CSE 101 - ALGORITHMS - SUMMER 2000

Lecture Notes 6

Monday, July 19, 2000.

4.1.7 Single Word Reusable Boggle

You are given an $n \times m$ matrix L of letters from a finite alphabet V, and a target word W in $V^\star$ of length p. You want to determine whether there is a (not necessarily simple, *i.e.* letters can be reused) path in the grid so that the letters along that path are W. Each step in the path can go from a point in the grid to any of its 8 neighboring points, including diagonal moves. You are asked to give an efficient algorithm to play SWoRB.

Step 1) Let $\text{MATCH?}(L, i, j, W, k)$ indicate whether there is a path on the grid L starting with position (i, j), so that the letters along the path are $W[k...p]$. Since the letters on the grid are reusable, one can easily see the following recurrent optimum condition:

$$\text{MATCH?}(L,i,j,W,k) = \begin{cases} \text{YES}, & \text{when } k = p+1 \text{ or when } i \in 1..n,\ j \in 1..m,\ L[i,j] = W[k] \text{ and at least one of} \\ & \text{MATCH?}(L,i,j+1,W,k+1), \text{MATCH?}(L,i+1,j+1,W,k+1), \\ & \text{MATCH?}(L,i+1,j,W,k+1), \text{MATCH?}(L,i+1,j-1,W,k+1), \\ & \text{MATCH?}(L,i,j-1,W,k+1), \text{MATCH?}(L,i-1,j-1,W,k+1), \\ & \text{MATCH?}(L,i-1,j,W,k+1), \text{MATCH?}(L,i-1,j+1,W,k+1) \text{ is YES}, \\ \text{NO}, & \text{otherwise.} \end{cases}$$

If one wants to trace the word on the grid, then one should maintain a table of local decissions, saying at each step in the game what's the next direction one should look at. For this reason, let $next[1..n+1, 1..m+1]$ be a table that may contain any of the elements $\{\bullet, \rightarrow, \searrow, \downarrow, \swarrow, \leftarrow, \nwarrow, \uparrow, \nearrow\}$, but which initially contains only $\bullet$. It is easy now to write a recursive inefficient pseudocode following the optimization condition. We proceed directly to the next step.

Step 2) We need to store the local decissions for all possible indexes i, j, k above. Therefore, in addition to the table $next$ above, let $s[1..n, 1..m, 1..k]$ be a three dimensional table that may contain any of the values {YES, NO, UNDEFINED}, which is assumed initially filled with only the value UNDEFINED. Then

$\text{MATCH?}(L, i, j, W, k)$

if $s[i,j,k] \neq$ UNDEFINED **then return** $m[i,j,k]$

if $k = p + 1$ **then return** $m[i,j,k] \leftarrow$ YES

if $i \notin 1..n$ or $j \notin 1..m$ or $L[i,j] \neq W[k]$ **then return** $s[i,j,k] \leftarrow$ NO

if $\text{MATCH?}(L,i,j+1,W,k+1)$ **then** { $next[i,j] \leftarrow$ "$\rightarrow$"; **return** $s[i,j,k] \leftarrow$ YES }

if $\text{MATCH?}(L,i+1,j+1,W,k+1)$ **then** { $next[i,j] \leftarrow$ "$\searrow$"; **return** $s[i,j,k] \leftarrow$ YES }

if $\text{MATCH?}(L,i+1,j,W,k+1)$ **then** { $next[i,j] \leftarrow$ "$\downarrow$"; **return** $s[i,j,k] \leftarrow$ YES }

if $\text{MATCH?}(L,i+1,j-1,W,k+1)$ **then** { $next[i,j] \leftarrow$ "$\swarrow$"; **return** $s[i,j,k] \leftarrow$ YES }

if $\text{MATCH?}(L,i,j-1,W,k+1)$ **then** { $next[i,j] \leftarrow$ "$\leftarrow$"; **return** $s[i,j,k] \leftarrow$ YES }

if $\text{MATCH?}(L,i-1,j-1,W,k+1)$ **then** { $next[i,j] \leftarrow$ "$\nwarrow$"; **return** $s[i,j,k] \leftarrow$ YES }

if $\text{MATCH?}(L,i-1,j,W,k+1)$ **then** { $next[i,j] \leftarrow$ "$\uparrow$"; **return** $s[i,j,k] \leftarrow$ YES }

if $\text{MATCH?}(L,i-1,j+1,W,k+1)$ **then** { $next[i,j] \leftarrow$ "$\nearrow$"; **return** $s[i,j,k] \leftarrow$ YES }

return $s[i,j,k] \leftarrow$ NO.

The main algorithm can be now devised as follows :

$\text{SWORB}(A, W)$

for $i \leftarrow 1$ **to** n **do**

  for $j \leftarrow 1$ **to** m **do**

    if $\text{MATCH?}(A,i,j,W,1)$ **then** { $\text{OUTPUT-WORD}(L,i,j,next)$; **return** YES }

return NO

where $\text{OUTPUT-WORD}(L, i, j, next)$ is a procedure that is intended to highlight the word-path in the letter matrix L. We only show how the letters can be succesively output:

OUTPUT-WORD$(L, i, j, next)$

case $next[i, j]$ **of**

  "•" : **return**

  "$\rightarrow$" : **output** $L[i, j]$; OUTPUT-WORD$(L, i, j + 1, next)$

  "$\searrow$" : **output** $L[i, j]$; OUTPUT-WORD$(L, i + 1, j + 1, next)$

  "$\downarrow$" : **output** $L[i, j]$; OUTPUT-WORD$(L, i + 1, j, next)$

  "$\swarrow$" : **output** $L[i, j]$; OUTPUT-WORD$(L, i + 1, j - 1, next)$

  "$\leftarrow$" : **output** $L[i, j]$; OUTPUT-WORD$(L, i, j - 1, next)$

  "$\nwarrow$" : **output** $L[i, j]$; OUTPUT-WORD$(L, i - 1, j - 11, next)$

  "$\uparrow$" : **output** $L[i, j]$; OUTPUT-WORD$(L, i - 1, j, next)$

  "$\nearrow$" : **output** $L[i, j]$; OUTPUT-WORD$(L, i - 1, j + 1, next)$

Step 3) ...

Note: A similar problem is that of *counting* the number of paths through M that form a word w. This could be accomplished by a similar algorithm that used the same subproblems, but stored a count of successful paths to a particular point instead of simply if one existed or not. The final step would sum over all counts on the final plane. This could be further extended to randomly sample from the paths uniformly by selecting each backward step with probability proportional to its share of the paths.

4.2 Other Dynamic Programming Exercises

Exercise 4.1 Write a dynamic programming algorithm for the following problem:

LARGEST-SQUARE

INPUT: A matrix $n \times m$ of bits, A.

OUTPUT: A largest squared submatrix of A containing only bits 1.

A bit can be either 0 or 1. A squared matrix is a matrix $k \times k$. Output the answer as a triple (i, j, k), where (i, j) are the coordinates of the upper left-hand corner of the squared submatrix and k is its dimension.

You algorithm should run in $O(n \cdot m)$ time. For partial credit, your algorithm can be slower.

Proof: Let $l[i, j]$ be the length of the largest square whose upper left-hand corner is positioned at coordinates (i, j) in A. To simplify writing, suppose that $l[n + 1, j]$ and $l[j, m + 1]$ are 0. Then one can see that the following recurrence holds:

$$l[i, j] = \begin{cases} 0, \text{ if } A[i, j] = 0 \text{ or } i > n \text{ or } j > m \\ 1 + \max\{l[i + 1, j + 1], l[i, j + 1], l[i + 1, j]\}, \text{ if } A[i, j] = 1 \end{cases}$$

A dynamic programming algorithm can be easily devised now, either using recursion with memoization or using the iterative method. The following one, for example, uses the iterative method (suppose that $l[1..n + 1, 1..m + 1]$ initially contains only 0):

LARGEST-SQUARE(A)

$K \leftarrow 0$

for $i \leftarrow n$ **downto** 1 **do**

  for $j \leftarrow m$ **downto** 1 **do**

    if $A[i, j] = 1$ **then** $\{l[i, j] \leftarrow 1 + \min\{l[i + 1, j + 1] + l[i, j + 1] + l[i + 1, j]$

      if $K < l[i, j]$ **then** $\{K \leftarrow l[i, j];\ I \leftarrow i;\ J \leftarrow j\ \}\ \}$

output (I, J, K).

The running time of this algorithm is obviously $O(n \cdot m)$.

Exercise 4.2 **Approximate String Matching** See Subsection 3.1.3 in the text book.

Exercise 4.3 **Polygon Triangulation** See Subsection 3.1.5 in the text book.

Exercise 4.4 **Problem 3-2 in Skiena** Consider the problem of storing n books on shelves in a library. The order of the books is fixed by the cataloging system and so cannot be rearranged. Therefore, we can speak of a book b_i, where $1 \leq i \leq n$, that has a thickness t_i and height h_i. The length of each bookshelf at the library is L. The values of h_i are not necessarily assumed to be all the same. In this case, we have the freedom to adjust the height of each bookshelf to that of the tallest book on the shelf. Thus, the cost of a particular layout is the sum of the heights of the largest book on each shelf.

1. Give an example to show that the greedy algorithm of stuffing each shelf as full as possible does not always give the minimum overall height.

2. Give an algorithm for this problem, and analyze it's time complexity.

Proof:

1. Consider a set of three books, each of thickness 1, and with the first book having height 1, and the second and third books having height 2. Suppose that $L = 2$. Clearly, the b_1 must be placed on the first shelf. The greedy algorithm then places b_2 on the first shelf. This now fills the first shelf, and so we then must place b_3 on the second shelf.

   Since the largest book on each shelf is of height 2, the cost of this layout is 4. However, we can do better. Instead of this layout, we can place the first book on the first shelf, and the last two books on the second shelf. In this case, the first shelf has only one book of height 1, and the second shelf has books only of height 2, and so the cost of this layout is 3, which is better than what the greedy algorithm produced. Notice that this example is simple, which is why it is the best example. The simplest example is the best example!

   You should write a proof of the optimality of the greedy algorithm for the case where all the book heights are constant (problem 3-1), and determine what in that proof goes wrong in the case of non-equal heights.

2. We'll do this with dynamic programming, and break the problem up into sub-problems. Think about the way in which the books are placed on the shelf. We can iteratively place the books on the shelf, at each step, we can make a decision to either place the book on the current shelf, or to start a new shelf. Sometimes, of course, we will have no choice: the current shelf may not be able to fit the book, and we would then be forced to start a new shelf.

   We'll solve this problem first by going backwards. We'll loop from n down to 1 and determine the cheapest way of placing books i through n if we start a new shelf with book i. We store the best cost in $COST[i]$. Assume that L represents the length of the shelves, and that H is an array containing the heights of the books. We will let $COST[i]$ be the best possible cost of placing book i through n. For each i, m can be determined by satisfying the inequality, $\sum_{q=i}^{m} H[q] \leq L$.

   $$cost[i] = \min_{1 \leq k \leq m} \{cost[i+k] + \max\{H[i], ..., H[i+k-1]\}\}$$

   The base case is $cost[n] = H[n]$. The algorithm would look something like the following:

   Inputs: An array of heights H and an array of thicknesses T, both indexed from 1 to n, containing positive real numbers. The length of each shelf L is assumed to be a global constant. We further assume that $T[i] \leq L$ for each i, $1 \leq i \leq n$ (no fat books that won't fit on any shelf!).

   Outputs: An array COST, indexed from 1 to n. COST[i] indicates the total cost of shelving books i through n starting on a new shelf. Also, a list of numbers indicating which books should mark the first book on the shelf.

prodecure PLACEBOOKS (H, T)
var
  $COST$: array($1..n$) of real;
  $currentshelf$: real;
  $i, j, start$: integer;
begin
  COST[n] := H[n];
  for $i := n - 1$ **downto** 1 **loop**
    $currentshelf := T[i]$;
        Append $start$ to left end list.
    for $j := i + 1$ **to** n **loop**
      $currentshelf := currentshelf + T[j]$;
      if $currentshelf \leq L$ **and**
        $\max\{H[i], \ldots, H[j]\} + COST[j+1] \leq COST[i]$ **then**
          $COST[i] = \max\{H[i], \ldots, H[j]\} + COST[j+1]$
          $NextShelf[i] = j + 1$
      end if;
    end for loop;
  end for loop;
end PLACEBOOKS;

The outer loop goes through the different subproblems in reverse order. For each subproblem, the inner loop iterates through the books to be placed on the shelf.

The if statement on the inside checks two things: First, if there is room to place the next book on the shelf. If there isn't, then there is no point in even attempting to place the book on the shelf. If there is, then we need to check to see if it will be cheaper to place the current book on the shelf we're on, or to start a new shelf. If we want to place the book on the current shelf, then we add to the current size of the shelf we're working on and increment the loop counter. Again, if not, then we have to add to the cost of the current subproblem and start a new shelf.

Performance: The execution of the inside of the innermost for loop is bounded by a constant amount. The number of times the innermost loop is executed is bounded by a n, and the outer loop is executed $n - 1$ times, so the performance will be $O(n^2)$.

Reconstruction: To reconstruct the book layout we store the book that will start the next shelf in $NextShelf[i]$. Then we place book 1 through $NextShelf[1] - 1$ on the first shelf. Then books $NextShelf[1]$ through $NextShelf[NextShelf[1] - 1]$, etc.

Exercise 4.5 **Problem 3-8 in Skiena** Let us assume that we are given a string $x = x_1 x_2 \ldots x_{n-1} x_n$ of characters from an alphabet $\{a_1, \ldots, a_k\}$, with a multiplication $*$ defined on ordered pairs of the alphabet, with n as the length of a given string, and k as the number of elements in the given alphabet. We are interested in determining whether or not it is possible to parenthesize x in such a way that the value of the resulting expression is a, where a is some element in the alphabet.

Proof:

Needs serious revision. Take the proof as a hint only.

The algorithm presented below is an iterative algorithm using dynamic programming. We will break the problem down into sub-problems. Given a string, we want to examine all ways to break the string up into two contiguous parts, and for each partition, we can determine whether or not the substrings can be parenthesized to form a left and right terms so that the resulting multiplication gives us the desired product. This algorithm can be expressed using a recursive scheme:

Recursive solution: Function ISP (short for "is string parenthesizable")

Inputs: A string S of alphabet elements indexed from $left$ to $right$, and a target element $target$.
Output: $parenthesizable$, a boolean representing true iff the string is parenthesizable to give a result $target$.

```
function ISP (S, left, right, target)
var
   partition, alpha1, alpha2 : integer;
   parenthesizable : boolean;
begin
   [base case]
   if left = right then
      if S[left] = target then
         return true;
      else
         return false;
      end if;
   end if;
   [recursive case]
   parenthesizable := false;
   for partition := left to right - 1 loop
      for alpha1 := 1 to k loop
         for alpha2 := 1 to k loop
            if alpha1 * alpha2 = target then
               if ISP(partition, S[left..partition], target) and
                  ISP(partition, S[partition + 1..right], target) then
                  parenthesizable :=true;
               end if;
            end if;
         end for loop;
      end for loop;
   end for loop;
   return parenthesizable;
end ISP;
```

Of course, a much better way to do this, as mentioned, is a dynamic programming approach using iteration instead of recursion. Having considered sub-problems as being the problem of determining whether or not a given substring has a parenthesization giving a particular product, we may simply start with the given string, and consider ALL possible contiguous substrings, and determine whether or not they have parenthesizations giving any particular target value.

We will iterate by first checking all substrings of length 2, then by checking all substrings of length 3, and so on until we check the given string itself. For each substring, we will have to test to see if it has a parenthesization for any possible target value in the alphabet. As an added bonus, if the string is in fact parenthesizable to get the given product, this algorithm will also output such a parenthesization.

Iterative solution: Procedure ISP (short for "is string parenthesizable")
Inputs: A string S of alphabet elements indexed from 1 to n.
Outputs: $parenthesizable$, a triply indexed array of boolean values representing whether or not a particular substring of S is parenthesizable to give a particular result. More specifically, the value $parenthesizable[i][L][R]$ is true iff the substring $S[L..R]$ has a parenthesization giving the target value a_i, which is one of the alphabet elements. The other output is, $order$, another triply indexed array, this one of string values. $order[i][L][R]$ is defined only when $parenthesizable[i][L][R]$ has the value **true**, and in that case will be a string a characters representing a correct parenthesization of $S[L..R]$ to acheive the target value a_i. Notice that both $parenthesizable[i][L][R]$ and $order[i][L][R]$ make sense only if $L \leq R$.

```
 procedure ISP (S, n)
var
  parenthesizable : array {integer,integer,integer} of boolean := false;
  order : array {integer,integer,integer} of string;
  size, partition, i, j, left, right : integer;
begin
  [single length strings]
  for i := 1 to n loop
    for j := 1 to k loop
      if S[i] = a_i then
        parenthesizable[j][i][i] = true;
        order[j][i][i] = S[i];
      end if;
    end for loop;
  end for loop;
  [strings of length greater than 1]
  for size := 2 to n loop
    for left := 1 to n − size + 1 loop
      for partition := left to left + size − 2 loop
        for i := 1 to k loop
          for leftterm := 1 to k loop
            for rightterm := 1 to k loop
              if leftterm ∗ rightterm = i and
                    parenthesizable[leftterm][left][partition] and
                    parenthesizable[rightterm][partition + 1][left + size − 1] then
                parenthesizable[i][left][left + size − 1] := true;
                order[i][left][left + size − 1] := "(" concat
                    order[leftterm][left][partition] concat
                    order[rightterm][partition + 1][left + size − 1] concat
                    ")";
              end if;
            end for loop;
          end for loop;
        end for loop;
      end for loop;
    end for loop;
  end for loop;
end ISP;
```

The **concat** operation is simply string concatenation. This is used to produce a correct parenthesization of a string (if one exists).

Let's look at this algorithm more carefully. The outermost two loop structures loop through the different sub-problems, ordered first by size, and then by where the left end of the sub-problem starts. The third loop loops through the different ways the sub-problem can be broken up. Remember that given a string, it can be broken up into two parts a number of different ways.

The fourth nested loop will loop through the k different alphabet symbols, and for each one, we wish to check to see if a parenthesization exists for that resulting target value. Finally, the innermost two loops will loop through the different values that each of the two terms can take on.

<u>Performance:</u> The first three nested loops are executed no more than n times each, and then on the inside of that, the three innermost nested loops are executed k times each, and the main body of the six nested loops is executed

a constant amount of time during each pass. Therefore, the performance of this algorithm should be $O(k^3n^3)$. Of course, we should check to see that the number of times the outer three loops are executed isn't asymptotically any LESS than n^3.

The outermost two loops will give us $\frac{n(n+1)}{2} - n = \frac{n(n-1)}{2}$ loops. (Why?) The inner most loop gives us the number of ways that the sub-problem can be broken into two parts. For a sub-problem of size s, there will be $s-1$ different way to break up the problem into two parts. The number of sub-problems of size s is $n-s+1$. (Why?) Thus we can add up the total number of times that the outermost three nested loops will execute, and we get:

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

And this quantity will be a cubic polynomial. (Why?) Hence, in fact, the running time of this algorithm will be $O(k^3n^3)$. Can you do better than this?

Exercise 4.6 **Problem 16-3 from CLR (p. 325)** ... (ignore it)

Exercise 4.7 Give an algorithm for the following problem. Given a list of n distinct positive integers, partition the list into two sublists, each of size $n/2$, such that the difference between the sums of the integers in the two sublists is minimized. Formulate a dynamic programming definition for the problem. Develop an algorithm based on your definition. What is the time complexity of your algorithm? You can assume that n is a power of 2.

Proof:

It also needs to be revised ...

Assume that we may assign numbers to either of two sets, perhaps calling them set A and set B. We first place the given n numbers in any order (it doesn't matter what order), and call the numbers l_i, for $1 \leq i \leq n$. We will define a sub-problem as follows: Assume that m numbers have been assigned to the sets, with k numbers assigned to set A and hence $m-k$ numbers assigned to set B, with a difference of sums d. Define the difference d to be

$$d := \sum_{x \in A} x - \sum_{x \in B} x$$

Bear in mind that d could be positive or negative.

The problem is as follows: with m numbers assigned to the sets and k numbers assigned to set A and a difference of sums d, where should the $m+1$th number be placed? We will define $DIFFSUM[m,k,d]$ to be the minimal difference that can be achieved under that situation.

Notice that $DIFFSUM[m,k,d]$ cannot be properly defined for certain values. Since ultimately we want to end up with two sets each of size $n/2$, we cannot properly define $DIFFSUM[m,k,d]$ if either k or $m-k$ exceeds $n/2$. For the purposes of the dynamic programming formulation, in this case, we will define $DIFFSUM[m,k,d] = \infty$. Furthermore, $DIFFSUM[m,k,d]$ makes NO sense whatsoever for when k exceeds m. We simply won't define $DIFFSUM[m,k,d]$ in this situation.

Another issue we need to address is the possible range of the values of d. Really, this needs to be the possible ranges of differences of sums of sublists not exceeding size $n/2$. This in itself may be difficult to determine, so at the very least, we can say that in absolute value, d can never exceed the sum of the absolute values of all of the given original numbers. This may be a crude bound, but at least it gives us a bound. Let D be this bound.

So we need to define $DIFFSUM[m,k,d]$ for values where $0 \leq m \leq n$, $0 \leq k \leq m$, and $-D \leq d \leq D$. Here is the dynamic programming formulation:

Exceptional case (when k or $m-k$ exceeds $n/2$):

$$\text{if } k > n/2 \text{ or } m-k > n/2 \text{ then } DIFFSUM[m,k,d] = \infty$$

Base cases (when $m = n$)

$$DIFFSUM[n, \frac{n}{2}, d] = d$$

Recursive case (when $m < n$)

$$DIFFSUM[m, k, d] = \texttt{minabs}\{DIFFSUM[m + 1, k + 1, d + l_{m+1}], DIFFSUM[m + 1, k, d - l_{m+1}]\}$$

The operator `minabs` chooses the smallest number in absolute value.

Let's examine these more clearly. The exceptional case was already discussed. The base case is obvious. If all n elements have been distributed, then the best that we can do is what is already in place!

The recursive case is a bit more complex. When determining where the mth item should be placed, we need to consider two possibilities. One is where that mth item is placed in set A, and one is where it is placed in set B. We check the previously computed values of those optimal values, and choose the smallest.

We'll put together an algorithm for this which along with computing these optimal values, actually outputs where the numbers should be placed. The output will in fact be an optimal partition of the input numbers together with the optimal difference. We'll get a partition after we've computed all of the numbers in the dynamic programming, then loop through, check each value one by one, and determine if the number should be placed in the left set or the right set.

Inputs: A list of n numbers, along with the value of n. D, the sum of the absolute values of the input numbers (this is our crude bound on the possible difference between partitions of a subset of the given numbers).

Outputs: A partition of the input list (in the form of two sets) along with the difference between the two.

Let us assume that a data structure "set" is available for sets of integers.

prodecure DIFFSUM $(LIST, n, D)$
var
  $DIFFSUM : array[1..n][1..n][-D..D] of integer;$
  $DECISION : array[1..n][1..n][-D..D] of enum LEFT, RIGHT;$
  $m, k, d : integer;$
  $LEFT, RIGHT : integer := emptyset;$
begin
  Initialize all entries of $DIFFSUM$ to be ∞;
  Initialize $DIFFSUM[n][n/2][d]$ to be d for all values of d;
  for $m := n - 1$ **downto** 1 **loop**
    for $k := 1$ **to** n **loop**
      for $d := -D$ **to** D **loop**
        if $k \leq n/2$ **and** $m - k \leq n/2$ **then**
          if $k \leq n/2$ **and** $m - k \leq n/2$ **then**
            if $|DIFFSUM[m + 1, k + 1, d + l_{m+1}]| < |DIFFSUM[m + 1, k, d - l_{m+1}]|$ **then**
              (Place LEFT) $DIFFSUM[m][k][d] := DIFFSUM[m + 1][k + 1][d + l_{m+1}];$
              $DECISION[m][k][d] := LEFT;$
            else;
              (Place RIGHT)
              $DIFFSUM[m][k][d] := DIFFSUM[m + 1][k][d - l_{m+1}];$
              $DECISION[m][k][d] := RIGHT;$
            end if;
          end if;

```
            end if;
        end for loop;
    end for loop;
  end for loop;
  (Now create partition.)
  d := 0;
  k := 0;
  for m := 1 to n loop
    if DECISION[m][k][d] = LEFT then
      LEFT := LEFT ∪ {l_{m+1}}
      k := k + 1;
      d := d + l_{m+1};
    else if DECISION[m][k][d] = LEFT then
      RIGHT := RIGHT ∪ {l_{m+1}}
      d := d − l_{m+1};
    end if;
  end for loop;
  return LEFT, RIGHT, d;
end DIFFSUM;
```

Performance:
The performance of this algorithm is quite clearly $\Theta(n^2 D)$. However, don't be misled into thinking that this means the given algorithm is a polynomial time algorithm. Remember, we really want to measure the performance of an algorithm to be a function of the SIZE of the input (say bit size). While n clearly measures an aspect of size of the input, D does not. D is a function of the actual numbers themselves. To give a bound on the running time of the algorithm based on the size of the input (bit size of the given numbers), we would have to give an exponential expression for the running time. This is known as pseudopolynomial time.

Although we haven't reached this topic in the course yet, this algorithm is classified as NP-complete, which roughly means that there no known polynomial time algorithm for solving the problem, although a polynomial time algorithm exists for checking the CORRECTNESS of a solution, given one. Moreover, should a polynomial time algorithm be discovered for this problem, it's existence would imply the existence of a polynomial time algorithm for ALL NP-complete problems.

Realistic performance:
You should clearly be wary of such an algorithm. From the get-go, we could have written a very simple brute-force algorithm that checks all possible partitions of the given set. The number of different possible partitions of a set of n elements into two equal size subsets is $\frac{1}{2}\binom{n}{n/2}$. Using Stirling's formula for the approximate value of $n!$, we have that this expression is approximately equal to $\frac{2^n}{\sqrt{2\pi n}}$. This is a lot of things to check, but in reality, it may actually be smaller than $n^2 D$, in which case the brute force method is really the way to go!

Comments on the problem statement:
Note that the fact that n can be assumed to be a power of 2 was not used in this solution, or was the fact that the given integers are distinct.

Exercise 4.8 **Problem 3-9 in Skienab:** Consider the following data compression technique. We have a table of m text strings, each of length at most k. We want to encode a data string D of length n using as few text strings as possible. For example, if our table contains $(a, ba, abab, b)$ and the data string is $bababbaababa$, the best way to encode it is $(b, abab, ba, abab, a)$ - a total of five code words. Give and $O(nmk)$ algorithm to find the length of the best encoding. You may assume that the string has an encoding in terms of the table.

Proof: The subproblems involve the suffixes (could also do prefixes) of the data string D. When considering the suffix $D[i..n]$, the shortest encoding can be determined by finding which substrings starting at $D[i]$ match any of the encoding strings, and if that encoding string was used what the smallest encoding of the rest of the data would be. $L[i]$ will store the minimal number of encoding strings needed to encode the data string suffix $D[1..n]$. Assume that the array S holds the m encoding strings.

$$L[i] = \min_{1 \leq j \leq m} \begin{cases} 1 + L[i + Len(S[j])] \\ \text{ if } D[i..(i + Len(S[j]) - 1)] = S[j] \text{ and } (i + Len(S[j])) \leq n \\ \\ \infty, \text{otherwise} \end{cases}$$

The base case is $L[n + 1] = 0$. In order to output the encoding strings used, the index of the encoding string that results in a minimal $L[i]$ with be stored in $ES[i]$.

Inputs: array of encoding strings S, data string D, number of encoding strings m, largest size of encoding string k, length of data string n
Outputs: the length of the best encoding

FINDENCODINGLENGTH(S,D,m,k,n)

```
for i = 1 to n
    L[i] = ∞
    ES[i] = nil
// use results from previous suffixes to compute length
for i = n downto 1
    for str = 1 to m
        if D[i..(i + length(S[str]) - 1)] = S[str] then
            if (1 + L[i + Len(S[str])]) < L[i] then
                L[i] = 1 + L[i + length(S[str])]
                ES[i] = str
return L[1]
```

This algorithm's worst-case running time is $\Theta(nmk)$. There are n subproblems and to calculate each one it is necessary to compare m strings of length at most k.

To reproduce the list of encoding strings used we can use the following algorithm.

OUTPUTSTRINGS(S,m,ES)

```
for i = 1 to n
    if ES[i] ≠ nil then
        output S[ES[i]]
```

Exercise 4.9 **Problem 3-6 in Skiena** In the United States, coins are minted with denominations of 1,5,10,25, and 50 cents. Now consider a country whose coins are minted with denominations of $\{d_1, ..., d_k\}$ units. They seek an algorithm that will enable them to make change of n units using the minimum number of coins.

a) The greedy algorithm for making change repeatedly uses the biggest coin smaller than the amount to be changed until it is zero. Provide a greedy algorithm for making change of n units using US denominations. Prove its correctness and analyze its time complexity.

b) Show that the greedy algorithm does not always give the minimum number of coins in a country whose denominations are $\{1, 6, 10\}$.

c) Give an efficient algorithm that correctly determines the minimum number of coins needed to make change of n units using denominations $\{d_1, ..., d_k\}$. Analyze its running time.

Proof:
a)
Inputs: number of units to make change for n
Outputs: number of half dollars, quarter, dimes, nickels, and pennies to use $(c_{50}, c_{25}, c_{10}, c_5, c_1)$.

MAKECHANGE(n)

1. $c_{50} = n$ div 50
2. $leftover = n \bmod 50$
3. $c_{25} = n$ div 25
4. $leftover = n \bmod 25$
5. $c_{10} = n$ div 10
6. $leftover = n \bmod 10$
7. $c_5 = n$ div 5
8. $leftover = n \bmod 5$
9. $c_1 = n$ div 1
10. $leftover = n \bmod 1$
11. return $(c_{50}, c_{25}, c_{10}, c_5, c_1)$

Because the algorithm always performs 10 calculations, it's worst-case running time is $\Theta(1)$.

Proof of Optimality

Assume that the best non-greedy solution for a given instance of the problem is $(b_{50}, b_{25}, b_{10}, b_5, b_1)$, where $n = 50b_{50} + 25b_{25} + 10b_{10} + 5b_5 + b_1$. We show that the greedy solution is as good as or better than the best solution. The greedy solution is $(c_{50}, c_{25}, c_{10}, c_5, c_1)$. We want to show that $c_{50} + c_{25} + c_{10} + c_5 + c_1 \leq b_{50} + b_{25} + b_{10} + b_5 + b_1$.

Since the best solution is not greedy at some point there will be fewer coins of some denomination in the best solution vs. the greedy solution. We will show that any combination of coins with lower denominations which make up for the difference could be replaced with fewer coins. Therefore, the best solution must be equivalent to the greedy solution.

If $b_50 < c_{50}$ then $25b_{25} + 10b_{10} + 5b_5 + b_1 \geq 50$. To satisfy the given inequality b_25 could be ≥ 2. If that is the case then each 2 quarters could be replaced by one half-dollar thus using fewer coins. For this inequality all the cases are listed below.

1. if $b_{25} \geq 2$, replace with 1 half-dollar
2. if $b_{25} = 1$ we must also have either 2 dimes and 1 nickel, 1 dime and 3 nickels, etc., any of these combinations can be replaced with 1 half-dollar therefore using fewer coins
3. if $b_{25} = 0$ we must also have either 5 dimes, 4 dimes and 2 nickels, etc., any of these combinations can be replaced with 1 half-dollar

If $b_{50} = c_{50}$ and $b_{25} < c_{25}$ then $10b_{10} + 5b_5 + b_1 \geq 25$.

1. if $b_{10} \geq 3$, replace with 1 quarter and 1 nickel

2. if $b_{10} = 2$ we must also have either 1 nickels or 5 pennies, all of which can be replaced with 1 quarter

3. if $b_{10} = 1$ we must also have either 3 nickels, 2 nickels and 5 pennies, etc., any of these combinations can be replaced with 1 quarter

3. if $b_{10} = 0$ we must also have either 5 nickels, 5 nickels and 5 pennies, etc., any of these combinations can be replaced with 1 quarter

The entire proof would continue through the case if $b_{50} = c_{50}$, $b_{25} = c_{25}$, $b_{10} = c_{10}$, and $b_5 < c_5$.

b)

We can show that the greedy algorithm doesn't work for all possible denominations by giving a counter-example. If $n = 12$ and $(d_1, d_2, d_3) = (1, 6, 10)$, then the greedy algorithm would return $(c_{10}, c_6, c_1) = (1, 0, 2)$. However, the optimal solution is $(c_{10}, c_6, c_1) = (0, 2, 0)$.

c)

Given a list of k coin values, $(d_1, d_2, ..., d_k)$, and a number n, we want to find the integers $(c_{d_1}, c_{d_2}, ..., c_{d_k})$ such that $n = \sum_{i=1}^{k} d_i c_{d_i}$ and that $\sum_{i=1}^{k} c_{d_i}$ is minimal.

Our subproblems consist of the optimal change set for 1 through n. To keep track of the optimal solution for each subproblem we will use an array called $sumc$ which is indexed by subproblem. (i.e. $sumc[i]$ contains the least number of coins needed to make change for i). $coin[i]$ designates which coin denomination was last used when making change for i units.

$$sumc[d_1] = 1, sumc[d_2] = 1, ..., sumc[d_k] = 1$$

$$sumc[i] = \min_{1 \le j \le k} sumc[i - d_j] + 1$$

Inputs: denominations $(d_1, d_2, ..., d_k)$, units n
Outputs: the count of each denomination $(c_{d_1}, c_{d_2}, ..., c_{d_k})$.

MAKECHANGE(n,$(d_1, d_2, ..., d_k)$)

// initialization
for $i = 1$ to n
    $sumc[i] = \infty$
for $j = 1$ to k
    $sumc[d_j] = 1$; $coin[d_j] = j$
// calculate $sumw[i]$ for $1 \le i \le n$
for $i = 1$ to n
    for $j = 1$ to k
        $temp = sumc[i - d_j] + 1$
        if $temp < sumc[i]$
            $sumc[i] = temp$; $coin[i] = j$
// determine if it is possible to make change
if $sumc[n] = \infty$ return "impossible"

```
else // generate answer
   for j = 1 to sumc[n]
      c_{d_j} = 0 // initialization
   // traverse through coins used to make best change
   total = n
   while total > 0
      c_{d_{coin[total]}} = c_{d_{coin[total]}} + 1
      total = total - d_{coin[total]}
   return (c_{d_1}, c_{d_2}, ..., c_{d_k})
```

The running time of the above algorithm is $O(nk)$. Notice that n doesn't properly describe our input size. The input parameter n can be represented with $\lg n$ bits. Assume that all d_i denominations are represented with a constant number of bits. The true input size is $O(p+k)$, where $p = \lg n$. In terms of the true input size the running time of the algorithm is $O(2^p k)$.

Exercise 4.10 **Blackjack Hand Card Counting** You are given an array A of n positive integers (cards with face values) with values from 1 to k, and positive integers $l < n, v < kn$. Count the number of sets of l array positions (hands of l cards) whose total value is equal to v. Give the best algorithm you can for this problem. Analyze your algorithm in terms of n, k and l. Your algorithm should take time polynomial in all 3 parameters.

Proof: First, if $v > nl$, we can output 0 as the number of hands. Let $H[I, L, V]$ be the number of hands from $A[I], ..A[n]$ using L cards totalling V. (Define $H[I, 0, 0] = 1$, and otherwise $H[I, L, V] = 0$ unless both L, V are positive.) Then there are two ways of forming such hands: either we keep $A[I]$ and pick a hand of $L-1$ cards summing to $V - A[I]$ or we pick the hand from $A[I+1], ..A[n]$. This gives the recursion $H[I, L, V] = H[I+1, L-1, V-A[I]] + H[I+1, L, V]$ provided $I < n$. If $I = n$, $H[n, L, V] = 1$ if $L = 1$ and $A[I] = A[n] = V$ (or if $L = V = 0$) and 0 otherwise. The recursive definition always increases I, so we want to fill it in the array H from $I = n$ to $I = 1$, and we can fill in L and V in any order. So our algorithm would first fill in , for $0 \le L \le l$ and $0 \le V \le v$, the entry $H(n, L, V)$ according to the rule for $I = n$. It would then, for $I = n-1$ to 1, $0 \le L \le l$, $0 \le V \le v$, fill in $H(I, L, V)$ according to the recursive definition. The program then returns $H(1, l, v)$. Since the recursion is constant time, the time is proportional to the size of the array H, $O(nlv) = O(nl^2k)$ since $v \le lk$.

Exercise 4.11 **Greedy algorithm**

Despite its appearance, this is not a dynamic programming algorithm!

You are given a list of n jobs $j_1, ..j_n$, each with a time $t(j)$ to perform the job, and a weight $w(j)$. You wish to order the jobs as $j_{\sigma(1)}, ..., j_{\sigma(n)}$ in such a way as to minimize: $\sum_i w(j_{\sigma(i)})(\sum_{1 \le k < i} t(j_{\sigma(k)}))$; the weighted sum of the time each job has to wait before being performed. Give an efficient algorithm for this problem. (Hint: the algorithm is trivial, but you must prove that it is correct.)

Proof: The algorithm is to compute $r(j) = w(j)/t(j)$ for each job j, sort the $r(j)'s$ from largest to smallest, and schedule the jobs according to this order. This takes $O(n \log n)$ time.

To see that this algorithm produces an optimal schedule, first note that the optimum schedule for $j_1, ...j_n$ given that j_i is the first to be performed is that which schedules j_i and then schedules the remaining $j's$ optimally. This is because the cost of first sceduling j_i and then the others in some order is $\sum_{i' \neq i} w(i')t(i)$ + the cost of the other jobs if j_i weren't run, and the first term does not depend on the order the others were run in. Therefore, if for every set of jobs there is an optimal schedule that performs the job j_i with largest value of $r(i)$ first, then it follows inductively that the sorted order is an optimal schedule.

To see this last, let j_i be such that $r(i) \ge r(i')$ for every i', and let $j_{\sigma(1)}, ...j_{\sigma(n)}$ be any schedule, and let k be such that $\sigma(k) = i$; i.e., j_i is the k'th job performed in the schedule. Then we claim that $j_{\sigma(1)}, ..j_{\sigma(k-2)}, j_i, j_{\sigma(k-1)}, j_{\sigma(k+1)}, ..j_{\sigma(n)}$

is a schedule whose cost is at most that of the original schedule. (I.e., swapping j_i and the job before it only makes the cost smaller.) This is because the delays for all jobs except j_i and $j_{\sigma(k-1)}$ stay the same, and the delay for j_i has decreased by $t(\sigma(k-1))$ and that for $j_{\sigma(k-1)}$ has increased by $t(i)$. Thus the change in cost is $w(\sigma(k-1))t(i) - w(i)t(\sigma(k-1)) = t(i)t(\sigma(k-1))(r(\sigma(k-1)) - r(i)) \leq 0$ since $r(i) \geq r(\sigma(k-1))$. Repeating this process k times, we can move j_i to be the first job without increasing the cost. Thus, there is an optimal schedule whose first job is j_i. Thus, using the above inductively, there is an optimal schedule whose first job is j_i and whose second is the job with next highest r, and so on, so the sorted order is an optimal schedule.

Exercise 4.12 **Addition Chain** An *addition-chain* for n is a sequence of numbers starting with 1 and ending with n so that any element of the sequence other than 1 is the sum of two (not necessarily distinct) earlier elements. For example, two addition-chains for 13 are $1, 2 = 1 + 1, 4 = 2 + 2, 8 = 4 + 4, 9 = 8 + 1, 13 = 9 + 4$ and $1, 2 = 1 + 1, 3 = 1 + 2, 5 = 3 + 2, 8 = 5 + 3, 13 = 8 + 5$. The cost of an addition-chain is the total number of additions, or, equivalently, the number of elements minus one. So both addition chains above have cost 5. The challange is to find the smallest cost addition chains for $n = 392$, $n = 2371$, and for $n = 12,509$.