USING INDUCTION TO DESIGN ALGORITHMS

An analogy between proving mathematical theorems and designing computer algorithms provides an elegant methodology for designing algorithms, explaining their behavior, and understanding their key ideas.

UDI MANBER

This article presents a methodology, based on mathematical induction, for approaching the design and the teaching of combinatorial algorithms. While this methodology does not cover all possible ways of designing algorithms it does cover many known techniques. It also provides an elegant intuitive framework for explaining the design of algorithms in more depth. The heart of the methodology lies in an analogy between the intellectual process of proving mathematical theorems and that of designing combinatorial algorithms. We claim that although these two processes serve different purposes and achieve different types of results, they are more similar than it seems. This claim is established here by a series of examples of algorithms, each developed and explained by the use of the methodology. We believe that students can get more motivation, greater depth, and better understanding of algorithms by this methodology.

Mathematical induction is a very powerful proof technique. It usually works as follows. Let T be a theorem that we want to prove. Suppose that T includes a parameter n whose value can be any natural number. Instead of proving directly that T holds for all values of n we prove that (1) T holds for $n = 1$, and (2) T holds for any $n > 1$ provided that T holds for $n - 1$. The first part is usually very simple to prove. Proving the second part is easier in many cases than proving the theorem directly since we can use the assumption that T holds for $n - 1$. (In some sense we get this assumption for free.) In other words, it is enough to *reduce* the theorem to one with a smaller value of n, rather than proving it from scratch. We concentrate on this reduction.

The same principle holds for algorithms. Induction allows one to concentrate on extending solutions of smaller subproblems to those of larger problems. One starts with an arbitrary instance of the problem at hand, and tries to solve it by using the assumption that the same problem but with smaller size has already been solved. For example, given a sequence of $n > 1$ numbers to sort (it is trivial to sort one number), we can assume that we already know how to sort $n - 1$ numbers. Then we can either sort the first $n - 1$ numbers and insert the nth number in its correct position (which leads to an algorithm called *insertion sort*), or start by putting the nth number in its final position and then sort the rest (which is called *selection sort*). We need only to address the operation on the nth number. (Of course, this is not the only way to sort, nor is it the only way to use induction for sorting.)

The use of induction in the example above is straightforward. There are, however, many different ways to use induction, leading to different algorithm

This research was supported in part by an NSF grant MCS-8303134, and an NSF Presidential Young Investigator Award (grant DCR-8451397), with matching funds from Tektronix, Digital Equipment Corporation, and Hewlett Packard.

© 1988 ACM 0001-0782/88/1100-1300 $1.50

design techniques. It is a very rich and flexible method. We will show that surprisingly many problems become much easier to solve if one uses this idea. We will survey some of these methods, and show their power in designing algorithms. Among the variations of induction that we discuss are choosing the induction sequence wisely, strengthening the induction hypothesis, strong induction, and maximal counterexample.

The novelty of our approach is twofold. First we collect seemingly different techniques of algorithm design under one umbrella. This makes the search for a new algorithm more organized. Second, we utilize known mathematical proof techniques for algorithm design. This is important since it opens the door to the use of powerful techniques that have been developed for many years in another discipline.

The use of induction, and mathematical proof techniques in general, in the algorithms area is not new. Induction has been used for a long time to prove correctness of algorithms by associating *assertions* with certain steps of the algorithm and proving that they hold initially and that they are invariant under certain operations. This method was originally suggested by Goldstine and von Neumann, and developed later by Floyd and others. Dijkstra [7] and Gries [8] present a methodology similar to ours to develop programs together with their proof of correctness. While we borrow some of their techniques, our emphasis is different: we concentrate on the high level algorithmic ideas without going down to the actual program level. PRL [2] and later NuPRL [6] use mathematical proofs as the basic part of a program development system. Recursion, of course, has been used extensively in algorithm design (for a general discussion on recursion see [5] and [13]).

Our goals are mainly pedagogical. We assume only that the reader is familiar with mathematical induction and basic data structures. For each proof technique we explain the analogy briefly and present one or more examples of algorithms. The emphasis of the presentation is on how to use this methodology. Our aim is not to explain an algorithm in a way that makes it easier for a programmer to translate it into a program, but rather to explain it in a way that makes it easier to *understand*. The algorithms are explained through a creative process rather than as finished products. Our goals in teaching algorithms are not only to show students how to solve particular problems, but also to help them solve new problems in the future. Teaching the thinking involved in designing an algorithm is as important as teaching the details of the solution, but this is usually much harder to do. We believe our methodology enhances the understanding of this thinking process.

Although induction suggests implementation by recursion, this is not necessarily the case. (Indeed, we call this approach *inductive* rather than *recursive* to de-emphasize recursive implementations.) In many cases, iterative implementation is just as easy, even if the algorithm was designed with induction in mind; iteration is also generally more efficient.

The algorithms presented in this article were selected to highlight the methodology. We chose relatively simple examples, with some more complicated examples toward the end. We have found that many of the algorithms regularly taught in the first course on algorithm design can be described using this method. (An introduction to algorithms book using this methodology will appear soon [12].) We begin with three simple examples (at least the use of induction makes them seem simple). Then we present several mathematical proof techniques and their analogous algorithm design techniques. In each case the analogy is illustrated by one or more examples.

THREE EXAMPLES

Evaluating Polynomials

The problem: Given a sequence of real numbers a_n, $a_{n-1}, \ldots, a_1, a_0$, and a real number x, compute the value of the polynomial

$$P_n(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0.$$

This problem may not seem like a natural candidate for an inductive approach. Nevertheless, we will show that induction can lead directly to a very good solution. We start with the simplest (almost trivial) approach, and then find variations that lead to better solutions.

The problem involves $n + 2$ numbers. The inductive approach is to solve this problem in terms of a solution of a smaller problem. In other words, we try to reduce the problem to one with smaller size, which is then solved recursively (or as we will call it *by induction*). The first attempt may be to remove a_n. This results in the problem of evaluating the polynomial

$$P_{n-1}(x) = a_{n-1} x^{n-1} + a_{n-2} x^{n-2} + \cdots + a_1 x + a_0.$$

This is the same problem, except for the size. Therefore, we can solve it by induction using the following hypothesis:

Induction hypothesis: *We know how to evaluate a polynomial represented by the input $a_{n-1}, \ldots, a_1, a_0$, at the point x (i.e., we know how to compute $P_{n-1}(x)$).*

We can now use the hypothesis to solve the problem by induction. First we have to solve the base case, which is computing a_0; this is trivial. Then, we must show how to solve the original problem (computing $P_n(x)$) with the aid of the solution of the smaller problem (which is the value of $P_{n-1}(x)$). In this case it is straightforward. We simply compute x^n, multiply it by a_n and add the result to $P_{n-1}(x)$.

At this point it may seem that the use of induction is frivolous because it just complicates a very simple solution. The algorithm above merely evaluates the polynomial from right to left as it is written. However, we will shortly see the power of the approach.

While the algorithm is correct, it is not very efficient. It requires $n + n - 1 + n - 2 + \cdots + 1 = n(n + 1)/2$ multiplications and n additions. We now use induction a little differently to obtain a better solution.

Improvements: The first improvement comes from the observation that there is a lot of redundant computation: the powers of x are computed from scratch. We can save many multiplications by using the value of x^{n-1} when we compute x^n. This is done by including the computation of x^n in the induction hypothesis:

Stronger induction hypothesis: *We know how to compute the value of the polynomial $P_{n-1}(x)$, and we know how to compute x^{n-1}.*

This induction hypothesis is stronger, since it requires computing x^{n-1}, but it is easier to extend: We need only perform one multiplication to compute x^n, then one more multiplication to get $a_n x^n$, and then one addition to complete the computation. (The induction hypothesis is not too strong, since we need to compute x^{n-1} anyway.) Overall, there are $2n$ multiplications and n additions. Even though the induction hypothesis requires more computation, it leads to less work overall. We will return to this point later. This algorithm looks very good by all measures. It is efficient, simple, and easy to implement. However, a better algorithm exists. It is a result of using induction in a different way.

We reduced the problem by removing the last coefficient, a_n (which was the straightforward thing to do). But, we can also remove the first coefficient, a_0. The smaller problem becomes the evaluation of the polynomial represented by the coefficients $a_n, a_{n-1}, \ldots, a_1$, which is

$$P'_{n-1}(x) = a_n x^{n-1} + a_{n-1}x^{n-2} + \cdots + a_1.$$

(Notice that a_n is now the $n - 1$st coefficient, and so on.) The new induction hypothesis is the following:

Induction hypothesis (reversed order): *We know how to evaluate the polynomial represented by the coefficients a_n, $a_{n-1}, \ldots, a_1$ at the point x (i.e., $P'_{n-1}(x)$ above).*

This hypothesis is better for our purposes since it is easier to extend. Clearly $P_n(x) = x \cdot P'_{n-1}(x) + a_0$. Therefore, only one multiplication and one addition are required to compute $P_n(x)$ from $P'_{n-1}(x)$. The complete algorithm can be described by the following expression:

$$a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$$
$$= ((\cdots((a_n x + a_{n-1})x + a_{n-2})\cdots)x + a_1)x + a_0.$$

This algorithm is known as *Horner's rule* after W. G. Horner. (It was also mentioned by Newton, see [10] p. 467.) The program to evaluate the polynomial is given below.

```
Algorithm Polynomial_Evaluation
  (a0, a1, a2, ..., an, x: real);
begin
  P := an;
  for i := 1 to n do
     P := x * P + a(n-i)
end;
```

Complexity: The algorithm requires only n multiplications, n additions, and one extra memory location. Even though the previous solutions seemed very simple and efficient, it was worthwhile to pursue a better algorithm. This algorithm is not only faster, its corresponding program is simpler.

Summary: The simple example above illustrates the flexibility in using induction. The trick that led to Horner's rule was merely considering the input from left to right instead of the intuitive right to left.

There are, of course, many other possibilities of using induction, and we will see more such examples.

Finding One-to-One Mappings

Let f be a function that maps a finite set A into itself (i.e., every element of A is mapped to another element of A). For simplicity, we denote the elements of A by the numbers 1 to n. We assume that the function f is represented by an array $f[1..n]$ such that $f[i]$ holds the value of $f(i)$ (which is an integer between 1 and n). The function f is called *one-to-one* if every element of A has exactly one element which is mapped to it. The function f can be represented by a diagram as in Figure 1, where both sides correspond to the same set and the edges indicate the mapping (which is done from left to right).

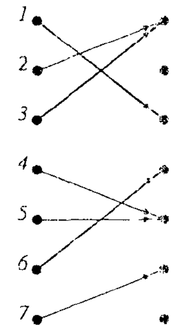

FIGURE 1. A Mapping from a Set into Itself

The problem: Find a subset $S \subseteq A$ with the maximum number of elements, such that (1) the function f maps every element of S to another element of S (i.e., f maps S into itself), and (2) every element of S has exactly one element of S which is mapped to it (i.e., f is one-to-one when restricted to S).

If f is originally one-to-one then the whole set A satisfies the conditions of the problem, and it is definitely maximal. If, on the other hand, $f(i) = f(j)$ for some $i \neq j$ then S cannot contain both i and j. For example, in Figure 1 S cannot contain both 2 and 3. The choice of which one of them to eliminate cannot be arbitrary. Suppose, for example, that we decide to eliminate 3. Since 1 is mapped to 3 we must eliminate 1 as well (the

mapping must be into S and 3 is no longer in S). But if 1 is eliminated, then 2 must be eliminated as well (for the same reason). But, this is not maximal, since it is easy to see that we could have eliminated 2 alone. The problem is to find a general method to decide which elements to include.

The idea of a solution by induction is to concentrate on reducing the size of the problem. We have some flexibility. We can reduce the size of the problem by finding either an element that belongs to S or an element that does not belong to S. We will do the latter. We use the straightforward induction hypothesis:

Induction hypothesis: *We know how to solve the problem for sets of $n - 1$ elements.*

The base case is trivial: if there is only one element in the set, then it must be mapped to itself, which is a one-to-one mapping. Assume now that we have a set A of n elements and we are looking for a subset S that satisfies the conditions of the problem. It turns out that it is easier to find an element that *cannot* belong to S than it is to find an element of S. We claim that any element i that has no other element mapped to it cannot belong to S. (In other words, an element i in the right side of the diagram, which is not connected to any edge, cannot be in S.) Otherwise, if $i \in S$ and S has, say, k elements, then those k elements are mapped into at most $k - 1$ elements, therefore the mapping cannot be one-to-one. If there is such an i then we simply remove it from the set. We now have a set $A' = A - \{i\}$ with $n - 1$ elements, which f maps into itself; by the induction hypothesis we know how to solve the problem for A'. If no such i exists then the mapping is one-to-one, and we are done.

The essence of this solution is that we *must* remove i. We proved above that it cannot belong to S. This is the strength of induction. Once we remove an element and reduce the size of the problem we are done. We have to be very careful, however, that the reduced problem is exactly the same (except for size) as the original problem. The only condition on the set A and the function f was that f maps A into itself. This condition is still maintained for the set $A - \{i\}$ since there was nothing that was mapped to i. The algorithm terminates when no elements can be removed.

Implementation: The algorithm was described above as a recursive procedure. In each step we find an element such that no other element is mapped to it, remove it, and continue recursively. The implementation, however, need not be recursive. We can maintain a counter $c[i]$ with each element i. Initially $c[i]$ should be equal to the number of elements that are mapped to i. This can be computed (in n steps) by scanning the array and incrementing the appropriate counters. We then put all the elements with a zero counter in a queue. In each step we remove an element j from the queue (and the set), decrement $c[f(j)]$, and if $c[f(j)] = 0$, we put $f(j)$ in the queue. The algorithm terminates when the queue is empty. The algorithm follows.

```
Algorithm Mapping (f: array [1..n] of integer);
begin
  S := A; {A is the set of numbers from 1 to n}
  for j := 1 to n do c[j] := 0;
  for j := 1 to n do increment c[f[j]];
  for j := 1 to n do
    if c[j] = 0 then put j in Queue;
  while Queue is not empty do
    remove i from the top of the queue;
    S := S - {i};
    decrement c[f[i]];
    if c[f[i]] = 0 then put f[i] in Queue
end;
```

Complexity: The initialization parts require $O(n)$ operations. Every element can be put on the queue at most once, and the steps involved in removing an element from the queue take constant time. The total number of steps is thus $O(n)$.

Summary: In this case reducing the size of the problem involves eliminating elements from a set. Therefore, we try to find the easiest way to remove an element without changing the conditions of the problem. Since the only requirement from the function is that it maps A into itself, the choice of an element which no other element is mapped to is natural.

Checking Intervals for Containment

The problem: The input is a set of intervals $I_1, I_2, \ldots, I_n$ on a line. Each interval I_j is given by its two endpoints L_j (left) and R_j (right). We want to *mark* all intervals that are contained in other intervals. In other words, an interval I_j should be marked if there exists another interval I_k ($k \neq j$) such that $L_k \leq L_j$ and $R_k \geq R_j$. For simplicity we assume that the intervals are distinct (i.e., no two intervals have the same left and right endpoints, but they may have one of them in common). Figure 2 shows such a set of intervals. (They are shown one on top of the other instead of all on one line for better illustration.)

FIGURE 2. A Set of Intervals

Using induction in a straightforward way involves removing an interval I, solving the problem recursively for the remaining intervals, and then checking the effects of adding I back. The problem is that putting I back requires checking all other intervals to see whether any of them either contains I or is contained in it. This will require checking I against $n - 1$ intervals, which will result in an algorithm that uses $n - 1 + n - 2 + \cdots + 1 = n(n - 1)/2$ comparisons. We want a better algorithm so we do two things: First we choose a special interval to remove, and second we try to use

as much of the information gathered from the solution of the smaller problem.

Let I be the interval with the largest (rightmost) left endpoint. There is no need to check left endpoints any further since all other intervals have smaller left endpoints. Therefore, to check whether I is contained in any of the other intervals, we only need to check whether any of them has a right endpoint larger than I's right endpoint. To get the interval with the largest left endpoint we can sort all intervals according to their left endpoints and scan them in order. Assume that the intervals are sorted in that order, so that $L_1 \leq L_2 \leq \cdots \leq L_n$. The induction hypothesis is the following:

Induction hypothesis: *We know how to solve the problem for* $I_1, I_2, \ldots, I_{n-1}$.

The base case is trivial: if there is only one interval, then it is not marked. Now consider I_n. By the induction hypothesis we know which of the intervals I_1 to I_{n-1} is contained in another interval. We need to determine whether I_n contains some (previously unmarked) intervals, and whether it is contained in another interval. Let's concentrate first on checking whether I_n is contained in another interval. If I_n is contained in an interval I_j, $j < n$, then $R_j \geq R_n$. But this is the only necessary condition (the sorting guarantees that $L_j \leq L_n$). Therefore, we only need to keep track of the *largest* right endpoint among the previous intervals. Comparing the largest right endpoint to R_n should give us the answer. We change the induction hypothesis slightly:

Stronger induction hypothesis: *We know how to solve the problem for* $I_1, I_2, \ldots, I_{n-1}$, *and find the largest right endpoint among them.*

Again, let I_n be the interval with the largest left endpoint, and let $MaxR$ be the largest right endpoint among the $n - 1$ first intervals. If $MaxR \geq R_n$ then I_n should be marked; otherwise ($MaxR < R_n$), R_n becomes the new maximal. (This last step is necessary since we are now not only marking intervals, but we are also looking for the largest right endpoint.) We are able to determine I_n's status with only one check.

To complete the algorithm we need to check whether I_n contains a previously unmarked interval. I_n contains an interval I_j, $j < n$, only if $L_j = L_n$ and $R_j < R_n$. We can handle this case by strengthening the sorting. We not only sort according to left endpoints, but among all intervals with the same left endpoint we sort according to the reversed order of right endpoints. This will eliminate the case above since I_j will be placed after I_n and its containment will therefore be found by the algorithm above. The complete algorithm is given below.

```
Algorithm Interval_Containment
  (I1, I2, ..., In: intervals);
{An interval Ij is represented as a pair Lj, Rj.}
{We assume that no two intervals are exactly the
  same.}
{Mark[j] will be set to true if Ij is contained in another
  interval.}
begin
  Sort the intervals in increasing order according to left
    endpoints;
  Intervals with equal left endpoints are sorted in
    decreasing order according to right endpoint;
  {for all j < k, either Lj < Lk or Lj = Lk and Rj > Rk}
  MaxR := R1;
  for j := 2 to n do
    if Rj ≤ MaxR then
      Mark[j] := true
    else
      Mark[j] := false;
      MaxR := Rj
end;
```

Complexity: Except for the sorting, the algorithm contains one loop involving $O(n)$ operations. Since sorting requires $O(n \log n)$ steps, it dominates the running time of the algorithm.

Summary: This example illustrates a less straightforward use of induction. First, we select the order under which the induction will take place. Then, we design the induction hypothesis so that (1) it implies the desired result, and (2) it is easy to extend. Concentrating on these steps makes the design of many algorithms simpler.

CHOOSING THE INDUCTION SEQUENCE WISELY

In the previous examples, the emphasis in the search for an algorithm was on reducing the size of the problem. This is the essence of the inductive approach. There are, however, many ways to achieve this goal. First, the problem may include several parameters (e.g., left endpoints and right endpoints, vertices and edges in a graph), and we must decide which of those should be reduced. Second, we may be able to eliminate many possible elements, and we want to choose the "easiest" one (e.g., the leftmost endpoint, the smallest number). Third, we may want to impose additional constraints on the problem (e.g., the intervals are in a sorted order). There are many other variations. For example, we can assume that we know how to solve the problem for some other values $< n$ instead of just $n - 1$. This is a valid assumption. Anything that reduces the size of the problem can be considered since it leads back to the base case which we solve directly. Going back to the sorting example discussed in the introduction, we can reduce the sorting of n elements to sorting two subsets of $n/2$ elements. The two sorted subsets can then be merged (leading to an algorithm called *merge sort*). Dividing the problem (inductively) into several equal parts is a very useful technique (which we will discuss later) called *divide and conquer*.

Some reductions are easy to make, some are hard. Some lead to good algorithms, some do not. In many cases this is the only hard part of the problem, and once the right choice is made the rest is easy (e.g., the choice of the element i in the mapping problem). This

is extremely important in mathematics. One never jumps into an induction proof without thinking first how to choose the induction sequence. As expected, it is also very important in algorithm design. In this section we show two examples in which the order of reduction is very important. The celebrity example, in particular, illustrates a nontrivial order.

Topological Sorting

Suppose there is a set of tasks that need to be performed one at a time. Some tasks depend on other tasks and cannot be started until the other tasks are completed. All the dependencies are known, and we want to arrange a schedule of performing the tasks that is consistent with the dependencies (i.e., every task is scheduled to be performed only after all the tasks it is dependent on are completed). We want to design a fast algorithm to generate such a schedule. This problem is called *topological sorting*. The tasks and their dependencies can be represented by a *directed graph*. A directed graph has a set of vertices V (corresponding to the tasks in this case), and a set of edges E which are pairs of vertices. There is an edge from vertex v to vertex w (denote as $(v, w) \in E$) if the task corresponding to v must be performed before the task corresponding to w. The graph must be acyclic, otherwise the tasks on the cycle can never be started. Here is a formulation of the problem in terms of graphs.

The problem: Given a directed acyclic graph $G = (V, E)$ with n vertices, label the vertices from 1 to n such that if v is labeled k then all vertices that can be reached from v by a directed path are labeled with labels $> k$. (A *path* is a sequence of vertices $v_1, v_2, \ldots, v_k$ that are connected by the edges $(v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k)$.)

We again try to solve the problem in terms of a smaller problem. The straightforward way to consider a smaller problem is to remove a vertex. In other words, the induction is on the number of vertices in the following way:

Induction hypothesis 1: *We know how to label all graphs with $n - 1$ vertices according to the conditions above.*

The base case of a graph with one vertex is trivial. If $n > 1$, we can remove one vertex, try to apply the induction hypothesis, and then try to extend the labeling. The first thing we need to verify is that the removal of a vertex results in the same problem as the original (except for the smaller size). It is essential to have the same problem, otherwise the induction hypothesis cannot be used. The only assumption in this case is that the graph is acyclic. Since removing a vertex cannot produce a cycle, this is a valid reduction.

The problem with this reduction is that although the hypothesis can be used, it is not clear how to extend the solution, i.e., how to label the removed vertex. Our method of solving this problem is to choose the vertex with care. We are free to choose any vertex as the nth vertex, since the removal of any vertex results in a valid smaller problem. Therefore, we should remove the vertex whose labeling is the easiest to achieve. An obvious choice is a vertex (task) with no dependencies, namely a vertex whose indegree (the number of edges coming into it) is zero. This vertex can be labeled 1 without any problems.

But, can we always find a vertex of indegree zero? The answer is intuitively yes since we must be able to start somewhere. Indeed, if all vertices had positive indegrees then we could traverse the graph "backwards" and never have to stop. But since there is only a finite number of vertices we must go through a cycle, which is a contradiction to our assumption. (By the same argument we can also start with a vertex of outdegree 0 and label it n.) The rest of the algorithm is now clear. Remove the chosen vertex with its adjacent edges and label the rest of the graph, which is still acyclic of course, with labels 2 to n. Since we need to label the remaining graph with labels 2 to n instead of 1 to $n - 1$ we need to change the induction hypothesis slightly.

Induction hypothesis 2: *We know how to label a graph with $n - 1$ vertices, according to the conditions of the problem, using any set of $n - 1$ distinct labels.*

The corresponding algorithm is given below.

```
Algorithm Topological_Sorting
  (G = (V, E): a directed acyclic graph);
begin
  Initialize v .indegree for all vertices;
    {e.g., by Depth-First Search}
  G_label := 0;
  for i := 1 to n do
    if v_i .indegree = 0 then put v_i in Queue;
  repeat
    remove vertex v from Queue;
    G_label := G_label + 1;
    v .label := G_label;
    for all edges (v, w) do
      w .indegree := w .indegree - 1;
      if w .indegree = 0 then put w in Queue
  until Queue is empty
end;
```

Complexity: Initializing the indegree counters requires $O(|V| + |E|)$ time (using depth first search for example). Finding a vertex with indegree 0 takes constant time (accessing a queue). Each edge (v, w) is considered once (when v is taken from the queue). Thus, the number of times the counters need to be updated is exactly the number of edges in the graph. The running time of the algorithm is therefore $O(|V| + |E|)$, which is linear in the size of the input.

Summary: This is another example in which the inductive approach leads almost directly to an algorithm. The trick here was to choose the order of the induction wisely. We did not reduce the problem arbitrarily, but chose a special vertex to remove. The size of any given problem can be reduced in many possible ways. The idea is to explore a variety of options and test the re-

sulting algorithms. We have seen in the polynomial evaluation example that going from left to right was better than going from right to left. Another common possibility is comparing top down versus bottom up. It is also possible to go in increments of 2 rather than 1, and there are many other possibilities. Sometimes the best induction sequence is not even the same for all inputs. It may be worthwhile to design a special algorithm just to find the best way to perform the reduction.

The Celebrity Problem

This is a popular exercise in algorithm design. It is a very nice example of a problem that has a solution that does not require scanning the whole data (or even a significant part of it). Among n persons, a *celebrity* is defined as someone who is known by everyone but does not know anyone. The problem is to identify the celebrity, if such exists, by only asking questions of the form "Excuse me, do you know this person over there?" (The assumption is that all the answers are correct, and that even the celebrity will answer.) The goal is to minimize the number of questions. Since there are $n(n-1)/2$ pairs of persons, there is potentially a need to ask $n(n-1)$ questions, in the worst case, if the questions are asked arbitrarily. It is not clear that one can do better than that in the worst case.

More technically, if we build a directed graph with the vertices corresponding to the persons and an edge from A to B if A knows B, then a celebrity corresponds to a *sink* (no pun intended). That is, a vertex with indegree $n-1$ and outdegree 0. A graph can be represented by an $n \times n$ *adjacency matrix* whose ijth entry is 1 if the ith person knows the jth person and 0 otherwise. The problem is to identify a sink by looking at as few entries from the matrix as possible.

Consider as usual the difference between the problem with $n-1$ persons and the problem with n persons. Since, by definition, there is at most one celebrity, there are three possibilities: (1) the celebrity is among the first $n-1$, (2) the celebrity is the nth person, and (3) there is no celebrity. The first case is the easiest to handle. We only need to check that the nth person knows the celebrity, and that the celebrity doesn't know the nth person. The other two cases are more difficult since, in order to determine whether the nth person is the celebrity, we may need to ask $2(n-1)$ questions. In the worst case, that leads to exactly $n(n-1)$ questions (which we tried to avoid). We need another approach.

The trick here is to consider the problem "backwards." It may be hard to identify a celebrity, but it is probably easier to identify someone as a noncelebrity. After all, there are definitely more noncelebrities than celebrities. Eliminating someone from consideration is enough to reduce the problem from n to $n-1$. Moreover, we do not need to eliminate someone specific; anyone will do. Suppose that we ask Alice whether she knows Bob. If she does then she cannot be a celebrity; if she doesn't then Bob cannot be a celebrity. We can eliminate one of them with one question.

Now consider again the three cases above. We do not just take an arbitrary person as the nth person. We use the idea above to eliminate either Alice or Bob, and then solve the problem for the other $n-1$ persons. We are guaranteed that case 2 will not occur since the person eliminated cannot be the celebrity. Furthermore, if case 3 occurs, namely there is no celebrity among the $n-1$ persons, then there is no celebrity among the n persons. Only case 1 remains, but, as was mentioned above, this case is easy. If there is a celebrity among the $n-1$ persons it takes two more questions to verify that this is a celebrity for the whole set. Otherwise there is no celebrity.

The algorithm proceeds as follows. We ask A whether she knows B, and eliminate one of them according to the answer. Let's assume that we eliminate A. We then find (by induction) a celebrity among the remaining $n-1$ persons. If there is no celebrity the algorithm terminates; otherwise, we check that A knows the celebrity and that the celebrity does not know A. A nonrecursive implementation of this algorithm is given below.

```
Algorithm Celebrity (Know: n × n boolean matrix);
begin
  i := 1;
  j := 2;
  next := 2;
  {in the first phase we eliminate all but one candidate}
  while next ≤ n do
    next := next + 1;
    if Know[i, j] then i := next
      else j := next;
    {one of either i or j is eliminated}
  if i = n + 1 then candidate := j else candidate := i;
  {Now we check that the candidate is indeed the
    celebrity}
  wrong := false; k := 1;
  Know[candidate, candidate] := false;
    {just a dummy variable to pass the test}
  while not wrong and k ≤ n do
    if Know[candidate, k] then wrong := true;
    if not Know[k, candidate] then
      if candidate ≠ k then wrong := true;
    k := k + 1;
  if not wrong then print "candidate is a celebrity!"
end;
```

Complexity: At most $3(n-1)$ questions will be asked: $n-1$ questions in the first phase so that $n-1$ persons can be eliminated from consideration, and at most $2(n-1)$ questions to verify that the candidate is indeed a celebrity. The solution above shows that it is possible to identify a celebrity by looking at only $O(n)$ entries in the adjacency matrix, even though *a priori* the solution may be sensitive to each of the $n(n-1)$ entries. (It is possible to save an additional $\lfloor \log_2 n \rfloor$ questions by being careful not to repeat, in the verification phase, questions asked during the elimination phase [9].)

Summary: The key idea to this elegant solution is to reduce the problem from n to $n - 1$ in a clever way. This example illustrates that it sometimes pays to spend some effort (in this case one question) to perform the reduction more effectively. In addition, a "backward" approach was found to be very useful for this problem. Instead of searching for a celebrity we tried to eliminate noncelebrities. This is very common in mathematical proofs. One need not approach a problem directly. It may be easier to solve related problems that lead to a solution of the original problem. To find such related problems, it is sometimes useful to start with the finished product (the theorem in a mathematical proof), and work our way backwards to figure out what is needed to construct it.

STRENGTHENING THE INDUCTION HYPOTHESIS

Strengthening the induction hypothesis is one of the most important techniques of proving mathematical theorems with induction. When attempting an inductive proof one often encounters the following scenario. Denote the theorem by $P(n)$. The induction hypothesis can be denoted by $P(n - 1)$ and the proof must conclude that $P(n - 1) \Rightarrow P(n)$. Many times one can add another assumption, call it Q, under which the proof becomes easier. That is, it is easier to prove $[P(n - 1)$ and $Q] \Rightarrow P(n)$. The combined assumption seems correct but it is not clear how to prove it. The trick is to include Q in the induction hypothesis (if it is possible). One now has to prove that $[P$ and $Q](n - 1) \Rightarrow [P$ and $Q](n)$. The combined theorem $[P$ and $Q]$ is a stronger theorem than just P, but sometimes stronger theorems are easier to prove (Polya [14] calls this principle *the inventor's paradox*). This process can be repeated and, with the right added assumptions, the proof becomes tractable. We have seen this principle briefly in the polynomial evaluation and interval containment problems.

In this section we present two examples that illustrate the use of strengthening the induction hypothesis. The first example is rather simple, but it illustrates the most common error made while using this technique, which is to ignore the fact that an additional assumption was added and forget to adjust the proof. In other words, proving that $[P(n - 1)$ and $Q] \Rightarrow P(n)$, without even noticing that Q was assumed. In our analogy this translates to "solving" a smaller problem that is not exactly the same as the original problem. The second example is much more complicated.

Computing Balance Factors in Binary Trees

Let T be a binary tree with root r. The *height* of a node v is the distance between v and the furthest leaf down the tree. The *balance factor* of a node v is defined as the difference between the height of its left child and the height of its right child (we assume that the children of a node are labeled by left or right). Figure 3 shows a tree in which each node is labeled by h/b where h is the node's height and b is its balance factor.

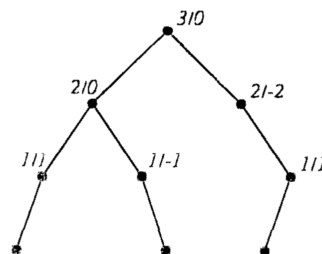

FIGURE 3. A Binary Tree with Heights and Balance Factors Marked

The problem: Given a tree T with n nodes compute the balance factors of all its nodes.

We use the regular inductive approach with the straightforward induction hypothesis:

Induction hypothesis: *We know how to compute balance factors of all nodes in trees with* $<n$ *nodes.*

The base case of $n = 1$ is trivial. Given a tree with $n > 1$ nodes, we remove the root, and solve the problem (by induction) for the two subtrees that remain. We chose to remove the root since the balance factor of a node depends only on the nodes below it. We now know the balance factors of all the nodes except for the root. But the root's balance factor depends not on the balance factors of its children but on their height. Hence, a simple induction does not work in this case. We need to know the heights of the children of the root. The idea is to include the height finding problem within the original problem:

Stronger induction hypothesis: *We know how to compute balance factors **and** heights of all nodes in trees with* $<n$ *nodes.*

Again, the base case is trivial. Now, when the root is considered, its balance factor can be easily determined by the difference between the heights of its children. Furthermore, the height of the root can also be easily determined—it is the maximal height of the two children plus 1.

The key to the algorithm is to solve a slightly extended problem. Instead of just computing balance factors, we also compute heights. The extended problem turns out to be an easier problem since the heights are easy to compute. In many cases, solving a stronger problem is easier. This is especially true for induction. With induction, we only need to extend a solution of a small problem to a solution of a larger problem. If the solution is broader (because the problem is extended) then the induction step may be easier since we have more to work with. It is a very common error to forget that there are two different parameters in this problem, and that each one should be computed separately.

Closest Pair

The problem: Given a set of points in the plane, find the distance between the two closest points.

The straightforward solution using induction would proceed by removing a point, solving the problem for $n-1$ points, and considering the extra point. However, if the only information obtained from the solution of the $n-1$ case is the minimum distance, then the distances from the additional point to all other $n-1$ points must be checked. As a result, the total number of distance computations is $n-1+n-2+\cdots+1=n(n-1)/2$. (This is, in fact, the straightforward algorithm consisting of comparing every pair of points.) We want to find a faster solution.

A Divide and Conquer Algorithm

The induction hypothesis is the following:

Induction hypothesis: *We know how to find the closest distance between two points in a set of $<n$ points in the plane.*

Since we assume that we can solve all subproblems of size $<n$ we can reduce the problem to two subproblems with $n/2$ points. We assume that n is a power of 2 so that it is always possible to divide the set into two equal parts. (We will discuss this assumption later.) There are many ways to divide a set of points into two equal parts. We are free to choose the best division for our purposes. We would like to get as much useful information from the solution of the smaller problems, thus we want as much of it to remain valid when the complete problem is considered. It seems reasonable here to divide the set by dividing the plane into two disjoint parts each containing half the set. After we find the minimal distance in each subset we only have to be concerned with distances between points close to the boundaries of the sets. The easiest way of doing this is by sorting all the points according to their x coordinates for example, and dividing the plane by the vertical line that bisects the set (see Figure 4). (If several points lie on the vertical line we assign them arbitrarily to one of the two sets making sure that the two sets have equal size.) The reason we choose this division is to minimize the work of combining the solutions. If the two parts interleave in some fashion then it will be harder to check for closest pairs. The sorting need be performed only once.

Given the set P we divide it as above into two equal size subsets P_1 and P_2. We then find the closest distance in each subset by induction. We assume that the minimal distance is d_1 in P_1 and d_2 in P_2. We assume further, without loss of generality, that $d_1 \leq d_2$. We need to find the closest distance in the whole set, namely we have to see whether there is a point in P_1 with a distance $<d_1$ to a point in P_2. First we notice that it is sufficient to consider only the points that lie in a strip of width $2d_1$ centered around the vertical separator of the two subjects (see Figure 4). No other points can possibly be of distance less than d_1 from points in the other subset. Using this observation we can usually eliminate many points from consideration, but, in the worst case, all the points can still reside in the strip and we cannot afford to use the straightforward algorithm on them.

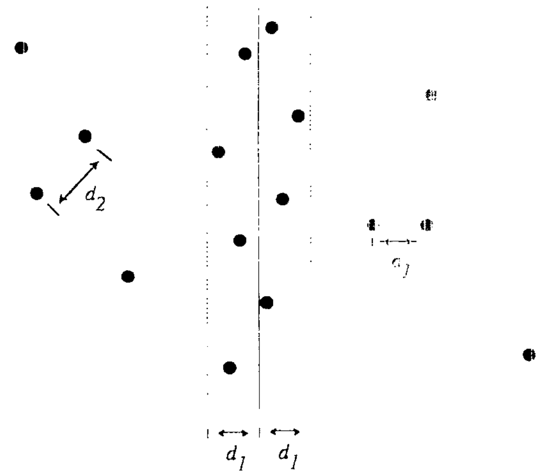

FIGURE 4. The Closest Pair Problem

Another less obvious observation is that for any point p in the strip there are very few points on the other side that can possibly be closer to p than d_1. The reason for this is that all the points in one side of the strip are at least d_1 apart. Let p be a point in the strip with y coordinate y_p, then only the points on the other side with a y coordinate y_q such that $|y_p - y_q| < d_1$ need be considered. There could be at most six such points on one side of the strip (see Figure 5 for the worst case). As a result, if we sort all points in the strip according to their y coordinates and scan the points in order, we only need to check each point against a constant number of its neighbors in the order (instead of all $n-1$ points). We omit the proof of this fact (see for example [15]).

Algorithm Closest_Pair {first attempt}
($p_1, p_2, \ldots, p_n$: points in the plane);
begin
Sort the points according to their x coordinates;
{this sorting is done only once at the beginning}
Divide the set into two equal size parts;
Recursively, compute the minimal distance in each part;
Let d be the minimal of the two minimal distances;
Eliminate points that lie further than d apart from the separation line;
Sort the points according to their y coordinates;
Scan the points in the y order and compute the distances of each point to its 5 neighbors;
{in fact, 4 is sufficient}
if any of these distances is less than d **then**
update d
end;

Complexity: It takes $O(n \log n)$ to sort according to the x coordinates, but this is done only once. We then solve two subproblems of size $n/2$. Eliminating the points outside of the strips can be done in $O(n)$ time. It then takes $O(n \log n)$ steps, in the worst case, to sort according to the y coordinates. Finally, it takes $O(n)$ steps to scan the points inside the strips and compare each one

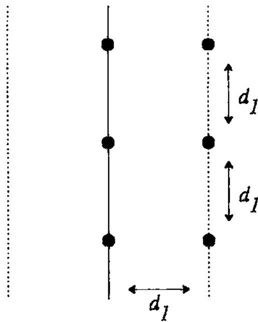

FIGURE 5. The Worst Case of Six Points d_1 Apart

to a constant number of its neighbors in the order. Overall, to find the closest pair in a set with n points we find two closest pairs in subsets with $n/2$ points and then use $O(n \log n)$ time to find the closest pair between the two subsets (plus $O(n \log n)$ once for sorting the x coordinates). This leads to the following recurrence relation:

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

The solution of this relation is $T(n) = O(n \log^2 n)$. This is better than a quadratic algorithm, but we can still do better than that. Now comes the clever use of induction.

An $O(n \log n)$ Algorithm

The key idea here is to strengthen the induction hypothesis. We have to spend $O(n \log n)$ time in the combining step because of the sorting. Although we know how to solve the sorting problem directly, it takes too long. Can we somehow solve the sorting problem at the same time we are solving the closest pair problem? In other words, we would like to strengthen the induction hypothesis to include the sorting as part of the closest pair problem to obtain a better solution.

Induction hypothesis: *Given a set of $<n$ points in the plane, we know how to find the closest distance and how to output the set sorted according to y coordinates.*

We have already seen how to find the minimal distance if we know how to sort. Hence, the only thing that needs to be done is to show how to sort the set of size n if we know the sorted order of the two subsets of size $n/2$. In other words, we need to *merge* the two sorted subsets into one sorted set. Merging can be done in linear time (see for example [1]). Therefore, the recurrence relation becomes

$$T(n) = 2T(n/2) + O(n), T(2) = 1,$$

which implies that $T(n) = O(n \log n)$. The only difference between this algorithm and the previous one is that the sorting according to the y coordinates is not done every time from scratch. We use the stronger induction hypothesis to perform the sorting as we go along. The revised algorithm is given below. This algorithm was developed by Shamos and Hoey [16] (see also [15]).

```
Algorithm Closest_Pair  {An improved version}
  (p1, p2, . . . , pn: points in the plane);
begin
  Sort the points according to their x coordinates;
    {this sorting is done only once at the beginning}
  Divide the set into two equal size parts;
  Recursively do the following:
    compute the minimal distance in each part;
    sort the points in each part according to their
      y coordinates;
  Merge the two sorted lists into one sorted list;
    {Notice that we must merge before we eliminate;
     we need to supply the whole set sorted to the next
     level of the recursion}
  Let d be the minimal of the minimal distances;
  Eliminate points that lie further than d apart from the
    separation line;
  Scan the points in the y order and compute the
    distances of each point to its 5 neighbors;
    {in fact, 4 is sufficient}
  if any of these distances is less than d then
    update d
end;
```

STRONG INDUCTION

The idea of strong induction (sometimes called *structured* induction) is to use not only the assumption that the theorem is true for $n - 1$ (or any other value $<n$), but the stronger assumption that the theorem is true *for all k, $1 \le k < n$.* Translating this technique to algorithm design requires maintaining a data structure with all the solutions of the smaller problems. Therefore, this technique usually leads to more space usage. We present one example of the use of this technique.

The Knapsack Problem

The knapsack problem is a very important optimization problem. It has many different variations, but we will discuss only one variation of it.

The problem: There are n items of different sizes. The ith item has an integer size k_i. The problem is to find a subset of the items whose sizes sum to exactly K, or determine that no such subset exists. In other words, we are given a *knapsack* of size K and we want to pack it fully with items. We denote the problem by $P(n, K)$, where the first parameter denotes the number of items and the second parameter denotes the size of the knapsack. We assume that the sizes are fixed. We denote by $P(j, k)$ (where $j \le n$ and $k \le K$) the knapsack problem with the first j items and a knapsack of size k. For simplicity, we concentrate for now only on the decision problem, which is to determine whether a solution exists. We show how to find a solution later.

We start with the straightforward induction approach.

Induction hypothesis: *We know how to solve $P(n - 1, K)$.*

The base case is easy; there is a solution only if the single element is of size K. If there is a solution to

$P(n-1, K)$, i.e., if there is a way to pack some of the $n-1$ items into the knapsack then we are done; we will simply not use the nth item. Suppose, however, that there is no solution for $P(n-1, K)$. Can we use this negative result? Yes, it means that the nth item must be included. In this case, the rest of the items must fit into a smaller knapsack of size $K - k_n$. We have reduced the problem to two smaller problems: $P(n, K)$ has a solution if and only if either $P(n-1, K)$ or $P(n-1, K-k_n)$ have solutions. To complete the algorithm we have to strengthen the hypothesis. We need to solve the problem not only for knapsacks of size K but for knapsacks of all sizes at most K. (It may be possible to limit the sizes only to those that result by subtracting one of the k_is, but this may not be limiting at all.)

Stronger induction hypothesis: *We know how to solve* $P(n-1, k)$ *for all* $0 \le k \le K$.

The reduction above did not depend on a particular value of k; it will work for any k. We can use this hypothesis to solve $P(n, k)$ for all $0 \le k \le K$: we reduce $P(n, k)$ to the two problems $P(n-1, k)$ and $P(n-1, k-k_n)$. (If $k - k_n < 0$ we ignore the second problem.) Both problems can be solved by induction. This is a valid reduction and we now have an algorithm, but it is very inefficient. We obtained two subproblems of only slightly smaller sizes. The number of subproblems is thus doubled with every reduction in size. Since the reduction in size may not be substantial the algorithm may be exponential (it depends on the values of the k_i's).

Fortunately, it is possible in many cases to improve the running time for these kinds of problems. The main observation is that the *total number of possible problems* is not too high. In fact, we introduced the notation of $P(n, k)$ especially to demonstrate it. There are n possibilities for the first parameter and K possibilities for the second one. Overall, there are only nK different possible problems! The exponential running time resulted from doubling the number of problems after every reduction, but if there are only nK different problems then we must have generated the same problem many many times. If we remember all the solutions we will never have to solve the same problem twice. This is really a combination of strengthening the induction hypothesis and using strong induction (which is using the assumption that all solutions to smaller cases are known and not only that for $n-1$). Let's see how we implement this approach.

We store all the known results in an $n \times K$ matrix. The ijth entry in the matrix contains the information about the solution of $P(i, j)$. The reduction using the stronger hypothesis above basically computes the nth row of the matrix. Each entry in the nth row is computed from two of the entries above it. If we are interested in finding the actual subset then we can add to each entry a flag which indicates whether the corresponding item was selected or not in that step. The flags can then be traced back from the (n, K)th entry and the subset can be recovered. The algorithm is presented at the top of the next column.

```
Algorithm Knapsack (k1, k2, . . . , kn, K: integer);
  {P[i, j].exist = true if there exists a solution to the
   knapsack problem with the first i elements and a
   knapsack of size j.
   P[i, j].belong = true if the ith element belongs to
   this solution.}
begin
  P[0, 0].exist := true;
  for j := 1 to K do
    P[0, j].exist := false;
  for i := 1 to n do
    for j := 0 to K do
      P[i, j].exist := false;  {the default value}
      if P[i − 1, j].exist then
        P[i, j].exist := true;
        P[i, j].belong := false
      else if j − ki ≥ 0 then
          if P[i − 1, j − ki].exist then
            P[i, j].exist := true;
            P[i, j].belong := true
end;
```

Complexity: There are nK entries in the matrix, and each one is computed in constant time from two other entries. Hence, the total running time is $O(nK)$. If the sizes of the items are not too large, then K cannot be too large and nK is much smaller than an exponential expression in n. (If K is very large or if it is a real number then this approach will not be efficient.) If we are only interested in determining whether a solution exists then the answer is in $P[n, K]$. If we are interested in finding the actual subset we can trace back from the (n, K)th entry, using, for example, the *belong* flag in the knapsack program, and recover the subset in $O(n)$ time.

Summary: The method we just used is an instance of a very general technique called *dynamic programming*, the essence of which is to build large tables with all known previous results. The construction of the tables is done iteratively. Each entry is computed from a combination of other entries above it or to the left of it in the matrix. The main problem is to organize the construction of the matrix in the most efficient way. The dynamic programming approach is very effective when the problem can only be reduced to several smaller, but not small enough, subproblems.

MAXIMAL COUNTEREXAMPLE

A distinctive and powerful technique for proving mathematical theorems is by assuming the contrary and finding a contradiction. Usually this is done in a completely nonconstructive manner, which is not very helpful in our analogy. Sometimes though the contradiction is achieved by a procedure similar to induction. Suppose we want to prove that a certain parameter P (in a given problem) can reach a certain value n. First, we show that P can reach a small value (the base case). Second, we assume that P cannot reach n, and we consider the maximal value $k < n$ that it can reach. The final and main step is to present a contradiction, usually to the maximality assumption. We present one

example in which this technique is very helpful in designing algorithms.

Perfect Matching in Very Dense Graphs

A *matching* in an undirected graph $G = (V, E)$ is a set of edges that have no vertex in common. (The edges serve to match their two vertices, and no vertex can be matched to more than one other vertex.) A *maximal* matching is one that cannot be extended, namely all other edges are connected to at least one of the matched vertices. A *maximum* matching is one with maximum cardinality. (A maximum matching is always maximal, but the other way around is not necessarily true.) A matching with n edges in a graph with $2n$ vertices is called a *perfect matching* (it is obviously maximum). In this example we consider a very restricted case. We assume that there are $2n$ vertices in the graph and all of them have degrees of at least n. It turns out that under these conditions a perfect matching always exists. We first present the proof of this fact, and then show how to modify the proof to get an algorithm for finding a perfect matching.

The proof is by maximal counterexample [11]. Consider a graph $G = (V, E)$ such that $|V| = 2n$, and the degree of each vertex is at least n. If $n = 1$ then the graph consists of exactly one edge connecting two vertices, which is a perfect matching. Assume that $n > 1$, and that a perfect matching does not exist. Consider a maximum matching $M \subset E$. $|M| < n$ by the assumption, and obviously $|M| \geq 1$ since any edge is by itself a matching. Since M is not perfect there are at least 2 nonadjacent vertices v_1 and v_2 which are not included in M (i.e., they are not incident to an edge in M). These two vertices have at least $2n$ distinct edges coming out of them. All of these edges lead to vertices that are covered by M since otherwise such an edge could be added to M. Since the number of edges in M is $<n$ and there are $2n$ edges from v_1 and v_2 adjacent to them, at least one edge from M, say (u_1, u_2), is adjacent to three edges from v_1 and v_2. Assume, without loss of generality, that those three edges are (u_1, v_1), (u_1, v_2), and (u_2, v_1) (see Figure 6). It is easy to see that by removing the edge (u_1, u_2) from M and adding the two edges (u_1, v_2), and (u_2, v_1) we get a larger matching, contradicting the maximality assumption.

It may seem at first that this proof cannot yield an algorithm since it starts with a maximum matching. Had we known how to find such a matching we could have solved the problem. However, the steps of the contradiction work for any maximal matching; they present a contradiction only for a maximum matching. A maximal matching is easier to find than a maximum matching. We can simply add disjoint edges (i.e., edges with no vertex in common) until it is no longer possible. Then, we can use the procedure above to transform the matching to a larger one. By the proof above we can continue doing so until a perfect matching is obtained.

OTHER PROOF TECHNIQUES

We concentrated in this article on proof techniques that are based on induction. There are many other techniques and more room for analogies, some obvious, some not so obvious. Many mathematical theorems are proved by series of lemmas, and this corresponds directly to the idea of modular design and structured programming. The idea of proof by contradiction has analogies in the design of algorithms. It would be interesting to study analogies to proofs "by similar arguments."

We briefly present here four more proof techniques (three of which are based on induction) and their analogies. More examples and analogies can be found in [12].

Reductions

A very powerful technique for proving theorems and designing algorithms is using reductions between problems. If problem A can be shown to be a special case of problem B, then the algorithm for B can be used as a "black box" for solving A. This technique is also very useful for proving lower bounds or classifying problems (e.g., NP-Complete problems). If problem A is known to be very hard, then problem B is at least as hard.

Double Induction

This is a method of applying induction to more than just one parameter at a time. It can be used in cases where the best order of induction is not clear and it is easier to apply it to several parameters making choices between them depending on the particular step. For example, if the problem involves n objects in k dimen-

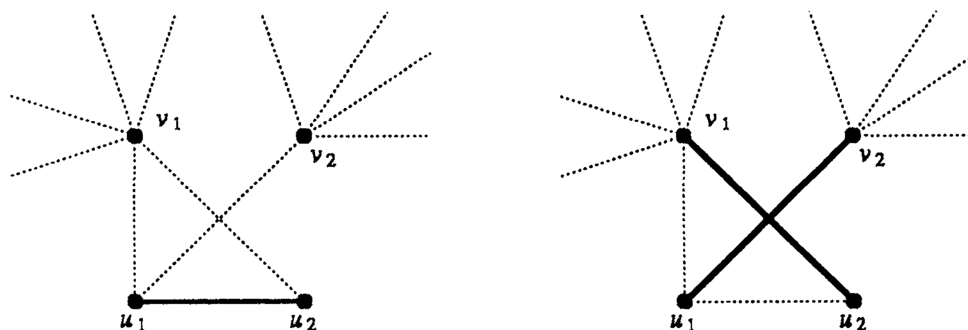

FIGURE 6. Extending a Matching

sional space, we may want to reduce the number of objects and/or the number of dimensions depending on the phase of the algorithm (see for example [4]).

Reversed Induction

This is a little known technique that is not often used in mathematics but is often used in computer science. Regular induction "covers" all natural numbers by starting from a base case ($n = 1$) and advancing. Suppose that we want to go backwards. We want to prove that the theorem is true for $n - 1$ assuming that it is true for n. We call this type of proof a *reversed induction*. But, what would be the base case? We can start by proving a base case of $n = M$, where M is a very large number. If we prove it for $n = M$ and then use reversed induction then we have a proof for all numbers $\leq M$. Although this is usually unsatisfactory, in some cases it is sufficient. For example, suppose we apply double induction on two parameters (e.g., number of vertices and number of edges in a graph). We can apply regular induction to one parameter, and reversed induction to the second parameter if the second parameter can be bounded in terms of the first one. For example, there are at most $n(n - 1)$ edges in directed graphs with n vertices. We can use regular induction on n with the assumption that all edges are present (namely, we consider only complete graphs), and then reversed induction on the number of edges.

A more common use of reversed induction is the following. Proving a base case for only one value of n limits the proof to those numbers less than the value. Suppose that we can prove the theorem directly for an infinite set of values of n. For example, the infinite set can consist of all powers of 2. Then we can use reversed induction and cover all values of n. This is a valid proof technique since for each value of n there is a value larger than it in the base case set (since the set is infinite).

A very good example of the use of this technique is the elegant proof (due to Cauchy) of the arithmetic mean versus geometric mean inequality (see for example [3]). When proving mathematical theorems, it is usually not easier to go from n to $n - 1$ than it is to go from $n - 1$ to n, and it is much harder to prove an infinite base case rather than a simple one. When designing algorithms, on the other hand, it is almost always easy to go from n to $n - 1$, that is, to solve the problem for smaller inputs. For example, one can introduce "dummy" inputs that do not affect the outcome. As a result, it is sufficient in many cases to design the algorithm not for inputs of all sizes, but only for sizes taken from an infinite set. The most common use of this principle is designing algorithms only for inputs of size n which is a power of 2. It makes the design much cleaner and eliminates many "messy" details. Obviously these details will have to be resolved eventually, but they are usually easy to handle. We used the assumption that n is a power of 2, for example, in the closest pair problem.

Choosing the Base of the Induction Wisely

Many recursive algorithms (e.g., quicksort) are not very good for small problems. When the problem is reduced to a small one, another simple algorithm (e.g., insertion sort) is used instead. Viewed in terms of induction, this corresponds to choosing the base case to be $n = k$ (for some k depending on the problem), and using a direct technique to solve the base case. There are examples where this approach can even improve the asymptotic running time of the algorithm [12].

Conclusions

We have presented a methodology for explaining and approaching the design of combinatorial algorithms. The benefits of having such a general methodology are twofold. First, it gives a more unified "line of attack" to an algorithm designer. Given a problem to solve, one can go through the techniques described and illustrated in this article and attempt a solution. Since these techniques have something in common (namely mathematical induction), the process of trying them all can be better understood and easier to carry out. Second, it gives a more unified way to explain existing algorithms and allows the student to be more involved in the creative process. Also, the proof of correctness of the algorithm becomes a more integral part of the description. We believe that this methodology should be included in the teaching of combinatorial algorithms.

Acknowledgments. Thanks to Rachel Manber for many inspiring conversations about induction and algorithms. Greg Andrews, Darrah Chavey, Irv Elshoff, Susan Horwitz, Sanjay Manchanda, Kirk Pruhs, and Sun Wu provided many helpful comments that improved the manuscript.

REFERENCES

1. Aho, A., Hopcroft, J., and Ullman, J. *The Design and Analysis of Computer Algorithms.* Addison Wesley, Reading, Mass., 1974.
2. Bates, J.L., and Constable, R.L. Proofs as programs. *ACM Trans. Prog. Lang. and Syst.* 7, 1 (Jan. 1985), pp. 113–136.
3. Beckenbach, E., and Bellman, R. *An Introduction to Inequalities.* New Mathematical Library, Random House, 1961.
4. Bently, J.L. Multidimensional divide-and-conquer. *Commun. ACM* 23, 4 (April 1980), 214–229.
5. Burge, W.H. *Recursive Programming Techniques*, Addison Wesley, Reading, Mass., 1975.
6. Constable, R.L. et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall, Englewood Cliffs, N.J., 1986.
7. Dijkstra, E.W. *A Discipline of Programming.* Prentice Hall, Englewood Cliffs, N.J., 1976.
8. Gries, D. *The Science of Programming*, Springer-Verlag, New York, 1981.
9. King, K.N., and Smith-Thomas, B. An optimal algorithm for sink-finding, *Inf. Proc. Letters* 14, 3 (1982), pp. 109–111.
10. Knuth, D.E. *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms.* 2nd ed., Addison Wesley, Reading, Mass., 1981.
11. Lovasz, L. *Combinatorial Problems and Exercises*, North Holland, 1979.
12. Manber, U. *Introduction to Algorithms—A Creative Approach.* Addison Wesley, Reading, Mass., 1989, to appear.
13. Paull, M.C. *Algorithm Design—A Recursion Transformation Framework*, John Wiley and Sons, New York, 1988.
14. Polya, G. *How to Solve It.* 2nd ed., Princeton University Press, 1957.
15. Preparata, F., and Shamos, M.I. *Computational Geometry*, Springer-Verlag, New York, 1985.
16. Shamos, M.I., and Hoey, D. Closest-point problems, In *Proceedings of the 16th Annual Symposium on Foundations of Computer Science*, (Berkeley, Calif., Oct. 1975). 1975.

CR Categories and Subject Descriptors: D.2.10 [**Software Engineering**]: Design—*methodologies*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*complexity of proof procedures, computation on discrete structures, geometrical problems and computation, sorting and searching*; G.2.1 [**Discrete Mathematics**]: Combinatorics—*combinatorial algorithms*; G.2.2 [**Discrete Mathematics**]: Graph Theory—*graph algorithms*; K.3.2 [**Computers and Education**]: Computer and Information Science Education—*computer science education*

General Terms: Algorithms, Design, Education, Theory

Additional Key Words and Phrases: Combinatorial algorithms, computational complexity, design of algorithms, mathematical induction, proof techniques

ABOUT THE AUTHOR:

UDI MANBER is an associate professor of computer science at the University of Arizona. He was previously an associate professor at the University of Wisconsin–Madison. His research interests include design of algorithms, distributed computing, and computer networks. He received the Presidential Young Investigator Award in 1985. He is currently completing a book entitled *Introduction to Algorithms—A Creative Approach*, which is based on this article. Author's present address: Udi Manber, Department of Computer Science, University of Arizona, Tucson, AZ 85721.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

ACM SPECIAL INTEREST GROUPS

ARE YOUR TECHNICAL INTERESTS HERE?

The ACM Special Interest Groups further the advancement of computer science and practice in many specialized areas. Members of each SIG receive as one of their benefits a periodical exclusively devoted to the special interest. The following are the publications that are available—through membership or special subscription.

SIGACT NEWS (Automata and Computability Theory)

SIGAda Letters (Ada)

SIGAPL Quote Quad (APL)

SIGARCH Computer Architecture News (Architecture of Computer Systems)

SIGART Newsletter (Artificial Intelligence)

SIGBDP DATABASE (Business Data Processing)

SIGBIO Newsletter (Biomedical Computing)

SIGCAPH Newsletter (Computers and the Physically Handicapped) Print Edition

SIGCAPH Newsletter, Cassette Edition

SIGCAPH Newsletter, Print and Cassette Editions

SIGCAS Newsletter (Computers and Society)

SIGCHI Bulletin (Computer and Human Interaction)

SIGCOMM Computer Communication Review (Data Communication)

SIGCPR Newsletter (Computer Personnel Research)

SIGCSE Bulletin (Computer Science Education)

SIGCUE Bulletin (Computer Uses in Education)

SIGDA Newsletter (Design Automation)

SIGDOC Asterisk (Systems Documentation)

SIGGRAPH Computer Graphics (Computer Graphics)

SIGIR Forum (Information Retrieval)

SIGMETRICS Performance Evaluation Review (Measurement and Evaluation)

SIGMICRO Newsletter (Microprogramming)

SIGMOD Record (Management of Data)

SIGNUM Newsletter (Numerical Mathematics)

SIGOIS Newsletter (Office Information Systems)

SIGOPS Operating Systems Review (Operating Systems)

SIGPLAN Notices (Programming Languages)

SIGPLAN FORTRAN FORUM (FORTRAN)

SIGSAC Newsletter (Security, Audit, and Control)

SIGSAM Bulletin (Symbolic and Algebraic Manipulation)

SIGSIM Simuletter (Simulation and Modeling)

SIGSMALL/PC Newsletter (Small and Personal Computing Systems and Applications)

SIGSOFT Software Engineering Notes (Software Engineering)

SIGUCCS Newsletter (University and College Computing Services)