Divide and conquer and dynamic programming

Divide and conquer is the algorithmic version of recursion. The term comes from the political doctrine *divide et impera*, but for algorithms, a more correct description would be *divide and combine*. The key idea is to

1. Split a problem into subproblems of the same type.
2. Recursively solve these problems.
3. Combine the results of the recursive calls to a solution of the original problem.

Step one and three can be very simple or complex, while step two is usually one or two recursive calls.

The binary search algorithm that we have seen several times by now is an example of a divide and conquer algorithm. Step one in the algorithm is identifying whether we should search to the left or to the right of the midpoint, the recursive step (step two) is searching in one of these intervals. Step three is almost non-existing since we just return the result of the recursive solution.

The recursive step(s) in divide and conquer algorithms are often implemented as recursive function calls, but need not be. Conceptually, we recurse, but as we saw in binary search, we can replace recursive calls with loops. It is not necessary to use recursion in your implementation of a divide and conquer algorithm; the defining component of this class of algorithms is that we solve a subproblem of the same type as the original problem. Since we are using recursion, even if it is only conceptually, we need to have basis cases and recursive cases. The basis case in binary search is when we have an empty interval or when the midpoint is the element we are looking for. The recursive case handles everything else.

Merge sort

As another example of divide and conquer, we can consider a sorting algorithm known as *merge sort*. This algorithm works as follows:

1. Split the initial input into two pieces of half the size of the original problem: the first and the second half of the input list.
2. Sort these two smaller lists recursively.
3. Combine the two sorted lists using merge.

The algorithm involves two recursive subproblems, so it is not easy to implement it as an iterative solution. We will, therefore, deal with it recursively. The basis cases for the recursion are when we have empty lists or lists of length one—these will be lists that are already sorted. The recursive case handles everything else.

A straightforward implementation of this could look as follows:

```
def merge_sort(x):
    if len(x) <= 1: return x
```

```
    mid = len(x) // 2
    return merge(merge_sort(x[:mid]), merge_sort(x[mid:]))
```

The function performs what we identified as the three steps of merge sort should in the most straightforward manner, but you might be uncomfortable with the slicing we do to split the input `x`. For `merge`, as we saw in sec. **??** using this form of slicing increased the running time from $O(n)$ to $O(n^2)$. It is not quite as bad in this algorithm since the linear time slice operation is slower than the time it takes to sort the sub-lists—we discuss the running time shortly, but it will be $O(n \log n)$. Still, we could avoid it by using indices into `x` instead:

```
def merge_sort_rec(x, low, high):
    if high - low <= 1: return x[low:high]
    mid = (low + high) // 2
    return merge(merge_sort_rec(x, low, mid),
                 merge_sort_rec(x, mid, high))

def merge_sort(x):
    return merge_sort_rec(x, 0, len(x))
```

I have implemented this using two separate functions, one that handles the actual sorting but takes indices as arguments, and one that only takes `x` as its argument and calls the former. We cannot set `low` and `high` as default arguments, since `high` should be set to the length of `x`, and we do not know what `x` is until we call the function (recall that the default arguments of a function must be known when we *define* the function, not when we call it). We could use the trick of setting them to `None` and then check if they are that, as we did `merge`, but then we would need to check the arguments in each recursive call. This wasn't a problem when we translated `merge` into an iterative algorithm, but we cannot do this with `merge_sort` since it is not tail recursive. Therefore, I prefer to split the algorithm into two functions.

Quick sort

Consider another famous divide and conquer sorting algorithm: *quick sort*. This algorithm picks one of the elements in its input, called the *pivot*, and then it splits the data into three sets, those elements that are less than the pivot, those that are equal to the pivot, and those elements that are greater than pivot. Naturally, the first set should go before the middel who should go before the last set in a sorted sequence, so if we sort the first set separately from the last set and then concatenate all three of them, then we have a sorted sequence.

The description is longer than a simple implementation:

```
def qsort(xs):
    if len(xs) < 2: return xs
    p = pick_pivot(xs)
```

```
    first = qsort_lc([x for x in xs if x < p])
    middle = [x for x in xs if x == p]
    last = qsort_lc([x for x in xs if x > p])

    return first + middle + last
```

Quick sort is called quick because picking the pivot and partitioning the data can be done, well, quickly. The list comprehension version above does follow the spirit of quick sort but creating new lists and filter them is not fast. And we haven't gotten to how to pick the pivot. For the latter, I will just pick the first element in the sequence we are to sort. This has some consequences for the expected running time but we return to that in the next section.

We will not split the sequence into three sets but two. It is slightly harder to split it into those less than the pivot, those equal to the pivot, and those greater than the pivot, compared to splitting into less than or equal to the pivot and greater than the pivot. I didn't do this above, in the list comprehension function, because I could risk in infinite recursion.

Exercise: Show how this function could end up recursing forever.

```
def qsort(xs):
    if len(xs) < 2: return x
    p = pick_pivot(xs)
    return qsort([x for x in xs if x <= p]) +
           qsort([x for x in xs if x > p])
```

To avoid infinite recursion we must make sure that at least one element is left out of the recursion in each step. We handle that by removing *one* element equal to the pivot instead of all of them.

We will use a function that satisfy this invariant

```
def partition(x, i, j):
    """
    Let pivot = x[i]. This function will
    arrange x[i:j] into x'[i:k] + x'[k:j] such that
    x'[h] <= pivot for h = i ... k and
    x'[k] == pivot and
    x'[h] > pivot for h = k + 1 ... j
    and then returns this k
    """
    ...
```

Preferably one that is fast. With this function, quick sort is reduced to this:

```
def qsort_rec(x, i, j):
    if j - i <= 1: return
    k = partition(x, i, j)
```

```
    qsort_rec(x, i, k)
    qsort_rec(x, k + 1, j)

def qsort(x):
    qsort_rec(x, 0, len(x))
```

After partitioning, an element equal to the pivot is at index k, and all elements before index k should go before k and all elements after index k should go after k. We leave out index k from the recursion—the first recursion is over $x[i:k]$ (i is included; k is not), and the second recursion is over $x[k+1:j]$. This means that we have to require of the `partition` function that `x[k]` is the pivot after calling the function.

The meat of the algorithm is in the `partition` function. The partition algorithm works as follows: we have an interval i to j of sequence x, i included and j not, and the pivot is at index i. We use two additional indices, k and h. We set k to $i+1$ and h to $j-1$. That is, k points one past the pivot and h points to the last element in $x[i:j]$, see fig. 1.

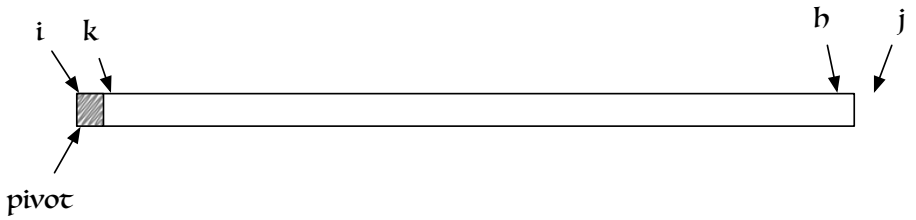

Figure 1: Initial setup for 'partition'. We set k to point one after i and h to point one before j.

We will have the invariant that all elements in $x[i:k]$ are less than or equal to the pivot and all elements in $x[h+1,j]$ are greater than the pivot (in both intervals, as always, we include the first index and exclude the second).

In each iteration we look at $x[k]$ to check if it is greater than the pivot or not. If it is less than or equal to the pivot we can increment k by one and still satisfy the invariant, fig. 2. If it is greater than the pivot we cannot increment k but we can move it to the end of the sequence where we keep those elements greater than the pivot. We swap it with $x[h]$ and decrement h by one, fig. 3.

The original $x[h]$ may or may not have been greater than the pivot; the invariant only tells us that elements in $x[h+1:j]$ are. This is irrelevant for the algorithm, however. If $x[h]$ was less than or equal to the pivot, then we would simply increment k in the next iteration. If it was greater than the pivot, then it would get swapped back in the next iteration. It might get swapped twice, but it is not worse than that. If swapping is very expensive, and you don't mind a slightly more complicated algorithm, you can decrease h until it points to an element

that is not greater than the pivot after each swap. The extra loop is hardly worth it for avoiding one swap, but you can do it.

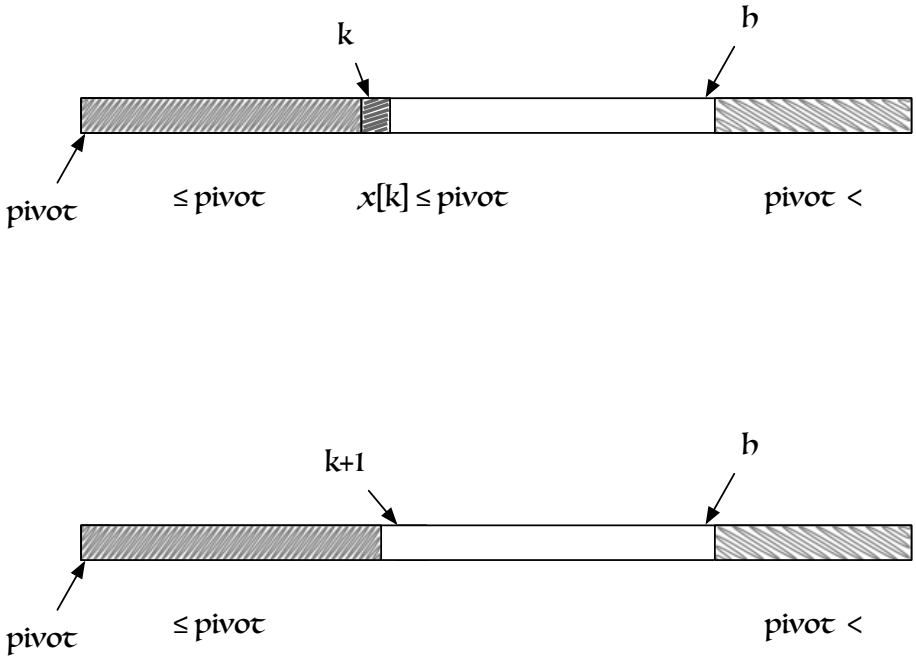

Figure 2: If 'x[k]' is less than the pivot we can increase k and maintain the invariant.

When h has moved past k we have partitioned the sequence such that all elements in $x[i:k]$ are less than or equal to the pivot and all elements in $x[h+1,j]$ are greater than the pivot, fig. 4. The index we want the function to return is the last element in the part that is less than the pivot, not one past it, so we will return $k-1$ rather than k. Sorry for the confusion. For reasons explained above, we need the element at that position to be equal to the pivot. We know that $x[k-1]$ belongs in the first part, and it could be anywhere in there for all we care. The pivot at $x[i]$ also belongs in the first part, and since we want the pivot at index $k-1$ we just swap the two. Then all is well and we are done.

An implementation could look like this:

```
def partition(x, i, j):
    pivot = x[i]
    k, h = i + 1, j - 1
    while k <= h:
        if x[k] <= pivot:
            k += 1
        elif x[k] > pivot:
```

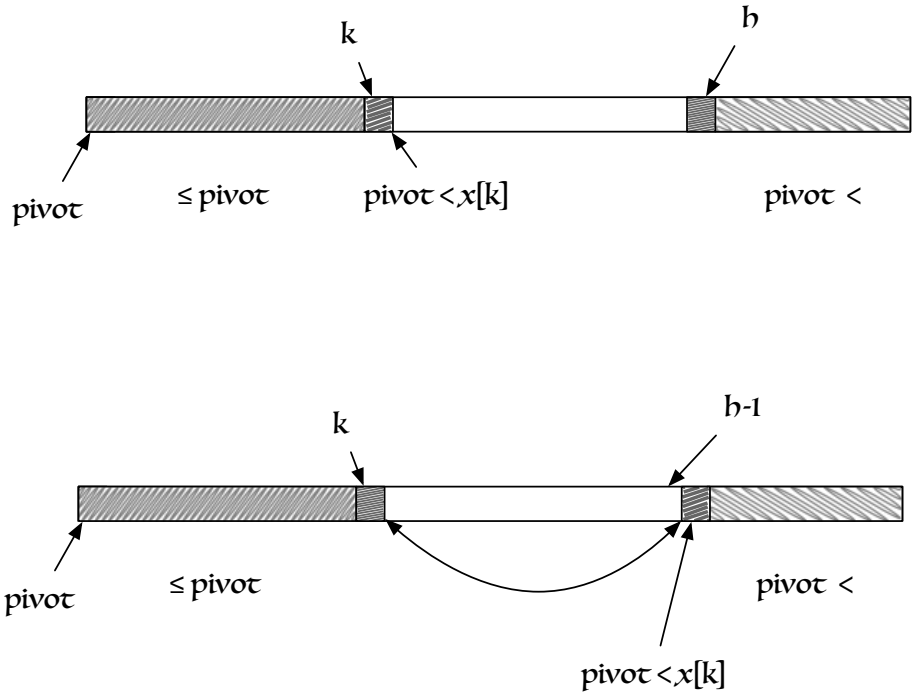

Figure 3: If 'x[k]' is greather than the pivot we cannot increase k and maintain the invariant but we can switch 'x[k]' and 'x[h-1]' and then decrease h. This will maintain the invariant.

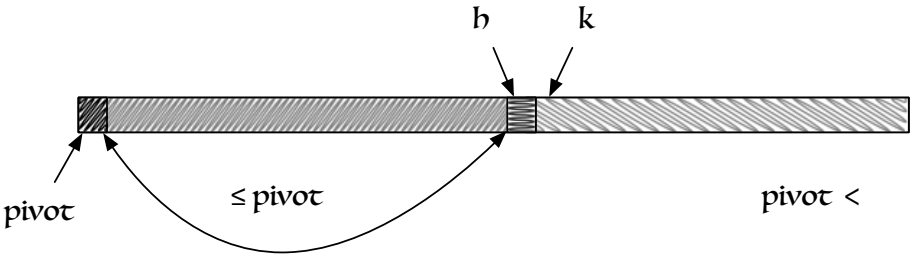

Figure 4: When we are done with the paritioning we need to put the pivot at the last position of the first part. We can do this simply by swapping the first and the k'th element. We need to return the last position in the first part and that is $k - 1$.

```
            x[k], x[h] = x[h], x[k]
            h -= 1

    x[i], x[k - 1] = x[k - 1], x[i]
    return k - 1
```

All the operations in `partition` are simple comparisons and swaps so it is fast, quick one might say, and it is that which makes quick sort a preferred algorithm in many usages.

Exercise: Prove that the partition algorithm is correct and that it runs in time $O(n)$.

Exercise: What is the best case and the worst case running time for quick sort?

Exercise: Is quick sort in place? Is it stable?

If the data is almost random then quick sort will spend linear time to split it in two almost equally sized parts and then recurse, giving us the time recurrence equation $T(n) = 2 \cdot T(n/2) + O(n)$, which, see case 1 below, is $O(n \log n)$. It might also pick the largest value as the pivot each time which will partition the data into two parts where the first contains all the elements. We take care always to remove one element before we call the recursion, but we could recurse on $n - 1$ elements, taking us to case 3 instead.

A common case for data we need to sort is that it is *almost* sorted to begin with. This happens, for example, when we are manipulating data that is sorted to begin with but only slightly sorted after our fiddling with it.

If we always pick the first element then almost sorted data means that we partition the input very unevenly in each recursion. Quick soft will run in $O(n^2)$ (case 3) instead of $O(n \log n)$ (case 1). Insertion sort, on the other hand will run in time $O(n)$ on almost-sorted data (but $O(n^2)$ on random data). A strategy for a better sorting algorithm could be to let quick sort handle the data while it is far from sorted—assuming that it is random to begin with—but then switch to insertion sort once it is close to sorted. Once quick sort has reached some minimal sequence length it can stop recursing. If all the recursions stop at that depth then the sequence will consist of small segments of randomly ordered data but each segment contains elements that are larger than the previous and smaller than the next. So the entire sequence is almost sorted. We can throw insertion sort at that sequence and expect the $O(n)$ best-case performance.

Exercise: Implement the `qsort_rec` function below to get the hybrid algorithm.

```
def qsort_rec(x, i, j, threshold):
    # Implement this

def qsort(x, threshold = 1):
    qsort_rec(x, 0, len(x), threshold)
```

```
def hybrid(x, threshold):
    qsort_rec(x, 0, len(x), threshold)
    insertion_sort(x)
```

Evaluate the hybrid algorithm empirically and compare it to basic quick sort and insertion sort algorithms. Explore how it performs for different choices of `threshold`.

An adaptive threshold would be better; if the data is already close to sorted at the beginning of the algorithm we shouldn't wait until we are deep in the recursion before we switch to insertion sort.

Exercise: Consider ways to determine if a sequence is close to sorted—for some measure of "close"—and consider how you would use it to adaptively switch to insertion sort.

Picking the first element for the pivot is a problem with almost sorted data. Picking the last element is a problem with almost reversed sorted data. Either will work well if the data is a random string. It doesn't matter which strategy we have for picking a pivot as long as it is deterministic; it is always possible to construct data that gives us the worst case behaviour. So called *randomised* algorithms adds stochastic choices to an algorithm in order to ignore what the data looks like and consider the runtime of the algorithm a random variable. If we pick a random element for our pivot in each call then we get the optimal average case performance regardless of the input.

However, picking random numbers is *not* a fast operation. Randomising the algorithm this way would defeat the purpose of having quick operations in each recursive call. A compromise often made is to pick the pivot deterministically but looking at more than one element in the input sequence before making the choice. If we look at three, say, and pick the median of these we are less likely to get an uneven partition.

Another worst case situation is if there are many equal elements. Then it doesn't help us to pick a random pivot. If we can split the sequence into three parts, as the list comprehension version we saw when we introduced quick sort, then having a sequence with only one value becomes a best case instead of worst case. When we partition we end up with one long middle part that we do not need to sort. It is possible to modify `partition` to do this and still be very efficient.

Divide and conquer running times

Figuring out the running time for recursive functions—or algorithms that are recursive even if they are not implemented as recursive functions—means solving recurrence equations. If $T(n)$ denotes the running time on an input of size n, then a recurrence equation could look like this:

$$T(n) = 2 \cdot T(n/2) + O(n)$$

This is the recurrence equation for merge sort and working it out will tell us what its worst case time complexity is. To sort a list of length n we solve two problems of half the size, $2T(n/2)$, and do some additional work in time $O(n)$. In the first version of merge sort, where we sliced the input, we used linear time both for the slicing and then for the merge; in the final version, we only spend linear time doing the merge. In either case, we spend linear time in addition to the recursive calls.

What characterises recurrence equations is similar to what defines recursive solutions to problems. The equations refer to themselves. Strictly speaking, we need basis cases for the recurrence equations to be well defined so we would have

$$T(n) = \begin{cases} O(1) & n \leq 1 \\ 2 \cdot T(n/2) + O(n) & \text{otherwise} \end{cases}$$

but when we consider the running time of an algorithm, the basis cases almost always involve constant time, so we often leave that out.

You can solve such recurrence equations by expanding them:

$$\begin{aligned} T(n) &= O(n) + 2 \cdot T(n/2) \\ &= O(n) + 2\left[O(n/2) + 2 \cdot T(n/4)\right] \\ &= O(n) + 2\left[O(n/2) + 2 \cdot [O(n/4) + T(n/8)]\right] \\ &= \ldots \end{aligned}$$

You have to be a little careful with the expansion and big-Oh notation, here. We see that we get an $O(n/2)$ in the first expansion, and normally we would consider this equal to $O(n)$. It is, but as we keep expanding, we see the series $O(n) + O(n/2) + O(n/4) + \cdot + O(n/n)$. If we pretend that all the $O(n/2^k)$ components are $O(n)$ this would give us $n \times O(n) = O(n^2)$. This *is* an upper bound on the expression, but it is not really tight. If you multiply into the parentheses in the equation, you will also get

$$2\left[O(n/2) + 2 \cdot T(n/4)\right] = 2O(n/2) + 4T(n/4) = O(n) + 4T(n/4)$$

which is actually okay, but if you translated $O(n/2)$ into $O(n)$, you would get

$$T(n) = 2O(n) + 4O(n) + 8O(n) + 16 \cdot T(n/8)$$

where each of the $O(n)$ components are multiplied by a number 2^k. You can consider that a constant in each step, but k depends on n, so it isn't really a constant. Neither is the number we divide n by inside the big-Oh.

The problem here is that we blindly translate the expanded numbers $2O(n/2)$ into $O(n)$ are do not take into account that, as we continue the expansion, the number we multiply with and the number we divide by, changes for each expansion. They depend on n in how many times we do this and what the numbers are in each step. They are not constants. The arithmetic rules we have learned for the big-Oh notation are correct; the problem is not that. The problem is that we consider the numbers in the expansion as constants when they are not. This is usually not a trap you will fall into when reasoning about iterative algorithms as we have done earlier, but it is easy to fall into here.

When expanding a recurrence equation, it is easier to translate $O(n)$ into cn for some constant c. We know such a constant exists such that cn is an upper bound for whatever the $O(n)$ is capturing. Then we get an expansion like this:

$$\begin{aligned} T(n) &= cn + 2 \cdot T(n/2) \\ &= cn + 2\left[cn/2 + 2 \cdot T(n/4)\right] \\ &= cn + 2\left[cn/2 + 2\left[cn/4) + 2 \cdot T(n/8)\right]\right] \\ &= \ldots \end{aligned}$$

If we take this expansion all the way down, we get

$$T(n) = \sum_{k=0}^{\log n} c \cdot 2^k n/2^k = \sum_{k=0}^{\log n} c \cdot n = c \cdot n \log n$$

where $\log n$ is the base-two logarithm, and we get that limit because $2^{\log n} = n$ is when we reach $n/n = 1$. This means that this recurrence is in $O(n \log n)$, so this is the big-Oh running time for merge sort.

We can conclude that the running time for merge sort is $O(n \log n)$.

What about quick sort? In the worst case the recursive calls will go n deep; in each recursion it can partition the sequence into one and the rest and thus recurse on $n - 1$ elements. The partitioning is fast compared to merging but the asymptotic worst-case performance suffers from it. To get the same complexity as merge sort we must have the recurrence $T(n) = 2T(n/2) + O(n)$. This happens when the partitioning gives us two parts that are roughly the same size. In the worst case, however, one has size n and the other is empty. We make sure to always reduce the sequence length by one by not including the pivot at index k but we still might end up by a recursive call on a sequence of length $n - 1$. In that case the depth of the call stack is linear and since we do linear work at each level we end up with a $O(n^2)$ time algorithm (see case 3 below).

If the sequence is random then each element is expected to be roughly in the middle of the numbers and then we expect that the partition will give us two equally sized parts. So the *average* case running time is $O(n \log n)$—and it is fast when that happens. We cannot assume that we hit the average case, though, and with almost sorted data picking the first element as the pivot is a particularly bad idea. There we will get the $O(n^2)$ running time. A hybrid algorithm, as in the earlier exercise, can give us the best of both worlds.

Frequently occurring recurrences and their running times

There are common cases of recurrence equations that almost all divide and conquer algorithms match. If you memorise these you will know the running time of most divide and conquer algorithms you run into. If that fails you have to get to work and derive the running time yourself. This usually involve expanding the recurrence until you have a series you recognise, but quite often I find that a wisely chosen drawing works as well as arithmetic. See below for examples.

Case 1:

$$T(n) = 2 \cdot T(n/2) + O(n) \in O(n \log n)$$

Merge sort and quick sort are examples of this recurrence. Whenever you can break a problem into two subproblems of half the length as the original, and you do not spend more than linear time splitting or combining, you get a $O(n \log n)$ algorithm. Merge sort can be shown to be optimal (in the sense of big-Oh) because all comparison-based sorting algorithms need to do $\Omega(n \log n)$ comparisons. Quick sort is only expected $O(n \log n)$ time.

When we deriving the running time from the recurrence equation we did a bit of arithmetic. Usually, you can solve recurrence equations just by expanding them and recognising the form of the sum you get back from it. This often takes the form of a series, and if you know what it converges to, you are done. Doing it this way doesn't give you much intuition about why the running time is at it is, though, but a drawing of how it works can.

Consider the $T(n) = 2T(n/2) + O(n)$ recurrence and fig. 5. What the figure shows is that we do linear work first, shown as the grey bar. Then we recurse on two halves and do linear work there as well, but on half the size. In total, though, it is linear work; it is just split in two different calls. At the next time we also do linear work, but now in four different calls. We continue like this until we hit base cases where to total work is linear again. The depth of the recursion, since we split the size of the data the recursive functions work on is $O(\log n)$ so summing it all up gives us $O(n \log n)$. The drawing tells us exactly the same as the arithmetic—which is reassuring—but I think the drawing is easier to follow and gives more intuition about why the recurrence gives us the time we got.

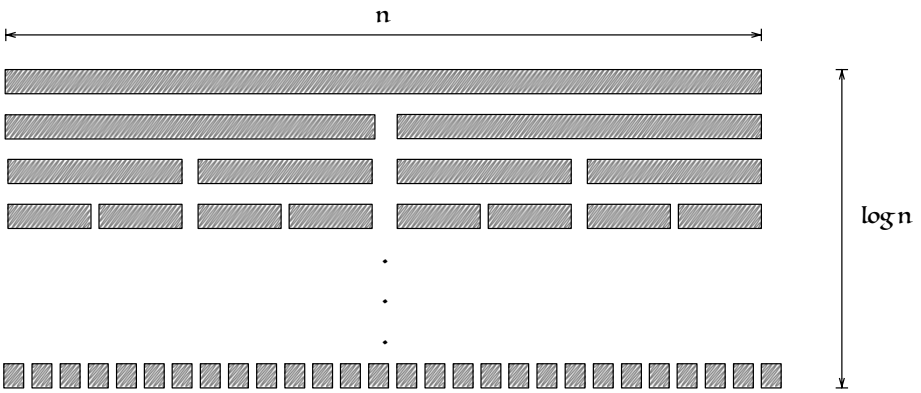

Figure 5: Work when we recurse on twice the half size and do linear work at each step.

Case 2:

$$T(n) = T(n-1) + O(1) \in O(n)$$

If we, in constant time, remove one from the problem size we have an algorithm that runs in linear time. If we consider linear search a recursive problem—the basis case is when we find the element, or we are at the end of the list, and the recursive case is doing a linear search on the rest of the input—then that would be an example of such an algorithm.

You can see a drawing that illustrates the running time in fig. 6, although drawing this is almost overkill. You can see that we read a linear call stack depth and if each call costs us constant time work we must end up with $O(n)$. Another way to see this is to consider the workload at each level (the grey blocks in the figure). The running time will be the sum of all these. If we project all of them to the bottom of the figure we see that we get n of them. That is certainly overkill for this recurrence but it is at times a helpful way to reason about the running time.

Case 3:

$$T(n) = T(n-1) + O(n) \in O(n^2)$$

If we need linear time to reduce the problem size by one, then we get a quadratic time algorithm. Selection sort, where we find the smallest element in the input, in linear time, swap it to the first element, and then recursively sort the rest of the list is an example of this.

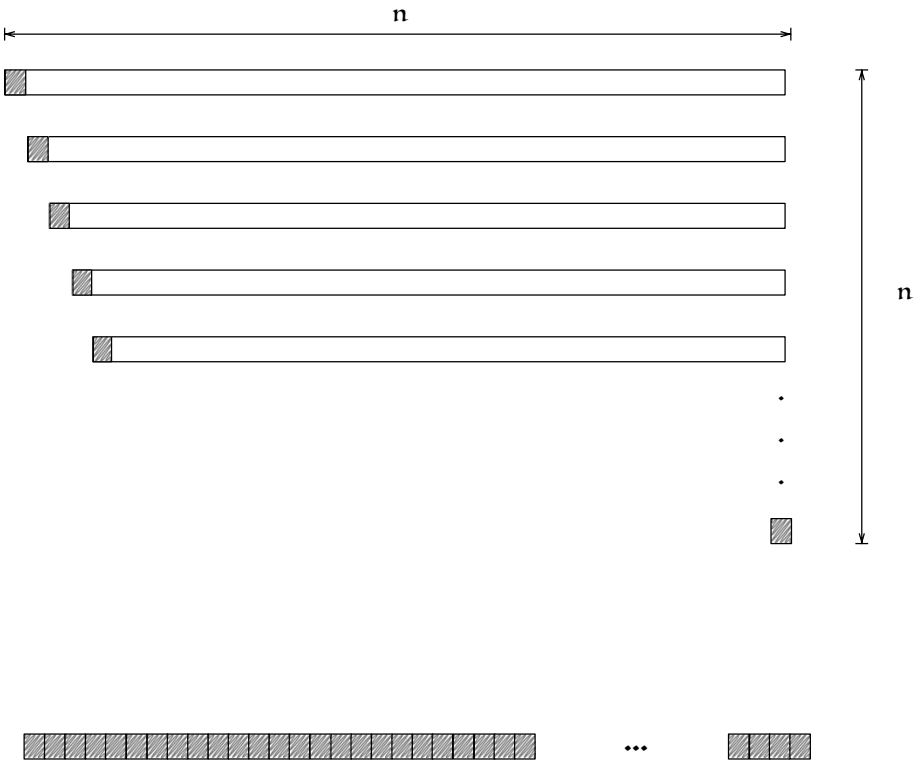

Figure 6: The work done if we only do constant work per call but we recurse on data that is only one smaller than the original.

For a proof of the running time consider fig. 7. If we have an $n \times n$ square its area is n^2. If we do linear work per level and remove one "work block" at each level we get the area in the figure. It is half the area of an $n \times n$ block so it is $n^2/2 \in O(n^2)$. It is actually $n(n+1)/2$ if you are more careful with the math—we cannot quite move down the diagonal and there is a little bit too much at each level. But $n(n+1)/2$ is still in $O(n^2)$ so the figure is not lying.

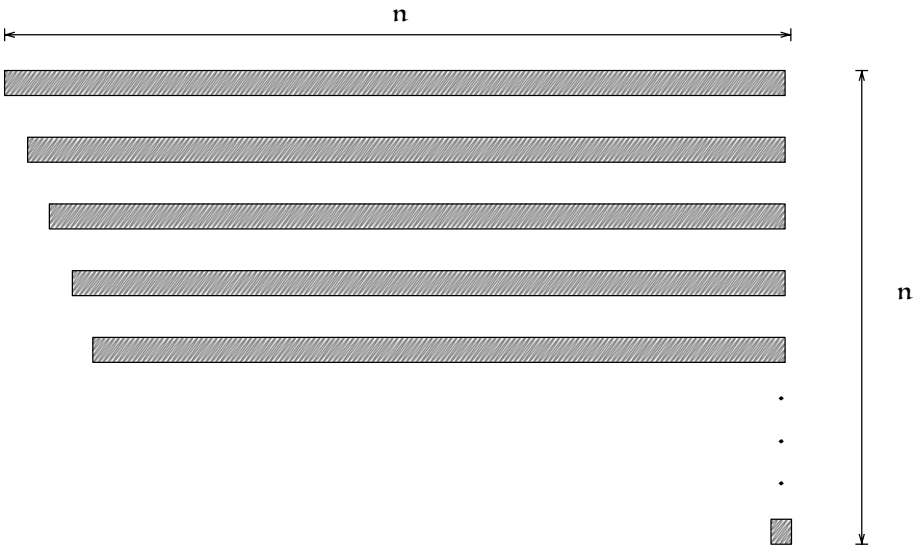

Figure 7: Work if we reduce the size by one in each iteration but do linear work.

Case 4:

$$T(n) = T(n/2) + O(1) \in O(\log n)$$

If you can reduce the problem to half its size in constant time, then you have a linear time algorithm. Binary search is an example of this.

Figure 8 is one way to illustrate why the running time is as it is. Here, again, I have shown the data and the work done in the recursive calls as white and grey bars, but I have drawn them along the x-axis and right-to-left. The rightmost chunk input has size n, to the immediate left if that we have the recursive call with data of size $n/2$, and so forth until we have the base case, of size 1, at the leftmost position.

The reason I have drawn it this way is that I can now read it from left to right. Every time I reach a grey square I put one above it such that the grey bars moving up along the y-axis are equidistance. Notice that each time I have added a grey square I have to move twice as long until I reach the next one than the

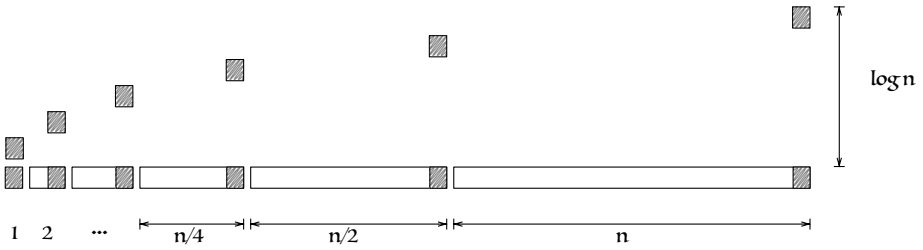

Figure 8: Work if we cut the problem size in half at each level and do one piece of work.

length I had to move from the previous to this one. Each time we double the size of the data on the x-axis we add one to the y-axis. This is pretty much the definition of the log base two function.

Case 5:

$$T(n) = T(n/2) + O(n) \in O(n)$$

If you can reduce the problem to half its size in linear time, and get the solution for the full problem after the recursive computation in linear time as well, then you have a linear time algorithm. Notice that this equation is different from the one we had for merge sort; in that recursion, we needed *two* recursive calls, in this we only need one.

As an example of this running time, consider a function that adds a list of numbers by first adding them pairwise and then adding all the pairwise sums in a recursive call:

```
def binary_sum(x):
    if len(x) == 0: return 0
    if len(x) == 1: return x[0]

    # O(n) work
    y = []
    for i in range(1, len(x), 2):
        y.append(x[i - 1] + x[i])
    if i + 1 < len(x): # handle odd lengths
        y[0] += x[i + 1]

    return binary_sum(y) # recurse on n//2 size
```

This looks like just a complicated way of adding numbers—and in some ways it is—but it can be relevant if you need to compute the sum of floating point

numbers (see the last section of this chapter). When you add two floating point numbers, you might not end up with their sum. If this surprises you, remember that a number must be represented in finite computer memory, but not all real numbers can. For example, the decimal representation of $1/3$ requires infinitely many decimals, $0.333333\ldots$. In your computer, you do not use decimal notation but binary, but the problem is the same. (You can, of course, represent rational numbers like $1/3$ as two integers, and you can represent arbitrarily large integers, but floating point numbers are much faster for the computer to work with).

If you add two floating point numbers, you will then lose bits of information proportional to how many orders of magnitude the numbers are apart, so the greater the difference, the less precise your addition is. If you add the numbers in a long list, starting from the left and moving right, as we have done many times before, then this can become a problem. The accumulator will grow and grow as we add more and more numbers, and if the numbers in the list are of roughly the same order of magnitude, the accumulator might end up many orders of magnitude larger than the next number to be added. If the difference gets large enough, adding a number to the accumulator results in just the value of the accumulator.

If you start with numbers of the same order of magnitude, then adding them pairwise as in the algorithm above, will keep them at roughly the same order of magnitude in the recursion, and this will alleviate the problem of losing precision in floating point numbers.

A classical algorithm, *k-select*, is an adaption of quick sort that has this recurrence as an expected case running time. As input it takes a a sequence x and an index i and it returns the value $x'[i]$ where x' is x sorted. If we sorted x, in $O(n \log n)$, and then returned $x'[i]$, in $O(1)$ we would have an $O(n \log n)$ algorithm. We can do better and get an $O(n)$ algorithm by not sorting x completely. We will partition the sequence exactly as we did in quick sort. This take time $O(n)$. Then, if the partitioning index k is larger than i we know that $x'[i]$ must be in the first part. If not, it must be in the second part. In the first case, we call recursively on the first part; in the second we call recursively on the second part but adjust the index to reflect that we have eliminated $k + 1$ elements from the left part.

Exercise: Modify your quick sort implementation to implement k-select.

Figure 9 shows how you can think about the running time. At each level we do linear work but we only take half the data with us in the recursion. The grey areas show the work we do at each level. I have put the recursion on the first half to the left at level two. Then the quarter work is to the right of it; the eights to right of that and so on. If we take all the pieces except the first n down to the bottom of the figure and lie them next to each other they add up to n. So the first n and the n from the rest of the calls sum to $2ninO(n)$.

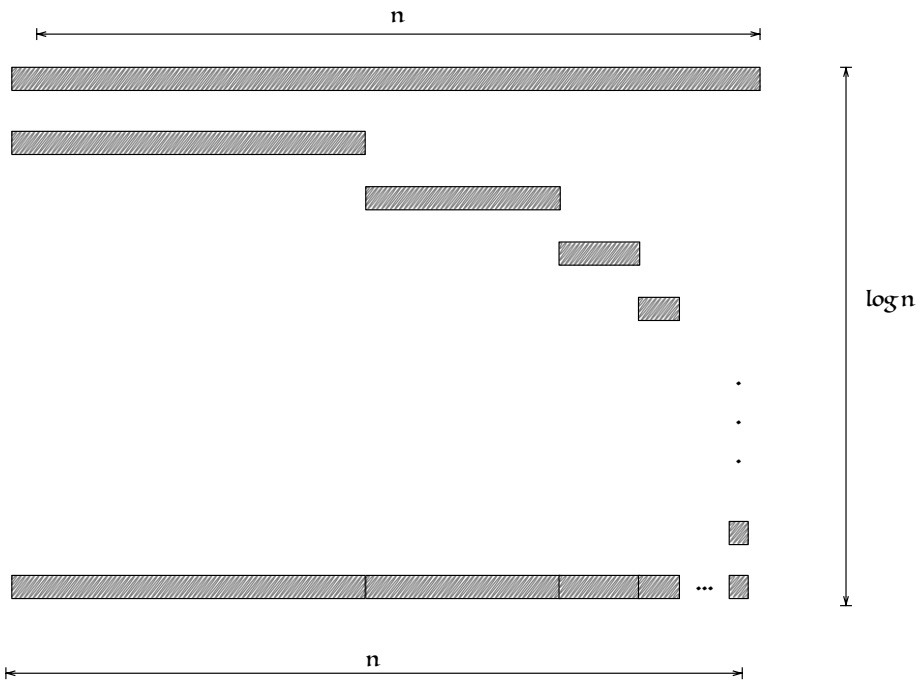

Figure 9: Work if we do linear time per level and then only take half the problem size with us to the next level.

Case 6:

$$T(n) = 2 \cdot T(n/2) + O(1) \in O(n)$$

If we can split and combine in constant time but require two recursive calls on half the size, we also get a linear time algorithm. Again, notice that this is different from the recurrence equation for merge sort where we needed linear time to merge the results of the two recursive calls.

An example of an algorithm that has this recurrence consider the problem of finding the largest difference between any two numbers in a list. You could, of course, run through all pairs and find the largest, but that would take time $O(n^2)$. Instead, you can solve a slightly harder problem and get the solution from there—whereby "harder" I mean that it solves more than just finding the maximum difference; I do not mean that this is a harder complexity class. We can actually solve this problem in linear time using a divide and conquer algorithm with the recurrence above.

The problem we will solve is to find the smallest and the largest number in a list as well as the largest difference. We do this recursively. We split the input into two halves, find the smallest and largest elements in both halves, as well as the largest difference in the two halves. We can then combine these. The smallest element for the full list is the smallest of the smallest in either half. The largest element is the maximum of the two we got from the recursive calls. The largest difference in the full list is either found in one of the two halves, or we can get it as the difference between the largest element in one half and the smallest in the other. We can implement the entire algorithm like this:

```
def min_max_maxdiff(x, i, j):
    # Invariant: j > i
    if i + 1 == j:
        return x[i], x[i], 0
    else:
        mid = (i + j) // 2
        min_l, max_l, maxdiff_l = min_max_maxdiff(x, i, mid)
        min_r, max_r, maxdiff_r = min_max_maxdiff(x, mid, j)
        min_res = min(min_l, min_r)
        max_res = max(max_l, max_r)
        maxdiff_res = max(maxdiff_l, maxdiff_r,
                          max_r - min_l, max_l - min_r)
        return min_res, max_res, maxdiff_res
```

If we only want the maximum difference, we can wrap the function:

```
def maxdiff(x):
    if len(x) == 0: return None
    _, _, md = min_max_maxdiff(x, 0, len(x))
    return md
```

Splitting the data and combining the results from the recursive calls can be done in constant time and we make two recursive calls of half the size, so by the recurrence equation above, the running time is $O(n)$.

It is a little harder to visualise why the running time is the way it is, but consider fig. 10. Here we show the work per level but this time I don't want you to add the grey blocks. I just want you to read the the figure from the bottom and up. At the bottom the work is n. At the second-to-last level it is $n/2$. It continues that way until we get to the top level where the work is 1. That is *exactly* the setting we saw in case 5. The setup is different and so are the figures but if you read this figure from the bottom and fig. 9 from the top you will see that the work per level is exactly the same. If case 5 is $O(n)$ then case 6 must be as well.

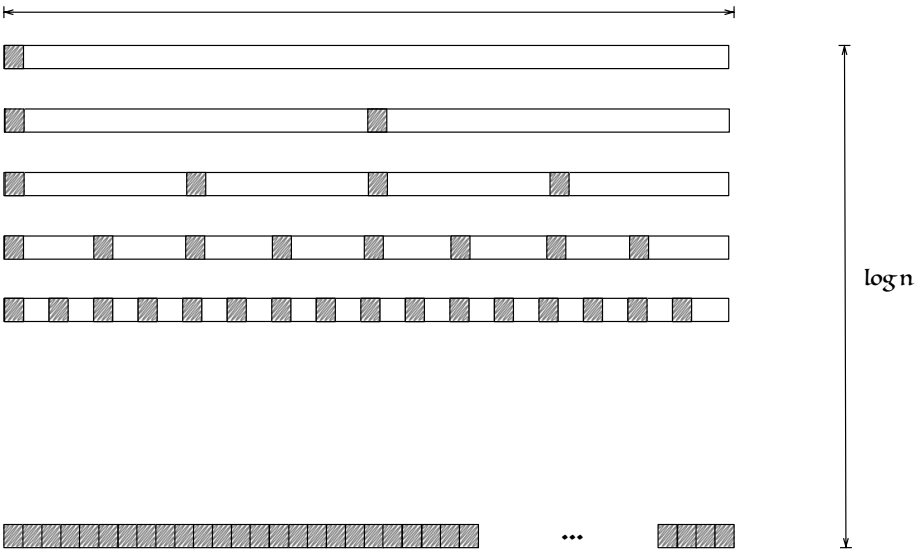

Figure 10: Work if we do constant work per recursion but do two recursive call on half the data each.

Dynamic programming

By now, we have seen how recursion is a powerful tool for both programming and algorithmic design, i.e., divide and conquer. But you will not be surprised to learn that not all recursive programs are efficient. For some recursive functions there is not much we can do about that but for others

Consider the recursive function for computing the n'th Fibonacci number.

$$F(n) = \begin{cases} 1 & n = 1 \text{ or } n = 0 \\ F(n-1) + F(n-2) & n > 1 \end{cases}$$

To compute $F(5)$ you must compute $F(4)$ and $F(3)$. To $F(4)$ you must compute $F(3)$ and $F(2)$, for $F(3)$ you must compute $F(2)$ and so on, see fig. 11 where I have marked the base cases with check-marks and the recursive cases with which recursive calls they make. The structure of the graph of recursive calls leads clearly leads to an explosion in the number of calls as n increases, see fig. 12. That makes this recursive calculation an unfeasible approach for computing $F(n)$ for large n.

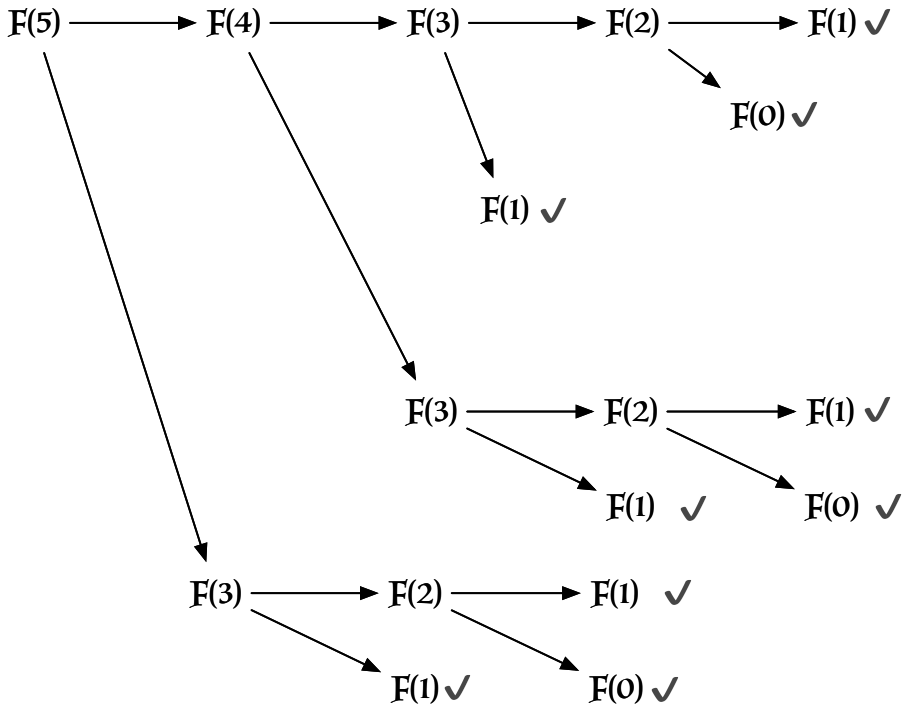

Figure 11: The graph of recursions necessary to compute the 5'th Fibonacci number.

If we could store the values of each function, illustrated infig. 13, we would get a linear time algorithm (fig. 14). You cannot determine that the algorithm's running time is linear just because the plot looks like a line but I leave it to you to prove it, or you can take my word for it. To go from an exponential to a linear algorithm like this only requires a table to store results in and to look up values in.

```
def fib(n):
    tbl = {}
    if n <= 1:
        return 1
    else:
```

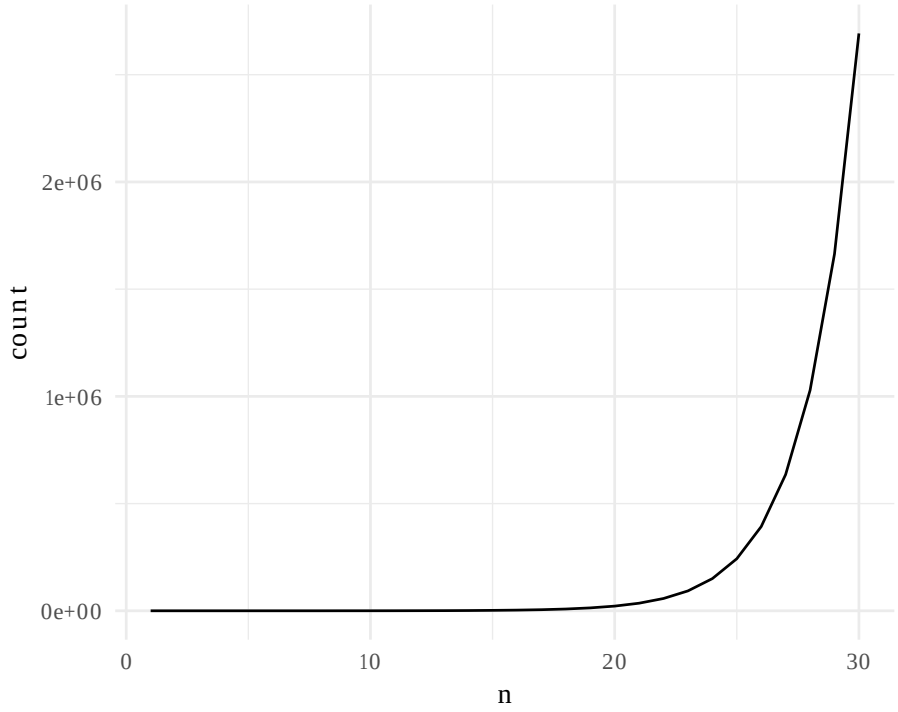

Figure 12: The number of recursive calls made when evaluating the Fibonacci function.

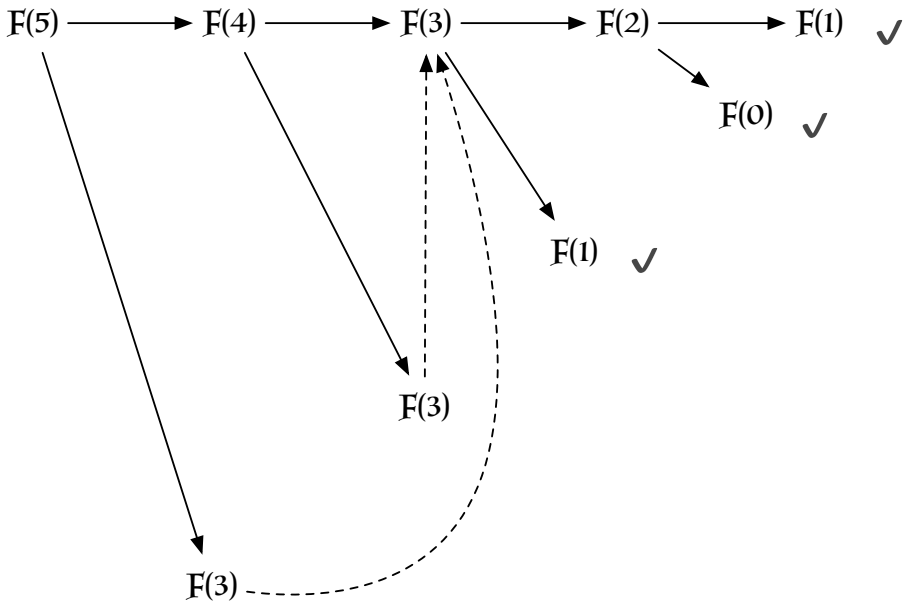

Figure 13: The recursivion graph for computing the 5'th Fibonacci number if we remember the problems we have already solved

```
    if n not in tbl:
        tbl[n] = fib(n - 1) + fib(n - 2)
    return tbl[n]
```

The $F(n)$ function is simple; we always know precisely which recursions we need and the sequence of recursions we need. This lets us turn the calculations around and start at the base cases and build $F(i) = F(i-1) + F(i-2)$ from 2 and up to n, see fig. 15.

We could build a table mapping the numbers $\{0, 1, 2, \ldots, n\}$ to the corresponding Fibonacci number, then fill the table starting from the smallest numbers and moving up. That way, whenever we need to compute $F(i)$ for some i, the values $F(i-1)$ and $F(i-2)$ are already in the table. This leads us to this algorithm that fills up a table and computes the Fibonacci numbers up to the one we want.

```
def fib(n):
    if n <= 1:
        return 1
    tbl = [1] * n
    for i in range(2,n):
        tbl[i] = tbl[i - 1] + tbl[i - 2]
    return tbl[n - 1] + tbl[n - 2]
```

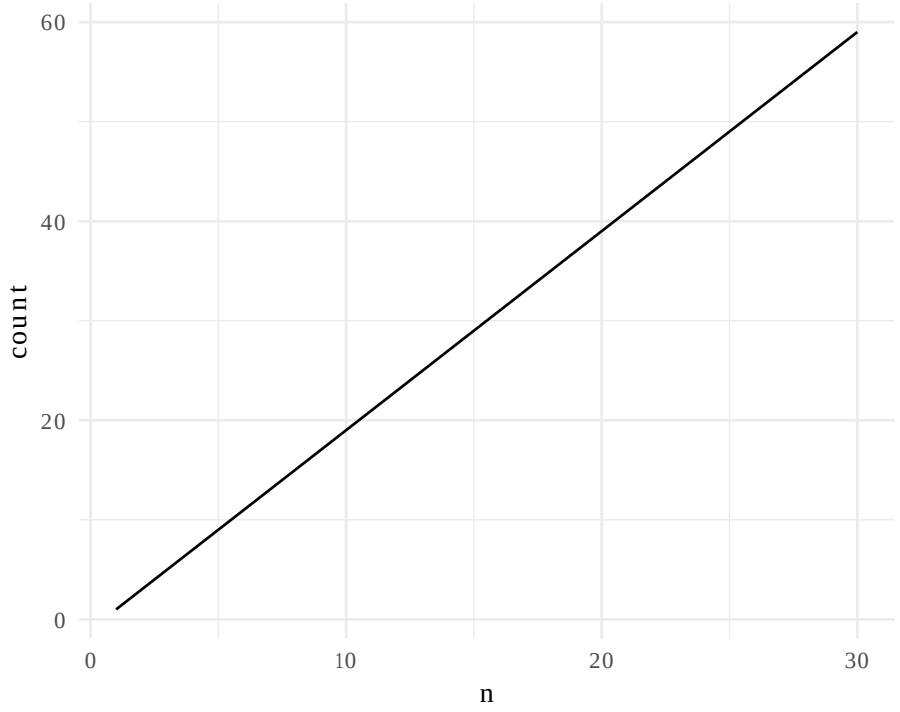

Figure 14: Number of recursions for increasing n.

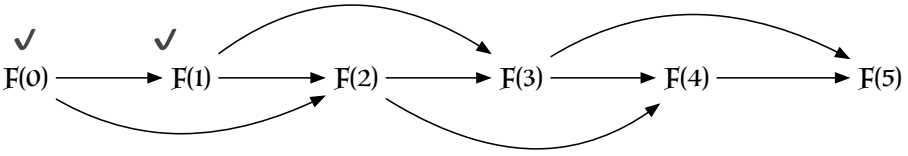

Figure 15: The graph for computing the 5'th Fibonacci number iteratively.

Of course, we only need the values $F(i-1)$ and $F(i-2)$; we do not need the full table when we compute $F(i)$ so we can save a bit of memory by only remembering the last two values. With two variables, representing $F(i-2)$ and $F(i-1)$, the iterative solution looks like this:

```
def fib(n):
    if n <= 1:
        return 1
    fi1, fi2 = 1, 1
    for i in range(n - 1):
        fi1, fi2 = fi1 + fi2, fi1
    return fi1
```

Dynamic programming is the straightforward idea that you should only calculate each value once. When we compute a value recursively and store results in a table, then we call it *memorisation* or *top-down* dynamic programming. When we build our way up from base cases to the one we want we call it *bottom-up* dynamic programming. I personally think that the term memorisation is more informative about what an implementation actually does so I will use memorisation rather than top-down dynamic programming. I will refer to button-up dynamic programming as dynamic programming.

For dynamic programming and memorisation to work you need a problem that you can split into subproblems that you can then solve independently. In that way, it matches the requirements for divide and conquer algorithms. For dynamic programming to give you any speedup, you also need the recursions to overlap, i.e., during the computation, your algorithm should solve the same instance of a subproblem multiple times. With dynamic programming we make sure to only solve each instance of a subproblem once—after that, we can look it up in a table when we need it. If all subproblems are unique, then we do not gain anything from dynamic programming. If the subproblems are only solved once we are in the domain of divide and conquer (notice that in the divide and conquer algorithms above we only solved each subproblem once). Frequently dynamic programming algorithms are recursive problems that would take exponential time if we implemented them naïvely but where we have a polynomial solution using dynamic programming or memorisation.

Engineering a dynamic programming algorithm

To construct a divide and conquer algorithm you first have to ask yourself if one is even possible. Does tabulating the results you get from recursive calls speed up your algorithm? If all function calls you do in your algorithm are unique, i.e., you never call a function with the same arguments twice, then tabulating the results is pointless. If you do have several calls with the same arguments, then you can always store the results. You will in essence trade running time for memory usage. You cannot save the result of a recursive call more often than

you make the call, though, so you can never use more memory than you must spend on time. (This is a general property of algorithms. Every piece of memory you use you must devote time to access).

Now, it is worth considering if memorisation will gain you anything and is even worthwhile. If the number of times you call the same function with the same parameters is bounded by a constant, then your memorisation algorithm will still be in the same asymptotic class. On the other hand, you could have a situation similar to the Fibonacci numbers. Here, the number of calls you need as you move from $F(n)$ down to the base cases grows exponentially, but almost all calls are identical to other calls. In such scenarios, you will often be able to move from an exponential running time to a polynomial time algorithm (and therefore also a polynomial space algorithm).

If you now have a memorisation algorithm the next step is to see if you can translate it into a dynamic programming algorithm. In the memorisation algorithm, you need a table lookup in each recursive call to see if you have already called the function with the arguments you call it with right now. If you have a table where you can look up a value in constant time, then dynamic programming algorithms will never be asymptotically faster than memorisation. However, there is overhead in table lookups, so if we can tabulate results without too much complexity, that is what we will want to do. That is the step where we move from memorisation to dynamic programming.

You would use dynamic programming instead of memorisation if 1) you know precisely which recursions you need to tabulate and 2) you know the order of recursions you will see in your algorithm.

Take the Fibonacci example once more. We can use dynamic programming because we know that $F(i-1)$ and $F(i-2)$ will be called by $F(i)$ for all i, so we know that we will use all $F(j), j = 0, \ldots, n$ to compute $F(n)$. There are no $F(j), j > n$ called in the algorithm and there are no $F(i)$ left out for $i = 0, \ldots, n$. If we do not know which calls will be made, then we might build too large a table. With memorisation, you store exactly the number of recursive calls your program makes. It is an optimal strategy for tabulating the necessary results. With dynamic programming, you can only get this if you know which calls will be made and which will not.

The reason that you need to know the order in which the results of recursions are required is that you need to tabulate the results of all recursions before they are made. If you do not do this correctly, then your algorithm will look up a value in a table that hasn't been stored there yet. That rarely goes well. If you have to check if a result is in the table, then you have simply taken the long way around to memorisation.

Now, if you have made it to a full-fledged dynamic programming algorithm, you can potentially reduce the memory further by once again examining the dependency graph of recursive calls. In the Fibonacci example, we know that to compute $F(i)$ we only need to know $F(i-1)$ and $F(i-2)$. This means that

we do not need to remember $F(j)$ for $j < i - 2$; we have already used them to compute $F(i-1)$ and $F(i-2)$, but we no longer need them. The final algorithm that we derived above remembers two values in each iteration, and it does so because we can work out from the recursion dependency graph that this is all we need.

Edit distance

The *edit distance* between two strings, x and y, is the minimal number of operations you need to transform x into y, where operations are

1. Replace one character with another.
2. Delete a character from x.
3. Insert a character into x.

Recursion

If we consider the minimal number of edits to go from a prefix of x, $x[:i]$, to a prefix of y, $y[:j]$.

A recursive function looks like this:

$$d(i,j) = \min \begin{cases} d(i-1,j-1) + \mathbf{1}(x[i-1] \neq y[j-1]) & \text{match/substitution} \\ d(i,j-1) + 1 & \text{deletion} \\ d(i-1,j) + 1 & \text{insertion} \end{cases}$$

The function $\mathbf{1}(x[i-1] \neq y[j-1])$ is 1 if $x[i-1] \neq y[j-1]$ and zero if $x[i-1] = y[j-1]$. The base cases for the recursion are $d(0,j) = j$ and $d(i,0) = i$. The edit distance between x and y is $d(n,m)$ where n is the length of x and m is the length of y.

Notice that we find the final result in $d(n,m)$. This means that we have indices from zero to n and m where we usually would not include the last two. If you work out an example, you will notice that $i = 0$ and $j = 0$ represent prefixes matched against the empty prefix of x or y, respectively. To match the first character in either string, we move into row one or column one. So the pair (i,j) is the match of $x[:i]$ and $y[:j]$. That is why we want the value at $d(n,m)$.

Once we have a recursion, implementing it as a recursive Python function is straightforward:

```
def edit_dist(x, y, i = None, j = None):
    if i is None: i = len(x)
    if j is None: j = len(y)
    if i == 0: return j
```

```
    if j == 0: return i
    return min(
        edit_dist(x, y, i - 1, j - 1) + int(x[i - 1] != y[j - 1]),
        edit_dist(x, y, i, j - 1) + 1,
        edit_dist(x, y, i - 1, j) + 1,
    )
```

In this function, I have included the strings x and y as parameters. The function needs them to check if $x[i-1] \neq y[j-1]$. Unless the strings are global variables, the function needs them as parameters. The indices, i and j, need to be parameters in the recursion but we will always use the length of the two strings. You could always call the function like this:

```
edit_dist(x,y,len(x),len(y))
```

However, the redundancy in the initial call is best avoided, and that is what the `None` default parameters do.

Memorisation

Until you hit a base case in the recursion, you have three directions to go in for each (i, j) so calculating the recursion as it stands gives you an exponential running time. With memorisation, the running time is $O(nm)$. To see this, observe that to compute $d(n, n)$ we never need to compute a value $d(i, j)$ where $i > n$ or $j > m$ and we have a base case whenever either $i = 0$ or $j = 0$. There are $O(nm)$ such pairs (i, j).

Implementing the memorisation solution is straightforward. Generally, once we have a recursion, and have convinced ourselves that there are sufficient identical recursive calls to save results, we simply add a table to the recursive function.

```
def edit_dist(x, y, i = None, j = None, tbl = None):
    if i is None:   i = len(x)
    if j is None:   j = len(y)
    if tbl is None: tbl = {}

    if i == 0:
        tbl[(i,j)] = j
    elif j == 0:
        tbl[(i,j)] = i
    else:
        tbl[(i,j)] = min(
            edit_dist(x, y, i - 1, j - 1, tbl) +
                        int(x[i - 1] != y[j - 1]),
            edit_dist(x, y, i, j - 1, tbl) + 1,
            edit_dist(x, y, i - 1, j, tbl) + 1,
        )
    return tbl[(i,j)]
```

Dynamic programming

To move from memorisation to dynamic programming first we need to introduce an $(n+1) \times (m+1)$ table$D$ to hold all $d(i,j)$ values.

Second, we need to figure out in which order we need to fill the table. From the recursion, we can see that we loop at one row above and one column to the left in the recursion. So, if we fill out the table left-to-right and row-by-row, then the algorithm with work. We can also fill the table column-by-column and up-to-down.

To build a two-dimensional table in Python you *can* use lists, but the straightforward way

```
D = [ [0] * m ] * n
```

does not do this. It builds the inner list `[0] * m` first and then it makes a list with `n` references to it. Any change to one row will change all the rows. You can fix this, but in general, when you need a table, you are better off by using the `Numpy` package. By convention you import it and give it the name `np` like this:

```
import Numpy as np
```

You can build tables or any number of dimensions. For a two dimensional table, with $n+1$ rows and $m+1$ columns, initialised as all zeros, you can use

```
np.zeros((n + 1, m + 1))
```

Back to the edit distance and dynamic programming. My implementation of the dynamic programming version of the edit distance looks like this:

```
def build_edit_table(x, y):
    n, m = len(x), len(y)
    D = np.zeros((n + 1, m + 1))

    # base cases
    for i in range(n + 1):
        D[i,0] = i
    for j in range(m + 1):
        D[0,j] = j

    # recursion
    for i in range(1, n + 1):
        for j in range(1, m + 1):
            D[i,j] = min(
                D[i - 1, j - 1] + int(x[i - 1] != y[j - 1]),
                D[i, j - 1] + 1,
                D[i - 1, j] + 1
            )
```

```
    return D

def edit_dist(x, y):
    D = build_edit_table(x, y)
    n, m = len(x), len(y)
    return D[n,m]
```

Exercise: Since we only need to look one row up and one column left we can reduce the memory usage to $O(n)$ or $O(m)$ depending on whether we remember two row or two columns. Show how to do this.

Backtracking

For backtracking we need, for each cell in the table, to figure out which of the (up to) three cells we got the optimal path from. We check the three cells in the recursion—up, diagonal, or left—for the value they contributed, remember what the step was, and then continue from the cell we found to be where our path came from. My implementation looks like this:

```
def backtrack_(D, x, y, i, j, path):
    if i == 0:
        path.extend('D' * j)
        return
    if j == 0:
        path.extend('I' * i)
        return

    left = D[i, j - 1] + 1
    diag = D[i - 1, j - 1] + int(x[i - 1] != y[j - 1])
    up = D[i - 1, j] + 1

    dist = left
    op = 'D'
    if diag < dist:
        op = 'X' if x[i - 1] != y[j - 1] else '='
        dist = diag
    if up < dist:
        op = 'I'

    path.append(op)
    if op == 'D':
        backtrack_(D, x, y, i, j - 1, path)
    if op in ('=','X'):
        backtrack_(D, x, y, i - 1, j - 1, path)
    if op == 'I':
        backtrack_(D, x, y, i - 1, j, path)
```

Lines 2 to 7 handles the base cases where either i or j are zero. Then, on lines 9 to 11, we compute the values in the value we got from the three cells our own value could have come from. We figure out which operation gave us the value in the (i, j) cell, lines 13 to 19. I have chosen to encode insertion as `I`, matches as `=`, substitutions as `X` and deletions as `D`.

We add the step that let to this cell to the path in 21. Finally, we call the function recursive accordingly, lines 22 to 27. The path is built in reverse order, i.e., it records the steps used to calculate the dynamic programming table from the lower right and up to the upper left. If you reverse it, you get the path in the opposite direction.

The function above was just the recursive backtracking. To give backtracking a better interface I call it form this function:

```
def backtrack(D, x, y):
    n, m = len(x), len(y)
    path = []
    backtrack_(D, x, y, n, m, path)
    path.reverse()
    return ''.join(path)
```

It sets `n` and `m` to the lengths of the strings. These are the indices, from which we must start the recursion and the `path`. We call `backtrack_` to compute the path. Then, because we get the steps in the wrong order, it reverses the list, and finally, it transforms the list into a string—just to make the result easier to read.

Exercise: Describe how you would use the table from the memorisation table to implement backtracking.

Exercise: Can you also backtrack if you reduce space usage to $O(n)$ by only looking at two rows at a time?

Partitioning

Consider this problem: You have to analyse some sequence of data, and you can parallelise the computation such that it can run on multiple CPUs. The computations will be faster on contiguous substrings. You want to make the slowest computation as swift as possible. This problem is called the *partitioning problem* and it takes as input a list, x, of length N, and you have to partition it into K contiguous parts such that the biggest part is as small as possible. A part is a contiguous block, so the elements of x from some index i to another $j \geq i$. The cost of the block is the sum of the elements

$$S(i, j) = \sum_{m=i}^{j-1} x[m]$$

If x is held in a global variable, `x`, then this function will compute $S(i, j)$:

```
def S(i, j):
    return sum(x[m] for m in range(i, j))
```

Computing $S(i, j)$ as a sum takes time $O(j-i)$ but we can bring it down constant time. If we preprocess our data by computing the cumulative sum

$$\mathrm{CS}[i] = \sum_{j=0}^{i-1} x[j]$$

then we can compute $S(i, j) = \mathrm{CS}[j] - \mathrm{CS}[i]$.

Exercise: Show that you can compute the CS array in linear time and explain why the code below does that.

```
import numpy as np
CS = np.zeros(N+1)
CS[1:] = np.cumsum(x)
```

Back to the partitioning problem. The cost of a given partitioning is the sum of the elements in the largest part. We are looking for the partitioning with the smallest cost, i.e. the partitioning where the largest cost part is as small as possible. For example, if `x` is the list

```
x = [2, 5, 3, 7, 5]
```

we can split it into two partitions in the following ways:

```
[] [2, 5, 3, 7, 5]
[2] [5, 3, 7, 5]
[2, 5] [3, 7, 5]
[2, 5, 3] [7, 5]
[2, 5, 3, 7] [5]
[2, 5, 3, 7, 5] []
```

The cost of the first partition is 22, the second 20, the third 15, the fourth 12, the fifth 17, and the sixth 22. The best of these is the fourth index with score 12.

If x has length n then there are $n+1$ ways to partition it into two (and $n-1$ to partition x into $K = 2$ non-empty parts[1]). If we want $K = 3$ non-empty and distinct, i.e., no two parts are identical, then we have $(n-1)(n-2)$ possibilities. We have $n-1$ positions to put the first split in and then $n-2$ positions for the next split. For $K = 3$ the number of partitions is $(n-1)(n-2)(n-3)$. For general K the number of partitions is $(n-1)!/(n-K-1)!$. This is super-exponential so clearly exhaustively trying out all partitions is not a feasible strategy.

[1] In the optimisation problem we do not require that the partitioning has non-empty parts, but any partitioning with empty parts can split the partition with the greatest cost and thus reduce the cost. An optimal partitioning will, therefore, not contain empty parts.

Exercise: Prove that the number of partitions is $\frac{(n-1)!}{(n-K-1)!}$.

Recursion

To derive a dynamic programming solution, we first derive a recursive solution to the problem. We can consider where to put the separator between the last partition and the previous partition. At some index i into x, we have the start index of the last partition. The cost of the last index is $S(i, N)$. The cost of this partitioning must be

$$\max\left[P(i, K-1), S(i, N)\right]$$

where $P(i, K-1)$ is the best partitioning of the array $x[0:i]$ into $K-1$ partitions. The best partitioning of x into K partitions is found by picking the optimal index i.

$$P(N, K) = \min_{i=0}^{N} \left\{\max\left[P(i, K-1), S(i, N)\right]\right\}$$

Notice that the maximisation is over two values, $P(i, K-1)$ and $S(i, N)$ while the minimisation is over all indices in x. We can handle the first in constant time and the latter in linear time if we can lookup all $P(i, K-1)$ in a table.

The basis cases of the recursion are single partitions, where we have no choice but to put all elements in the same partition, $P(n, 1) = S(0, n)$ for all n, and the empty prefix of x where the cost of any number of partitions is zero, $P(0, k) = 0$, for all k.

In summary we have

$$P(i, k) = \begin{cases} S(0, n) & k = 1 \\ 0 & i = 0 \\ \min_{j=0}^{i} \left\{\max\left[P(j, K-1), S(j, N)\right]\right\} & \text{otherwise} \end{cases} \tag{1}$$

We can already look up $S(i, N)$ for all i (and N) but $P(i, K-1)$ we need to compute recursively.

Exercise: Implement a recursive function that computes $P(n, k)$ for all $0 \leq n \leq N$ and $k \geq 0$, i.e., implement (eq. 1) as a Python function.

Exercise: Implement memorisation in the recursive function. You can simply check if you already have a value for the recursive call—identified by i and $K-1$—compute it if we do not, and in either case return it. You can use a `dict` for the table.

Exercise: Instead of using a `dict` in the previous exercise, how would you use an $N \times K$ Numpy table? You will need a way to indicate that a value has not been computed yet.

Dynamic programming

Once we have our recursion, we can always build a memorisation algorithm, but we need to examine it carefully to see if we can build it bottom up and get a dynamic programming algorithm. First, do we use all the values and second can we compute them such that we can guarantee that a result is available when we need it.

We can see directly from the recursion that to compute $P(i, k)$ we look at $P(j, k-1)$ for each $0 \leq j < i$. The recursion on k will go down to the base case for k so we will use all values of j and k, and we will use them one k at a time. If we know all $P(j, k-1), j < i$ when we compute $P(i, k)$ then we have all the values we need. By these observations we know we can build a table, `PT`, and fill it up either by row or by column.

```
def P(M, K):
    PT = np.zeros((N+1,K+1))
    # Base cases
    for i in range(N+1):
        PT[i,1] = ST[0,i]
    for j in range(2,K+1):
        PT[0,j] = 0

    # Recursive case
    for i in range(1,N+1):
        for j in range(2,K+1):
            PT[i,j] = min(max(PT[m,j-1], ST[m,i])
                          for m in range(i))
    return PT[N, K]
```

Exercise: What is the running time of the dynamic programming solution? What about the memorisation solution?

Exercise: Reduce the space complexity to $O(n)$ by only storing the previous row or column in the P table.

Exercise: Since we want to know the optimal partition and not just the cost of the optimal partition, we need to backtrack to get it. Show how this can be done and implement your solution.

Representing floating point numbers

Floating point numbers, the computer analogue to real numbers, can be represented in different ways, but they are all variations of the informal presentation I give in this section. If you are not particularly interested in how these numbers are represented, you can safely skip this section. You can already return to it if you think you are getting weird behaviour when working with floating point numbers.

The representation used by modern computers is standardised as IEEE 754. It fundamentally represents numbers as explained below, but with some tweaks that let you represent plus and minus infinity, "not a number" (NaN), and with higher precision for numbers close to zero than the presentation here would allow. It also uses a sign bit for the coefficient while it represents the exponent as a signed integer for reasons lost deep in numerical analysis. All that you need to know is that floating point numbers work roughly as I have explained here, but with lots of technical complications. If you find yourself a heavy user of floating point numbers, you will need to study numerical analysis beyond what we can cover in this book, and you can worry about the details of number representations there.

Floating point numbers are similar to the *scientific notation* for base-b numbers, where numbers are represented as

$$x = \pm a \times b^{\pm q}$$

where $a = a_1.a_2a_3 \ldots a_n, a_i \in \{0, 1, \ldots, b-1\}$ is the *coefficient* and $q = q_1q_2q_3 \ldots q_m, q_i \in \{0, 1, \ldots, b-1\}$ is the *exponent* of the number. To get a binary notation, replace b by 2. For non-zero numbers, a_1 must be 1, so we do not represent it explicitly, which gives us one more bit to work with. Not all real numbers can be represented with this notation if we require that both a are q are finite sequences,[2] but if we allow them to be infinite we can. We can approximate any number arbitrarily close by using sufficiently long sequences numbers of digits; n for the coefficient and m for the exponent. We usually assume that if $x \neq 0$ then $a_1 \neq 0$ since, if $a_1 = 0$ we can update a to $a_2.a_3 \ldots a_n$ and decrease q by one if positive or increase it by one of positive.

Where floating point numbers differ from the real numbers is that we have a fixed limit on how many digits we have available for the coefficient and the exponent. To represent any real number, we can choose sufficiently high values for n and m, but with floating point numbers there is a fixed number of digits for a and b. You cannot approximate all numbers arbitrarily close. For example, with $b = 2$ and $n = m = 1$, we have $\pm a \in \{-1, 0, 1\}, \pm q \in \{-1, 0, 1\}$, so we can only

[2] Which real numbers are representable using a finite number of digits depends on the base, b. You cannot represent $1/3$ using a finite decimal ($b = 10$) notation but in base $b = 3$ it is simply 1×3^{-1}. Likewise, you cannot represent $1/10$ in binary in a finite number of digits, where you trivially can in base 10.

represent the numbers $\{-1, -1/2, 0, 1/2, 1\}$: $\pm 1/2 = \pm 1 \times 2^{-1}$, $\pm 0 = \pm 0 \times 2^{q}$, and $\pm 1 = \pm 1 \times 2^{\pm 0}$ (where ± 0 might be represented as two different numbers, signed and unsigned zero, or as a single unsigned zero, depending on the details of the representation). If we use two bits for the exponent, we get the number line shown in fig. 16.

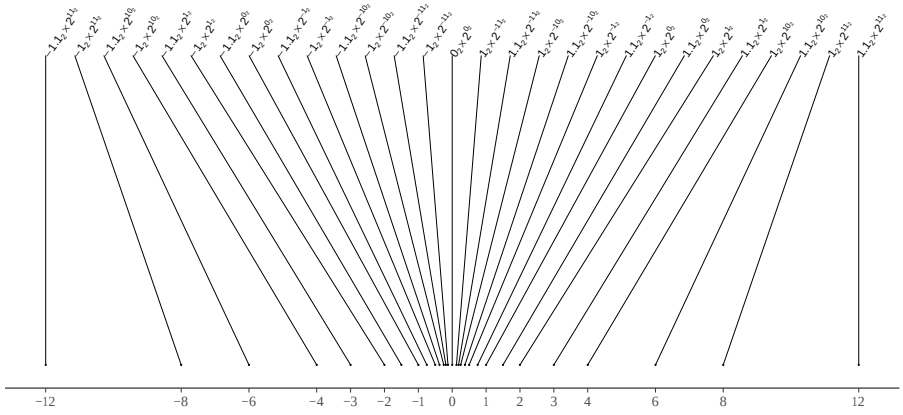

Figure 16: Number line when we have one bit for the coefficient and two bits for the exponent.

As a rule of thumb, floating point numbers have the property that is illustrated in fig. 16. The numbers are closer together when you get closer to zero and further apart when their magnitude increase. There is a positive and a negative minimal number; you cannot get closer than those to zero except by being zero. If you need to represent a non-zero number of magnitude less than this, we say that you have an *underflow* error. There are also a smallest and a largest number (the positive and negative numbers furthest from zero). If you need to represent numbers of magnitude larger than these, we say you have an *overflow* error.

There isn't really much you can do about underflow and overflow problems except to try to avoid them. Translating numbers into their logarithm is often a viable approach if you only multiply numbers, but can be tricky if you also need to add them.

In the binary sum example from earlier, the problem is not underflow or overflow, but rather losing significant bits when adding numbers. The problem there is the fixed number of bits set aside for the coefficient. If you want to add two numbers of different magnitude, i.e. their exponents are different, then you first have to make the exponents equal, which you can do by moving the decimal point. Consider 1.01101×2^3 to 1.11010×2^0—where we have five bits for the

coefficients (plus one that is always 1, i.e. $n = 6$). If you want to add 1.01101×2^3 to 1.11010×2^0 you have to move the decimal point in one of them. With the representation we have for the coefficients, $a = a_1.a_2 \ldots a_n$, we can only have one digit before the decimal point so we cannot translate 1.01101×2^3 into 1011.01×2^0, so we have to translate 1.11010×2^0 into 0.00111010×2^3. We want the most significant bit to be one, of course, but we make this representation for the purpose of addition; once we have added the numbers, we put it in a form where the most significant bit in the coefficient is one. The problem with addition is that we cannot represent 0.00111010×2^3 if we only have five bits in the coefficient. We have five bits because our numbers are six bits long and the first one must always be one. So we have to round the number off and get 0.00111×2^3. The difference in the sum is $2^{-4} = 0.0625$, so not a large difference in the final result, but we have lost three bits of accuracy from the smaller number.

Without limiting the number of bits we have this calculation.

$$1.01101 \times 2^3 + 1.11010 \times 2^0 = \tag{2}$$
$$1.01101 \times 2^3 + 0.00111010 \times 2^3 = \tag{3}$$
$$1.01101 \times 2^3 + 0.00111010 \times 2^3 = 1.10100010 \times 2^3 \tag{4}$$
$$\tag{5}$$

If we cannot go beyond five bits, translating 1.11010 into 0.00111010×2^3 will get us 0.00111×2^3 and using that we get:

$$1.01101 \times 2^3 + 0.00111 \times 2^3 = 1.10100 \times 2^3$$

This is clearly different from the calculation with more bits.

In general, you expect to lose bits equal to the difference in the exponents of the numbers. The actual loss depends on the details of the representation, but as a rule of thumb, this is what you will lose.

Assume we have one informative bit for the coefficient ($n = 2$) and two for the exponent, ($m = 2$), and we wanted to add six ones together $6 \times 1.0_2 \times 2^0 = 1.1_2 \times 2^2$. Adding the numbers one at a time we get:

$$1.0 \times 2^0 + 1.0 \times 2^0 = 1.0 \times 2^1 \tag{6}$$
$$1.0 \times 2^1 + 0.1 \times 2^1 = 1.1 \times 2^1 \tag{7}$$
$$1.1 \times 2^1 + 0.1 \times 2^1 = 1.0 \times 2^2 \tag{8}$$
$$1.0 \times 2^2 + 0.01 \times 2^2 = 1.01 \times 2^2 = 1.0 \times 2^2 \tag{9}$$
$$\tag{10}$$

which is off by 0.1×2^2. If we add the numbers as we did with our `binary_sum` function, we instead have

$$1.0 \times 2^0 + 1.0 \times 2^0 = 1.0 \times 2^1 \quad (\times 3) \tag{11}$$

$$1.0 \times 2^1 + 1.0 \times 2^1 = 1.0 \times 2^2 \tag{12}$$

$$1.0 \times 2^2 + 1.0 \times 2^2 = 1.1 \times 2^2 \tag{13}$$

which is correct.

Obviously, the floating point numbers you use in Python have a much higher precision than one bit per coefficient and two per exponent so you will not run into problems with accuracy as fast as in this example. The principle, is the same, however. If you add enough numbers, you risk that the accumulator becomes too large for the addition with the next number to have an effect. If you run into this, then adding the numbers pairwise as in `binary_sum` can alleviate this.