextStates(state):
  (r, c) = state
  sts = []
  for dir in DIRS: # try all dirs
    r1 = r+dir[0]; c1 = c+dir[1]
    if isValid(r1,c1):
      sts.append((r1,c1))
  return sts

def isValid(r, c):
  return ((r >= 0) and (r < len(maze)) and
       (c >= 0) and (c < len(maze[0])) and
       (maze[r][c] == ' '))
```

Listing 5.17. maze: isGoal() and nextStates()

The available directions are implemented as step offsets in a DIRS list:

```
DIRS = [(1,0), (0,1), (-1,0), (0,-1)]  # down, left, up, right
```

5.5.4 The 8-Puzzle. The 8-puzzle problem is a 3-by-3 board holding 8 slideable tiles (numbered 1 to 8) and an empty space (which we'll label as 0). The goal is to rearrange the tiles into the numbered ordered:

```
0 1 2
3 4 5
6 7 8
```

It's possible to move a horizontal or vertical neighbor of the 0 tile into its space, which causes the tile's number and 0 to be swapped.

This is a version of the famous 15-puzzle, which for many years was believed to have been devised by the great American game designer Sam Loyd. Martin Gardner edited two collections of Loyd's puzzles which are well worth a study [**LG59**, **LG60**]; the 15-puzzle appears in Vol. 1.

Extensive game play with the 8-puzzle has established that reaching the solution from some arbitrary starting state takes an average of 22 moves. Also, the number of choices (or branching factor) at each move is about 3. (When the

empty tile is in the middle, four moves are possible; when it's in a corner, two; and when it's at an edge, three.) This implies that an exhaustive tree search to depth 22 will look at about $3^{22} \approx 3.1 \times 10^{10}$ states. Things are nowhere near that bad in practice since only about $9!/2 = 181,440$ distinct states are reachable based on permissible tile operations.

Nevertheless, it's necessary to add heuristics to guide the search to the goal state within a reasonable amount of time. We'll utilize A* (AStar) which applies a cost function $f()$ to each node in the search tree, and then considers the nodes in increasing cost order. The function has two parts: $g()$, the cost of getting from the start state to this node (usually represented by the length of the path), and $h()$ an estimate of the cost of getting from this node to the goal state. This is expressed as:

$$f(n) = g(n) + h(n)$$

where n is the current node. $h(n)$ is a heuristic since we don't have exact knowledge about the cost of reaching the goal.

Probably the most natural $h()$ estimate for sliding puzzles is to sum the 'taxicab' distances between the tiles and their final positions in the goal. This is the smallest number of horizontal and vertical moves between the two positions, and is sometimes called the Manhattan distance since it's similar to the way that we count the number of city blocks between two locations.

For example the Manhattan distance between the start state and goal in Fig. 5.14 is $2+3+1+2+2+2+3+3+2 = 20$ where the tiles are considered in the order 0 to 8.

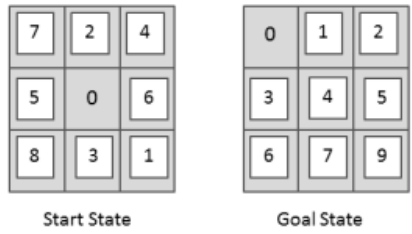


Figure 5.14. 8-puzzle start and goal states

A* is much faster than BFS or DFS but may not return the shortest path to the goal because of the way that the cost function rearranges the nodes.

The implementation shown below (Listing 5.18; searchUtils.py) looks similar to bfs() but the queue data structure is replaced by a priority queue (Python's heapq class). When a node is added to a priority queue it's stored in increasing order based on its cost field. When a node is removed, the one with the smallest cost is selected.

```
def aStar(startState):
  priQueue = []
  startNode = node(startState, None)
  startNode.setCost()
  heapq.heappush(priQueue, startNode)
  visited = []
  while priQueue != []:
    minNode = heapq.heappop(priQueue)
    visited.append(minNode.state)
    if isGoal(minNode.state):
      return minNode.getPath(), len(visited)
    for s in nextStates(minNode.state):
      if not (s in visited):
        # allow duplicate states to be inserted
        # since they are sorted by cost
        nd = node(s, minNode)
        nd.setCost()
        heapq.heappush(priQueue, nd)
  return None, len(visited)
```

Listing 5.18. A* search

The heap combines the access speeds of an array with a tree ordering of its elements, which makes it faster that using a sorted queue. Both the pushing and popping operations have $O(\log n)$ running time, where n is the size of the heap.

Another change is that aStar() uses the setCost() method in the node class:

```
  def setCost(self):
    self.cost = len(self.getPath())-1 + goalDist(self.state)
```

goalDist() is the heuristic function $h()$, and is defined along with isGoal() and nextStates() in puzzle8.py (see Listing 5.19).

```
def goalDist(loc):
  ''' The sum of the Manhattan distances
      (sum of the vertical and horizontal dists)
      from the tiles to their goal positions
  '''
  tot = 0
  for i in range(len(loc)):
    idx =  loc.index(i)   # loc of tile i
    row, col = idx//SIZE, idx%SIZE
    idxG =  GOAL.index(i) # goal loc of tile i
    rowG, colG = idxG//SIZE, idxG%SIZE
    tot += abs(row-rowG) + abs(col-colG)
  return tot
```

Listing 5.19. 8-puzzle: goalDist()

puzzle8.py gives the user the option of employing BFS, A*, or DFS to search for an answer. All of these utilize the same isGoal() and nextStates() functions.

In addition to reporting the path (if there is one) and the number of visited states, the time and tracemalloc Python modules are employed to measure the elapsed time and the peak amount of memory used. The results for four different starting states are shown in Table 5.1.

Start Tile	Algorithm	Path length	No. of states	Duration (secs)	Memory (KB)
1 2 5	BFS	4	11	0.000	9
3 4 0	A*	4	4	0.000	5
6 7 8	DFS	4	3222	0.440	479
1 4 2	BFS	9	194	0.020	102
6 5 8	A*	9	12	0.000	10
7 3 0	DFS	19	1776	0.170	268
1 0 2	BFS	24	1906531	11 mins	451473
7 5 4	A*	24	2420	0.340	1128
8 6 3	DFS	no path	7165	1.810	1006
0 8 7	BFS	aborted			
6 5 4	A*	31	16871	12.152	7402
3 2 1	DFS	no path	7500	2.051	1047

Table 5.1. 8-Puzzle algorithm statistics

A graphical representation of BFS's search with the tile setup in row 1 of Table 5.1 is shown in Fig. 5.15. Fig. 5.15 illustrates why the path length is 4 and the search visits 11 states. Recall that BFS traverses the levels left-to-right in a top-down fashion.

Table 5.1 shows that DFS is the obvious loser, partly due to it using a maximum depth setting of only 20 which means that it couldn't investigate a large enough subtree to find answers for the last two tile configurations.

The last two tests also show the drawbacks of basic BFS – the waiting time grew so long on the last test that we aborted the execution, and the time and memory usage for the previous tile settings in rows 2 and 3 were very poor. A* was the easy winner, and only required a 10-line goalDist() function to achieve it.

5.5.5 The n-Queens Problem. We wish to *peacefully* place 8 queens on an 8x8 chessboard, meaning that there are no queens in the same row, column, or diagonal (otherwise they'd *attack* each other). Our solution will generalize the problem to *n*-queens on a *nxn* chessboard.

A particularly poor search approach for 8-queens would be to consider all $\binom{64}{8}$ placements, which results in around 4.42×10^9 tests. Fortunately, position

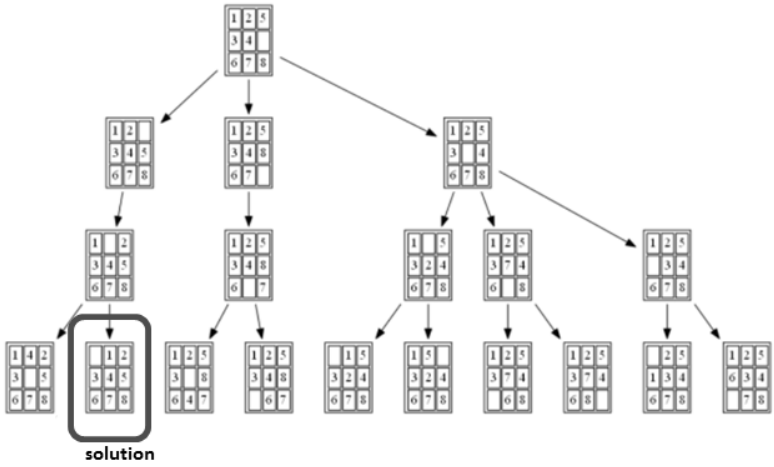


Figure 5.15. BFS search for tile 1

constraints can reduce the search space considerably: the queen in the first column has 8 position choices, the queen in the second column has only 7 choices, the queen in the third column has 6 choices, and so on, resulting in a search space of just $8! = 40,320$. We can also add diagonal constraints to further reduce the size to 2,057.

This approach suggests that we don't need to employ a 2D list for board placements, since there's only one queen per column. A single list will suffice, with an element's index acting as the column number, and the row position stored as the element's value. For instance, the solution in Fig. 5.16 can be written as [3, 5, 7, 1, 6, 0, 2, 4]. Note: we start counting rows and columns from 0, starting from the top-left corner of the board.

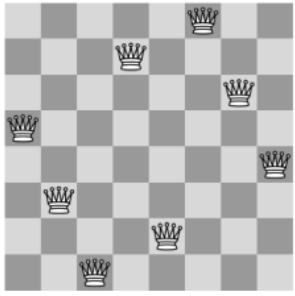


Figure 5.16. One solution for the 8-queens problem

The 8-queens version of the puzzle has 92 distinct solutions, and if symmetric placements based on rotation and reflection aren't counted separately, then there are just 12 unique configurations.

We could again use A* or BFS, but the solution steps described above suggests that we could use recursive DFS based around an examineColumn() function. It places queens column-wise, beginning with the

first column, by setting queens[0] = 0 (we're counting from 0). Then we tentatively move the second queen down the second column until we find a square that isn't attacked (e.g. queens[1] = 2). Then we move the third queen in the third column down until we find a square not attacked by the first two queens, and so on.

If we can't place a queen because each square of its column is attacked, then examineColumn() must backtrack and try to move the preceding queen further down, otherwise the function will have to backtrack further, to the previous column.

row[i] = True	Row i is not attacked
upl[k] = True	Diagonal with $i + j = k$ is not attacked; k takes values from 2 to $2n$.
upr[k] = True	Diagonal with $i - j = k$ is not attacked; k takes values from $1 - n$ to $n - 1$.

Table 5.2. Attack tests for a cell

Unlike the previous examples, we want the search to continue after the first solution. An answer will be printed, a global counter incremented, and then the search will continue as if the solution had never been encountered. In this way, the code will report all solutions in increasing row order, meaning that even the first queen will tried out in the n-th row eventually.

examineColumn() in Listing 5.20 (nqueens.py) utilizes the three boolean lists in Table 5.2, where upl (short for "up-and-left") and upr ("up-and-right") store the results of diagonal tests.

```
def examineColumn(i):
  global count
  for j in range(n):
    if isValid(i,j):
      queens[i] = j
      # col i queen is placed in row j
      # this queen attacks these squares
      uprIdx = (i-j) + (n-1)
      row[j] = False
      upl[i+j] = False; upr[uprIdx] = False
      if i < n-1:
        examineColumn(i+1)
      else:  # found a new solution
        count += 1
      # undo changes for backtracking
      row[j] = True
      upl[i+j] = True; upr[uprIdx] = True
```

Listing 5.20. Solving n-queens

The results when $n = 8$ are:

```
Board 1
Q . . . . . . .
. . . . . . Q .
. . . . Q . . .
. . . . . . . Q
. Q . . . . . .
. . . Q . . . .
. . . . . Q . .
. . Q . . . . .

Board 2
Q . . . . . . .
. . . . . . Q .
. . . Q . . . .
. . . . . Q . .
. . . . . . . Q
. Q . . . . . .
. . . . Q . . .
. . Q . . . . .

  : many results not shown

Board 92
. . Q . . . . .
. . . . . Q . .
. . . Q . . . .
. Q . . . . . .
. . . . . . . Q
. . . . Q . . .
. . . . . . Q .
Q . . . . . .

Found 92 solutions
```

The initial call to the examineColumn(0) is surrounded by code for measuring the elapsed time and memory usage (which we've not listed here). The results are reported in Table 5.3.

n	no. of places	time (secs)	memory (bytes)
8	92	0.015	1152
9	352	0.015	1288
10	724	0.078	1328
11	2680	0.356	1368
12	14280	1.807	1408
13	73712	10.406	1448
14	365596	62.492	1488
15	2279184	404.118	1528
16	14772512	2757.066	1568
17	95815104	19566.369	1672

Table 5.3. n-queens statistics

The time column suggests that we are dealing with exponential running time. To be a little more precise, an upper bound for the running time for examineColumn() on an n-by-n board considers n values for column 0, $n-1$ values for column 1, and so on, resulting in $O(n!)$. Factorial running time is actually worse than exponential.

The memory column is pleasingly small since the only significant data structures are the positions list and the stack of recursive calls, which both increase linearly with the size of n.

History of the problem: 8-queens was first posed in 1848 by Max Bezzel in a chess magazine, but went unnoticed at the time. It was suggested again by Franz Nauck in the popular *Illustrierte Zeitung* (June 1, 1850) and this time caught people's attention, including that of Carl Friedrich Gauss. On 21 September 1850, the blind Dr. Nauck published all 92 solutions in the same journal, beating Gauss who had only discovered 72 answers up to that time. For a much fuller survey of the problem, consult Bell [**BS09**].

Exercises

(1) A knight is placed on any square of an *nxn* chessboard with the task of visiting every square exactly once. Write a backtracking algorithm for this "knight's tour" when $n = 8$. Use a 2D board to store the current move number. Don't forget to set board[x][y] = 0 when you have to backtrack.

(2) Solve the knight's tour problem for the 8 x 8 board using divide-and-conquer. First, tackle the problem for the lower left 4x5 board in Fig. 5.17, extend it to the lower right 4 x 5 board, and finally to the upper 8 x 3 board. One possible solution is shown in Fig. 5.17 where the numbers indicate the sequence of visits.

60	63	50	53	56	43	48	45
51	54	59	62	49	46	57	42
64	61	52	55	58	41	44	47
3	8	15	20	27	22	35	40
16	11	4	9	34	39	28	23
7	2	19	14	21	26	31	36
12	17	10	5	38	33	24	29
1	6	13	18	25	30	37	32

Figure 5.17. One solution to the knight's tour (Ex. 2)

A. J. Schwenk ('Which rectangular chessboards have a knight's tour?', *Math. Magazine* 64, 1991, 325-332, `https://www.mimuw.edu.pl/~rytter/TEACHING/ALCOMB/schwenk.pdf`) generalized this method to construct closed knight tours for all boards where closed tours exist.

(3) *A Hamiltonian circuit* or *path* of a graph visits every node exactly once. The exercises above asked us to find Hamiltonian paths in the graphs whose nodes are squares of a board, with two nodes connected if they are a knight-jump apart.

Finding a Hamiltonian circuit or path is an NP-complete problem. For these problems no method is known which always works *and* is always faster than

simply trying every possibility. However, there are methods which find a Hamiltonian cycle quite quickly for most graphs that have one.

Write a program to find closed knight's tours using the following method developed by Lajos Pósa (details can be found in Schwenk's paper cited above). Construct a path $H_0, H_1, ..., H_l$ by adding nodes until we can not extend it any further. Then H_l must be connected to some H_r with $r \leq l - 2$. Then $H_0, ..., H_r, H_l, H_{l-1}, ..., H_{r+1}$ is also a path. We call the operation of replacing our previous path by this one a *rotation*. Try to extend the new path, and if it can't be done, then rotate again since there are usually choices when making extensions or rotations. When all the nodes have been added to the path, perform more rotations to generate a Hamiltonian cycle. A good way to avoid falling into a loop is to insert a random element into the choices, but the process may still get stuck.

(4) Implement Warnsdorff's rule (1823): the knight should always jumps to a cell from which it can jump to the *fewest* squares not previously visited. In case of several suitable moves choose any one of them. This rule also seems to work for *mxn* boards which contain Hamiltonian paths, but there's no proof that it always works. For instance, sometimes we reach a dead end as a result of having made an unfortunate choice earlier. However, there's no known example where a tour can not be completed without violating Warnsdorff's rule. The rule generates Hamiltonian paths, not cycles.

(5) Show that the 15-puzzle is not solvable if and only if its start state is an odd number of swaps away from the goal state. This property was used by Loyd to make his swapped 14-15 version of the puzzle impossible to solve. For details, see the paper by Keith Conrad at `https://kconrad.math.uconn.edu/blurbs/grouptheory/15puzzle.pdf`.

(6) Another popular heuristic for the 8-puzzle is the *number of misplaced tiles*. In Fig. 5.14 that value would be 9. Implement this heuristic and compare its performance to the Manhattan distance.

(7) Solve the *n*-queen problem using A*, which will require you to devise an *h*() function for the state. One approach is to calculate the number of queens that attack a square, which can utilize code in examineColumns().

(8) Aside from the brothers and sisters version of the Missionaries and Cannibals problem, Alcuin also debuted the wolf, goat, and cabbage puzzle. A farmer with a wolf, a goat, and a cabbage must cross a river by boat. The boat can carry only the farmer and a single item. If left unattended, the wolf will eat the goat, or the goat will eat the cabbage. How can they all cross the river without anything being eaten?

(9) Four missionaries are on one side of the river and four cannibals on the other side. Each group wishes to get to the opposite bank and the only means of transport is a rowing boat big enough for three people. Only one missionary and one cannibal know how to row. The cannibals will eat the missionaries as soon as they outnumber them, whether in the boat or on either bank. Can the crossing be completed successfully?

What if there are five missionaries and five cannibals and the boat still carries only three people? What if there are six missionaries and six cannibals?

(10) An entire category of search puzzle that we left out are *train shunting* problems, which involve rearranging train carriages on a track based on a limited set of permitted moves.

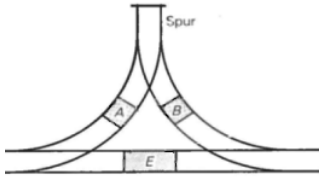


Figure 5.18. Switching trucks A and B

For example, two trucks A and B are in a siding as in Fig. 5.18. Can engine E shunt A into B's position and B into A's position and finish up exactly where it started? Each truck will just fit on the spur (the top part of the two sidings), so it's possible to shunt a truck onto the spur from one siding and pull it off from the other. However, the engine is too big to fit into the spur.

5.6 Permutations

5.6.1 Permutations as Arrangements. An arrangement of three different objects, such as 1, 2, and 3, is called a *permutation*. There are six permutations of 1, 2, and 3: 123, 132, 213, 231, 312, 321. More generally, the product rule tells us that there are $n!$ permutations of n distinct objects.

Listing 5.21 (perms.py) permutes a list by random shuffling its elements.

```
def shuff(elems):
  n = len(elems)
  for i in range(n-1, 1, -1):
    rIdx = random.randint(0, i-1)
    elems[i], elems[rIdx] = elems[rIdx], elems[i]
```

Listing 5.21. Randonly shuffling a list

For example:

```
>>> from perms import *
>>> elems = [1, 2, 3]
>>> shuff(elems); print(elems)
[1, 3, 2]
>>> shuff(elems); print(elems)
[1, 2, 3]
>>> shuff(elems); print(elems)
[3, 2, 1]
```

Python's random module includes a shuffle() function that does much the same thing.

```
>>> import random
>>> random.shuffle(elems); print(elems)
[1, 3, 2]
>>> random.shuffle(elems); print(elems)
[1, 2, 3]
>>> random.shuffle(elems); print(elems)
[3, 1, 2]
```

This approach isn't suitable if you want to iterate through all the possible permutations since each call produces a random result. Listing 5.22 (perms.py) implements a simple backtracking algorithm for printing all the possibilities.

```
def permute1(elems):
  perm(elems, 0, len(elems))

def perm(elems, left, right):
  if left == right:
    print(elems)
  else:
    for i in range(left, right):
      elems[left], elems[i] = elems[i], elems[left]
      perm(elems, left+1, right)
      elems[left], elems[i] = elems[i], elems[left]
```

Listing 5.22. Printing all the permutations

For example:

```
>>> elems = [1, 2, 3]
>>> permute(elems)
[1, 2, 3]      [1, 3, 2]      [2, 1, 3]
[2, 3, 1]      [3, 2, 1]      [3, 1, 2]
```

This function prints all the permutations and then returns, which is not always ideal. Typically, we'd like to obtain each permutation as a separate value or collected together in a list. This requirement is a good reason for using Python's `yield`, as in Listing 5.23 (perms.py).

```
def permuteY(elems):
  if len(elems) <= 1:
    yield elems
    return
  for p in permuteY(elems[1:]):
    for i in range(len(elems)):
      yield p[:i] + elems[0:1] + p[i:]
```

Listing 5.23. Permutations one-by-one

For example:

```
>>> elems = [1, 2, 3]
>>> pg = permuteY(elems)
>>> next(pg)
[1, 2, 3]
>>> next(pg)
[2, 1, 3]
>>> next(pg)
[2, 3, 1]
```

The generator can be induced to store all of the results in a list by surrounding it with a `list()`:

```
>>> elems = [1, 2, 3]
>>> ps = list( permuteY(elems))
>>> print(ps)
[[1, 2, 3], [2, 1, 3], [2, 3, 1], [1, 3, 2], [3, 1, 2], [3, 2, 1]]
```

5.6.2 Permutations as Rearrangements. A permutation can also be regarded as a *rearrangement* function that specifies how one ordering is transformed into another. For instance, the permutation 5 4 6 9 7 8 10 2 3 1 can be thought of as the output of the function f_1 that rearranges {1, 2, ..., 10}:

$$f_1 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 5 & 4 & 6 & 9 & 7 & 8 & 10 & 2 & 3 & 1 \end{pmatrix}.$$

Some other permutations of 1..10:

$$f_2 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 3 & 4 & 5 & 1 & 2 & 9 & 7 & 8 & 10 & 6 \end{pmatrix}$$

$$f_3 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 3 & 9 & 2 & 1 & 6 & 5 & 4 & 10 & 7 & 8 \end{pmatrix}$$

$$f_4 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 2 & 4 & 8 & 3 & 10 & 9 & 1 & 6 & 5 & 7 \end{pmatrix}$$

A permutation function f can be represented by a graph. Choose n points and label them from 1 to n. For each $i \in 1..n$ draw an arrow from i to $f(i)$. Fig. 5.19 shows the graphs for f_1 to f_4.

Each graph typically contains several cycles, which can be represented by the cycle notation underneath each one in Fig. 5.19. Note that fixed points, i.e. points where $f(i) = i$, are included. The permutation f_4 is termed *cyclic* because it has just one cycle.

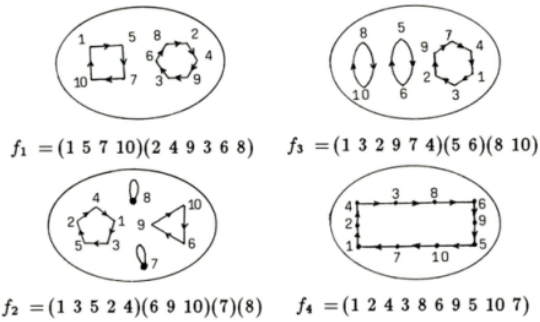


Figure 5.19. Permutation functions as graphs

Listing 5.24 (cycles.py) determines the cycles in the rearrangements stored in the ps permutation of 1..*n*.

```
n = int(input("n=? "))
ps = list(range(1,n+1))  # [1..n]
random.shuffle(ps)
print("Rearrangements:")
for i in range(n):
  print(f"{(i+1):2d} --> {ps[i]:2d}")
print("Cycles:")
for i in range(n):
  first = i
  if ps[i] > 0:  # start a cycle
    print("(", end =' ')
    while True:
      r = first
      print(r+1, end = ' ')
      first = ps[first]-1
      ps[r] = -ps[r]
      if first == i:
        break
    print(")") # end cycle
```

Listing 5.24. Finding cycles

An example:

```
> python cycles.py
n=? 10
Rearrangements:
 1 -->  5
 2 -->  7
 3 -->  3
 4 -->  1
 5 --> 10
 6 -->  9
 7 -->  8
 8 -->  6
 9 -->  4
10 -->  2
Cycles:
( 1 5 10 2 7 8 6 9 4 )
( 3 )
```

The algorithm starts with $i = 0$ and sets first=i. Then it repeatedly updates first = ps[first]-1 until i reoccurs, thereby closing the cycle. When an element joins a cycle it's marked by reversing its sign. When a cycle is completed, the code looks for the next unmarked element to start the next cycle.

5.6.3 The Josephus Permutation. We return to the Josephus Problem (n,k) of Sec. 1.7: n people are arranged in a circle, and every k-th person is eliminated, beginning with number k. Find the number x of the s-th eliminated person.

We now want to print the sequence $J(n,k)$ of eliminations – the *Josephus Permutation* of $1..n$. Only a slight modification of Listing 1.28 is needed, resulting in Listing 5.25 (josPerm.py).

```
n,k = map(int, input("n k=? ").split())
s = 0
while s < n:
  s += 1
  x = k*s
  while x > n:
    x = int((k*(x-n)-1)/(k-1))
  print(x, end =' ')
```

Listing 5.25. The Josephus Permutation

An example, represented by Fig. 5.20:

```
> python josPerm.py
n k=? 10 2
2 4 6 8 10 3 7 1 9 5
```

The last man standing is number 5.

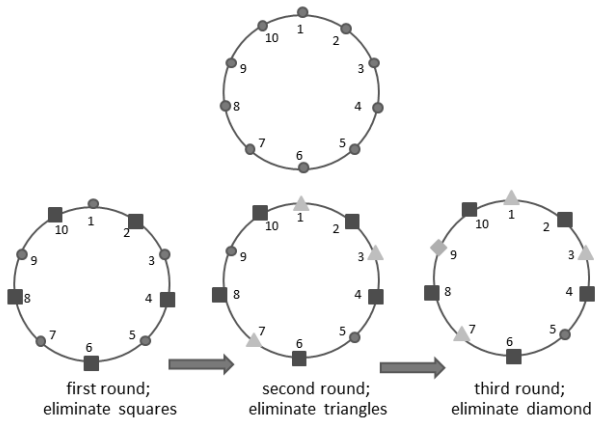


Figure 5.20. The Josephus problem (10,2)

Exercises

(1) Modify Listing 5.24 so that it counts the number of cycles.

(2) Generate multiple permutations of $1..n$, count the number of cycles in each one, and estimate the expected number of cycles in a random n-permutation. It can be shown that it equals the Harmonic series for n terms, $H_n = 1 + 1/2 + 1/3 + ... + 1/n$.

(3) We have n boxes, each with its own unique key. After mixing the keys at random, we drop one into each box which automatically locks shut. Now we smash open k random boxes. What's the probability $p(n, k)$ that we can use the k recovered keys to open the remaining boxes?

This was a problem in the 1971 Kürschák competition in Hungary (with $k = 2$). There are short but difficult ways of deriving the surprisingly simple formula for $p(n, k)$, but it may be easier to find the formula by simulation. First we translate the problem into the language of permutations: choose a random permutation p of $1..n$. What's the probability that the cycles in p containing $1..k$ also cover all the elements from $1..n$?

(4) Consider the following variation of Ex. 3: break open just one box and unlock all the boxes you can. Now break open another box and unlock all the additional boxes. Repeat this until you have opened every box. How many boxes do you have to break open on average?

(5) Show that the following Josephus permutations are cyclic:

a) $J(n, 2)$ for $n = 2, 5, 6, 9, 14, 18$;

b) $J(n, 3)$ for $n = 3, 5, 27, 89, 1139, 1219, 1921, 2155$;

c) $J(n, 4)$ for $n = 5, 10, 369, 609$;

d) $J(n, 7)$ for $n = 11, 21, 35, 85, 103, 161, 231$.

(6) *The golden permutation.* Let $t = (\sqrt{5}-1)/2$. The n pairs of numbers $(i, \text{frac}(i \times t))$ for $i = 1..n$ are sorted so their second values are in increasing order. (frac() returns the fractional part of a float.) When the corresponding first values are printed, the result is the so-called "golden permutation" of $1..n$. Because of its properties it's sometimes preferred by statisticians to a random permutation of $1..n$.

a) Write a program which prints the golden permutation of $1..n$ for $n = 100$.

With suitable coding additions check the following surprising properties:

b) Calculate the *absolute difference* between an element of the permutation and the previous one. (The last element can be regarded as preceding the first one.) Only three different numbers will occur in the entire permutation.

c) Elements with a '1' difference have a *distance* of 38, 39, or 62. (A distance is one more than the number of elements between two numbers with the same difference.) If these elements are organized to form the vertices of a regular 100-gon, only two shortest distances will be left over – 38 and 39.

d) Suppose a random sample of ten elements is to be selected from a sequence of consecutive integers, e.g. from 38..77. We start anywhere in the permutation, for example at 48, and choose successively those elements that lie in our

chosen interval 38..77. For instance, we might obtain 57, 70, 49, 62, 41, 75, 54, 67, 46, and 59, which is unusually uniformly distributed in the 38..77 range. Note also that if we order the ten numbers according to magnitude, only the differences 3 and 5 occur. Also, if we list numbers in their order of occurrence, then only the differences 13, 21, and 34 appear.

e) Consider the segment 38..77 on the number line. If the points 57, 70, 49,... are marked successively, then the current point falls in one of the currently largest free intervals, or in an end-interval.

f) Replace t in the code by t^2. Which of the b) to e) properties remain valid?

g) Replace t in the code by a different irrational number, such as $\sqrt{2}$, π , or e. Which properties remain valid?

h) Investigate b) to g) for values of n other than 100.

i) Sort the pairs of numbers $(i, 55 \times i \bmod 89)$ for $i = 0..88$, so that their second values are in increasing order. Check properties b) to e) for this permutation. Note that 55 and 89 are successive Fibonacci numbers. Repeat this question but with two relatively prime integers that are not Fibonacci numbers.

5.7 Coupled Difference Equations

How many words consisting of n digits from the alphabet A= {0, 1, 2, 3, 4} can be formed with the restriction that neighboring digits inside a word differ by exactly 1?

Table 5.4 outlines what this means for the words that can be built.

Words starting with ...	Digit Sequences
0	0 1 ...
1	1 0 ...
	1 2 ...
2	2 1 ...
	2 3 ...
3	3 2 ...
	3 4 ...
4	4 3 ...

Table 5.4. Possible word forms

Let x_n be the total number of words, and y_n, z_n, u_n the number of n-words starting with 0 (or 4), 1 (or 3), and 2. Then we can write four difference equations

$$\begin{aligned} x_n &= 2y_n + 2z_n + u_n \\ y_n &= z_{n-1} \\ z_n &= y_{n-1} + u_{n-1} \\ u_n &= 2z_{n-1} \end{aligned} \tag{5.7}$$

The boundary cases for these are when a word consists of just 1 digit, which means that $y_1 = 1, z_1 = u_1 = 1, x_1 = 5$.

Each of the equations can be mapped to a function that returns a count of words based on a supplied n argument, as in Listing 5.26 (coupledRecurrences.py).

```
def x(n):
  return 2*y(n) + 2*z(n) + u(n)

def y(n):
  if n == 1: return 1
  else:
    return z(n-1)

def z(n):
  if n == 1: return 1
  else:
    return y(n-1) + u(n-1)

def u(n):
  if n == 1: return 1
  else:
    return 2*z(n-1)
```

Listing 5.26. Implementing the difference equations

Table 5.5 shows the output when $n = 17$.

A convenient way to visualize the grammar is using a graph with four states for the alphabet's digits (see Fig. 5.21). If there is a two-way connection between two states, they're linked with a single line without an arrow.

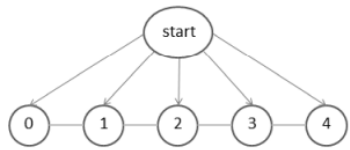


Figure 5.21. Graph of the word grammar

n	x_n	y_n	z_n	u_n
1	5	1	1	1
2	8	1	2	2
3	14	2	3	4
4	24	3	6	6
5	42	6	9	12
6	72	9	18	18
7	126	18	27	36
8	216	27	54	54
9	378	54	81	108
10	648	81	162	162
11	1134	162	243	324
12	1944	243	486	486
13	3402	486	729	972
14	5832	729	1458	1458
15	10206	1458	2187	2916
16	17496	2187	4374	4374
17	30618	4374	6561	8748

Table 5.5. Output from Listing 5.26 for $n = 17$.

Exercises

(1) Study Table 5.5 and find closed formulas for x_n, y_n, z_n, u_n. *Hint*: consider even and odd n values separately.

(2) How many n-words from the alphabet {0, 1, 2, 3} do *not* contain the two words 01 or 10? Write a program based on the difference equations for this problem.

(3) How many n-words from the alphabet {0, 1, 2} have neighboring digits differing by at most 1?

5.8 Games

In this section, we'll look at turn-based games for two players, A and B. Player A always moves first but the rules are otherwise the same for both participants.

We're given a starting state and the set M of legal moves. A player loses if he finds himself in a position from which no legal move can be made.

We can think of each game position as a vertex of a graph, and each move as a directed edge. We'll restrict our games to only having finitely many vertices (positions) without circuits (i.e. a position can only be used once). This ensures that one of the players will eventually lose.

It can be shown by induction on the number of moves that the set P of all positions can be split in two: a set L of losing positions, and a set W of winning positions: $P = L \cup W, L \cap W = 0$ (see Fig. 5.22).

A player finding himself in an L position will lose provided his opponent plays optimally, while a player in a W position can force a win whatever his opponent may choose to do. In order to win, a player must always move so as to force his opponent to choose a move to an L position. From each L position, every move may end in a W position, but for every W position, a move to an L position is possible. L must contain at least one final position f from which there is no escape. The player who leaves his opponent facing such a position has won the game. These criteria are illustrated in Fig. 5.22.

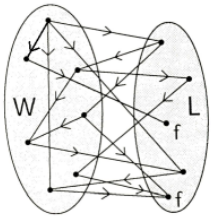


Figure 5.22. Winning (W) and losing (L) game positions

Our problem is to identify the set L of losing positions for a particular game.

5.8.1 Nim. Nim has two players take turns removing chips from several piles. On each turn, a player must remove at least one chip, and may remove any number provided they all come from the same pile. The player removing the last chip (or chips) wins.

A complete mathematical analysis of Nim was worked out by C.L. Bouton in 1902, and it's been a popular topic ever since, discussed by Rouse Ball and Coxeter [**BC87**], Kasner and Newman [**KN01**], and Beasley [**Bea06**].

Nim Game 1. Two players take turns to reduce a *single* pile of n chips. A move consists in removing m chips, where m is a number in the set $\{1, 3, 8\}$. The winner is the one who removes the last chip. Listing 5.27 (nim.py) prints the losing positions for this version.

```
n = int(input("n=? "))
wins = [False]*(2*n)
legalMoves = [1, 3, 8]
for i in range(n):
  if not wins[i]:
    for m in legalMoves:
      wins[i+m] = True
print("Losing Positions:")
for i in range(n-1):
  if not wins[i]:
    print(f"{i:2d} ", end='')
  else:  # a win
    print(" - ", end ='')
  if (i+1)%11 == 0:
    print()
```

Listing 5.27. First Nim game

The output is for a game starting with 100 chips.

```
> python nim.py
n=? 100
Losing Positions:
 0  -  2  -  4  -  6  -  -  -  -
11  - 13  - 15  - 17  -  -  -  -
22  - 24  - 26  - 28  -  -  -  -
33  - 35  - 37  - 39  -  -  -  -
44  - 46  - 48  - 50  -  -  -  -
55  - 57  - 59  - 61  -  -  -  -
66  - 68  - 70  - 72  -  -  -  -
77  - 79  - 81  - 83  -  -  -  -
88  - 90  - 92  - 94  -  -  -  -
```

Position '0' is the initial move (by Player A) which has the potential of being a losing position. The last position is for the 99th move, which is printed as '-' indicating that it is a winning position, since the player can remove the last chip.

The output strongly suggest a pattern with a period of 11. The L set consists of all the non-negative integers of the form $11k, 11k+2, 11k+4, 11k+6$.

This variant of Nim is sometimes called the *Subtraction Game* and its analysis is particularly simple. Player A will lose if and only if $n \pmod{m+1} = 0$, where n is the size of the pile and m is one of the legal moves. In this example, $n = 100$, and the modulo produces 0 when $m = 1$ or $m = 3$. If Player B knows what they are doing, Player A has no chance of winning.

Nim Game 2. In this version, the legal moves are the set of squares {1, 4, 9, 16, 25, 36, ...}. Listing 5.28 (squareNim.py) prints the losing positions.

```
n = int(input("n=? "))
wins = [False]*(n*n)
legalMoves = [(i*i)
  for i in range(1, int( math
      .sqrt(n))+1) ]
print(legalMoves)
for i in range(n):
  if not wins[i]:
    for m in legalMoves:
      wins[i+m] = True
print("Losing Positions:")
for i in range(n-1):
  if not wins[i]:
    print(f"{i:3d} ", end='')
  else:  # a win
    print(" -- ", end ='')
  if (i+1)%10 == 0:
    print()
print()
```

Listing 5.28. Second Nim game

The output is for a game starting with 130 chips.

```
> python squareNim.py
n=? 130
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
Losing Positions:
  0  --   2  --  --   5  --   7  --  --
 10  --  12  --  --  15  --  17  --  --
 20  --  22  --  --  --  --  --  --  --
 --  --  --  --  34  --  --  --  --  39
 --  --  --  --  44  --  --  --  --  --
 --  --  52  --  --  --  --  57  --  --
 --  --  62  --  --  65  --  67  --  --
 --  --  72  --  --  --  --  --  --  --
 --  --  --  --  --  85  --  --  --  --
 --  --  --  --  --  95  --  --  --  --
 --  --  --  --  --  --  --  --  -- 109
 --  --  --  --  --  --  --  --  -- 119
 --  --  --  -- 124  --  -- 127  --
```

There's no apparent periodicity, but there are suggestive patterns. Losing positions tend to end in 0, 2, 4, 5, 7, or 9, and the other digits seem very uncommon. More details on this "Subtract a square" version of Nim can be found at `https://en.wikipedia.org/wiki/Subtract_a_square`.

5.8.2 Wythoff's Nim. There's an old Chinese game called *tsyan-shidzi*, meaning "choosing stones", with a beautiful mathematical theory. The game was reinvented by the Dutch mathematician W. A. Wythoff, who published an analysis of it in 1907, and so it's now commonly known as "Wythoff's Nim".

Here are the rules: on a table there are two piles of chips, and two players who take turns. A move consists of taking any number of chips from one pile *or* the same number of chips from *both* piles (which isn't allowed in standard Nim). The winner is the one who takes the last chip.

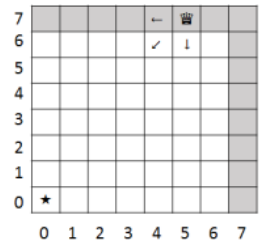


Figure 5.23. The "Corner the Lady" game

We can translate the game into the board game shown in Fig. 5.23, invented in the early 1960s by Rufus P. Isaacs, a mathematician at Johns Hopkins University. Martin Gardner named it: "Corner the Lady" in an informative article about Wythoff's Nim in *Scientific American* [**Gar97**].

Player A puts a queen on any cell in the top row or in the column farthest to the right; the relevant cells appear in gray in Fig. 5.23. The queen moves in the usual way but only west, south, or southwest. Player B moves first, then the players alternate. The player who gets the queen to the starred cell at the lower left corner is the winner.

The isomorphism between the Queen-Cornering game and Wythoff's Nim is fairly apparent. In Fig. 5.24a, we number the columns and rows along the x and y axes, so that each cell can be assigned an (x, y) pair. These numbers correspond to the number of chips in piles x and y in Wythoff's Nim. When the queen moves west, pile x is diminished. When the queen moves south, pile y is reduced. When it moves diagonally southwest, both piles shrink by the same amount. Moving the queen to cell (0,0) is equivalent to reducing both piles to 0.

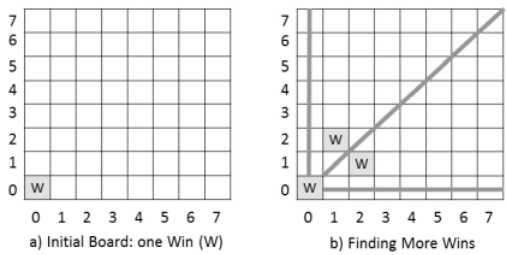


Figure 5.24. Winning the "Corner the Lady" game

In the following analysis, one of the authors will assume the role of player A, and (0,0) is a winning position (W) for me if on my turn I can move to that square.

My possible starting positions necessary for this winning move are marked with lines in Fig. 5.24b. I want Player B to land in one of these marked squares, so how do I ensure that? I must end my previous move in one of the two W squares

in Fig. 5.24b, so that Player B starts from one of them. He can only move to a marked square, from which I can win. Note that these W squares are (1, 2) and (2, 1).

In Fig. 5.25, I extend this reasoning two more stages backwards.

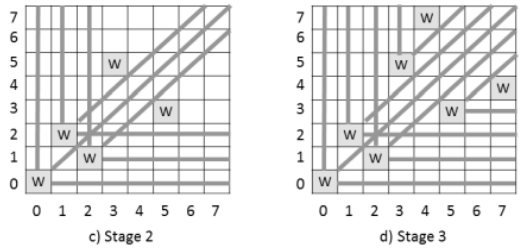


Figure 5.25. Winning "Corner the Lady" (stages 2 and 3)

Fig. 5.25c uses lines to mark all the ways I can move to the winning spots at (1, 2) or (2,1). Therefore on my previous move I want Player B to start from one of the two new W squares ((3,5) and (5,3)) because then he can only move to a marked square, from where I can win.

Fig. 5.25d continues this backwards reasoning to find two more winning spots for me – (4,7) and (7,4). I should finish on one of these squares so that Player B must move to a marked square from where I can win.

I've identified seven W positions: (0,0), (1,2), (2,1), (3,5), (5,3), (4,7) and (7,4). There's an obvious symmetry about the main diagonal, so I'll focus on (0,0), (1,2), (3,5), and (4,7). Wythoff carried out a similar analysis of his game, and came up with Table 5.6.

n	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
xs(n)	0	1	3	4	6	8	9	11	12	14	16	17	19	21	22	24	25	27	29
ys(n)	0	2	5	7	10	13	15	18	20	23	26	28	31	34	36	39	41	44	47

Table 5.6. Losing positions for player B in Wythoff's Nim

Note that Wythoff classified the Table 5.6 positions as *losing ones for Player B*, which is equivalent to my looking for winning moves for Player A.

The table suggests the following algorithm for constructing the set L of losing positions for Player B: suppose the losing positions (xs(i), ys(i)) for $i < n$ are known already. Then xs(n) is the smallest positive integer not yet part of a pair, and ys(n) = xs(n) + n. This is a lot easier to implement in code than 'marking' horizontal, vertical , and diagonal lines.

Also note that every positive integer occurs exactly once as a member of a pair.

The algorithm is translated into Listing 5.29 (wythoff.py) which lets us easily generate Table 5.6, and also plot the pairs.

```
n = int(input("n=? "))
avails = [True]*2*(n+1)
  # ints for use in pairs
xs = [0]*(n+1)
ys = [0]*(n+1)
i = 0
for j in range(1,n+1):
  if avails[j]:
    i += 1
    xs[i] = j
      # smallest not yet used
    ys[i] = j+i
      # sum of xs[i] and i
    avails[i] = False  # i, j+1  used
    avails[j+i] = False
print(" n   xs  ys")
for j in range(i+1):
  print(f"{j:2d}:  {xs[j]:2d}  {ys[j]:2d}")
```

Example output:

```
> python wythoff.py
n=? 30
 n   xs  ys
 0:   0   0
 1:   1   2
 2:   3   5
 3:   4   7
 : # more lines
16:  25  41
17:  27  44
18:  29  47
19:  30  49
```

Listing 5.29. Generating Wythoff losing moves for player B

The plot of these (x, y) pairs is shown in Fig. 5.26.

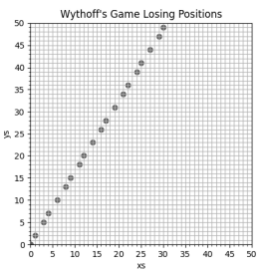


Figure 5.26. Plot of Wythoff losing moves for player B

We should also verify that L is indeed the set of losing positions for Player B.

Let W be the set of pairs not in L. It's easy to check that every move from L leads to a position in W. It remains to show that from any position in W we can move into L.

Let $(p,q) \in W, p \le q$. If $p = q$, I (player A) can move to $(0,0) \in$ L and win in one move. If $p \ne q$, let (p, p') or (p', p) be the element of L with a p value in the pair. Note that there are two cases since the p number of chips may be in either pile.

If $p' < q$, I reduce q to p'. If $q < p'$, then we know that $p < q < p'$ which is equivalent to $0 < q - p < p' - p$ (by subtracting p from all terms). To win I must reduce both piles by equal amounts, which will leave a number of chips equal to the difference $q - p$, which is in L.

5.8.2.1 The Hidden Golden Ratio. Wythoff noticed a hidden feature of Table 5.6 – the numbers in the xs sequence are multiples of the golden ratio rounded down to integers, and the ys sequence are multiples of the squared golden ratio rounded down to integers. (He later wrote that he pulled this discovery "out of a hat".)

Listing 5.30 (phis.py) can be used to confirm this.

```
phi = (math.sqrt(5)+1)/2
phi2 = phi * phi
print("n   xs  ys")
for i in range(19):
  print(f"{i:2d}  {int(i*phi):2d}  {int(i*phi2):2d}")
```

Listing 5.30. The hidden golden ratio

The hidden ϕ^2 in the ys sequence isn't hard to find since $\phi^2 = \phi + 1$, and so

$$\lfloor \phi^2 n \rfloor = \lfloor \phi n + n \rfloor = \lfloor \phi n \rfloor + n$$

which defines the ys sequence in terms of xs ($\lfloor \phi n \rfloor$).

The fact that every positive integer appears just once among the pairs can now be rephrased as all the integers that lie between successive multiples of ϕ and successive multiples of ϕ^2.

Two sequences of increasing positive integers that together contain every positive integer only once are called *complementary*.

ϕ isn't the only irrational number that generates such sequences, although it's the only one that matches the pairs of Wythoff's Nim. In 1926 Sam Beatty, a Canadian mathematician, reported that *any* positive irrational number (e.g. $\sqrt{2}, \pi, e$) can generate complementary sequences. His theorem states that if $\alpha > 1$ is irrational then the complementary sequence of $\lfloor \alpha n \rfloor$ is the sequence $\lfloor \beta n \rfloor$, such that $1/\alpha + 1/\beta = 1$.

For example, let $r = \sqrt{2} \approx 1.414$, and $s = 2 + \sqrt{2} \approx 3.414$. Note that $s = r/(r-1)$ and so $1/s + 1/r = 1$.

The sequence $\lfloor rn \rfloor$ is 1, 2, 4, 5, 7, 8, 9, 11, 12, 14, 15, 16, 18, 19, 21, 22, 24, ...

The sequence $\lfloor sn \rfloor$ is 3, 6, 10, 13, 17, 20, 23, 27, 30, 34, 37, 40, 44, 47, 51, 54, 58, ...

A comparison of these show that they are indeed complementary.

This theorem applies to the golden ratio sequences since $\phi^2 = \phi + 1$, so $1/\phi + 1/\phi^2 = 1$, and hence $\lfloor \phi n \rfloor$ and $\lfloor \phi^2 n \rfloor$ form complementary sequences.

Exercises

(1) Run Listing 5.28 for large values of n and plot the strange fluctuations of the L elements.

(2) For Wythoff's Nim:

a) Write a program which transforms a position in W into a position in L.

b) Write a program which plays against a human opponent.

c) Write a program which plays against a human opponent, based on the golden ratio formulas.

(3) In a game with a pile of n chips, the legal moves are the set $M = \{1, 2, 3, 5, ...\}$ of Fibonacci numbers. Write a program which finds the losing positions.

(4) *Bachet's game.* Initially there are n chips on the table. The set of legal moves is $M = \{1, 2, ..., k\}$. The winner is the person who removes the last chip. Find the losing positions.

(5) In Ex. 4 let $M = \{1, 2, 4, 8, ...\}$ (any power of 2). Find the set L.

(6) In Ex. 4 let $M = \{1, 2, 3, 5, 7, 11, ...\}$ (1 and all the primes). Find L.

(7) In Ex. 4 let $M = \{1, 3, 5, 8, 13\}$. Find the set L.

(8) Write a program that reads in a finite sequence M of positive integers which will act as the game's legal moves. Make predictions about the set L of losing positions.

(9) Write a recursive version of Listing 5.27, which recognizes if an input n belongs to L.

(10) Let xs(n) = $\lfloor \phi n \rfloor$, and consider the sequence as(n) = xs(n+1) - xs(n) for $n = 0$, 1, 2, ... Its first few terms are shown in Table 5.7.

1	2	1	2	2	1	2	1	2	2	1	2	2	1	2
12	122	12	122	122	12	122	12	122	122	12	122	122	12	122

Table 5.7. Terms of the as(n) sequence

The second line arises from the first by inflation: $1 \to 12, 2 \to 122$. This seems to transform the sequence into itself, which would make it self-similar.

a) Test this self-similarity as far out as possible.

b) What's the proportion of 2's among all the digits? Are you surprised? (See the solution of Ex. 3 in Sec. 3.14.]

(11) We start with 0 and repeatedly perform the substitutions $0 \to 1, 1 \to 10$ to generate 0, 1, 10, 101, 10110, 10110101, ... In this way we produce an infinite binary sequence, which is more complicated than the Morse Thue sequence.

Investigate the sequence, especially its self-similarity, proportion of 1's, etc. Compare it with Ex. 24 in Sec. 2.1, and Ex. 3 in Sec. 3.14 where the more difficult substitution $0 \to 001, 1 \to 0$ were examined.

(12) In a Nim-style game, there are 10,000 chips on the table. If it's your turn then you may remove any p^n amount of chips, where p is a prime and $n = 0, 1, 2, 3, \ldots$. The winner is the player who takes the last checker. Is that player A or B, if he plays optimally?

(13) In this question's modification of Wythoff's game, you're allowed to remove an *even* number of chips from a single pile, or equal numbers from both piles. By modifying Listing 5.29 find the set L and closed formulas for its elements. *Hint*: there will be two formulas, depending on the parity of the initial number of chips. In case of an odd initial number the final position will be (0,1).

(14) In this game start with $n = 2$ chips. The players A and B move alternately by adding to the current pile of size n a proper divisor of n. The goal is a number ≥ 1990. Who wins, A or B?

(15) In this version of Wythoff's game, you may remove any number of chips from one pile, or m chips from one pile and n chips from the other pile where $|m - n| < a$. For $a = 3$ find the losing positions and guess closed formulas for them. You've met these formulas before in Ex. 3 in Sec. 3.14.

5.9 The Subset Sum Problem (Part 2)

We're given a set of n positive reals and the task of selecting a subset which sums to a value as close as possible to a goal number, G, without exceeding it. We'll focus on a special case of this famous and notorious problem, due to R.W. Floyd, which appears as Ex. 1.4 in Hopcroft [**HUA83**]:

> Abby and Carl inherit n gold coins with weights $\sqrt{1}, \sqrt{2}, \ldots, \sqrt{n}$ kgs. Divide them into two piles whose total weights differ from each other by as little as possible.

This version of the problem is treating G as:

$$G = \frac{\sqrt{1} + \sqrt{2} + \cdots + \sqrt{n}}{2}$$

We can obtain a good, but not necessarily optimal, solution by employing a FFD (First-Fit-Decreasing) heuristic: arrange the coins into decreasing weight order, and assign the heaviest to Abby if the new total weight doesn't exceed G, otherwise give it to Carl. Repeat this until all the coins have been allocated.

Listing 5.31 (FFD.py) implements this approach, and indicates the final distribution of weights to Abby as a binary word. Its k-th digit is 1 if Abby has been assigned the coin of weight $\sqrt{k}$.

```
n = int(input("n=? "))
wghts = []
for i in range(n):
  wghts.append(math.sqrt(i+1))
goal = sum(wghts)/2
print("goal:", goal)
print("Abby's weights: ",end='')
abby = 0; carl = 0
for i in range(n-1,-1,-1):
  # large to small
  if (abby + wghts[i]) < goal:
    # assign w_i to Abby
    abby += wghts[i]
    print(1, end='')
  else:   # assign to Carl
    carl += wghts[i]
    print(0, end ='')
print()
print("Total Weights")
print(f"abby: {abby:.9f}")
print(f"carl: {carl:.9f}")
```

Listing 5.31. Implementing FFD

The result when $n = 20$:

```
> python FFD.py
n=? 20
goal: 30.832988905709904
Abby's weights:
   11111110000000001000
Total Weights
abby: 30.811421944
carl: 30.854555867
```

The FFD algorithm yields a good result (accurate to 1 d.p.) so it might even be optimal. To check, we should estimate how close we can get to an even division of the weights.

The sum of the seven largest weights is so much smaller than $G/2$ that we can add a light coin and still stay under $G/2$. Hence the share of weights for each player should consist of at least 8 coins and, by symmetry, at most 12.

There are

$$\binom{20}{8} + \binom{20}{9} + \cdots + \binom{20}{12} = 772,616$$

subsets with 8 to 12 coins, with their corresponding weights lying between 16 and 46 kgs. If the weights were uniformly distributed across this interval, each subinterval would be $\approx 30/770000 \approx 3.9 * 10^{-5}$. That's a lot smaller than the current difference between Abby's and Carl's weights, so it seems likely that our solution can be improved.

How do we find the optimal solution? We could try testing every possible subset, stepping through all 2^{20} and choosing the one with a summed weight as close to G as possible. Ideally, in each step of this search, exactly one weight should be added or removed, so the subset sum is easy to update. In terms of the binary word representation of Abby's weights, this approach corresponds to changing one bit in each step. Is this even possible, when all of the 2^{20} subsets

must be visited? The answer is yes, and is relatively easy to implement if we use *Gray codes* [**Gar20**].

5.9.1 Gray Codes. A Gray code represents a sequence of numbers using a positional notation where any adjacent pair will differ in their digits at one position only, with an absolute difference of 1. For instance, 183 and 193 could be adjacent numbers in a decimal Gray code (the middle digits differ by 1), but not 173 and 193, nor 134 and 143.

Binary Gray codes are the simplest since two successive values in a sequence only differ by one bit. They've proved so useful that they've become a common tool for error detection and recovery in digital communication and data storage.

If we limit the code to one bit, the only possible numbers are 0 and 1, which we can graph as a straight line (see Fig. 5.27a). A Gray code is obtained by moving along the line in either direction, and we've chosen to define the code as {0, 1}

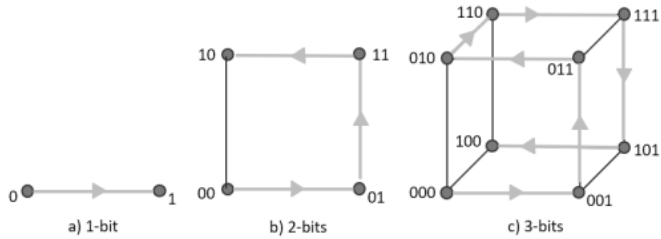


Figure 5.27. Gray codes

A Gray code with two bits is { 00, 01, 11, 10 }, which labels the corners of the square in Fig. 5.27b. Note that the binary numbers at any pair of adjacent corners differ in only one place. The path starting at 00 could be made cyclic by having it return to 00 from 10.

A 3-bit Gray code is used to label the corners of the cube in Fig. 5.27c. The path shown by the lines marks out the Gray code { 000, 001, 011, 010, 110, 111, 101, 100 }. Once again it could be made cyclic by linking from 100 back to 000.

These are called Hamiltonian paths after the Irish mathematician William Rowan Hamilton, and n-bit Gray codes can always be mapped to Hamiltonian paths on cubes of n dimensions. The 4-bit Gray code is shown mapped to a hypercube in Fig. 5.28.

For practical purposes a Gray code should have simple conversion rules for translating standard binary coded decimals to their Gray code equivalents and vice versa. The simplest is called a reflected Gray code.

Start with a 1-bit Gray code {0, 1}, duplicate it, and reverse the duplicate. Add 0s to the left of the original sequence, and 1s to the reversed duplicate, and

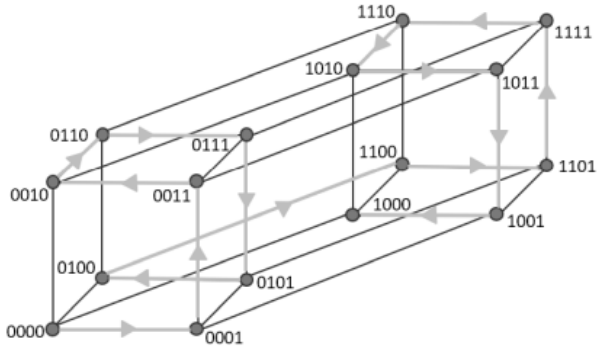


Figure 5.28. A 4-bit Gray code on a hypercube

the combination of the two sequences is a 2-bit Gray code. The single-bit change property is upheld by the reflection property, and also makes the sequence potentially cyclic. This process can be repeated, as illustrated in Fig. 5.29, to generate Gray codes with more bits.

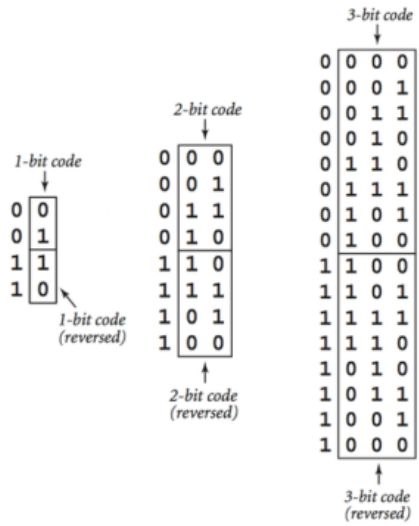


Figure 5.29. Reflected Gray codes

Listing 5.32 (grays.py) implements this approach.

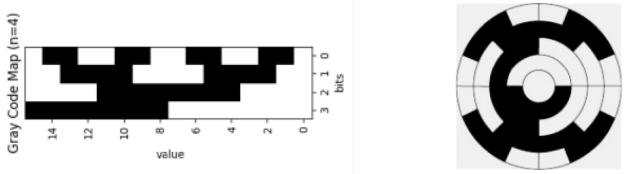
```
def grayCodes(n):
  if n <= 0:
    return ['0']
  elif n == 1:
    return ['0', '1']
  half1 = grayCodes(n-1)
  revhalf2 = reversed(half1.copy())
  return ['0'+bs for bs in half1] + \
         ['1'+bs for bs in revhalf2]
```

Listing 5.32. Generating reflected Gray codes

Examples:

```
>>> from grays import *
>>> grayCodes(3)
['000', '001', '011', '010',
 '110', '111', '101', '100']
>>> grayCodes(4)
['0000', '0001', '0011', '0010',
 '0110', '0111', '0101', '0100',
 '1100', '1101', '1111', '1110',
 '1010', '1011', '1001', '1000']
```

A useful visualization of reflected gray codes is a grid of their bit values, with black squares for 1s and white squares for 0s (see Fig. 5.30a produced by gcMap.py). This pattern (suitably rotated as in Fig. 5.30b) is utilized in some rotary position encoders as conductive strips around concentric rings.



(a) As a grid (b) As a ring

Figure 5.30. 4-bit Gray code patterns

Listing 5.33 (grays.py) converts between binary and reflected Gray codes.

```
def grayEncode(n):
  return n ^ (n >> 1) # xor

def grayDecode(n):
  mask = n >> 1
  while mask:
    n ^= mask; mask >>= 1
  return n
```

Listing 5.33. To/from a Gray code

Convert 1000_2 to the Gray code value 1100, and back again:

```
>>> from grays import *
>>> bin(grayEncode(0b1000))
'0b1100'
>>> bin(grayDecode(0b1100))
'0b1000'
```

bin() is a Python built-in for converting an integer to a binary string.

5.9.1.1 Gray Code Transition Sequences. A Gray code can be represented by its *transition sequence* – a list of the bits position changes as we move from one

Gray code value to the next. We assume that the positions are numbered from right to left, starting at 1.

The transition sequences for the 1-bit, 2-bit, and 3-bit Gray Codes in Fig. 5.27 are:

$$T_1 = 1, \quad T_2 = 1, 2, 1, \quad T_3 = 1, 2, 1, 3, 1, 2, 1.$$

The T_4 transition sequence for Fig. 5.28 is:

$$1, 2, 1, 3, 1, 2, 1, 4, 1, 2, 1, 3, 1, 2, 1$$

The underlying reflection mechanism used to create the Gray codes is apparent in their bit changing indices. The general equation for the transition sequence is

$$T_{n+1} = T_n, (n + 1), T_n \text{ reversed}.$$

We can easily prove by induction that for all n

$$T_n = T_n \text{ reversed}$$

and so the recursive definition simplifies to

$$T_1 = 1, \quad T_{n+1} = T_n, (n + 1), T_n.$$

We can generate the T_n sequence efficiently by using a stack. It initially contains the values $n, n - 1, ..., 1$ (as in Fig. 5.31).

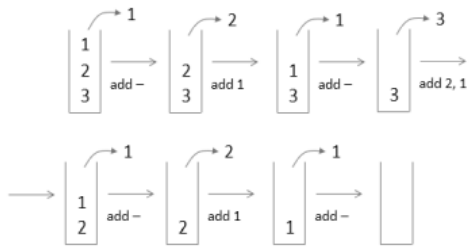


Figure 5.31. 3-bit Gray code transitions using a stack

The algorithm (see Listing 5.34; grayTransStack.py) removes the top number i and adds it to the sequence. Then the numbers $i - 1, i - 2, ..., 1$ are pushed onto the stack. This approach is from Reingold, Nievergelt and Deo [**RND77**], p. 178.

```
n = int(input("n=? "))
stk = [i for i in range(n, 0, -1)]
while stk != []:
  t = stk.pop()
  print(t, end = ' ')
  for i in range(t-1, 0, -1):
    stk.append(i)
print()
```

Listing 5.34. A stack for transitions

Examples:

```
> python grayTransStack.py
n=? 3
1 2 1 3 1 2 1
> python grayTransStack.py
n=? 4
1 2 1 3 1 2 1 4 1 2 1 3 1 2 1
```

T_n is also known as the *ruler* function because when its values are plotted as bars, they look very similar to the markings on a ruler (see Fig. 5.32).

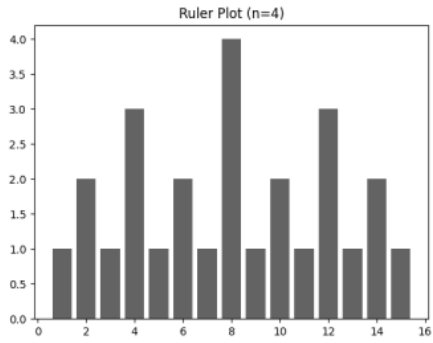


Figure 5.32. 4-bit Gray code transitions as a ruler

The code for generating Fig. 5.32 can be found in `ruler.py`.

5.9.1.2 Transition Sequences for Search. A transition sequence can be used to efficiently iterate through every possible subset of an n-element set. Consider the 3-bit Gray code:

$$000, 001, 011, 010, 110, 111, 101, 100$$

If we have a three element set, say {a,b,c}, then each Gray code number can represent a subset of that set, where 0 in the i-th position means do not include the value, and 1 means inclusion. Applying the Gray code above in this way produces the subsets:

$$\{\}, \{c\}, \{b, c\}, \{b\}, \{a, b\}, \{a, b, c\}, \{a, c\}, \{a\}$$

A key benefit of this particular ordering of the {a,b,c} powerset is that there's only one change between adjacent subsets. The change is made explicit by the corresponding transition sequence for the 3-bit Gray code $\{1, 2, 1, 3, 1, 2, 1\}$. Remember

that we're counting indices from right-to-left in transitions, and starting at 1. So the first '1' denotes the powerset change from {} to {c}. The '2' is the transition {c} to {b,c}. The next '1' means a change to index position 1, which will *remove* the *c*, leaving the subset {b}.

There are a surprisingly large number of search problems which can benefit from the use of Gray codes to efficiently traverse the search space. As an example, consider the problem of generating all the multiples of {2, 3, 5, 7}. We'll utilize a generator version of Listing 5.34 to produce a 4-bit Gray code transition sequence one value at a time. These values will guide how the program (see Listing 5.35; multAll.py) iterates through the multiples.

```
nums = [2, 3, 5, 7]
n = len(nums)
grayGen = grayTransY(n)
 # transitions for n-bit Gray codes
prod = 1
for i in range(2**n-1):
  idx = next(grayGen)
     # get a transition
  val = nums[idx-1]
  if val > 0:
    prod *= val
  else: # flag inclusion
    prod = prod//-val
  nums[idx-1] = -nums[idx-1]
  print(idx, ":", prod)
```

Listing 5.35. Generating all multiples

The output shows the transition sequence value and the multiple that it generates:

```
> python multAll.py

1 : 2          1 : 70
2 : 6          2 : 210
1 : 3          1 : 105
3 : 15         3 : 21
1 : 30         1 : 42
2 : 10         2 : 14
1 : 5          1 : 7
4 : 35
```

The only problem with using Gray code transition sequences is the double meaning of a value. We saw this issue in the set example above, where '1' initially meant include *c*, but later meant remove it. This dual meaning is dealt with in Listing 5.35 by making a number negative when it becomes part of the multiple, and positive again upon its removal.

The generator version of the stack-based transition sequence code (Listing 5.34) is in Fig. 5.36.

```
def grayTransY(n):
  # generate gray code transitions from 0 to 2^n -1
  stk = [i for i in range(n, 0, -1)]
  while stk != []:
    t = stk.pop()
    for i in range(t-1, 0, -1):
      stk.append(i)
    yield t
```

Listing 5.36. Transition sequence generator

5.9.2 Subset Sums and Gray Codes. Armed with Gray codes, we return to our collection of 20 golden coins, with weights ranging from $\sqrt{1}, \sqrt{2}, ..., \sqrt{20}$ kgs. The task is to find a subset closest to half the total weight (i.e. $G = 30.832988905709904$).

We'll employ the transition sequence for a 20-bit gray code to efficiently search through all the possible subsets of the coin weights looking for the closest match. Nowadays, 2^{20} subsets isn't a large number, but let's still try to reduce the search space.

We can immediately cut the computation time in half by checking only the first 2^{19} subsets, excluding $\sqrt{20}$. The excluded subsets are the complements of the ones we are checking, so can be ignored.

Another trick cuts the number of subsets by an additional factor of 16. We remove the set S of *integer roots* in the range 1 to 20 = $\{\sqrt{1}, \sqrt{4}, \sqrt{9}, \sqrt{16}\} = \{1, 2, 3, 4\}$. The powerset of S represents all the integers from 0 to 10, which we can more efficiently represent as an integer variable k. Now we look for a subset with weight w such that $|G - w - k|$ is as small as possible. We will eventually choose a k value that makes $|G - w - k|$ the smallest.

Instead of trying 11 different values for k, the best choice can be obtained directly by calculating round(G-w). Now the problem becomes one of finding a w such that $|G - w - \text{round}(G - w)|$ is as small as possible, provided $0 \leq$ round(G - w) ≤ 10.

Listing 5.37 (subsetSum.py) is hardwired to look for the 16 non-integer square roots in the range 2 to 19. They are generated with `math.sqrt(i+round(math.sqrt(i))`, an approach we first used in Ex. 1 of "Additional Exercises for Chapters 1 to 2".

```
N = 16  # non-squares

nis = [0]*N  # non-integer squares
print("nis[] entries")
for j in range(N):
  nis[j] = math.sqrt(j+1 + round( math.sqrt(j+1)))
grayGen = grayTransY(N)

goal = (10+sum(nis))/2
minW = 1; wghts = 0; i = 0
while i < N-1: # reduce comparisons
  i = next(grayGen)
  wghts += nis[i]
  nis[i] = -nis[i] # -ve means used in weights
  rd = round(goal - wghts)
  nWeight = abs((goal - wghts) - rd)
  if nWeight <= minW:
    if (rd >= 0) and (rd <= 10): # 0 to 10
```

```
    for j in range(N):
      print(int(nis[j] < 0),end ='')
      # 0 (not used) or 1 (used)
    minW = nWeight
    print(f"; min == {minW:.11f}")
```

Listing 5.37. Summing subsets

`minW` will shrink as it is assigned smaller values of $|G - w - \text{round}(G - w)|$. Its current value is printed as the program progresses, along with a binary word indicating which of the 16 weights have been used.

```
> python subsetSum.py
0111111110000000; min == 0.00180212361
0111110011010000; min == 0.00143429476
0001000111011000; min == 0.00113251014
0101011101111000; min == 0.00082114602
0010001011001100; min == 0.00075593530
0101110011101100; min == 0.00021332943
0100001010010110; min == 0.00003584597
```

We use the transition sequence generator, grayTransY(), retrieving the next transition in a while-loop. Once again we've also using the trick of negating a value when it becomes part of the subset, so that next time, the program knows to remove it.

The binary word output shown above requires a little deciphering. The weights $\{\sqrt{2}, \sqrt{3}, \sqrt{5}, \sqrt{6}, \cdots\}$ are denoted from left-to-right, with no integer square roots included. This means that the final 0100001010010110 result corresponds to:

$$\begin{aligned} 0100001010010110 &= \sqrt{3} + \sqrt{10} + \sqrt{12} + \sqrt{15} + \sqrt{18} + \sqrt{19} \\ &\approx 1.732 + 3.162 + 3.464 + 3.873 + 4.243 + 4.359 \\ &= 20.833 \end{aligned}$$

We're aiming for a number close to G, which is approximately 30.832988905709904. Our 20.833 result is not an error, but a consequence of the exclusion of the four integer square roots $(\sqrt{1}, \sqrt{4}, \sqrt{9}, \sqrt{16})$. We manually decide how many of these to add to 20.833 to get a result close to G. The best combination is to add all of them (i.e. 10), to get a final weight for Abby's coins of approximately 30.833. To be more precise, our final result is 30.8329530597.

The absolute and relative deviations from G are

$$|\text{result} - G| \approx 3.6 * 10^{-5}, \quad \frac{|\text{result} - G|}{G} \approx 10^{-6}$$

You may recall that we estimated that we could get as close as $3.9 * 10^{-5}$ to G, which turns out to be (approximately) the case.

There are actually two versions of our result. Instead of using $\{\sqrt{3}, \sqrt{10}, \sqrt{12}, \sqrt{15}, \sqrt{18}, \sqrt{19}\}$ we could replace $\sqrt{18}$ by $\sqrt{2}$ and $\sqrt{8}$. Listing 5.37

didn't report this other answer because our coding only reports reductions to `minW`; probably the other value was slightly bigger due to rounding errors and so was skipped.

Exercises

(1) Suppose k people inherit n gold pieces weighing $x_1, x_2, ..., x_n$. We want to split the inheritance into k heaps which are as nearly equal as possible.

For this case, the FFD algorithm is defined as follows: first order the items so that $x_1 \geq x_2 \geq ... \geq x_n$. Then proceed to distribute the items in order, starting with x_1, which we place in the first heap H_1. In general, if there are heaps H_j which have room for x_i, i.e. the total size of items in H_j is not more than goal$-x_i$, then we place x_i on the heap with the smallest index j. Otherwise we place it on the smallest heap; if there are several equally small heaps, we pick the one with the smallest index.

Write a program for this method and apply it to $k = 3$, $x_i = \sqrt{i}$, and $n = 20$, 30, and 40.

(2) Prove that Listing 5.34 works correctly. (Not easy.)

(3) Partition $\sqrt{1}, ..., \sqrt{25}$ into two heaps differing as little as possible.

What changes are needed in `subsetSum.py`? In this version, $G = 42.81689013753907$. Our program reports `minW` = 1.735072416E-06, for the binary 01001110111101110100.

Let's try to quantify the roundoff error in this computation. First, consider the error introduced by one rounding operation, i.e. replacing a real number x by the closest binary fraction with a k-bit mantissa.

For any integer j, there are $2^{k-1} + 1$ binary fractions with a k-bit mantissa in the interval $[2^{j-1}, 2^j]$. They divide the range into subintervals of length 2^{j-k}. Hence, if j is the integer such that $2^{j-1} < x \leq 2^j$, then there is a binary fraction with a k-bit mantissa at a distance $\leq \frac{1}{2}2^{j-k} < 2^{-k}|x|$ from it. Thus, rounding x changes it in the worst case by less than $2^{-k}|x|$.

If we take into consideration that $|x|$ will in general not be exactly in the middle of two k-bit binary fractions, and that x could be almost twice 2^{j-1}, we see that $2^{-(k+1)}|x|$ is, on average, a generous estimate of the rounding error.

In this version of Listing 5.37, we modify `minW` by additions and subtractions a total of 2^{19} times. The intention is to get a sum close to G (42.81689013753907) which we'll approximate as 40. Since floats in Python are represented with a mantissa of $k = 52$ binary digits, the average roundoff in each operation is roughly $\epsilon = 40 \times 2^{-53} \approx$ 4.4E-15.

Each roundoff error is added to the sum of the previous roundoff errors, so we can obtain a rough estimate of the probable total from the theory of random

walks. Suppose we take n steps of length ϵ, moving randomly left or right each time. One can show that the expected value of the square of the final distance from the starting point is $\epsilon^2 n$. We don't have such a simple formula for the expected value of the magnitude of the distance itself but $\epsilon\sqrt{n}$, is a reasonable guess for our purposes. If we replace ϵ by the average error in one rounding, we get that the magnitude of the error at the end of our computation is probably around $40 \times 2^{19/2} \times 2^{-53} \approx 3.2\text{E-}12$.

Our program reports `minW` = 1.735072416E-6, for the word 01001110111101110100. If we calculate `minW` ourselves, using the square roots in the binary word, plus the integer square roots, then the actual minimum is 1.7350750346E-6, so the actual roundoff error is 2.6E-12. This suggests that we can be confident, although not absolutely certain, that we have found two heap weights which differ as little as possible.

We could reduce the roundoff errors by using variables of type decimal instead of float, but this would slow down the program. Another change would be to make certain that we don't miss solutions by printing out not only the subsets which reduce `minW` but also those which come close, within some range based on our rounding error calculations.