A PRINCIPLE OF ALGORITHM DESIGN ON LIMITED PROBLEM DOMAIN

Jayadev Misra
Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

Abstract

This paper studies the problem of algorithm design on well defined data structures. A general principle is presented which is shown to be useful in designing algorithms which operate on sequences (strings). A generalization of the principle is presented for more general data structures. Implications of these results are discussed.

Introduction

Lately there has been considerable interest in identifying approaches to problem solving rather than a specific algorithm for a specific problem. The goal of such research is to systematize the intuitive process of algorithm construction. Once certain systematic schemes for handling a class of problems are known, a programmer would attempt to apply these methods rather than trying to develop one by some ad hoc technique. An example of a general principle is "depth first search" which has been explored in great detail for problem solving on undirected graphs [7].

Admittedly, the approaches usually taken for problem solving depend on human intuition and differ considerably from problem to problem and person to person. However it turns out that in a surprisingly large number of cases similar methods of attack are often successful on problems having similar structure. Clearly a mathematical study of problem solving must include a rigorous definition of the structure of a problem (so that conditions may be derived under which two problems may considered to be of identical structure), a rigorous formulation of the problem solving approach and conditions under which a certain approach is applicable to a certain problem. There has been very little work on any of the above areas. Classical work of Bellman on dynamic programming [1] formulated a certain approach to solving problems which are of a decomposable type.

Though a mathematical formulation is desirable and is an important research area, it is hard (with the current methods) to characterize any nontrivial class of problems. Difficulty arises due to many special restrictions attached to a specific problem. It is thus worthwhile to study approaches which may be termed systematic though not algorithmic. Such schemes usually involve some human intuition. Dynamic programming may be viewed as a process where the applicability criterion is dependent on human intuition.

In this paper, we will mainly study the class of problems which arise in connection with sequences (strings/one dimensional files). The problems are usually of the type where a certain quantity needs to be computed from a given sequence by a "left to right scan" type of algorithm. We will present a systematic way of generating such algorithms by a process of iteration. Human intuition would be involved in answering questions such as, "given x, is it possible to compute y?" or "what is needed to compute y, given x?". We believe that algorithm designers follow a similar process without explicit recognition of the iterations. (Brighter ones, of course, skip all the iterations). We will consider certain nontrivial problems and apply the principle to obtain algorithms. In a later section, we discuss the general problem of designing algorithms which compute specific functions on nonlinear structures such as trees and graphs. A generalization of the principle for sequences is shown to result in a principle for more general structures. Implications of these results are discussed in the final section.

Problems on Sequences

We will consider sequences of the form $x:(x_1\ x_2\ x_3 \ldots x_n)$, $n \geq 1$, where each x_i is of a certain given type. Character strings, sequences of integers etc. fit into the above definition. We will not put any a-priori bound on n, the length of the sequence; the implication is that the algorithms which would be designed, would be general enough to work for all nonnull sequences.

Let D be a certain function that is defined on any sequence: $D(x_1\ x_2 \ldots x_n)$ denotes the value of the function on the sequence $(x_1\ x_2 \ldots x_n)$. Our problem is to design an algorithm which computes D for any input sequence x. For instance, D may be a simple term (the value of the maximum element on an integer sequence), a substring (a longest contiguous subsequence of a character sequence that does not contain the character ",") or another sequence (the sorted sequence from the given one). In fact D could be any conceivable function on sequences.

Usual methods of attack for computing D, is to compute $D(x_1)$ and then successively $D(x_1\ x_2)$, $D(x_1\ x_2\ x_3)$ etc. where $D(x_1\ x_2 \ldots x_{i+1})$ is computed from $D(x_1 \ldots x_i)$ and x_{i+1}. Thus finally $D(x_1\ x_2 \ldots x_n)$ is obtained.

Example 1: Find the maximum of any sequence $(x_1 \ldots x_n)$, $n \geq 1$, where each x_i is a positive integer. Clearly, $\text{Max}\ (x_1) = x_1$. and $\text{Max}\ (x_1\ x_2 \ldots x_{i+1}) = \text{maximum}\ (\text{Max}\ (x_1\ x_2 \ldots x_i)\ ,\ x_{i+1})$. Thus an iterative algorithm as shown below may be employed.

```
begin
    Max: = $x_1$;
    for i: = 2 to n do
        Max: = maximum (Max, $x_i$);
    end for;
end;
```

However it is often impossible to compute $D(x_1\ x_2 \cdots x_{i+1})$ solely from $D(x_1\ x_2 \ldots x_i)$ and x_{i+1}. Suppose for instance that $D(x_1\ x_2 \ldots x_i)$ represents the length of the longest substring (contiguous elements of the sequence) that does not contain the character ",". Then the knowledge of $D(x_1\ x_2 \ldots x_i)$ and x_{i+1} is not sufficient for computing $D(x_1 \ldots x_{i+1})$. Consider the string"a, bc, pqr,". It is not possible to compute D(a,bc,pqr,) given only that the last character is a "," and D for the substring excluding the "," is equal to 2. In this case, we need to compute and carry along something more than D.

Example 2: Suppose we want to compute the length of the longest substring not containing ",". Then we may compute

$D_1(x_1\ x_2 \ldots x_i)$ = length of the longest substring in $(x_1\ x_2 \ldots x_i)$ not containing ","

$D_2\ (x_1\ x_2 \ldots x_i)$ = length of the substring following the last (rightmost) "," in $(x_1\ x_2 \ldots x_i)$. (It is equal to i if there is no "," in $x_1\ x_2 \ldots x_i$).

Thus D_1(a,bc,pqr) = 2

D_2(a,bc,pqr) = 3

Now, if x_1 = "," then $D_1(x_1) = 0$ else $D_1(x_1) = 1$

Similarly if x_1 = "," then $D_2(x_1) = 0$ else $D_2(x_1) = 1$.

Next suppose that $D_1(x_1 \ldots x_i)$ and $D_2(x_1 \ldots x_i)$ have been computed. Then $D_1(x_1 \ldots x_{i+1})$ and $D_2(x_1 \ldots x_{i+1})$ may be computed as follows:

```
If $x_{i+1} \neq$ "," then
    $D_1\ (x_1\ x_2 \ldots x_{i+1}) = D_1(x_1 \ldots x_i)$
    $D_2\ (x_1\ x_2 \ldots x_{i+1}) = 1 + D_1(x_1 \ldots x_i)$.
If $x_{i+1}$ = "," then
    $D_2(x_1 \ldots x_{i+1}) = 0$
    $D_1\ (x_1 \ldots x_{i+1})$ = maximum ($D_1(x_1 \ldots x_i)$,
        $D_2(x_1 \ldots x_i)$)
```

These equations follow from the definition of D_1, D_2.□

We needed to compute and carry along D_1, D_2 even though we only need the final value of D_1. This situatión occurs quite often in solving problems on sequences as well as on more general data structures. Thus the major aspect of algorithm construction for computing D involves identifying certain set of quantities D' such that

(i) $D'(x_1)$ is easy to compute.

(ii) $D'(x_1 \ldots x_{i+1})$ can be obtained easily from $D'(x_1 \ldots x_i)$ and x_{i+1}.

(iii) $D(x_1 \ldots x_i)$ can be computed easily from $D'(x_1 \ldots x_i)$.

Then the algorithm for computing D looks as follows:

```
begin
    compute D': = $D'(x_1)$
    for i: = 2 to n do
        compute new D' from old D' and $x_i$;
    compute D from D';
    end for;
end;
```

In the example 2, D' = $\{D_1, D_2\}$. Trivially $D'(x_1 \ldots x_i)$ may just be the string $(x_1 \ldots x_i)$. This D' however yields little clue as to how to proceed from one step to the next; to compute $D'(x_1 \ldots x_{i+1})$ from $D'(x_1 \ldots x_i)$ and x_{i+1}. Principle A, given below will almost always yield a nontrivial D'. Since D' represents the amount of information that needs to be carried along, D' should be as small as possible. Furthermore, we should be able to compute (ii), (iii) as fast as possible. The following principle uses a scheme similar to successive approximation, for locating D', starting with D as an initial estimate of D'. We will assume through out that $D(x_1)$ is easy to compute. In the following description, D_i,D'_i would denote $D(x_1 \ldots x_i)$ and $D'(x_1 \ldots x_i)$ respectively.

Principle A: (for locating D', given D)

```
begin
    Let D': = D;
    While $D'_{i+1}$ can not in general be obtained
        easily from $D'_i$, $x_{i+1}$ (for arbitrary $i \geq 1$) do
    Let D" be some quantity such that $D'_{i+1}$ can in
        general be computed (easily) from $D''_i$, $x_{i+1}$;
        D': = D"
    end do;
end;
```

This principle may be applied for computing successive D' such that the final D' obtained meets condition (ii). Usually condition (i) is trivially met. Condition (iii) would be met since computation of D' would include the computation of D.

Example 2 (continued):

We apply the principle to the problem of locating the length of a longest substring not containing ",", in a character string. Initially let D'_i be the length of the longest substring in $(x_1 \ldots x_i)$ not containing ",". As we noticed earlier, D'_{i+1} can not be computed from D'_i and x_{i+1}. Hence we have to locate D" such that, we can compute the length of the longest substring in $(x_1 \ldots x_{i+1})$ given D''_i and x_{i+1}. D" = $\{D_1, D_2\}$ is a suitable condidate, where D_1, D_2 are as defined in example 2. Next we consider whether D'_i and x_{i+1} are sufficient to compute D'_{i+1}. Since this is affirmative, we terminate the process. □

The next example illustrates the power of the proposed method on a nontrivial problem.

Example 3: Given a sequence $x = (x_1\ x_2 \ldots x_n)$, $n \geq 1$, of positive integers, it is required to find a longest ascending subsequence (not necessarily contiguous); i.e. a subsequence $(x_{i_1}\ x_{i_2} \ldots x_{i_r})$ such that

$i_1 < i_2 \ldots < i_\gamma$ and $x_{i_1} < x_{i_2} \ldots < x_{i_\gamma}$ and γ is as large as possible. For the sequence x = (6 7 3 5 1 9 2 12), two longest ascending subsequences are (3 5 9 12) and (6 7 9 12).

We will abbreviate longest ascending subsequence by LAS. Algorithms for the problem appear in [3,6]. The following algorithm derived from Principle A closely resembles an algorithm developed independently by Matuszek [5].

We start the process with D' = D = LAS. Next we ask the question whether it is possible to obtain D'_{i+1} (easily) from D'_i and x_{i+1}; i.e. given any LAS for $(x_1 \ldots x_i)$ and x_{i+1}, is it possible to obtain an LAS for $(x_1 \ldots x_{i+1})$? Suppose the last element of the LAS for $(x_1 \ldots x_i)$ is larger than x_{i+1}. In this case the current LAS can not be extended. However, if there is another LAS whose last element is less than x_{i+1}, then that LAS can be extended. We thus conclude that from an arbitrary LAS for $(x_1 \ldots x_i)$ and x_{i+1}, we can not in general get another LAS for $(x_1 \ldots x_{i+1})$.

However, these arguments show that if we pick an LAS from $(x_1 \ldots x_i)$ whose last element is as small as possible, we can compute an LAS for $(x_1 \ldots x_{i+1})$, from this LAS and x_{i+1}: if x_{i+1} is smaller than the last element of the current LAS, the current LAS is an LAS for $(x_1 \ldots x_{i+1})$; otherwise the current LAS can be extended to include x_{i+1}. Thus let D'_i be the (unique) LAS from $(x_1 \ldots x_i)$ whose last element is as small as possible. We call such an LAS, a best LAS or BLAS.

At the next iteration, we ask the question whether a $BLAS_{i+1}$ can be obtained easily from $BLAS_i$ and x_{i+1}. Suppose we can extend the $BLAS_i$ by addition of x_{i+1} at the end. Then this must necessarily be $BLAS_{i+1}$ (since there is no other ascending subsequence of equal length whose last element is other than x_{i+1}). Next suppose that $BLAS_i$ can not be extended by adding x_{i+1} at the end, which would be the case if x_{i+1} is smaller than the last element of $BLAS_i$. Is $BLAS_i$ equal to $BLAS_{i+1}$ in that case? Note that BLAS (1 2 5) = (1 2 5). However, BLAS(1 2 5 3) = (1 2 3). Thus unfortunately the answer is in negative to the question posed above. Using the arguments presented previously, we see that $BLAS_{i+1}$ can be obtained from x_{i+1}, $BLAS_i$ and the best ascending subsequence whose length is one less than $BLAS_i$. Thus D' for the next iteration is BLAS and best ascending sequence of length one less than BLAS (we denote it by BLAS(-1)).

On the next iteration, using similar arguments, we conclude that in order to compute $BLAS_{i+1}$ and $BLAS_{i+1}(-1)$, we need to have x_{i+1}, $BLAS_i(-1)$ and $BLAS_i(-2)$. It is then easy to see that continuing iterations will lead us to require D' = $\{$BLAS, BLAS(-1), BLAS(-2)....$\}$, i.e. D' will be the set of best ascending subsequences of all possible lengths starting from length 1 and including the best longest ascending subsequence. It may be easily verified that such a D' indeed satisfies the condition for the iteration in Principle A, to terminate.

Now that we have located a proper D' to carry forward the computation, the major design problems have almost been solved. It remains to design the proper data structure for D' such that D'_{i+1} may be quickly computed from D'_i and x_{i+1}. We omit these details here, noting that only the last elements of various BLAS need to be stored. Since these elements would be sorted (a best ascending sequence of length j must have a last element which is strictly smaller than the last element in the best ascending sequence of length j+1), a binary search can be carried out with x_{i+1} to locate that particular best ascending sequence which should be modified (extended). $\square$

We note the following important facts above the proposed scheme, as exemplified above.

(1) We had no need to decide the complexity of computing D'_{i+1} from D'_i and x_{i+1}. Through out the example (except at the very end) it was impossible to compute D'_{i+1} from D'_i and x_{i+1}. Thus we did not have to make any difficult design decisions of comparing alternate D". Fortunately, this turns out to be the case in all the examples that we have studied.

(2) D can almost always be computed easily from D'. This is a consequence of the initialization and our insistence that we only consider those D' from which the previous D' can be computed.

(3) D" usually includes D' in successive iterations. In the example studied, the successive approximations led from any LAS to BLAS to best ascending sequence of all lengths. Such refinements are almost always encountered in dynamic programming type situations, where in order to compute a certain function, a more general function is computed. The value of the desired function is obtained by setting certain arguments in the generalized function to certain specific values.

(4) D' finally obtained may be regarded as being closed in that D'_i and x_{i+1} permit computation of D'_{i+1}. Closure in this sense is very general and is almost always a property of dynamic programming problems. There are two conceivable techniques of creating such a closed set: we start with a large set and shrink its size. Or we start from a small set and increase its size. The first technique corresponds to starting with D'_i being the string $(x_1 \ldots x_i)$, which is the maximum amount of information we would ever need to compute any function. The difficulty with this approach is that little, if any, clue exists as to how to proceed. We have adopted the second technique in this paper; in addition to providing a systematic scheme for progressing toward a solution, we are assured at every step that D can indeed be computed from D'.

(5) Human intuition played an important role in deciding whether a certain quantity can be computed from some other quantity. This question is usually quite simple to answer (by enumerating several possibilities) and determining whether there is enough information to carry forward the computation.

Extensions to Other Data Structures

Ideas of the previous section can be extended to compute useful functions on data structures other than sequences, such as trees, etc. Usually there are two kinds of generalization involved in computing D over such a data structure.

Structural Generalization: We may compute the quantity over various substructures of the original structure, each substructure being of the same type as the original structure. Finally, we combine the various D's computed on substructures to compute D over the original structure. Note that D for each of the substructures may be computed by the same technique by decomposing it into its substructures.

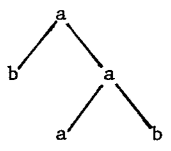
Examples of structural generalization abound: finding the maximum element in an integer sequence (where substructures are the subsequences which start from the beginning of the sequence) and finding the maximum path length in a tree(substructures are subtrees) are two simple applications. Substructure generalization is a basic property of many algorithms that work on recursively defined data structures.

Functional Generalization: It is often necessary to compute something more than D on substructures so that D may be computed over the original structure. We have discussed the need for such a generalization on sequences. Similar arguments apply to other recursive data structures. A generalization of principle A for such structures is given below.

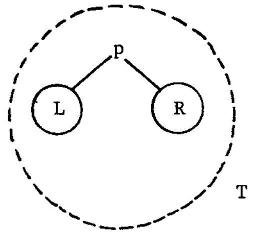
Principle A': Do a substructure decomposition; i.e., identify the different substructures of the original structure and substructures of substructures etc.; Apply Principle A to locate D' such that for any substructure S, given D' values of all its substructures (and some nominal information about the substructure S itself) D' for S can be computed. $\square$

Clearly, we could then successively compute D' of larger substructures and then compute D of the original structure from it. We illustrate the application of Principle A' in the next example.

Example 4: Consider a binary tree each node of which holds a certain symbol. Symbols may be repeated. An example of such a tree is shown below.



Given some symbol s, it is required to find out if the symbol s occurs on any longest path from root to a terminal node. An usual substructure decomposition with trees is to take each subtree as the substructure. D is a yes/no type of answer. Applying Principle A, we first ask whether, given yes/no from the two subtrees L,R and the symbol p (as shown below), we could compute yes/no for the entire tree T. For the moment, we ignore the possibility that either or both L,R may be null .



Consider the case when D_L = "no" and D_R = "yes" and $p \neq s$. Suppose the maximum path length in L is larger than R; then the answer should be "no". However, if maximum pathlength in L is less than or equal to that in R then the answer should be "yes". Hence we conclude that it is impossible to compute D_T from D_L, D_R and p.

Using the above argument, we may set $D'_J = \{D_J$, maximum path length in $J\}$ for any subtree J. In the next iteration we question whether D'_L, D'_R and p can be used to compute D_T and maximum pathlength in T. Let P_L,P_R denote the maximum pathlength from L,R respectively. Clearly, $P_T = 1 + \max(P_L, P_R)$. Following table lists the value of D_T for various possibilities.

D_L	D_R	p	D_T
no	no	$\neq$s	no
no	no	=s	yes
yes	no	$\neq$s	yes if $P_L \geq P_R$; No otherwise.
yes	no	=s	yes
no	yes	$\neq$s	yes if $P_R \geq P_L$; No otherwise.
no	yes	=s	yes
yes	yes	$\neq$s	yes
yes	yes	=s	yes

From the above table, it is clear what needs to be done for the case of null tree: maximum pathlength for a null tree is zero and D should be "no". $\square$

Conclusion and Summary

The problem of designing algorithms on certain classes of data structures has been considered. A principle of successive approximation is presented which yields the required information to be computed of the substructures in order to (recursively) compute a particular function on the original structure. The method could be applied systematically where every iteration involves answering questions of the type "can x be computed from y?" or "what is needed to compute x?". Interestingly, answers to such questions are often reduced to enumerating several mutually exclusive possibilities and answering questions individually for each one.

The algorithms generated by principle A are of "left to right scan" type. Hence, a sorting algorithm designed by using Principle A, would most likely be similar to insertion sort. An efficient algorithm such as quick sort [4] requires a deeper understanding of the problem structure. The proposed method is not intended to replace a mathematical analysis of the problem. There are problems for which no left to right scan algorithm exists and hence they can not be solved by techniques presented in this paper. Furthermore, the analysis involved in applying the principle will vary from person to person leading to different algorithms. However, we believe that such unified principles are useful heuristics for construction of a large class of algorithms.

Acknowledgment

This research was partially supported by National Science Foundation Grant DCR75-09842.

References

[1] Bellman, R. E., Dynamic Programming, Princeton University Press, 1957.

[2] Dijkstra, E. W., O. J. Dahl, C.A. R. Hoare, Structured Programming, Academic Press, 1972.

[3] Fredman, M. L., "On Computing the Length of the Longest Increasing Subsequence", Discrete Mathematics, vol. 11, Jan. 1975, pp. 29-35.

[4] Knuth, D. E., Art of Computer Programming: Sorting and Searching, Addison-Wesley, 1972.

[5] Matuszek, D., Personal Communication

[6] Szymanski, T. G., "A Special Case of the Maximal Common Subsequence Problem", Tech. Report, Electrical Engineering Dept., Princeton University, 1975.

[7] Tarjan, R. E., "Dept First Search and Linear Graph Algorithms", Siam J. of Computing, vol.1, no. 2, June 1972.