

# Algorithms and Complexity

Herbert S. Wilf  
University of Pennsylvania  
Philadelphia, PA 19104-6395

## Copyright Notice

Copyright 1994 by Herbert S. Wilf. This material may be reproduced for any educational purpose, multiple copies may be made for classes, etc. Charges, if any, for reproduced copies must be just enough to recover reasonable costs of reproduction. Reproduction for commercial purposes is prohibited. This cover page must be included in all distributed copies.

## Internet Edition, Summer, 1994

This edition of Algorithms and Complexity is the file "`pub/wilf/AlgComp.ps.Z`" at the anonymous ftp site "`ftp.cis.upenn.edu`". It may be taken at no charge by all interested persons. Comments and corrections are welcome, and should be sent to `wilf@central.cis.upenn.edu`

## Chapter 2: Recursive Algorithms

### 2.1 Introduction

Here are two different ways to define  $n!$ , if  $n$  is a positive integer. The first definition is nonrecursive, the second is recursive.

- (1) ' $n!$  is the product of all of the whole numbers from 1 to  $n$ , inclusive.'
- (2) 'If  $n = 1$  then  $n! = 1$ , else  $n! = n \cdot (n - 1)!$ .'

Let's concentrate on the second definition. At a glance, it seems illegal, because we're defining something, and in the definition the same 'something' appears. Another glance, however, reveals that the value of  $n!$  is defined in terms of the value of the same function at a *smaller* value of its argument, *viz.*  $n - 1$ . So we're really only using mathematical induction in order to validate the assertion that a function has indeed been defined for all positive integers  $n$ .

What is the practical import of the above? It's monumental. Many modern high-level computer languages can handle recursive constructs directly, and when this is so, the programmer's job may be considerably simplified. Among recursive languages are Pascal, PL/C, Lisp, APL, C, and many others. Programmers who use these languages should be aware of the power and versatility of recursive methods (conversely, people who like recursive methods should learn one of those languages!).

A formal 'function' module that would calculate  $n!$  *nonrecursively* might look like this.

```
function fact(n);
{computes n! for given n > 0}
  fact := 1;
  for i := 1 to n do fact := i · fact;
end.
```

On the other hand a *recursive*  $n!$  module is as follows.

```
function fact(n);
  if n = 1 then fact := 1
    else fact := n · fact(n - 1);
end.
```

The hallmark of a recursive procedure is that it *calls itself*, with arguments that are in some sense smaller than before. Notice that there are no visible loops in the recursive routine. Of course there will be loops in the compiled machine-language program, so in effect the programmer is shifting many of the bookkeeping problems to the compiler (but *it* doesn't mind!).

Another advantage of recursiveness is that the *thought* processes are helpful. Mathematicians have known for years that induction is a marvellous method for proving theorems, making constructions, etc. Now computer scientists and programmers can profitably think recursively too, because recursive compilers allow them to express such thoughts in a natural way, and as a result many methods of great power are being formulated recursively, methods which, in many cases, might not have been developed if recursion were not readily available as a practical programming tool.

Observe next that the 'trivial case,' where  $n = 1$ , is handled separately, in the recursive form of the  $n!$  program above. This trivial case is in fact essential, because it's the only thing that stops the execution of the program. In effect, the computer will be caught in a loop, reducing  $n$  by 1, until it reaches 1, then it will actually know the value of the function *fact*, and after that it will be able to climb back up to the original input value of  $n$ .

The overall structure of a recursive routine will always be something like this:

```

procedure calculate(list of variables);
  if {trivialcase} then do {trivialthing}
  else do
    {call calculate(smaller values of the variables)};
    {maybe do a few more things}
end.

```

In this chapter we're going to work out a number of examples of recursive algorithms, of varying sophistication. We will see how the recursive structure helps us to analyze the running time, or complexity, of the algorithms. We will also find that there is a bit of art involved in choosing the list of variables that a recursive procedure operates on. Sometimes the first list we think of doesn't work because the recursive call seems to need more detailed information than we have provided for it. So we try a larger list, and then perhaps it works, or maybe we need a still larger list ..., but more of this later.

### Exercises for section 2.1

1. Write a *recursive* routine that will find the digits of a given integer  $n$  in the base  $b$ . There should be no visible loops in your program.

## 2.2 Quicksort

Suppose that we are given an array  $x[1], \dots, x[n]$  of  $n$  numbers. We would like to rearrange these numbers as necessary so that they end up in nondecreasing order of size. This operation is called *sorting* the numbers.

For instance, if we are given  $\{9, 4, 7, 2, 1\}$ , then we want our program to output the sorted array  $\{1, 2, 4, 7, 9\}$ .

There are many methods of sorting, but we are going to concentrate on methods that rely on only two kinds of basic operations, called *comparisons* and *interchanges*. This means that our sorting routine is allowed to

- (a) pick up two numbers ('keys') from the array, compare them, and decide which is larger.
- (b) interchange the positions of two selected keys.

Here is an example of a rather primitive sorting algorithm:

- (i) find, by successive comparisons, the smallest key
- (ii) interchange it with the first key
- (iii) find the second smallest key
- (iv) interchange it with the second key, etc. etc.

Here is a more formal algorithm that does the job above.

```

procedure slowsort(X: array[1..n]);
{sorts a given array into nondecreasing order}
  for  $r := 1$  to  $n - 1$  do
    for  $j := r + 1$  to  $n$  do
      if  $x[j] < x[r]$  then swap( $x[j], x[r]$ )
end.{slowsort}

```

If you are wondering why we called this method 'primitive,' 'slowsort,' and other pejorative names, the reason will be clearer after we look at its complexity.

What is the cost of sorting  $n$  numbers by this method? We will look at two ways to measure that cost. First let's choose our unit of cost to be one comparison of two numbers, and then we will choose a different unit of cost, namely one interchange of position ('swap') of two numbers.

How many paired comparisons does the algorithm make? Reference to *procedure slowsort* shows that it makes one comparison for each value of  $j = r + 1, \dots, n$  in the inner loop. This means that the total number of comparisons is

$$\begin{aligned} f_1(n) &= \sum_{r=1}^{n-1} \sum_{j=r+1}^n 1 \\ &= \sum_{r=1}^{n-1} (n-r) \\ &= (n-1)n/2. \end{aligned}$$

The number of comparisons is  $\Theta(n^2)$ , which is quite a lot of comparisons for a sorting method to do. Not only that, but the method does that many comparisons regardless of the input array, *i.e.* its best case and its worst case are equally bad.

The Quicksort\* method, which is the main object of study in this section, does a *maximum* of  $cn^2$  comparisons, but *on the average* it does far fewer, a mere  $O(n \log n)$  comparisons. This economy is much appreciated by those who sort, because sorting applications can be immense and time consuming. One popular sorting application is in alphabetizing lists of names. It is easy to imagine that some of those lists are very long, and that the replacement of  $\Theta(n^2)$  by an average of  $O(n \log n)$  comparisons is very welcome. An insurance company that wants to alphabetize its list of 5,000,000 policyholders will gratefully notice the difference between  $n^2 = 25,000,000,000,000$  comparisons and  $n \log n = 77,124,740$  comparisons.

If we choose as our unit of complexity the number of swaps of position, then the running time may depend strongly on the input array. In the ‘slowsort’ method described above, some arrays will need no swaps at all while others might require the maximum number of  $(n-1)n/2$  (which arrays need that many swaps?). If we average over all  $n!$  possible arrangements of the input data, assuming that the keys are distinct, then it is not hard to see that the average number of swaps that *slowsort* needs is  $\Theta(n^2)$ .

Now let’s discuss Quicksort. In contrast to the sorting method above, the basic idea of Quicksort is sophisticated and powerful. Suppose we want to sort the following list:

$$26, 18, 4, 9, 37, 119, 220, 47, 74 \tag{2.2.1}$$

The number 37 in the above list is in a very intriguing position. Every number that precedes it is smaller than it is and every number that follows it is larger than it is. What that means is that *after sorting the list, the 37 will be in the same place it now occupies, the numbers to its left will have been sorted but will still be on its left, and the numbers on its right will have been sorted but will still be on its right.*

If we are fortunate enough to be given an array that has a ‘splitter,’ like 37, then we can

- (a) sort the numbers to the left of the splitter, and then
- (b) sort the numbers to the right of the splitter.

Obviously we have here the germ of a recursive sorting routine.

The fly in the ointment is that most arrays don’t have splitters, so we won’t often be lucky enough to find the state of affairs that exists in (2.2.1). However, we can make our own splitters, with some extra work, and that is the idea of the Quicksort algorithm. Let’s state a preliminary version of the recursive procedure as follows (look carefully for how the procedure handles the trivial case where  $n=1$ ).

```

procedure quicksortprelim(x : an array of n numbers);
{sorts the array x into nondecreasing order}
  if n ≥ 2 then
    permute the array elements so as to create a splitter;
    let x[i] be the splitter that was just created;
    quicksortprelim(the subarray x1, . . . , xi-1) in place;
    quicksortprelim(the subarray xi+1, . . . , xn) in place;
  end.{quicksortprelim}

```

---

\* C. A. R. Hoare, *Comp. J.*, **5** (1962), 10-15.

This preliminary version won't run, though. It looks like a recursive routine. It seems to call itself twice in order to get its job done. But it doesn't. It calls something that's just slightly different from itself in order to get its job done, and that won't work.

Observe the exact purpose of Quicksort, as described above. We are given an array of length  $n$ , and we want to sort it, *all of it*. Now look at the two 'recursive calls,' which really aren't quite. The first one of them sorts the array to the left of  $x_i$ . That is indeed a recursive call, because we can just change the ' $n$ ' to ' $i - 1$ ' and call Quicksort. The second recursive call is the problem. It wants to sort a portion of the array that doesn't begin at the beginning of the array. The routine Quicksort as written so far doesn't have enough flexibility to do that. So we will have to give it some more parameters.

Instead of trying to sort *all* of the given array, we will write a routine that sorts only the portion of the given array  $\mathbf{x}$  that extends from  $x[left]$  to  $x[right]$ , inclusive, where  $left$  and  $right$  are input parameters. This leads us to the second version of the routine:

```

procedure qksort( $\mathbf{x}$ :array; left, right:integer);
{sorts the subarray  $x[left], \dots, x[right]$ }
  if  $right - left \geq 1$  then
    create a splitter for the subarray in the  $i^{th}$  array position;
    qksort( $\mathbf{x}, left, i - 1$ );
    qksort( $\mathbf{x}, i + 1, right$ )
  end.{qksort}

```

Once we have *qksort*, of course, Quicksort is no problem: we call *qksort* with  $left := 1$  and  $right := n$ .

The next item on the agenda is the little question of how to create a splitter in an array. Suppose we are working with a subarray

$$x[left], x[left + 1], \dots, x[right].$$

The first step is to choose one of the subarray elements (the element itself, not the *position* of the element) to be the splitter, and the second step is to make it happen. The choice of the splitter element in the Quicksort algorithm is done very simply: *at random*. We just choose, using our favorite random number generator, one of the entries of the given subarray, let's call it  $T$ , and declare it to be the splitter. To repeat the parenthetical comment above,  $T$  is the *value* of the array entry that was chosen, not its *position* in the array. Once the value is selected, the position will be what it has to be, namely to the right of all smaller entries, and to the left of all larger entries.

The reason for making the random choice will become clearer after the smoke of the complexity discussion has cleared, but briefly it's this: the analysis of the average case complexity is relatively easy if we use the random choice, so that's a plus, and there are no minuses.

Second, we have now chosen  $T$  to be the value around which the subarray will be split. The entries of the subarray must be moved so as to make  $T$  the splitter. To do this, consider the following algorithm.\*

---

\* Attributed to Nico Lomuto by Jon Bentley, *CACM* 27 (April 1984).

```

procedure split(x, left, right, i)
{chooses at random an entry  $T$  of the subarray
 [ $x_{left}, x_{right}$ ], and splits the subarray around  $T$ }
{the output integer  $i$  is the position of  $T$  in the
 output array:  $x[i] = T$ };
1   $L :=$  a random integer in [ $left, right$ ];
2  swap( $x[left], x[L]$ );
3  {now the splitter is first in the subarray}
4   $T := x[left]$ ;
5   $i := left$ ;
6  for  $j := left + 1$  to  $right$  do
   begin
7     if  $x[j] < T$  then
   begin
8        $i := i + 1$ 
       swap( $x[i], x[j]$ )
   end;
   end
9  swap( $x[left], x[i]$ )
10 end.{split}

```

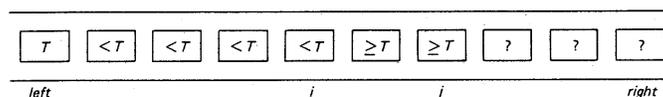
We will now prove the correctness of *split*.

**Theorem 2.2.1.** *Procedure split correctly splits the array  $\mathbf{x}$  around the chosen value  $T$ .*

**Proof:** We claim that as the loop in lines 7 and 8 is repeatedly executed for  $j := left + 1$  to  $right$ , the following three assertions will always be true just *after* each execution of lines 7, 8:

- (a)  $x[left] = T$  and
- (b)  $x[r] < T$  for all  $left < r \leq i$  and
- (c)  $x[r] \geq T$  for all  $i < r \leq j$

Fig. 2.2.1 illustrates the claim.



**Fig. 2.2.1: Conditions (a), (b), (c)**

To see this, observe first that (a), (b), (c) are surely true at the beginning, when  $j = left + 1$ . Next, if for some  $j$  they are true, then the execution of lines 7, 8 guarantee that they will be true for the next value of  $j$ .

Now look at (a), (b), (c) when  $j = right$ . It tells us that just prior to the execution of line 9 the condition of the array will be

- (a)  $x[left] = T$  and
- (b)  $x[r] < T$  for all  $left < r \leq i$  and
- (c)  $x[r] \geq T$  for all  $i < r \leq right$ .

When line 9 executes, the array will be in the correctly split condition. ■

Now we can state a ‘final’ version of *qksort* (and therefore of Quicksort too).

```

procedure qksort(x:array; left, right:integer);
{sorts the subarray  $x[\textit{left}], \dots, x[\textit{right}]$ };
if right - left  $\geq$  1 then
    split(x, left, right, i);
    qksort(x, left, i - 1);
    qksort(x, i + 1, right)
end.{qksort}

```

```

procedure Quicksort(x :array; n:integer)
{sorts an array of length  $n$ };
    qksort(x, 1, n)
end.{Quicksort}

```

Now let's consider the complexity of Quicksort. How long does it take to sort an array? Well, the amount of time will depend on exactly which array we happen to be sorting, and furthermore it will depend on how lucky we are with our random choices of splitting elements.

If we want to see Quicksort at its worst, suppose we have a really unlucky day, and that the random choice of the splitter element happens to be the smallest element in the array. Not only that, but suppose this kind of unlucky choice is repeated on each and every recursive call.

If the splitter element is the smallest array entry, then it won't do a whole lot of splitting. In fact, if the original array had  $n$  entries, then one of the two recursive calls will be to an array with no entries at all, and the other recursive call will be to an array of  $n - 1$  entries. If  $L(n)$  is the number of paired comparisons that are required in this extreme scenario, then, since the number of comparisons that are needed to carry out the call to *split* an array of length  $n$  is  $n - 1$ , it follows that

$$L(n) = L(n - 1) + n - 1 \quad (n \geq 1; L(0) = 0).$$

Hence,

$$L(n) = (1 + 2 + \dots + (n - 1)) = \Theta(n^2).$$

The worst-case behavior of Quicksort is therefore quadratic in  $n$ . In its worst moods, therefore, it is as bad as '*slowsort*' above.

Whereas the performance of *slowsort* is pretty much always quadratic, no matter what the input is, Quicksort is usually a lot faster than its worst case discussed above.

We want to show that *on the average* the running time of Quicksort is  $O(n \log n)$ .

The first step is to get quite clear about what the word 'average' refers to. We suppose that the entries of the input array  $\mathbf{x}$  are all distinct. Then the performance of Quicksort can depend only on the sequence of size relationships in the input array and the choices of the random splitting elements.

The actual numerical values that appear in the input array are not in themselves important, except that, to simplify the discussion we will assume that they are all different. The only thing that will matter, then, will be the set of outcomes of all of the paired comparisons of two elements that are done by the algorithm. Therefore, we will assume, for the purposes of analysis, that the entries of the input array are exactly the set of numbers  $1, 2, \dots, n$  in some order.

There are  $n!$  possible orders in which these elements might appear, so we are considering the action of Quicksort on just these  $n!$  inputs.

Second, for each particular one of these inputs, the choices of the splitting elements will be made by choosing, at random, one of the entries of the array at each step of the recursion. We will also average over all such random choices of the splitting elements.

Therefore, when we speak of the function  $F(n)$ , the *average* complexity of Quicksort, we are speaking of the average number of *pairwise comparisons* of array entries that are made by Quicksort, where the averaging

is done first of all over all  $n!$  of the possible input orderings of the array elements, and second, for each such input ordering, we average also over all sequences of choices of the splitting elements.

Now let's consider the behavior of the function  $F(n)$ . What we are going to show is that  $F(n) = O(n \log n)$ .

The labor that  $F(n)$  estimates has two components. First there are the pairwise comparisons involved in choosing a splitting element and rearranging the array about the chosen splitting value. Second there are the comparisons that are done in the two recursive calls that follow the creation of a splitter.

As we have seen, the number of comparisons involved in splitting the array is  $n - 1$ . Hence it remains to estimate the number of comparisons in the recursive calls.

For this purpose, suppose we have rearranged the array about the splitting element, and that it has turned out that the splitting entry now occupies the  $i^{\text{th}}$  position in the array.

Our next remark is that each value of  $i = 1, 2, \dots, n$  is equally likely to occur. The reason for this is that we chose the splitter originally by choosing a random array entry. Since all orderings of the array entries are equally likely, the one that we happened to have chosen was just as likely to have been the largest entry as to have been the smallest, or the  $17^{\text{th}}$ -from-largest, or whatever.

Since each value of  $i$  is equally likely, each  $i$  has probability  $1/n$  of being chosen as the residence of the splitter.

If the splitting element lives in the  $i^{\text{th}}$  array position, the two recursive calls to Quicksort will be on two subarrays, one of which has length  $i - 1$  and the other of which has length  $n - i$ . The average numbers of pairwise comparisons that are involved in such recursive calls are  $F(i - 1)$  and  $F(n - i)$ , respectively. It follows that our average complexity function  $F$  satisfies the relation

$$F(n) = n - 1 + \frac{1}{n} \sum_{i=1}^n \{F(i - 1) + F(n - i)\} \quad (n \geq 1). \quad (2.2.2)$$

together with the initial value  $F(0) = 0$ .

How can we find the solution of the recurrence relation (2.2.2)? First let's simplify it a little by noticing that

$$\begin{aligned} \sum_{i=1}^n \{F(n - i)\} &= F(n - 1) + F(n - 2) + \dots + F(0) \\ &= \sum_{i=1}^n \{F(i - 1)\} \end{aligned} \quad (2.2.3)$$

and so (2.2.2) can be written as

$$F(n) = n - 1 + \frac{2}{n} \sum_{i=1}^n F(i - 1). \quad (2.2.4)$$

We can simplify (2.2.4) a lot by getting rid of the summation sign. This next step may seem like a trick at first (and it is!), but it's a trick that is used in so many different ways that now we call it a 'method.' What we do is first to multiply (2.2.4) by  $n$ , to get

$$nF(n) = n(n - 1) + 2 \sum_{i=1}^n F(i - 1). \quad (2.2.5)$$

Next, in (2.2.5) we replace  $n$  by  $n - 1$ , yielding

$$(n - 1)F(n - 1) = (n - 1)(n - 2) + 2 \sum_{i=1}^{n-1} F(i - 1). \quad (2.2.6)$$

Finally we subtract (2.2.6) from (2.2.5), and the summation sign obligingly disappears, leaving behind just

$$nF(n) - (n - 1)F(n - 1) = n(n - 1) - (n - 1)(n - 2) + 2F(n - 1). \quad (2.2.7)$$

After some tidying up, (2.2.7) becomes

$$F(n) = \left(1 + \frac{1}{n}\right)F(n-1) + \left(2 - \frac{2}{n}\right). \quad (2.2.8)$$

which is exactly in the form of the general first-order recurrence relation that we discussed in section 1.4.

In section 1.4 we saw that to solve (2.2.8) the winning tactic is to change to a new variable  $y_n$  that is defined, in this case, by

$$\begin{aligned} F(n) &= \frac{n+1}{n} \frac{n}{n-1} \frac{n-1}{n-2} \cdots \frac{2}{1} y_n \\ &= (n+1)y_n. \end{aligned} \quad (2.2.9)$$

If we make the change of variable  $F(n) = (n+1)y_n$  in (2.2.8), then it takes the form

$$y_n = y_{n-1} + 2(n-1)/n(n+1) \quad (n \geq 1) \quad (2.2.10)$$

as an equation for the  $y_n$ 's ( $y_0 = 0$ ).

The solution of (2.2.10) is obviously

$$\begin{aligned} y_n &= 2 \sum_{j=1}^n \frac{j-1}{j(j+1)} \\ &= 2 \sum_{j=1}^n \left\{ \frac{2}{j+1} - \frac{1}{j} \right\} \\ &= 2 \sum_{j=1}^n \frac{1}{j} - 4n/(n+1). \end{aligned}$$

Hence from (2.2.9),

$$F(n) = 2(n+1) \left\{ \sum_{j=1}^n 1/j \right\} - 4n \quad (2.2.11)$$

is the average number of pairwise comparisons that we do if we Quicksort an array of length  $n$ . Evidently  $F(n) \sim 2n \log n$  ( $n \rightarrow \infty$ ) (see (1.1.7) with  $g(t) = 1/t$ ), and we have proved

**Theorem 2.2.2.** *The average number of pairwise comparisons of array entries that Quicksort makes when it sorts arrays of  $n$  elements is exactly as shown in (2.2.11), and is  $\sim 2n \log n$  ( $n \rightarrow \infty$ ).*

Quicksort is, on average, a very quick sorting method, even though its worst case requires a quadratic amount of labor.

### Exercises for section 2.2

1. Write out an array of 10 numbers that contains no splitter. Write out an array of 10 numbers that contains 10 splitters.
2. Write a program that does the following. Given a positive integer  $n$ . Choose 100 random permutations of  $[1, 2, \dots, n]$ ,\* and count how many of the 100 had at least one splitter. Execute your program for  $n = 5, 6, \dots, 12$  and tabulate the results.
3. Think of some method of sorting  $n$  numbers that isn't in the text. In the worst case, how many comparisons might your method do? How many swaps?

\* For a fast and easy way to do this see A. Nijenhuis and H. S. Wilf, *Combinatorial Algorithms*, 2nd ed. (New York: Academic Press, 1978), chap. 6.

4. Consider the array

$$\mathbf{x} = \{2, 4, 1, 10, 5, 3, 9, 7, 8, 6\}$$

with  $left = 1$  and  $right = 10$ . Suppose that the procedure *split* is called, and suppose the random integer  $L$  in step 1 happens to be 5. Carry out the complete *split* algorithm (not on a computer; use pencil and paper). Particularly, record the condition of the array  $\mathbf{x}$  after each value of  $j$  is processed in the *for*  $j = \dots$  loop.

5. Suppose  $H(0) = 1$  and  $H(n) \leq 1 + \frac{1}{n} \sum_{i=1}^n H(i-1)$  ( $n \geq 1$ ). How big might  $H(n)$  be?

6. If  $Q(0) = 0$  and  $Q(n) \leq n^2 + \sum_{i=1}^n Q(i-1)$  ( $n \geq 1$ ), how big might  $Q(n)$  be?

7. (Research problem) Find the asymptotic behavior, for large  $n$ , of the probability that a randomly chosen permutation of  $n$  letters has a splitter.

### 2.3 Recursive graph algorithms

Algorithms on graphs are another rich area of applications of recursive thinking. Some of the problems are quite different from the ones that we have so far been studying in that they seem to need exponential amounts of computing time, rather than the polynomial times that were required for sorting problems.

We will illustrate the dramatically increased complexity with a recursive algorithm for the ‘maximum independent set problem,’ one which has received a great deal of attention in recent years.

Suppose a graph  $G$  is given. By an *independent set* of vertices of  $G$  we mean a set of vertices no two of which are connected by an edge of  $G$ . In the graph in Fig. 2.3.1 the set  $\{1, 2, 6\}$  is an independent set and so is the set  $\{1, 3\}$ . The largest independent set of vertices in the graph shown there is the set  $\{1, 2, 3, 6\}$ . The problem of finding the size of the largest independent set in a given graph is computationally very difficult. All algorithms known to date require exponential amounts of time, in their worst cases, although no one has proved the nonexistence of fast (polynomial time) algorithms.

If the problem itself seems unusual, and maybe not deserving of a lot of attention, be advised that in Chapter 5 we will see that it is a member in good standing of a large family of very important computational problems (the ‘NP-complete’ problems) that are tightly bound together, in that if we can figure out better ways to compute any one of them, then we will be able to do all of them faster.

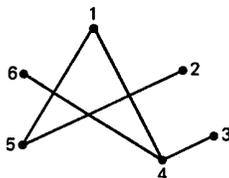


Fig. 2.3.1

Here is an algorithm for the independent set problem that is easy to understand and to program, although, of course, it may take a long time to run on a large graph  $G$ .

We are looking for the size of the largest independent set of vertices of  $G$ . Suppose we denote that number by  $maxset(G)$ . Fix some vertex of the graph, say vertex  $v^*$ . Let’s distinguish two kinds of independent sets of vertices of  $G$ . There are those that contain vertex  $v^*$  and those that don’t contain vertex  $v^*$ .

If an independent set  $S$  of vertices contains vertex  $v^*$ , then what does the rest of the set  $S$  consist of? The remaining vertices of  $S$  are an independent set in a smaller graph, namely the graph that is obtained from  $G$  by deleting vertex  $v^*$  as well as all vertices that are connected to vertex  $v^*$  by an edge. This latter set of vertices is called the *neighborhood* of vertex  $v^*$ , and is written  $Nbhd(v^*)$ .

The set  $S$  consists, therefore, of vertex  $v^*$  together with an independent set of vertices from the graph  $G - \{v^*\} - Nbhd(v^*)$ .

Now consider an independent set  $S$  that doesn’t contain vertex  $v^*$ . In that case the set  $S$  is simply an independent set in the smaller graph  $G - \{v^*\}$ .

We now have all of the ingredients of a recursive algorithm. Suppose we have found the two numbers  $maxset(G - \{v^*\})$  and  $maxset(G - \{v^*\} - Nhd(v^*))$ . Then, from the discussion above, we have the relation

$$maxset(G) = \max\{maxset(G - \{v^*\}), 1 + maxset(G - \{v^*\} - Nhd(v^*))\}.$$

We obtain the following recursive algorithm.

```
function maxset1(G);
{returns the size of the largest independent set of
  vertices of G}
if G has no edges
  then maxset1 := |V(G)|
  else
    choose some nonisolated vertex v* of G;
    n1 := maxset1(G - {v*});
    n2 := maxset1(G - {v*} - Nhd(v*));
    maxset1 := max(n1, 1 + n2)
end.{maxset1}
```

**Example:**

Here is an example of a graph *G* and the result of applying the *maxset1* algorithm to it. Let the graph *G* be a 5-cycle. That is, it has 5 vertices and its edges are (1, 2), (2, 3), (3, 4), (4, 5), (1, 5). What are the two graphs on which the algorithm calls itself recursively?

Suppose we select vertex number 1 as the chosen vertex *v* in the algorithm. Then *G* - {1} and *G* - {1} - *Nhd*(1) are respectively the two graphs shown in Fig. 2.3.2.



**Fig. 2.3.2:**  $G - \{1\}$                        $G - \{1\} - Nhd(1)$

The reader should now check that the size of the largest independent set of *G* is equal to the larger of the two numbers  $maxset1(G - \{1\})$ ,  $1 + maxset1(G - \{1\} - Nhd(1))$  in this example.

Of course the creation of these two graphs from the original input graph is just the beginning of the story, as far as the computation is concerned. Unbeknownst to the programmer, who innocently wrote the recursive routine *maxset1* and then sat back to watch, the compiler will go ahead with the computation by generating a tree-full of graphs. In Fig. 2.3.3 we show the collection of all of the graphs that the compiler might generate while executing a single call to *maxset1* on the input graph of this example. In each case, the graph that is below and to the left of a given one is the one obtained by deleting a single vertex, and the one below and to the right of each graph is obtained by deleting a single vertex and its entire neighborhood.

Now we are going to study the complexity of *maxset1*. The results will be sufficiently depressing that we will then think about how to speed up the algorithm, and we will succeed in doing that to some extent.

To open the discussion, let's recall that in Chapter 0 it was pointed out that the complexity of a calculation is usefully expressed as a function of the number of bits of input data. In problems about graphs, however, it is more natural to think of the amount of labor as a function of *n*, the number of vertices of the graph. In problems about matrices it is more natural to use *n*, the size of the matrix, and so forth.

Do these distinctions alter the classification of problems into 'polynomial time do-able' vs. 'hard'? Take the graph problems, for instance. How many bits of input data does it take to describe a graph? Well, certainly we can march through the entire list of  $n(n - 1)/2$  pairs of vertices and check off the ones that are actually edges in the input graph to the problem. Hence we can describe a graph to a computer by making



algorithm that one might think of for this problem, which is to examine every single subset of the vertices of  $G$  and ask if it is an independent set or not. That algorithm would take  $\Theta(2^n)$  time units because there are  $2^n$  subsets of vertices to look at. Hence we have traded in a  $2^n$  for a  $1.619^n$  by being a little bit cagey about the algorithm. Can we do still better?

There have in fact been a number of improvements of the basic *maxset1* algorithm worked out. Of these the most successful is perhaps the one of Tarjan and Trojanowski that is cited in the bibliography at the end of this chapter. We are not going to work out all of those ideas here, but instead we will show what kind of improvements on the basic idea will help us to do better in the time estimate.

We can obviously do better if we choose  $v^*$  in such a way as to be certain that it has at least *two* neighbors. If we were to do that then although we wouldn't affect the number of vertices of  $G - \{v^*\}$  (always  $n - 1$ ) we would at least reduce the number of vertices of  $G - \{v^*\} - N_{bhd}(v^*)$  as much as possible.

So, as our next thought, we might replace the instruction 'choose some nonisolated vertex  $v^*$  of  $G$ ' in *maxset1* by an instruction 'choose some vertex  $v^*$  of  $G$  that has at least two neighbors.' Then we could be quite certain that  $G - \{v^*\} - N_{bhd}(v^*)$  would have at most  $n - 3$  vertices.

What if there isn't any such vertex in the graph  $G$ ? Then  $G$  would contain only vertices with 0 or 1 neighbors. Such a graph  $G$  would be a collection of  $E$  disjoint edges together with a number  $m$  of isolated vertices. The size of the largest independent set of vertices in such a graph is easy to find. A maximum independent set contains one vertex from each of the  $E$  edges and it contains all  $m$  of the isolated vertices. Hence in this case,  $maxset = E + m = |V(G)| - |E(G)|$ , and we obtain a second try at a good algorithm in the following form.

```

procedure maxset2( $G$ );
{returns the size of the largest independent set of
 vertices of  $G$ }
if  $G$  has no vertex of degree  $\geq 2$ 
then  $maxset2 := |V(G)| - |E(G)|$ 
else
  choose a vertex  $v^*$  of degree  $\geq 2$ ;
   $n_1 := maxset2(G - \{v^*\})$ ;
   $n_2 := maxset2(G - \{v^*\} - N_{bhd}(v^*))$ ;
   $maxset2 := max(n_1, 1 + n_2)$ 
end.{maxset2}

```

How much have we improved the complexity estimate? If we apply to *maxset2* the reasoning that led to (2.3.2) we find

$$f(n) \leq cn^2 + f(n-1) + f(n-3) \quad (f(0) = 0; n = 2, 3, \dots), \quad (2.3.3)$$

where  $f(n)$  is once more the worst-case time bound for graphs of  $n$  vertices.

Just as before, (2.3.3) is a recurrent inequality of the form that was studied at the end of section 1.4, in theorem 1.4.1. Using the conclusion of that theorem, we find from (2.3.3) that  $f(n) = O((c + \epsilon)^n)$  where  $c = 1.46557\dots$  is the positive root of the equation  $c^3 = c^2 + 1$ .

The net result of our effort to improve *maxset1* to *maxset2* has been to reduce the running-time bound from  $O(1.619^n)$  to  $O(1.47^n)$ , which isn't a bad day's work. In the exercises below we will develop *maxset3*, whose running time will be  $O(1.39^n)$ . The idea will be that since in *maxset2* we were able to insure that  $v^*$  had at least two neighbors, why not try to insure that  $v^*$  has at least 3 of them?

As long as we have been able to reduce the time bound more and more by insuring that the selected vertex has lots of neighbors, why don't we keep it up, and insist that  $v^*$  should have 4 or more neighbors? Regrettably the method runs out of steam precisely at that moment. To see why, ask what the 'trivial case' would then look like. We would be working on a graph  $G$  in which no vertex has more than 3 neighbors. Well, what 'trivialthing' shall we do, in this 'trivial case'?

The fact is that there isn't any way of finding the maximum independent set in a graph where all vertices have  $\leq 3$  neighbors that's any faster than the general methods that we've already discussed. In fact, if one could find a fast method for that restricted problem it would have extremely important consequences, because we would then be able to do all graphs rapidly, not just those special ones.

We will learn more about this phenomenon in Chapter 5, but for the moment let's leave just the observation that the general problem of *maxset* turns out to be no harder than the special case of *maxset* in which no vertex has more than 3 neighbors.

Aside from the complexity issue, the algorithm *maxset* has shown how recursive ideas can be used to transform questions about graphs to questions about smaller graphs.

Here's another example of such a situation. Suppose  $G$  is a graph, and that we have a certain supply of colors available. To be exact, suppose we have  $K$  colors. We can then attempt to *color* the vertices of  $G$  properly in  $K$  colors (see section 1.6).

If we don't have enough colors, and  $G$  has lots of edges, this will not be possible. For example, suppose  $G$  is the graph of Fig. 2.3.4, and suppose we have just 3 colors available. Then there is no way to color the vertices without ever finding that both endpoints of some edge have the same color. On the other hand, if we have four colors available then we can do the job.

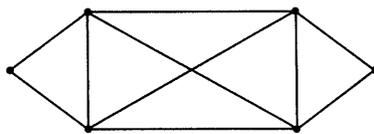


Fig. 2.3.4

There are many interesting computational and theoretical problems in the area of coloring of graphs. Just for its general interest, we are going to mention the four-color theorem, and then we will turn to a study of some of the computational aspects of graph coloring.

First, just for general cultural reasons, let's slow down for a while and discuss the relationship between graph colorings in general and the four-color problem, even though it isn't directly relevant to what we're doing.

The original question was this. Suppose that a delegation of Earthlings were to visit a distant planet and find there a society of human beings. Since that race is well known for its squabbling habits, you can be sure that the planet will have been carved up into millions of little countries, each with its own ruling class, system of government, etc., and of course, all at war with each other. The delegation wants to escape quickly, but before doing so it draws a careful map of the 5,000,000 countries into which the planet has been divided. To make the map easier to read, the countries are then colored in such a way that whenever two countries share a stretch of border they are of two different colors. Surprisingly, it was found that the coloring could be done using only red, blue, yellow and green.

It was noticed over 100 years ago that no matter how complicated a map is drawn, and no matter how many countries are involved, it seems to be possible to color the countries in such a way that

- (a) every pair of countries that have a common stretch of border have different colors and
- (b) no more than *four* colors are used in the entire map.

It was then conjectured that four colors are always sufficient for the proper coloring of the countries of any map at all. Settling this conjecture turned out to be a very hard problem. It was finally solved in 1976 by K. Appel and W. Haken\* by means of an extraordinary proof with two main ingredients. First they showed how to reduce the general problem to only a *finite* number of cases, by a mathematical argument. Then, since the 'finite number' was over 1800, they settled all of those cases with quite a lengthy computer calculation. So now we have the 'Four Color Theorem,' which asserts that no matter how we carve up the plane or the sphere into countries, we will always be able to color those countries with at most four colors so that countries with a common frontier are colored differently.

We can change the *map* coloring problem into a *graph* coloring problem as follows. Given a map. From the map we will construct a graph  $G$ . There will be a vertex of  $G$  corresponding to each country on the map. Two of these vertices will be connected by an edge of the graph  $G$  if the two countries that they correspond to have a common stretch of border (we keep saying 'stretch of border' to emphasize that if the two countries have just a single point in common they are allowed to have the same color). As an illustration

---

\* Every planar map is four colorable, *Bull. Amer. Math. Soc.*, **82** (1976), 711-712.

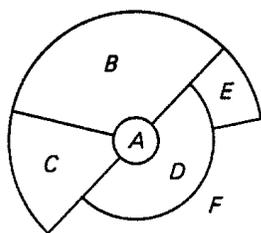


Fig. 2.3.5(a)

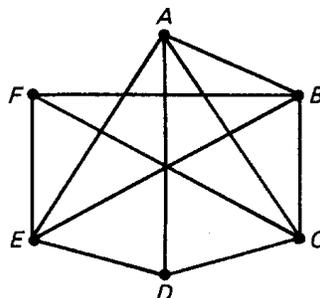


Fig. 2.3.5(b)

of this construction, we show in Fig. 2.3.5(a) a map of a distant planet, and in Fig. 2.3.5(b) the graph that results from the construction that we have just described.

By a ‘planar graph’ we mean a graph  $G$  that can be drawn in the plane in such a way that two edges never cross (except that two edges at the same vertex have that vertex in common). The graph that results from changing a map of countries into a graph as described above is always a planar graph. In Fig. 2.3.6(a) we show a planar graph  $G$ . This graph doesn’t look planar because two of its edges cross. However, that isn’t the graph’s fault, because with a little more care we might have drawn *the same graph* as in Fig. 2.3.6(b), in which its planarity is obvious. Don’t blame the graph if it doesn’t look planar. It might be planar anyway!

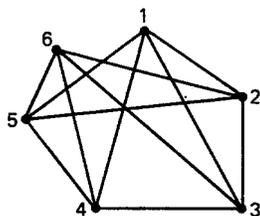


Fig. 2.3.6(a)

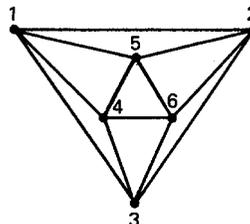


Fig. 2.3.6(b)

The question of recognizing whether a given graph is planar is itself a formidable problem, although the solution, due to J. Hopcroft and R. E. Tarjan,\* is an algorithm that makes the decision in *linear time*, *i.e.* in  $O(V)$  time for a graph of  $V$  vertices.

Although every *planar* graph can be properly colored in four colors, there are still all of those other graphs that are not planar to deal with. For any one of those graphs we can ask, if a positive integer  $K$  is given, whether or not its vertices can be  $K$ -colored properly.

As if that question weren’t hard enough, we might ask for even more detail, namely about the *number* of ways of properly coloring the vertices of a graph. For instance, if we have  $K$  colors to work with, suppose  $G$  is the *empty* graph  $\bar{K}_n$ , that is, the graph of  $n$  vertices that has no edges at all. Then  $G$  has quite a large number of proper colorings,  $K^n$  of them, to be exact. Other graphs of  $n$  vertices have fewer proper colorings than that, and an interesting computational question is to count the proper colorings of a given graph.

We will now find a recursive algorithm that will answer this question. Again, the complexity of the algorithm will be exponential, but as a small consolation we note that no polynomial time algorithm for this problem is known.

Choose an edge  $e$  of the graph, and let its endpoints be  $v$  and  $w$ . Now delete the edge  $e$  from the graph, and let the resulting graph be called  $G - \{e\}$ . Then we will distinguish two kinds of proper colorings of  $G - \{e\}$ : those in which vertices  $v$  and  $w$  have the same color and those in which  $v$  and  $w$  have different colors. Obviously the number of proper colorings of  $G - \{e\}$  in  $K$  colors is the sum of the numbers of colorings of each of these two kinds.

\* Efficient planarity testing, *J. Assoc. Comp. Mach.* **21** (1974), 549-568.

Consider the proper colorings in which vertices  $v$  and  $w$  have the same color. We claim that the number of such colorings is equal to the number of *all* colorings of a certain new graph  $G/\{e\}$ , whose construction we now describe:

The vertices of  $G/\{e\}$  consist of the vertices of  $G$  other than  $v$  or  $w$  and one new vertex that we will call ' $vw$ ' (so  $G/\{e\}$  will have one less vertex than  $G$  has).

Now we describe the *edges* of  $G/\{e\}$ . First, if  $a$  and  $b$  are two vertices of  $G/\{e\}$  neither of which is the new vertex ' $vw$ ', then  $(a, b)$  is an edge of  $G/\{e\}$  if and only if it is an edge of  $G$ . Second,  $(vw, b)$  is an edge of  $G/\{e\}$  if and only if either  $(v, b)$  or  $(w, b)$  (or both) is an edge of  $G$ .

We can think of this as 'collapsing' the graph  $G$  by imagining that the edges of  $G$  are elastic bands, and that we squeeze vertices  $v$  and  $w$  together into a single vertex. The result is  $G/\{e\}$  (anyway, it is if we replace any resulting double bands by single ones!).

In Fig. 2.3.7(a) we show a graph  $G$  of 7 vertices and a chosen edge  $e$ . The two endpoints of  $e$  are  $v$  and  $w$ . In Fig. 2.3.7(b) we show the graph  $G/\{e\}$  that is the result of the construction that we have just described.

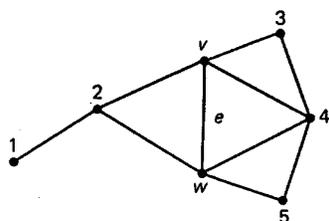


Fig. 2.3.7(a)

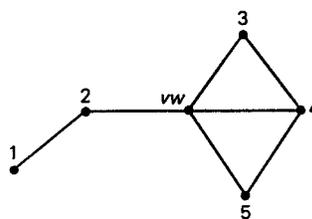


Fig. 2.3.7(b)

The point of the construction is the following

**Lemma 2.3.1.** *Let  $v$  and  $w$  be two vertices of  $G$  such that  $e = (v, w) \in E(G)$ . Then the number of proper  $K$ -colorings of  $G - \{e\}$  in which  $v$  and  $w$  have the same color is equal to the number of all proper colorings of the graph  $G/\{e\}$ .*

**Proof:** Suppose  $G/\{e\}$  has a proper  $K$ -coloring. Color the vertices of  $G - \{e\}$  itself in  $K$  colors as follows. Every vertex of  $G - \{e\}$  other than  $v$  or  $w$  keeps the same color that it has in the coloring of  $G/\{e\}$ . Vertex  $v$  and vertex  $w$  each receive the color that vertex  $vw$  has in the coloring of  $G/\{e\}$ . Now we have a  $K$ -coloring of the vertices of  $G - \{e\}$ .

It is a proper coloring because if  $f$  is any edge of  $G - \{e\}$  then the two endpoints of  $f$  have different colors. Indeed, this is obviously true if neither endpoint of  $f$  is  $v$  or  $w$  because the coloring of  $G/\{e\}$  was a proper one. There remains only the case where one endpoint of  $f$  is, say,  $v$  and the other one is some vertex  $x$  other than  $v$  or  $w$ . But then the colors of  $v$  and  $x$  must be different because  $vw$  and  $x$  were joined in  $G/\{e\}$  by an edge, and therefore must have gotten different colors there. ■

To get back to the main argument, we were trying to compute the number of proper  $K$ -colorings of  $G - \{e\}$ . We observed that in any  $K$ -coloring  $v$  and  $w$  have either the same or different colors. We have shown that the number of colorings in which they receive the same color is equal to the number of all proper colorings of a certain smaller (one less vertex) graph  $G/\{e\}$ . It remains to look at the case where vertices  $v$  and  $w$  receive different colors.

**Lemma 2.3.2.** *Let  $e = (v, w)$  be an edge of  $G$ . Then the number of proper  $K$ -colorings of  $G - \{e\}$  in which  $v$  and  $w$  have different colors is equal to the number of all proper  $K$ -colorings of  $G$  itself.*

**Proof:** Obvious (isn't it?). ■

Now let's put together the results of the two lemmas above. Let  $P(K; G)$  denote the number of ways of properly coloring the vertices of a given graph  $G$ . Then lemmas 2.3.1 and 2.3.2 assert that

$$P(K; G - \{e\}) = P(K; G/\{e\}) + P(K; G)$$

or if we solve for  $P(K; G)$ , then we have

$$P(K; G) = P(K; G - \{e\}) - P(K; G/\{e\}). \quad (2.3.4)$$

The quantity  $P(K; G)$ , the number of ways of properly coloring the vertices of a graph  $G$  in  $K$  colors, is called *the chromatic polynomial of  $G$* .

We claim that it is, in fact, a polynomial in  $K$  of degree  $|V(G)|$ . For instance, if  $G$  is the complete graph of  $n$  vertices then obviously  $P(K, G) = K(K - 1) \cdots (K - n + 1)$ , and that is indeed a polynomial in  $K$  of degree  $n$ .

**Proof of claim:** The claim is certainly true if  $G$  has just one vertex. Next suppose the assertion is true for graphs of  $< V$  vertices, then we claim it is true for graphs of  $V$  vertices also. This is surely true if  $G$  has  $V$  vertices and no edges at all. Hence, suppose it is true for all graphs of  $V$  vertices and fewer than  $E$  edges, and let  $G$  have  $V$  vertices and  $E$  edges. Then (2.3.4) implies that  $P(K; G)$  is a polynomial of the required degree  $V$  because  $G - \{e\}$  has fewer edges than  $G$  does, so its chromatic polynomial is a polynomial of degree  $V$ .  $G/\{e\}$  has fewer vertices than  $G$  has, and so  $P(K; G/\{e\})$  is a polynomial of lower degree. The claim is proved, by induction. ■

Equation (2.3.4) gives a recursive algorithm for computing the chromatic polynomial of a graph  $G$ , since the two graphs that appear on the right are both ‘smaller’ than  $G$ , one in the sense that it has fewer edges than  $G$  has, and the other in that it has fewer vertices. The algorithm is the following.

```
function chrompoly( $G$ :graph): polynomial;
{computes the chromatic polynomial of a graph  $G$ }
if  $G$  has no edges then chrompoly:= $K^{|V(G)|}$ 
else
  choose an edge  $e$  of  $G$ ;
  chrompoly:=chrompoly( $G - \{e\}$ )-chrompoly( $G/\{e\}$ )
end.{chrompoly}
```

Next we are going to look at the complexity of the algorithm *chrompoly* (we will also refer to it as ‘the delete-and-identify’ algorithm). The graph  $G$  can be input in any one of a number of ways. For example, we might input the full list of edges of  $G$ , as a list of pairs of vertices.

The first step of the computation is to choose the edge  $e$  and to create the edge list of the graph  $G - \{e\}$ . The latter operation is trivial, since all we have to do is to ignore one edge in the list.

Next we call *chrompoly* on the graph  $G - \{e\}$ .

The third step is to create the edge list of the collapsed graph  $G/\{e\}$  from the edge list of  $G$  itself. That involves some work, but it is rather routine, and its cost is linear in the number of edges of  $G$ , say  $c|E(G)|$ .

Finally we call *chrompoly* on the graph  $G/\{e\}$ .

Let  $F(V, E)$  denote the maximum cost of calling *chrompoly* on any graph of at most  $V$  vertices and at most  $E$  edges. Then we see at once that

$$F(V, E) \leq F(V, E - 1) + cE + F(V - 1, E - 1) \quad (2.3.5)$$

together with  $F(V, 0) = 0$ . If we put, successively,  $E = 1, 2, 3$ , we find that  $F(V, 1) \leq c$ ,  $F(V, 2) \leq 4c$ , and  $F(V, 3) \leq 11c$ . Hence we seek a solution of (2.3.5) in the form  $F(V, E) \leq f(E)c$ , and we quickly find that if

$$f(E) = 2f(E - 1) + E \quad (f(0) = 0) \quad (2.3.6)$$

then we will have such a solution.

Since (2.3.6) is a first-order difference equation of the form (1.4.5), we find that

$$\begin{aligned} f(E) &= 2^E \sum_{j=0}^E j 2^{-j} \\ &\sim 2^{E+1}. \end{aligned} \quad (2.3.7)$$

The last ‘ $\sim$ ’ follows from the evaluation  $\sum j2^{-j} = 2$  that we discussed in section 1.3.

To summarize the developments so far, then, we have found out that the chromatic polynomial of a graph can be computed recursively by an algorithm whose cost is  $O(2^E)$  for graphs of  $E$  edges. This is exponential cost, and such computations are prohibitively expensive except for graphs of very modest numbers of edges.

Of course the mere fact that our proved time estimate is  $O(2^E)$  doesn’t necessarily mean that the algorithm can be that slow, because maybe our complexity analysis wasn’t as sharp as it might have been. However, consider the graph  $G(s, t)$  that consists of  $s$  disjoint edges and  $t$  isolated vertices, for a total of  $2s + t$  vertices altogether. If we choose an edge of  $G(s, t)$  and delete it, we get  $G(s - 1, t + 2)$ , whereas the graph  $G/\{e\}$  is  $G(s - 1, t + 1)$ . Each of these two new graphs has  $s - 1$  edges.

We might imagine arranging the computation so that the extra isolated vertices will be ‘free,’ *i.e.*, will not cost any additional labor. Then the work that we

do on  $G(s, t)$  will depend only on  $s$ , and will be twice as much as the work we do on  $G(s - 1, \cdot)$ . Therefore  $G(s, t)$  will cost at least  $2^s$  operations, and our complexity estimate wasn’t a mirage, there really are graphs that make the algorithm do an amount  $2^{|E(G)|}$  of work.

Considering the above remarks it may be surprising that there is a slightly different approach to the complexity analysis that leads to a time bound (for the same algorithm) that is a bit sharper than  $O(2^E)$  in many cases (the work of the complexity analyst is never finished!). Let’s look at the algorithm *chrompoly* in another way.

For a graph  $G$  we can define a number  $\gamma(G) = |V(G)| + |E(G)|$ , which is rather an odd kind of thing to define, but it has a nice property with respect to this algorithm, namely that whatever  $G$  we begin with, we will find that

$$\gamma(G - \{e\}) = \gamma(G) - 1; \quad \gamma(G/\{e\}) \leq \gamma(G) - 2. \quad (2.3.8)$$

Indeed, if we delete the edge  $e$  then  $\gamma$  must drop by 1, and if we collapse the graph on the edge  $e$  then we will have lost one vertex and at least one edge, so  $\gamma$  will drop by at least 2.

Hence, if  $h(\gamma)$  denotes the maximum amount of labor that *chrompoly* does on any graph  $G$  for which

$$|V(G)| + |E(G)| \leq \gamma \quad (2.3.9)$$

then we claim that

$$h(\gamma) \leq h(\gamma - 1) + h(\gamma - 2) \quad (\gamma \geq 2). \quad (2.3.10)$$

Indeed, if  $G$  is a graph for which (2.3.9) holds, then if  $G$  has any edges at all we can do the delete-and-identify step to prove that the labor involved in computing the chromatic polynomial of  $G$  is at most the quantity on the right side of (2.3.10). Else, if  $G$  has no edges then the labor is 1 unit, which is again at most equal to the right side of (2.3.10), so the result (2.3.10) follows.

With the initial conditions  $h(0) = h(1) = 1$  the solution of the recurrent inequality (2.3.10) is obviously the relation  $h(\gamma) \leq F_\gamma$ , where  $F_\gamma$  is the Fibonacci number. We have thereby proved that the time complexity of the algorithm *chrompoly* is

$$\begin{aligned} O(F_{|V(G)|+|E(G)|}) &= O\left(\left(\frac{1+\sqrt{5}}{2}\right)^{|V(G)|+|E(G)|}\right) \\ &= O(1.62^{|V(G)|+|E(G)|}). \end{aligned} \quad (2.3.11)$$

This analysis does not, of course, contradict the earlier estimate, but complements it. What we have shown is that the labor involved is always

$$O\left(\min(2^{|E(G)|}, 1.62^{|V(G)|+|E(G)|})\right). \quad (2.3.12)$$

On a graph with ‘few’ edges relative to its number of vertices (how few?) the first quantity in the parentheses in (2.3.12) will be the smaller one, whereas if  $G$  has more edges, then the second term is the smaller one. In either case the overall judgment about the speed of the algorithm (it’s slow!) remains.

### Exercises for section 2.3

1. Let  $G$  be a cycle of  $n$  vertices. What is the size of the largest independent set of vertices in  $V(G)$ ?
2. Let  $G$  be a path of  $n$  vertices. What is the size of the largest independent set of vertices in  $V(G)$ ?
3. Let  $G$  be a connected graph in which every vertex has degree 2. What must such a graph consist of? Prove.
4. Let  $G$  be a connected graph in which every vertex has degree  $\leq 2$ . What must such a graph look like?
5. Let  $G$  be a not-necessarily-connected graph in which every vertex has degree  $\leq 2$ . What must such a graph look like? What is the size of the largest independent set of vertices in such a graph? How long would it take you to calculate that number for such a graph  $G$ ? How would you do it?
6. Write out algorithm *maxset3*, which finds the size of the largest independent set of vertices in a graph. Its trivial case will occur if  $G$  has no vertex of degree  $\geq 3$ . Otherwise, it will choose a vertex  $v^*$  of degree  $\geq 3$  and proceed as in *maxset2*.
7. Analyze the complexity of your algorithm *maxset3* from exercise 6 above.
8. Use (2.3.4) to prove by induction that  $P(K; G)$  is a polynomial in  $K$  of degree  $|V(G)|$ . Then show that if  $G$  is a tree then  $P(K; G) = K(K - 1)^{|V(G)|-1}$ .
9. Write out an algorithm that will change the vertex adjacency matrix of a graph  $G$  to the vertex adjacency matrix of the graph  $G/\{e\}$ , where  $e$  is a given edge of  $G$ .
10. How many edges must  $G$  have before the second quantity inside the ‘ $O$ ’ in (2.3.12) is the smaller of the two?
11. Let  $\alpha(G)$  be the size of the largest independent set of vertices of a graph  $G$ , let  $\chi(G)$  be its chromatic number, and let  $n = |V(G)|$ . Show that, for every  $G$ ,  $\alpha(G) \geq n/\chi(G)$ .

### 2.4 Fast matrix multiplication

Everybody knows how to multiply two  $2 \times 2$  matrices. If we want to calculate

$$\begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} \quad (2.4.1)$$

then, ‘of course,’

$$c_{i,j} = \sum_{k=1}^2 a_{i,k} b_{k,j} \quad (i, j = 1, 2). \quad (2.4.2)$$

Now look at (2.4.2) a little more closely. In order to calculate each one of the 4  $c_{i,j}$ ’s we have to do 2 multiplications of numbers. The cost of multiplying two  $2 \times 2$  matrices is therefore 8 multiplications of numbers. If we measure the cost in units of additions of numbers, the cost is 4 such additions. Hence, the matrix multiplication method that is shown in (2.4.1) has a complexity of 8 multiplications of numbers and 4 additions of numbers.

This may seem rather unstartling, but the best ideas often have humble origins.

Suppose we could find another way of multiplying two  $2 \times 2$  matrices in which the cost was only 7 multiplications of numbers, together with more than 4 additions of numbers. Would that be a cause for dancing in the streets, or would it be just a curiosity, of little importance? In fact, it would be extremely important, and the consequences of such a step were fully appreciated only in 1969 by V. Strassen, to whom the ideas that we are now discussing are due.\*

What we’re going to do next in this section is the following:

- (a) describe another way of multiplying two  $2 \times 2$  matrices in which the cost will be only 7 multiplications of numbers plus a bunch of additions of numbers, and
- (b) convince you that it was worth the trouble.

The usefulness of the idea stems from the following amazing fact: if two  $2 \times 2$  matrices can be multiplied with only 7 multiplications of numbers, then two  $N \times N$  matrices can be multiplied using only  $O(N^{2.81\dots})$

---

\* V. Strassen, Gaussian elimination is not optimal, *Numerische Math.* **13** (1969), 354-6.

multiplications of numbers instead of the  $N^3$  such multiplications that the usual method involves (the number ‘2.81...’ is  $\log_2 7$ ).

In other words, if we can reduce the number of multiplications of numbers that are needed to multiply two  $2 \times 2$  matrices, then that improvement will show up in the *exponent* of  $N$  when we measure the complexity of multiplying two  $N \times N$  matrices. The reason, as we will see, is that the little improvement will be pyramided by numerous recursive calls to the  $2 \times 2$  procedure— but we get ahead of the story.

Now let’s write out another way to do the  $2 \times 2$  matrix multiplication that is shown in (2.4.1). Instead of doing it *à là* (2.4.2), try the following 11-step approach.

First compute, from the input  $2 \times 2$  matrices shown in (2.4.1), the following 7 quantities:

$$\begin{aligned}
 I &= (a_{12} - a_{22}) \times (b_{21} + b_{22}) \\
 II &= (a_{11} + a_{22}) \times (b_{11} + b_{22}) \\
 III &= (a_{11} - a_{21}) \times (b_{11} + b_{12}) \\
 IV &= (a_{11} + a_{12}) \times b_{22} \\
 V &= a_{11} \times (b_{12} - b_{22}) \\
 VI &= a_{22} \times (b_{21} - b_{11}) \\
 VII &= (a_{21} + a_{22}) \times b_{11}
 \end{aligned} \tag{2.4.3}$$

and then calculate the 4 entries of the product matrix  $C = AB$  from the 4 formulas

$$\begin{aligned}
 c_{11} &= I + II - IV + VI \\
 c_{12} &= IV + V \\
 c_{21} &= VI + VII \\
 c_{22} &= II - III + V - VII.
 \end{aligned} \tag{2.4.4}$$

The first thing to notice about this seemingly overelaborate method of multiplying  $2 \times 2$  matrices is that only 7 multiplications of numbers are used (count the ‘ $\times$ ’ signs in (2.4.3)). ‘Well yes,’ you might reply, ‘but 18 additions are needed, so where is the gain?’

It will turn out that multiplications are more important than additions, not because computers can do them faster, but because when the routine is called *recursively* each ‘ $\times$ ’ operation will turn into a multiplication of two big matrices whereas each ‘ $\pm$ ’ will turn into an addition or subtraction of two big matrices, and that’s much cheaper.

Next we’re going to describe how Strassen’s method (equations (2.4.3), (2.4.4)) of multiplying  $2 \times 2$  matrices can be used to speed up multiplications of  $N \times N$  matrices. The basic idea is that we will partition each of the large matrices into four smaller ones and multiply them together using (2.4.3), (2.4.4).

Suppose that  $N$  is a power of 2, say  $N = 2^n$ , and let there be given two  $N \times N$  matrices,  $A$  and  $B$ . We imagine that  $A$  and  $B$  have each been partitioned into four  $2^{n-1} \times 2^{n-1}$  matrices, and that the product matrix  $C$  is similarly partitioned. Hence we want to do the matrix multiplication that is indicated by

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \tag{2.4.5}$$

where now each of the capital letters represents a  $2^{n-1} \times 2^{n-1}$  matrix.

To do the job in (2.4.5) we use exactly the 11 formulas that are shown in (2.4.3) and (2.4.4), except that the lower-case letters are now all upper case. Suddenly we very much appreciate the reduction of the number of ‘ $\times$ ’ signs because it means one less multiplication of large matrices, and we don’t so much mind that it has been replaced by 10 more ‘ $\pm$ ’ signs, at least not if  $N$  is very large.

This yields the following recursive procedure for multiplying large matrices.

```

function MatrProd(A, B: matrix; N:integer):matrix;
{MatrProd is AB, where A and B are  $N \times N$ }
{uses Strassen method}
if N is not a power of 2 then
  border A and B by rows and columns of 0's until
  their size is the next power of 2 and change N;
if N = 1 then MatrProd := AB
  else
    partition A and B as shown in (2.4.5);
    I := MatrProd(A11 - A22, B21 + B22, N/2);
    II := MatrProd(A11 + A22, B11 + B22, N/2);
    etc. etc., through all 11 of the formulas
    shown in (2.4.3), (2.4.4), ending with ...
    C22 := II - III + V - VII
  end.{MatrProd}

```

Note that this procedure calls itself recursively 7 times. The plus and minus signs in the program each represent an addition or subtraction of two *matrices*, and therefore each one of them involves a call to a matrix addition or subtraction procedure (just the usual method of adding, nothing fancy!). Therefore the function *MatrProd* makes 25 calls, 7 of which are recursively to itself, and 18 of which are to a matrix addition/subtraction routine.

We will now study the complexity of the routine in two ways. We will count the number of multiplications of numbers that are needed to multiply two  $2^n \times 2^n$  matrices using *MatrProd* (call that number  $f(n)$ ), and then we will count the number of additions of numbers (call it  $g(n)$ ) that *MatrProd* needs in order to multiply two  $2^n \times 2^n$  matrices.

The multiplications of numbers are easy to count. *MatrProd* calls itself 7 times, in each of which it does exactly  $f(n-1)$  multiplications of numbers, hence  $f(n) = 7f(n-1)$  and  $f(0) = 1$  (why?). Therefore we see that  $f(n) = 7^n$  for all  $n \geq 0$ . Hence *MatrProd* does  $7^n$  multiplications of numbers in order to do one multiplication of  $2^n \times 2^n$  matrices.

Let's take the last sentence in the above paragraph and replace ' $2^n$ ' by  $N$  throughout. It then tells us that *MatrProd* does  $7^{\log N / \log 2}$  multiplications of numbers in order to do one multiplication of  $N \times N$  matrices. Since  $7^{\log N / \log 2} = N^{\log 7 / \log 2} = N^{2.81\dots}$ , we see that Strassen's method uses only  $O(N^{2.81})$  multiplications of numbers, in place of the  $N^3$  such multiplications that are required by the usual formulas.

It remains to count the additions/subtractions of numbers that are needed by *MatrProd*.

In each of its 7 recursive calls to itself *MatrProd* does  $g(n-1)$  additions of numbers. In each of its 18 calls to the procedure that adds or subtracts matrices it does a number of additions of numbers that is equal to the square of the size of the matrices that are being added or subtracted. That size is  $2^{n-1}$ , so each of the 18 such calls does  $2^{2n-2}$  additions of numbers. It follows that  $g(0) = 0$  and for  $n \geq 1$  we have

$$\begin{aligned}
 g(n) &= 7g(n-1) + 18 \cdot 4^{n-1} \\
 &= 7g(n-1) + \frac{9}{2}4^n.
 \end{aligned}$$

We follow the method of section 1.4 on this first-order linear difference equation. Hence we make the change of variable  $g(n) = 7^n y_n$  ( $n \geq 0$ ) and we find that  $y_0 = 0$  and for  $n \geq 1$ ,

$$y_n = y_{n-1} + \frac{9}{2}(4/7)^n.$$

If we sum over  $n$  we obtain

$$\begin{aligned}
 y_n &= \frac{9}{2} \sum_{j=1}^n (4/7)^j \\
 &\leq \frac{9}{2} \sum_{j=0}^{\infty} (4/7)^j \\
 &= 21/2.
 \end{aligned}$$

Finally,  $g(n) = 7^n y_n \leq (10.5)7^n = O(7^n)$ , and this is  $O(N^{2.81})$  as before. This completes the proof of

**Theorem 2.4.1.** *In Strassen's method of fast matrix multiplication the number of multiplications of numbers, of additions of numbers and of subtractions of numbers that are needed to multiply together two  $N \times N$  matrices are each  $O(N^{2.81})$  (in contrast to the  $\Theta(N^3)$  of the conventional method). ■*

In the years that have elapsed since Strassen's original paper many researchers have been whittling away at the exponent of  $N$  in the complexity bounds. Several new, and more elaborate algorithms have been developed, and the exponent, which was originally 3, has progressed downwards through 2.81 to values below 2.5. It is widely believed that the true minimum exponent is  $2 + \epsilon$ , *i.e.*, that two  $N \times N$  matrices can be multiplied in time  $O(N^{2+\epsilon})$ , but there seems to be a good deal of work to be done before that result can be achieved.

### Exercises for section 2.4

1. Suppose we could multiply together two  $3 \times 3$  matrices with only 22 multiplications of numbers. How fast, recursively, would we then be able to multiply two  $N \times N$  matrices?
2. (cont.) With what would the '22' in problem 1 above have to be replaced in order to achieve an improvement over Strassen's algorithm given in the text?
3. (cont.) Still more generally, with how few multiplications would we have to be able to multiply two  $M \times M$  matrices in order to insure that recursively we would then be able to multiply two  $N \times N$  matrices faster than the method given in this section?
4. We showed in the text that if  $N$  is a power of 2 then two  $N \times N$  matrices can be multiplied in at most time  $CN^{\log_2 7}$ , where  $C$  is a suitable constant. Prove that if  $N$  is not a power of 2 then two  $N \times N$  matrices can be multiplied in time at most  $7CN^{\log_2 7}$ .

### 2.5 The discrete Fourier transform

It is a lot easier to multiply two numbers than to multiply two polynomials.

If you should want to multiply two polynomials  $f$  and  $g$ , of degrees 77 and 94, respectively, you are in for a lot of work. To calculate just one coefficient of the product is already a lot of work. Think about the calculation of the coefficient of  $x^{50}$  in the product, for instance, and you will see that about 50 numbers must be multiplied together and added in order to calculate just that one coefficient of  $fg$ , and there are 171 other coefficients to calculate!

Instead of calculating the *coefficients* of the product  $fg$  it would be much easier just to calculate the *values* of the product at, say, 172 points. To do that we could just multiply the values of  $f$  and of  $g$  at each of those points, and after a total cost of 172 multiplications we would have the values of the product.

The values of the product polynomial at 172 distinct points determine that polynomial completely, so that sequence of values *is* the answer. It's just that we humans prefer to see polynomials given by means of their coefficients instead of by their values.

The Fourier transform, that is the subject of this section, is a method of converting from one representation of a polynomial to another. More exactly, it converts from the sequence of *coefficients* of the polynomial to the sequence of *values* of that polynomial at a certain set of points. Ease of converting between these two representations of a polynomial is vitally important for many reasons, including multiplication of polynomials, high precision integer arithmetic in computers, creation of medical images in CAT scanners and NMR scanners, etc.

Hence, in this section we will study the discrete Fourier transform of a finite sequence of numbers, methods of calculating it, and some applications.

This is a computational problem which at first glance seems very simple. What we're asked to do, basically, is to evaluate a polynomial of degree  $n - 1$  at  $n$  different points. So what could be so difficult about that?

If we just calculate the  $n$  values by brute force, we certainly won't need to do more than  $n$  multiplications of numbers to find each of the  $n$  values of the polynomial that we want, so we surely don't need more than  $O(n^2)$  multiplications altogether.

The interesting thing is that this particular problem is so important, and turns up in so many different applications, that it really pays to be very efficient about how the calculation is done. We will see in this section that if we use a fairly subtle method of doing this computation instead of the obvious method, then the work can be cut down from  $O(n^2)$  to  $O(n \log n)$ . In view of the huge arrays on which this program is often run, the saving is very much worthwhile.

One can think of the Fourier transform as being a way of changing the description, or *coding* of a polynomial, so we will introduce the subject by discussing it from that point of view.

Next we will discuss the obvious way of computing the transform.

Then we will describe the ‘Fast Fourier Transform’, which is a rather un-obvious, but very fast, method of computing the same creature.

Finally we will discuss an important application of the subject, to the fast multiplication of polynomials.

There are many different ways that might choose to describe (‘encode’) a particular polynomial. Take the polynomial  $f(t) = t(6 - 5t + t^2)$ , for instance. This can be uniquely described in any of the following ways (and a lot more).

It is the polynomial whose

- (i) *coefficients* are 0, 6, -5, 1 or whose
- (ii) *roots* are 0, 2 and 3, and whose highest coefficient is 1 or whose
- (iii) *values* at  $t = 0, 1, 2, 3$  are 0, 2, 0, 0, respectively, or whose
- (iv) *values* at the fourth-roots of unity  $1, i, -1, -i$  are 2,  $5 + 5i$ ,  $-12, 5 - 5i$ , or etc.

We want to focus on two of these ways of representing a polynomial. The first is by its coefficient sequence; the second is by its sequence of values at the  $n^{\text{th}}$  roots of unity, where  $n$  is 1 more than the degree of the polynomial. The process by which we pass from the coefficient sequence to the sequence of values at the roots of unity is called forming the *Fourier transform* of the coefficient sequence. To use the example above, we would say that the Fourier transform of the sequence

$$0, 6, -5, 1 \tag{2.5.1}$$

is the sequence

$$2, 5 + 5i, -12, 5 - 5i. \tag{2.5.2}$$

In general, if we are given a sequence

$$x_0, x_1, \dots, x_{n-1} \tag{2.5.3}$$

then we think of the polynomial

$$f(t) = x_0 + x_1 t + x_2 t^2 + \dots + x_{n-1} t^{n-1} \tag{2.5.4}$$

and we compute its values at the  $n^{\text{th}}$  roots of unity. These roots of unity are the numbers

$$\omega_j = e^{2\pi i j/n} \quad (j = 0, 1, \dots, n-1). \tag{2.5.5}$$

Consequently, if we calculate the values of the polynomial (2.5.4) at the  $n$  numbers (2.5.5), we find the Fourier transform of the given sequence (2.5.3) to be the sequence

$$\begin{aligned} f(\omega_j) &= \sum_{k=0}^{n-1} x_k \omega_j^k \\ &= \sum_{k=0}^{n-1} x_k e^{2\pi i j k/n} \quad (j = 0, 1, \dots, n-1). \end{aligned} \tag{2.5.6}$$

Before proceeding, the reader should pause for a moment and make sure that the fact that (2.5.1)-(2.5.2) is a special case of (2.5.3)-(2.5.6) is clearly understood. The Fourier transform of a sequence of  $n$  numbers is another sequence of  $n$  numbers, namely the sequence of values at the  $n^{\text{th}}$  roots of unity of the very same polynomial whose coefficients are the members of the original sequence.

The Fourier transform moves us from *coefficients* to *values at roots of unity*. Some good reasons for wanting to make that trip will appear presently, but for the moment, let's consider the computational side of the question, namely how to compute the Fourier transform efficiently.

We are going to derive an elegant and very speedy algorithm for the evaluation of Fourier transforms. The algorithm is called the Fast Fourier Transform (FFT) algorithm. In order to appreciate how fast it is, let's see how long it would take us to calculate the transform without any very clever procedure.

What we have to do is to compute the values of a given polynomial at  $n$  given points. How much work is required to calculate the value of a polynomial at *one* given point? If we want to calculate the value of the polynomial  $x_0 + x_1t + x_2t^2 + \dots + x_{n-1}t^{n-1}$  at exactly one value of  $t$ , then we can do (think how you would do it, before looking)

```
function value(x :coeff array; n:integer; t:complex);
{computes value := x0 + x1t + ... + xn-1tn-1}
value := 0;
for j := n - 1 to 0 step -1 do
    value := t · value + xj
end.{value}
```

This well-known algorithm (= 'synthetic division') for computing the value of a polynomial at a single point  $t$  obviously runs in time  $O(n)$ .

If we calculate the Fourier transform of a given sequence of  $n$  points by calling the function *value*  $n$  times, once for each point of evaluation, then obviously we are looking at a simple algorithm that requires  $\Theta(n^2)$  time.

With the FFT we will see that the whole job can be done in time  $O(n \log n)$ , and we will then look at some implications of that fact. To put it another way, the cost of calculating all  $n$  of the values of a polynomial  $f$  at the  $n^{\text{th}}$  roots of unity is much less than  $n$  times the cost of one such calculation.

First we consider the important case where  $n$  is a power of 2, say  $n = 2^r$ . Then the values of  $f$ , a polynomial of degree  $2^r - 1$ , at the  $(2^r)^{\text{th}}$  roots of unity are, from (2.5.6),

$$f(\omega_j) = \sum_{k=0}^{n-1} x_k \exp\{2\pi ijk/2^r\} \quad (j = 0, 1, \dots, 2^r - 1). \quad (2.5.7)$$

Let's break up the sum into two sums, containing respectively the terms where  $k$  is even and those where  $k$  is odd. In the first sum write  $k = 2m$  and in the second put  $k = 2m + 1$ . Then, for each  $j = 0, 1, \dots, 2^r - 1$ ,

$$\begin{aligned} f(\omega_j) &= \sum_{m=0}^{2^{r-1}-1} x_{2m} e^{2\pi ijm/2^{r-1}} + \sum_{m=0}^{2^{r-1}-1} x_{2m+1} e^{2\pi ij(2m+1)/2^r} \\ &= \sum_{m=0}^{2^{r-1}-1} x_{2m} e^{2\pi ijm/2^{r-1}} + e^{2\pi ij/2^r} \sum_{m=0}^{2^{r-1}-1} x_{2m+1} e^{2\pi ijm/2^{r-1}}. \end{aligned} \quad (2.5.8)$$

Something special just happened. Each of the two sums that appear in the last member of (2.5.8) is itself a Fourier transform, of a shorter sequence. The first sum is the transform of the array

$$x[0], x[2], x[4], \dots, x[2^r - 2] \quad (2.5.9)$$

and the second sum is the transform of

$$x[1], x[3], x[5], \dots, x[2^r - 1]. \quad (2.5.10)$$

The stage is set (well, almost set) for a recursive program.

There is one small problem, though. In (2.5.8) we want to compute  $f(\omega_j)$  for  $2^r$  values of  $j$ , namely for  $j = 0, 1, \dots, 2^r - 1$ . However, the Fourier transform of the shorter sequence (2.5.9) is defined for only  $2^{r-1}$  values of  $j$ , namely for  $j = 0, 1, \dots, 2^{r-1} - 1$ . So if we calculate the first sum by a recursive call, then we will need its values for  $j$ 's that are outside the range for which it was computed.

This problem is no sooner recognized than solved. Let  $Q(j)$  denote the first sum in (2.5.8). Then we claim that  $Q(j)$  is a *periodic* function of  $j$ , of period  $2^{r-1}$ , because

$$\begin{aligned}
 Q(j + 2^{r-1}) &= \sum_{m=0}^{2^{r-1}-1} x_{2m} \exp\{2\pi i m(j + 2^{r-1})/2^{r-1}\} \\
 &= \sum_{m=0}^{2^{r-1}-1} x_{2m} \exp\{2\pi i m j/2^{r-1}\} e^{2\pi i m} \\
 &= \sum_{m=0}^{2^{r-1}-1} x_{2m} \exp\{2\pi i m j/2^{r-1}\} \\
 &= Q(j)
 \end{aligned} \tag{2.5.11}$$

for all integers  $j$ . If  $Q(j)$  has been computed only for  $0 \leq j \leq 2^{r-1} - 1$  and if we should want its value for some  $j \geq 2^{r-1}$  then we can get that value by asking for  $Q(j \bmod 2^{r-1})$ .

Now we can state the recursive form of the Fast Fourier Transform algorithm in the (most important) case where  $n$  is a power of 2. In the algorithm we will use the type *complexarray* to denote an array of complex numbers.

```

function FFT(n:integer; x :complexarray):complexarray;
{computes fast Fourier transform of  $n = 2^k$  numbers  $\mathbf{x}$  }
if n = 1 then FFT[0] := x[0]
else
  evenarray := {x[0], x[2], ..., x[n-2]};
  oddarray := {x[1], x[3], ..., x[n-1]};
  {u[0], u[1], ..., u[n/2-1]} := FFT(n/2, evenarray);
  {v[0], v[1], ..., v[n/2-1]} := FFT(n/2, oddarray);
  for j := 0 to n-1 do
     $\tau := \exp\{2\pi i j/n\}$ ;
    FFT[j] := u[j mod n/2] +  $\tau$ v[j mod n/2]
end.{FFT}

```

Let  $y(k)$  denote the number of multiplications of complex numbers that will be done if we call *FFT* on an array whose length is  $n = 2^k$ . The call to *FFT*( $n/2$ , *evenarray*) costs  $y(k-1)$  multiplications as does the call to *FFT*( $n/2$ , *oddarray*). The 'for  $j := 0$  to  $n$ ' loop requires  $n$  more multiplications. Hence

$$y(k) = 2y(k-1) + 2^k \quad (k \geq 1; y(0) = 0). \tag{2.5.12}$$

If we change variables by writing  $y(k) = 2^k z_k$ , then we find that  $z_k = z_{k-1} + 1$ , which, together with  $z_0 = 0$ , implies that  $z_k = k$  for all  $k \geq 0$ , and therefore that  $y(k) = k2^k$ . This proves

**Theorem 2.5.1.** *The Fourier transform of a sequence of  $n$  complex numbers is computed using only  $O(n \log n)$  multiplications of complex numbers by means of the procedure *FFT*, if  $n$  is a power of 2.*

Next\* we will discuss the situation when  $n$  is not a power of 2.

The reader may observe that by 'padding out' the input array with additional 0's we can extend the length of the array until it becomes a power of 2, and then call the *FFT* procedure that we have already

---

\* The remainder of this section can be omitted at a first reading.

discussed. In a particular application, that may or may not be acceptable. The problem is that the original question asked for the values of the input polynomial at the  $n^{\text{th}}$  roots of unity, but after the padding, we will find the values at the  $N^{\text{th}}$  roots of unity, where  $N$  is the next power of 2. In some applications, such as the multiplication of polynomials that we will discuss later in this section, that change is acceptable, but in others the substitution of  $N^{\text{th}}$  roots for  $n^{\text{th}}$  roots may not be permitted.

We will suppose that the FFT of a sequence of  $n$  numbers is wanted, where  $n$  is not a power of 2, and where the padding operation is not acceptable. If  $n$  is a prime number we will have nothing more to say, *i.e.* we will not discuss any improvements to the obvious method for calculating the transform, one root of unity at a time.

Suppose that  $n$  is not prime ( $n$  is ‘composite’). Then we can factor the integer  $n$  in some nontrivial way, say  $n = r_1 r_2$  where neither  $r_1$  nor  $r_2$  is 1.

We claim, then, that the Fourier transform of a sequence of length  $n$  can be computed by recursively finding the Fourier transforms of  $r_1$  different sequences, each of length  $r_2$ . The method is a straightforward generalization of the idea that we have already used in the case where  $n$  was a power of 2.

In the following we will write  $\xi_n = e^{2\pi i/n}$ . The train of ‘=’ signs in the equation below shows how the question on an input array of length  $n$  is changed into  $r_1$  questions about input arrays of length  $r_2$ . We have, for the value of the input polynomial  $f$  at the  $j^{\text{th}}$  one of the  $n$   $n^{\text{th}}$  roots of unity, the relations

$$\begin{aligned}
 f(e^{2\pi i j/n}) &= \sum_{s=0}^{n-1} x_s \xi_n^{js} \\
 &= \sum_{k=0}^{r_1-1} \sum_{t=0}^{r_2-1} \{x_{tr_1+k} \xi_n^{j(tr_1+k)}\} \\
 &= \sum_{k=0}^{r_1-1} \sum_{t=0}^{r_2-1} \{x_{tr_1+k} \xi_n^{tj r_1} \xi_n^{kj}\} \\
 &= \sum_{k=0}^{r_1-1} \left\{ \sum_{t=0}^{r_2-1} x_{tr_1+k} \xi_{r_2}^{tj} \right\} \xi_n^{kj} \\
 &= \sum_{k=0}^{r_1-1} a_k(j) \xi_n^{kj}.
 \end{aligned} \tag{2.5.13}$$

We will discuss (2.5.13), line-by-line. The first ‘=’ sign is the definition of the  $j^{\text{th}}$  entry of the Fourier transform of the input array  $\mathbf{x}$ . The second equality uses the fact that every integer  $s$  such that  $0 \leq s \leq n-1$  can be uniquely written in the form  $s = tr_1 + k$ , where  $0 \leq t \leq r_2 - 1$  and  $0 \leq k \leq r_1 - 1$ . The next ‘=’ is just a rearrangement, but the next one uses the all-important fact that  $\xi_n^{r_1} = \xi_{r_2}$  (why?), and in the last equation we are simply defining a set of numbers

$$a_k(j) = \sum_{t=0}^{r_2-1} x_{tr_1+k} \xi_{r_2}^{tj} \quad (0 \leq k \leq r_1 - 1; 0 \leq j \leq n - 1). \tag{2.5.14}$$

The important thing to notice is that for a fixed  $k$  the numbers  $a_k(j)$  are *periodic* in  $n$ , of period  $r_2$ , *i.e.*, that  $a_k(j+r_2) = a_k(j)$  for all  $j$ . Hence, even though the values of the  $a_k(j)$  are needed for  $j = 0, 1, \dots, n-1$ , they must be computed only for  $j = 0, 1, \dots, r_2 - 1$ .

Now the entire job can be done recursively, because for fixed  $k$  the set of values of  $a_k(j)$  ( $j = 0, 1, \dots, r_2 - 1$ ) that we must compute is itself a Fourier transform, namely of the sequence

$$\{x_{tr_1+k}\} \quad (t = 0, 1, \dots, r_2 - 1). \tag{2.5.15}$$

Let  $g(n)$  denote the number of complex multiplications that are needed to compute the Fourier transform of a sequence of  $n$  numbers. Then, for  $k$  fixed we can recursively compute the  $r_2$  values of  $a_k(j)$  that we need with  $g(r_2)$  multiplications of complex numbers. There are  $r_1$  such fixed values of  $k$  for which we must do the

computation, hence all of the necessary values of  $a_k(j)$  can be found with  $r_1 g(r_2)$  complex multiplications. Once the  $a_k(j)$  are all in hand, then the computation of the one value of the transform from (2.5.13) will require an additional  $r_1 - 1$  complex multiplications. Since  $n = r_1 r_2$  values of the transform have to be computed, we will need  $r_1 r_2 (r_1 - 1)$  complex multiplications.

The complete computation needs  $r_1 g(r_2) + r_1^2 r_2 - r_1 r_2$  multiplications if we choose a particular factorization  $n = r_1 r_2$ . The factorization that should be chosen is the one that minimizes the labor, so we have the recurrence

$$g(n) = \min_{n=r_1 r_2} \{r_1 g(r_2) + r_1^2 r_2\} - n. \quad (2.5.16)$$

If  $n = p$  is a prime number then there are no factorizations to choose from and our algorithm is no help at all. There is no recourse but to calculate the  $p$  values of the transform directly from the definition (2.5.6), and that will require  $p - 1$  complex multiplications to be done in order to get each of those  $p$  values. Hence we have, in addition to the recurrence formula (2.5.16), the special values

$$g(p) = p(p - 1) \quad (\text{if } p \text{ is prime}). \quad (2.5.17)$$

The recurrence formula (2.5.16) together with the starting values that are shown in (2.5.17) completely determine the function  $g(n)$ . Before proceeding, the reader is invited to calculate  $g(12)$  and  $g(18)$ .

We are going to work out the exact solution of the interesting recurrence (2.5.16), (2.5.17), and when we are finished we will see which factorization of  $n$  is the best one to choose. If we leave that question in abeyance for a while, though, we can summarize by stating the (otherwise) complete algorithm for the fast Fourier transform.

```
function FFT(x:complexarray; n:integer):complexarray;
{computes Fourier transform of a sequence x of length n}
if n is prime
then
  for j:=0 to n-1 do
    FFT[j] :=  $\sum_{k=0}^{n-1} x[k] \xi_n^{jk}$ 
else
  let n = r1r2 be some factorization of n;
  {see below for best choice of r1, r2}
  for k:=0 to r1-1 do
    {ak[0], ak[1], ..., ak[r2-1]}
    := FFT({x[k], x[k+r1], ..., x[k+(r2-1)r1]}, r2);
  for j:=0 to n-1 do
    FFT[j] :=  $\sum_{k=0}^{r_1-1} a_k[j \bmod r_2] \xi_n^{kj}$ 
end.{FFT}
```

Our next task will be to solve the recurrence relations (2.5.16), (2.5.17), and thereby to learn the best choice of the factorization of  $n$ .

Let  $g(n) = nh(n)$ , where  $h$  is a new unknown function. Then the recurrence that we have to solve takes the form

$$h(n) = \begin{cases} \min_d \{h(n/d) + d\} - 1, & \text{if } n \text{ is composite;} \\ n - 1, & \text{if } n \text{ is prime.} \end{cases} \quad (2.5.18)$$

In (2.5.18), the 'min' is taken over all  $d$  that divide  $n$  other than  $d = 1$  and  $d = n$ . The above relation determines the value of  $h$  for all positive integers  $n$ . For example,

$$\begin{aligned} h(15) &= \min_d (h(15/d) + d) - 1 \\ &= \min(h(5) + 3, h(3) + 5) - 1 \\ &= \min(7, 7) - 1 = 6 \end{aligned}$$

and so forth.

To find the solution in a pleasant form, let

$$n = p_1^{a_1} p_2^{a_2} \cdots p_s^{a_s} \quad (2.5.19)$$

be the canonical factorization of  $n$  into primes. We claim that the function

$$h(n) = a_1(p_1 - 1) + a_2(p_2 - 1) + \cdots + a_s(p_s - 1) \quad (2.5.20)$$

is the solution of (2.5.18) (this claim is obviously (?) correct if  $n$  is prime).

To prove the claim in general, suppose it to be true for  $1, 2, \dots, n-1$ , and suppose that  $n$  is not prime. Then every divisor  $d$  of  $n$  must be of the form  $d = p_1^{b_1} p_2^{b_2} \cdots p_s^{b_s}$ , where the primes  $p_i$  are the same as those that appear in (2.5.19) and each  $b_i$  is  $\leq a_i$ . Hence from (2.5.18) we get

$$h(n) = \min_{\mathbf{b}} \{(a_1 - b_1)(p_1 - 1) + \cdots + (a_s - b_s)(p_s - 1) + p_1^{b_1} \cdots p_s^{b_s}\} - 1 \quad (2.5.21)$$

where now the ‘min’ extends over all admissible choices of the  $b$ ’s, namely exponents  $b_1, \dots, b_s$  such that  $0 \leq b_i \leq a_i$  ( $\forall i = 1, s$ ) and not all  $b_i$  are 0 and not all  $b_i = a_i$ .

One such admissible choice would be to take, say,  $b_j = 1$  and all other  $b_i = 0$ . If we let  $H(b_1, \dots, b_s)$  denote the quantity in braces in (2.5.21), then with this choice the value of  $H$  would be  $a_1(p_1 - 1) + \cdots + a_s(p_s - 1) + 1$ , exactly what we need to prove our claim (2.5.20). Hence what we have to show is that the above choice of the  $b_i$ ’s is the best one. We will show that if one of the  $b_i$  is larger than 1 then we can reduce it without increasing the value of  $H$ .

To prove this, observe that for each  $i = 1, s$  we have

$$\begin{aligned} H(b_1, \dots, b_i + 1, \dots, b_s) - H(b_1, \dots, b_s) &= -p_i + d(p_i - 1) \\ &= (d - 1)(p_i - 1). \end{aligned}$$

Since the divisor  $d \geq 2$  and the prime  $p_i \geq 2$ , the last difference is nonnegative. Hence  $H$  doesn’t increase if we decrease one of the  $b$ ’s by 1 unit, as long as not all  $b_i = 0$ . It follows that the minimum of  $H$  occurs among the *prime* divisors  $d$  of  $n$ . Further, if  $d$  is prime, then we can easily check from (2.5.21) that it doesn’t matter which prime divisor of  $n$  that we choose to be  $d$ , the function  $h(n)$  is always given by (2.5.20). If we recall the change of variable  $g(n) = nh(n)$  we find that we have proved

**Theorem 2.5.2.** (*Complexity of the Fast Fourier Transform*) *The best choice of the factorization  $n = r_1 r_2$  in algorithm FFT is to take  $r_1$  to be a prime divisor of  $n$ . If that is done, then algorithm FFT requires*

$$g(n) = n(a_1(p_1 - 1) + a_2(p_2 - 1) + \cdots + a_s(p_s - 1))$$

*complex multiplications in order to do its job, where  $n = p_1^{a_1} \cdots p_s^{a_s}$  is the canonical factorization of the integer  $n$ . ■*

Table 2.5.1 shows the number  $g(n)$  of complex multiplications required by *FFT* as a function of  $n$ . The saving over the straightforward algorithm that uses  $n(n-1)$  multiplications for each  $n$  is apparent.

If  $n$  is a power of 2, say  $n = 2^q$ , then the formula of theorem 2.5.2 reduces to  $g(n) = n \log n / \log 2$ , in agreement with theorem 2.5.1. What does the formula say if  $n$  is a power of 3? if  $n$  is a product of distinct primes?

## 2.6 Applications of the FFT

Finally, we will discuss some applications of the FFT. A family of such applications begins with the observation that the FFT provides the fastest game in town for multiplying two polynomials together. Consider a multiplication like

$$(1 + 2x + 7x^2 - 2x^3 - x^4) \cdot (4 - 5x - x^2 - x^3 + 11x^4 + x^5).$$

n	g(n)	n	g(n)
2	2	22	242
3	6	23	506
4	8	24	120
5	20	25	200
6	18	26	338
7	42	27	162
8	24	28	224
9	36	29	812
10	50	30	210
11	110	31	930
12	48	32	160
13	156	33	396
14	98	34	578
15	90	35	350
16	64	36	216
17	272	37	1332
18	90	38	722
19	342	39	546
20	120	40	280
21	168	41	1640

**Table 2.5.1: The complexity of the FFT**

We will study the amount of labor that is needed to do this multiplication by the straightforward algorithm, and then we will see how the FFT can help.

If we do this multiplication in the obvious way then there is quite a bit of work to do. The coefficient of  $x^4$  in the product, for instance, is  $1 \cdot 11 + 2 \cdot (-1) + 7 \cdot (-1) + (-2) \cdot (-5) + (-1) \cdot 4 = 8$ , and 5 multiplications are needed to compute just that single coefficient of the product polynomial.

In the general case, we want to multiply

$$\left\{ \sum_{i=0}^n a_i x^i \right\} \cdot \left\{ \sum_{j=0}^m b_j x^j \right\}. \quad (2.6.1)$$

In the product polynomial, the coefficient of  $x^k$  is

$$\sum_{r=\max(0, k-m)}^{\min(k, n)} a_r b_{k-r}. \quad (2.6.2)$$

For  $k$  fixed, the number of terms in the sum (2.6.2) is  $\min(k, n) - \max(0, k - m) + 1$ . If we sum this amount of labor over  $k = 0, m + n$  we find that the total amount of labor for multiplication of two polynomials of degrees  $m$  and  $n$  is  $\Theta(mn)$ . In particular, if the polynomials are of the same degree  $n$  then the labor is  $\Theta(n^2)$ .

By using the FFT the amount of labor can be reduced from  $\Theta(n^2)$  to  $\Theta(n \log n)$ .

To understand how this works, let's recall the definition of the Fourier transform of a sequence. It is the sequence of values of the polynomial whose coefficients are the given numbers, at the  $n^{\text{th}}$  roots of unity, where  $n$  is the length of the input sequence.

Imagine two universes, one in which the residents are used to describing polynomials by means of their coefficients, and another one in which the inhabitants are fond of describing polynomials by their values at roots of unity. In the first universe the locals have to work fairly hard to multiply two polynomials because they have to carry out the operations (2.6.2) in order to find each coefficient of the product.

In the second universe, multiplying two polynomials is a breeze. If we have in front of us the values  $f(\omega)$  of the polynomial  $f$  at the roots of unity, and the values  $g(\omega)$  of the polynomial  $g$  at the same roots of unity, then what are the values  $(fg)(\omega)$  of the product polynomial  $fg$  at the roots of unity? To find each one requires only a single multiplication of two complex numbers, because the value of  $fg$  at  $\omega$  is simply  $f(\omega)g(\omega)$ .

Multiplying values is easier than finding the coefficients of the product.

Since we live in a universe where people like to think about polynomials as being given by their coefficient arrays, we have to take a somewhat roundabout route in order to do an efficient multiplication.

Given: A polynomial  $f$ , of degree  $n$ , and a polynomial  $g$  of degree  $m$ ; by their coefficient arrays. Wanted: The coefficients of the product polynomial  $fg$ , of degree  $m + n$ .

Step 1: Let  $N - 1$  be the smallest integer that is a power of 2 and is greater than  $m + n + 1$ .

Step 2. Think of  $f$  and  $g$  as polynomials each of whose degrees is  $N - 1$ . This means that we should adjoin  $N - n$  more coefficients, all = 0, to the coefficient array of  $f$  and  $N - m$  more coefficients, all = 0, to the coefficient array of  $g$ . Now both input coefficient arrays are of length  $N$ .

Step 3. Compute the FFT of the array of coefficients of  $f$ . Now we are looking at the values of  $f$  at the  $N^{\text{th}}$  roots of unity. Likewise compute the FFT of the array of coefficients of  $g$  to obtain the array of values of  $g$  at the same  $N^{\text{th}}$  roots of unity. The cost of this step is  $O(N \log N)$ .

Step 4. For each of the  $N^{\text{th}}$  roots of unity  $\omega$  multiply the number  $f(\omega)$  by the number  $g(\omega)$ . We now have the numbers  $f(\omega)g(\omega)$ , which are exactly the values of the unknown product polynomial  $fg$  at the  $N^{\text{th}}$  roots of unity. The cost of this step is  $N$  multiplications of numbers, one for each  $\omega$ .

Step 5. We now are looking at the *values* of  $fg$  at the  $N^{\text{th}}$  roots, and we want to get back to the *coefficients* of  $fg$  because that was what we were asked for. To go backwards, from values at roots of unity to coefficients, calls for the *inverse Fourier transform*, which we will describe in a moment. Its cost is also  $O(N \log N)$ . ■

The answer to the original question has been obtained at a total cost of  $O(N \log N) = O((m + n) \log(m + n))$  arithmetic operations. It's true that we did have to take a walk from our universe to the next one and back again, but the round trip was a lot cheaper than the  $O((m + n)^2)$  cost of a direct multiplication.

It remains to discuss the inverse Fourier transform. Perhaps the neatest way to do that is to juxtapose the formulas for the Fourier transform and for the inverse transform, so as to facilitate comparison of the two, so here they are. If we are given a sequence  $\{x_0, x_1, \dots, x_{n-1}\}$  then the Fourier transform of the sequence is the sequence (see (2.5.6))

$$f(\omega_j) = \sum_{k=0}^{n-1} x_k e^{2\pi i j k / n} \quad (j = 0, 1, \dots, n-1). \quad (2.6.3)$$

Conversely, if we are given the numbers  $f(\omega_j)$  ( $j = 0, \dots, n-1$ ) then we can recover the coefficient sequence  $x_0, \dots, x_{n-1}$  by the inverse formulas

$$x_k = \frac{1}{n} \sum_{j=0}^{n-1} f(\omega_j) e^{-2\pi i j k / n} \quad (k = 0, 1, \dots, n-1). \quad (2.6.4)$$

The differences between the inverse formulas and the original transform formulas are first the appearance of '1/n' in front of the summation and second the '-' sign in the exponential. We leave it as an exercise for the reader to verify that these formulas really do invert each other.

We observe that if we are already in possession of a computer program that will find the FFT, then we can use it to calculate the inverse Fourier transform as follows:

- (i) Given a sequence  $\{f(\omega)\}$  of values of a polynomial at the  $n^{\text{th}}$  roots of unity, form the complex conjugate of each member of the sequence.
- (ii) Input the conjugated sequence to your FFT program.
- (iii) Form the complex conjugate of each entry of the output array, and divide by  $n$ . You now have the inverse transform of the input sequence.

The cost is obviously equal to the cost of the FFT plus a linear number of conjugations and divisions by  $n$ .

An outgrowth of the rapidity with which we can now multiply polynomials is a rethinking of the methods by which we do ultrahigh-precision arithmetic. How fast can we multiply two integers, each of which has ten million bits? By using ideas that developed directly (though not at all trivially) from the ones that we have been discussing, Schönhage and Strassen found the fastest known method for doing such large-scale multiplications of integers. The method relies heavily on the FFT, which may not be too surprising since an integer  $n$  is given in terms of its bits  $b_0, b_1, \dots, b_m$  by the relation

$$n = \sum_{i \geq 0} b_i 2^i. \quad (2.6.5)$$

However the sum in (2.6.5) is seen at once to be the value of a certain polynomial at  $x = 2$ . Hence in asking for the bits of the product of two such integers we are asking for something very similar to the coefficients of the product of two polynomials, and indeed the fastest known algorithms for this problem depend upon the Fast Fourier Transform.

### Exercises for section 2.6

1. Let  $\omega$  be an  $n^{\text{th}}$  root of unity, and let  $k$  be a fixed integer. Evaluate

$$1 + \omega^k + \omega^{2k} + \dots + \omega^{k(n-1)}.$$

2. Verify that the relations (2.6.3) and (2.6.4) indeed are inverses of each other.

3. Let  $f = \sum_{j=0}^{n-1} a_j x^j$ . Show that

$$\frac{1}{n} \sum_{\omega^n=1} |f(\omega)|^2 = |a_0|^2 + \dots + |a_{n-1}|^2$$

4. The values of a certain cubic polynomial at  $1, i, -1, -i$  are  $1, 2, 3, 4$ , respectively. Find its value at  $2$ .
5. Write a program that will do the FFT in the case where the number of data points is a power of 2. Organize your program so as to minimize additional array storage beyond the input and output arrays.
6. Prove that a polynomial of degree  $n$  is uniquely determined by its values at  $n + 1$  distinct points.

### 2.7 A review

Here is a quick review of the algorithms that we studied in this chapter.

Sorting is an easy computational problem. The most obvious way to sort  $n$  array elements takes time  $\Theta(n^2)$ . We discussed a recursive algorithm that sorts in an average time of  $\Theta(n \log n)$ .

Finding a maximum independent set in a graph is a hard computational problem. The most obvious way to find one might take time  $\Theta(2^n)$  if the graph  $G$  has  $n$  vertices. We discussed a recursive method that runs in time  $\Theta(1.39^n)$ . The best known methods run in time  $\Theta(2^{n/3})$ .

Finding out if a graph is  $K$ -colorable is a hard computational problem. The most obvious way to do it takes time  $\Theta(K^n)$ , if  $G$  has  $n$  vertices. We discussed a recursive method that runs in time  $O(1.62^{n+E})$  if  $G$  has  $n$  vertices and  $E$  edges. One recently developed method \* runs in time  $O((1 + \sqrt[3]{3})^n)$ . We will see in section 5.7 that this problem can be done in an *average* time that is  $O(1)$  for fixed  $K$ .

Multiplying two matrices is an easy computational problem. The most obvious way to do it takes time  $\Theta(n^3)$  if the matrices are  $n \times n$ . We discussed a recursive method that runs in time  $O(n^{2.82})$ . A recent method \*\* runs in time  $O(n^\gamma)$  for some  $\gamma < 2.5$ .

---

\* E. Lawler, A note on the complexity of the chromatic number problem, *Information Processing Letters* **5** (1976), 66-7.

\*\* D. Coppersmith and S. Winograd, On the asymptotic complexity of matrix multiplication, *SIAM J. Comp.* **11** (1980), 472-492.

Finding the discrete Fourier transform of an array of  $n$  elements is an easy computational problem. The most obvious way to do it takes time  $\Theta(n^2)$ . We discussed a recursive method that runs in time  $O(n \log n)$  if  $n$  is a power of 2.

When we write a program recursively we are making life easier for ourselves and harder for the compiler and the computer. A single call to a recursive program can cause it to execute a tree-full of calls to itself before it is able to respond to our original request.

For example, if we call Quicksort to sort the array

$$\{5, 8, 13, 9, 15, 29, 44, 71, 67\}$$

then the tree shown in Fig. 2.7.1 might be generated by the compiler.

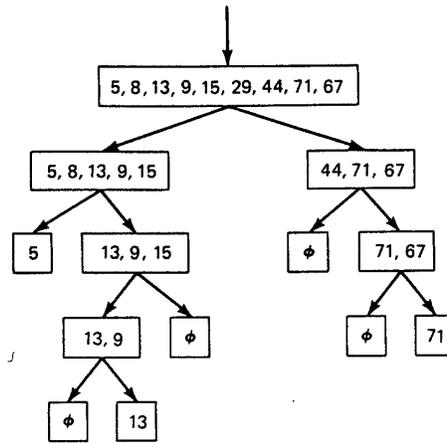


Fig. 2.7.1: A tree of calls to *Quicksort*

Again, if we call *maxset1* on the 5-cycle, the tree in Fig. 2.3.3 of calls may be created.

A single invocation of *chrompoly*, where the input graph is a 4-cycle, for instance, might generate the tree of recursive calls that appears in Fig. 2.7.2.

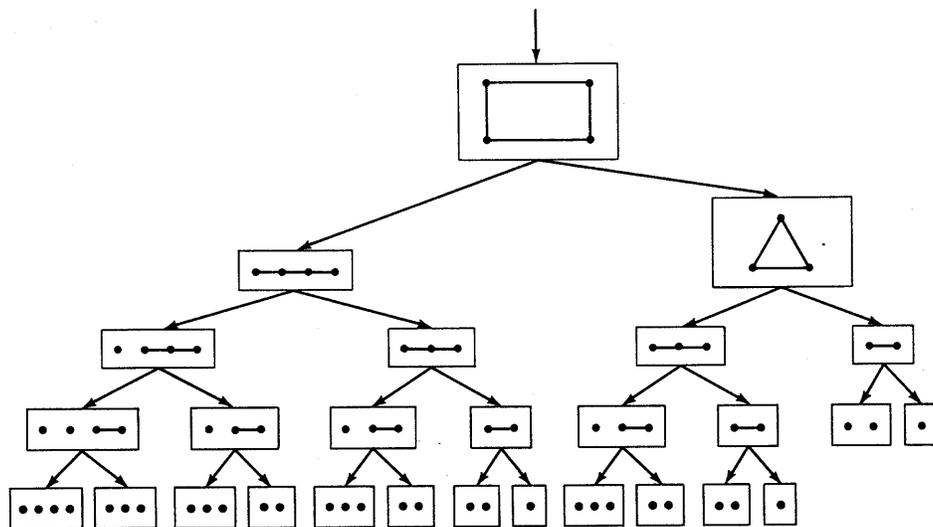
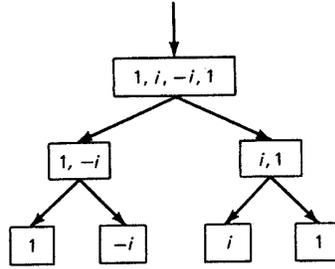


Fig. 2.7.2: A tree of calls to *chrompoly*



**Fig. 2.7.3: The recursive call tree for *FFT***

Finally, if we call the ‘power of 2’ version of the *FFT* algorithm on the sequence  $\{1, i, -i, 1\}$  then *FFT* will proceed to manufacture the tree shown in Fig. 2.7.3.

It must be emphasized that the creation of the tree of recursions is done by the compiler without any further effort on the part of the programmer. As long as we’re here, how does a compiler go about making such a tree?

It does it by using an auxiliary *stack*. It adopts the philosophy that if it is asked to do two things at once, well after all, it can’t do that, so it does one of those two things and drops the other request on top of a stack of unfinished business. When it finishes executing the first request it goes to the top of the stack to find out what to do next.

### Example

Let’s follow the compiler through its tribulations as it attempts to deal with our request for maximum independent set size that appears in Fig. 2.3.3. We begin by asking for the *maxset1* of the 5-cycle. Our program immediately makes two recursive calls to *maxset1*, on each of the two graphs that appear on the second level of the tree in Fig. 2.3.3. The stack is initially empty.

The compiler says to itself ‘I can’t do these both at once’, and it puts the right-hand graph (involving vertices 3,4) on the stack, and proceeds to call itself on the left hand graph (vertices 2,3,4,5).

When it tries to do that one, of course, two more graphs are generated, of which the right-hand one (4,5) is dropped onto the stack, on top of the graph that previously lived there, so now two graphs are on the stack, awaiting processing, and the compiler is dealing with the graph (3,4,5).

This time the graph of just one vertex (5) is dropped onto the stack, which now holds three graphs, as the compiler works on (4,5).

Next, that graph is broken up into (5), and an empty graph, which is dutifully dropped onto the stack, so the compiler can work on (5).

Finally, something fruitful happens: the graph (5) has no edges, so the program *maxset1* gives, in its trivial case, very specific instructions as to how to deal with this graph. We now know that the graph that consists of just the single vertex (5) has a *maxset1* values of 1.

The compiler next reaches for the graph on top of the stack, finds that it is the empty graph, which has no edges at all, and therefore its *maxset* size is 0.

It now knows the  $n_1 = 1$  and the  $n_2 = 0$  values that appear in the algorithm *maxset1*, and therefore it can execute the instruction  $maxset1 := \max(n_1, 1 + n_2)$ , from which it finds that the value of *maxset1* for the graph (4,5) is 1, and it continues from there, to dig itself out of the stack of unfinished business.

In general, if it is trying to execute *maxset1* on a graph that has edges, it will drop the graph  $G - \{v^*\} - N_{bhd}(v^*)$  on the stack and try to do the graph  $G - \{v^*\}$ .

The reader should try to write out, as a formal algorithm, the procedure that we have been describing, whereby the compiler deals with a recursive computation that branches into two sub-computations until a trivial case is reached. ■

### Exercise for section 2.7

1. In Fig. 2.7.3, add to the picture the *output* that each of the recursive calls gives back to the box above it that made the call.

### Bibliography

A definitive account of all aspects of sorting is in

D. E. Knuth, *The art of computer programming*, Vol. 3: *Sorting and searching*, Addison Wesley, Reading MA, 1973.

All three volumes of the above reference are highly recommended for the study of algorithms and discrete mathematics.

A  $O(2^{n/3})$  algorithm for the maximum independent set problem can be found in

R. E. Tarjan and A. Trojanowski, Finding a maximum independent set, *SIAM J. Computing* 6 (1977), 537-546.

Recent developments in fast matrix multiplication are traced in

Victor Pan, How to multiply matrices faster, Lecture notes in computer science No. 179, Springer-Verlag, 1984.

The realization that the Fourier transform calculation can be speeded up has been traced back to

C. Runge, *Zeits. Math. Phys.*, **48** (1903) p. 443.

and also appears in

C. Runge and H. König, *Die Grundlehren der math. Wissensch.*, 11, Springer Verlag, Berlin 1924.

The introduction of the method in modern algorithmic terms is generally credited to

J. M. Cooley and J. W. Tukey, An algorithm for the machine calculation of complex Fourier series, *Mathematics of Computation*, **19** (1965), 297-301.

A number of statistical applications of the method are in

J. M. Cooley, P. A. W. Lewis and P. D. Welch, The Fast Fourier Transform and its application to time series analysis, in *Statistical Methods for Digital Computers*, Enslin, Ralston and Wilf eds., John Wiley & Sons, New York, 1977, 377-423.

The use of the FFT for high precision integer arithmetic is due to

A Schönhage and V. Strassen, Schnelle Multiplikation grosser Zahlen, *Computing*, **7** (1971), 281-292.

An excellent account of the above as well as of applications of the FFT to polynomial arithmetic is by

A. V. Aho, J. E. Hopcroft and J. D. Ullman, The design and analysis of computer algorithms, Addison Wesley, Reading, MA, 1974 (chap. 7).