3 Divide and Conquer

The first major algorithmic technique we cover is **divide and conquer**. Part of the trick of making a good divide and conquer algorithm is determining how a given problem could be separated into two or more similar, but smaller, subproblems. More generally, when we are creating a divide and conquer algorithm we will take the following steps:

Divide and Conquer Methodology

1. Given a problem, identify a small number of significantly smaller subproblems of the same type
2. Solve each subproblem recursively (the smallest possible size of a subproblem is a base-case)
3. Combine these solutions into a solution for the main problem

The first algorithm we'll present using this methodology is the merge sort.

3.1 Merge Sort

The problem that **merge sort** solves is general sorting: given an unordered array of elements that have a total ordering, create an array that has the same elements sorted. More precisely, for an array a with indexes 1 through n, if the condition

for all i, j such that $1 \leq i < j \leq n$ then $a[i] \leq a[j]$

holds, then a is said to be **sorted**. Here is the interface:

```
// sort -- returns a sorted copy of array a
function sort(array a): array
```

Following the divide and conquer methodology, how can a be broken up into smaller subproblems? Because a is an array of n elements, we might want to start by breaking the array into two arrays of size $n/2$ elements. These smaller arrays will also be unsorted and it is meaningful to sort these smaller problems; thus we can consider these smaller arrays "similar". Ignoring the base case for a moment, this reduces the problem into a different one: Given two sorted arrays, how can they be combined to form a single sorted array that contains all the elements of both given arrays:

```
// merge -- given a and b (assumed to be sorted) returns a merged array that
// preserves order
function merge(array a, array b): array
```

So far, following the methodology has led us to this point, but what about the base case? The base case is the part of the algorithm concerned with what happens when the problem cannot be broken into smaller subproblems. Here, the base case is when the array only has one element. The following is a sorting algorithm that faithfully sorts arrays of only zero or one elements:

```
// base-sort -- given an array of one element (or empty), return a copy of the
// array sorted
function base-sort(array a[1..n]): array
  assert (n <= 1)
  return a.copy()
end
```

Putting this together, here is what the methodology has told us to write so far:

```
// sort -- returns a sorted copy of array a
function sort(array a[1..n]): array
  if n <= 1: return a.copy()
  else:
    let sub_size := n / 2
    let first_half := sort(a[1,..,sub_size])
    let second_half := sort(a[sub_size + 1,..,n])

    return merge(first_half, second_half)
  fi
end
```

And, other than the unimplemented merge subroutine, this sorting algorithm is done! Before we cover how this algorithm works, here is how merge can be written:

```
// merge -- given a and b (assumed to be sorted) returns a merged array that
// preserves order
function merge(array a[1..n], array b[1..m]): array
  let result := new array[n + m]
  let i, j := 1

  for k := 1 to n + m:
    if i >= n: result[k] := b[j]; j += 1
    else-if j >= m: result[k] := a[i]; i += 1
    else:
      if a[i] < b[j]:
        result[k] := a[i]; i += 1
      else:
        result[k] := b[j]; j += 1
      fi
    fi
  repeat
end
```

[TODO: how it works; including correctness proof] This algorithm uses the fact that, given two sorted arrays, the smallest element is always in one of two places. It's either at the head of the first array, or the head of the second.

3.1.1 Analysis

Let $T(n)$ be the number of steps the algorithm takes to run on input of size n.

Merging takes linear time and we recurse each time on two sub-problems of half the original size, so

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n).$$

By the master theorem, we see that this recurrence has a "steady state" tree. Thus, the runtime is:

$$T(n) = O(n \cdot \log n).$$

3.1.2 Iterative Version

This merge sort algorithm can be turned into an iterative algorithm by iteratively merging each subsequent pair, then each group of four, et cetera. Due to a lack of function overhead, iterative algorithms tend to be faster in practice. However, because the recursive version's call tree is logarithmically deep, it does not require much run-time stack space: Even sorting 4 gigs of items would only require 32 call entries on the stack, a very modest amount considering if even each call required 256 bytes on the stack, it would only require 8 kilobytes.

The iterative version of mergesort is a minor modification to the recursive version - in fact we can reuse the earlier merging function. The algorithm works by merging small, sorted subsections of the original array to create larger subsections of the array which are sorted. To accomplish this, we iterate through the array with successively larger "strides".

```
// sort -- returns a sorted copy of array a
function sort_iterative(array a[1..n]): array
   let result := a.copy()
   for power := 0 to log2(n-1)
     let unit := 2^power
     for i := 1 to n by unit*2
       let a1[1..unit] := result[i..i+unit-1]
       let a2[1..unit] := result[i+unit..min(i+unit*2-1, n)]
       result[i..i+unit*2-1] := merge(a1,a2)
     repeat
   repeat

   return result
end
```

This works because each sublist of length 1 in the array is, by definition, sorted. Each iteration through the array (using counting variable i) doubles the size of sorted sublists by

merging adjacent sublists into sorted larger versions. The current size of sorted sublists in the algorithm is represented by the *unit* variable.

3.2 Binary Search

Once an array is sorted, we can quickly locate items in the array by doing a binary search. Binary search is different from other divide and conquer algorithms in that it is mostly divide based (nothing needs to be conquered). The concept behind binary search will be useful for understanding the partition and quicksort algorithms, presented in the randomization chapter.

Finding an item in an already sorted array is similar to finding a name in a phonebook: you can start by flipping the book open toward the middle. If the name you're looking for is on that page, you stop. If you went too far, you can start the process again with the first half of the book. If the name you're searching for appears later than the page, you start from the second half of the book instead. You repeat this process, narrowing down your search space by half each time, until you find what you were looking for (or, alternatively, find where what you were looking for would have been if it were present).

The following algorithm states this procedure precisely:

```
// binary-search -- returns the index of value in the given array, or
// -1 if value cannot be found.  Assumes array is sorted in ascending order
function binary-search(value, array A[1..n]): integer
  return search-inner(value, A, 1, n + 1)
end

// search-inner -- search subparts of the array; end is one past the
// last element
function search-inner(value, array A, start, end): integer
  if start == end:
     return -1                  // not found
  fi

  let length := end - start
  if length == 1:
    if value == A[start]:
      return start
    else:
      return -1
    fi
  fi

  let mid := start + (length / 2)
  if value == A[mid]:
    return mid
  else-if value > A[mid]:
    return search-inner(value, A, mid + 1, end)
  else:
    return search-inner(value, A, start, mid)
  fi
end
```

Note that all recursive calls made are tail-calls, and thus the algorithm is iterative. We can explicitly remove the tail-calls if our programming language does not do that for us already by turning the argument values passed to the recursive call into assignments, and

then looping to the top of the function body again:

```
// binary-search -- returns the index of value in the given array, or
// -1 if value cannot be found.  Assumes array is sorted in ascending order
function binary-search(value, array A[1,..n]): integer
  let start := 1
  let end := n + 1

  loop:
    if start == end: return -1 fi                 // not found

    let length := end - start
    if length == 1:
      if value == A[start]: return start
      else: return -1 fi
    fi

    let mid := start + (length / 2)
    if value == A[mid]:
      return mid
    else-if value > A[mid]:
      start := mid + 1
    else:
      end := mid
    fi
  repeat
end
```

Even though we have an iterative algorithm, it's easier to reason about the recursive version. If the number of steps the algorithm takes is $T(n)$, then we have the following recurrence that defines $T(n)$:

$$T(n) = 1 \cdot T\left(\frac{n}{2}\right) + O(1).$$

The size of each recursive call made is on half of the input size (n), and there is a constant amount of time spent outside of the recursion (i.e., computing *length* and *mid* will take the same amount of time, regardless of how many elements are in the array). By the master theorem, this recurrence has values $a = 1, b = 2, k = 0$, which is a "steady state" tree, and thus we use the steady state case that tells us that

$$T(n) = \Theta(n^k \cdot \log n) = \Theta(\log n).$$

Thus, this algorithm takes logarithmic time. Typically, even when n is large, it is safe to let the stack grow by $\log n$ activation records through recursive calls.

difficulty in initially correct binary search implementations

The article on wikipedia on Binary Search also mentions the difficulty in writing a correct binary search algorithm: for instance, the java Arrays.binarySearch(..) overloaded function implementation does an interative binary search which didn't work when large integers overflowed a simple expression of mid calculation mid = (end + start) / 2 i.e. end + start > max_positive_integer . Hence the above algorithm is more correct in using a length =

end - start, and adding half length to start. The java binary Search algorithm gave a return value useful for finding the position of the nearest key greater than the search key, i.e. the position where the search key could be inserted.

i.e. it returns - *(keypos+1)* , if the search key wasn't found exactly, but an insertion point was needed for the search key (insertion_point = *-return_value - 1*). Looking at boundary values[1], an insertion point could be at the front of the list (ip = 0, return value = -1), to the position just after the last element, (ip = length(A), return value = - *length(A) - 1*) .

As an exercise, trying to implement this functionality on the above iterative binary search can be useful for further comprehension.

3.3 Integer Multiplication

If you want to perform arithmetic with small integers, you can simply use the built-in arithmetic hardware of your machine. However, if you wish to multiply integers larger than those that will fit into the standard "word" integer size of your computer, you will have to implement a multiplication algorithm in software or use a software implementation written by someone else. For example, RSA encryption needs to work with integers of very large size (that is, large relative to the 64-bit word size of many machines) and utilizes special multiplication algorithms.[2]

3.3.1 Grade School Multiplication

How do we represent a large, multi-word integer? We can have a binary representation by using an array (or an allocated block of memory) of words to represent the bits of the large integer. Suppose now that we have two integers, X and Y, and we want to multiply them together. For simplicity, let's assume that both X and Y have n bits each (if one is shorter than the other, we can always pad on zeros at the beginning). The most basic way to multiply the integers is to use the grade school multiplication algorithm. This is even

1 `http://en.wikibooks.org/wiki/boundary%20values`

2 A (mathematical) integer larger than the largest "int" directly supported by your computer's hardware is often called a "BigInt". Working with such large numbers is often called "multiple precision arithmetic". There are entire books on the various algorithms for dealing with such numbers, such as:

- Modern Computer Arithmetic `^{http://www.loria.fr/~zimmerma/mca/pub226.html}` , Richard Brent and Paul Zimmermann, Cambridge University Press, 2010.
- Donald E. Knuth, The Art of Computer Programming , Volume 2: Seminumerical Algorithms (3rd edition), 1997.

People who implement such algorithms may

- write a one-off implementation for one particular application
- write a library that you can use for many applications, such as GMP, the GNU Multiple Precision Arithmetic Library `^{http://gmplib.org/}` or McCutchen's Big Integer Library `^{https://mattmccutchen.net/bigint/}` or various libraries `http://www.leemon.com/crypto/BigInt.html` `https://github.com/jasondavies/jsbn` `https://github.com/libtom/libtomcrypt` `http://www.gnu.org/software/gnu-crypto/` `http://www.cryptopp.com/` used to demonstrate RSA encryption
- put those algorithms in the compiler of a programming language that you can use (such as Python and Lisp) that automatically switches from standard integers to BigInts when necessary

easier in binary, because we only multiply by 1 or 0:

```
          x6 x5 x4 x3 x2 x1 x0
        ×  y6 y5 y4 y3 y2 y1 y0
        -----------------------
          x6 x5 x4 x3 x2 x1 x0 (when y0 is 1; 0 otherwise)
        x6 x5 x4 x3 x2 x1 x0  0 (when y1 is 1; 0 otherwise)
     x6 x5 x4 x3 x2 x1 x0  0  0 (when y2 is 1; 0 otherwise)
  x6 x5 x4 x3 x2 x1 x0  0  0  0 (when y3 is 1; 0 otherwise)
    ... et cetera
```

As an algorithm, here's what multiplication would look like:

```
// multiply -- return the product of two binary integers, both of length n
function multiply(bitarray x[1,..n], bitarray y[1,..n]): bitarray
  bitarray p = 0
  for i:=1 to n:
    if y[i] == 1:
      p := add(p, x)
    fi
    x := pad(x, 0)          // add another zero to the end of x
  repeat
  return p
end
```

The subroutine **add** adds two binary integers and returns the result, and the subroutine **pad** adds an extra digit to the end of the number (padding on a zero is the same thing as shifting the number to the left; which is the same as multiplying it by two). Here, we loop n times, and in the worst-case, we make n calls to **add**. The numbers given to **add** will at most be of length $2n$. Further, we can expect that the **add** subroutine can be done in linear time. Thus, if n calls to a $O(n)$ subroutine are made, then the algorithm takes $O(n^2)$ time.

3.3.2 Divide and Conquer Multiplication

As you may have figured, this isn't the end of the story. We've presented the "obvious" algorithm for multiplication; so let's see if a divide and conquer strategy can give us something better. One route we might want to try is breaking the integers up into two parts. For example, the integer x could be divided into two parts, x_h and x_l, for the high-order and low-order halves of x. For example, if x has n bits, we have

$$x = x_h \cdot 2^{n/2} + x_l$$

We could do the same for y:

$$y = y_h \cdot 2^{n/2} + y_l$$

But from this division into smaller parts, it's not clear how we can multiply these parts such that we can combine the results for the solution to the main problem. First, let's write out $x \times y$ would be in such a system:

$$x \times y = x_h \times y_h \cdot (2^{n/2})^2 + (x_h \times y_l + x_l \times y_h) \cdot (2^{n/2}) + x_l \times y_l$$

This comes from simply multiplying the new hi/lo representations of x and y together. The multiplication of the smaller pieces are marked by the "$\times$" symbol. Note that the multiplies by $2^{n/2}$ and $(2^{n/2})^2 = 2^n$ does not require a real multiplication: we can just pad on the right number of zeros instead. This suggests the following divide and conquer algorithm:

```
// multiply -- return the product of two binary integers, both of length n
function multiply(bitarray x[1,..n], bitarray y[1,..n]): bitarray
  if n == 1: return x[1] * y[1] fi          // multiply single digits:  O(1)

  let xh := x[n/2 + 1, .., n]               // array slicing, O(n)
  let xl := x[0, .., n / 2]                 // array slicing, O(n)
  let yh := y[n/2 + 1, .., n]               // array slicing, O(n)
  let yl := y[0, .., n / 2]                 // array slicing, O(n)

  let a := multiply(xh, yh)                 // recursive call; T(n/2)
  let b := multiply(xh, yl)                 // recursive call; T(n/2)
  let c := multiply(xl, yh)                 // recursive call; T(n/2)
  let d := multiply(xl, yl)                 // recursive call; T(n/2)

  b := add(b, c)                            // regular addition; O(n)
  a := shift(a, n)                          // pad on zeros; O(n)
  b := shift(b, n/2)                        // pad on zeros; O(n)
  return add(a, b, d)                       // regular addition; O(n)
end
```

We can use the master theorem to analyze the running time of this algorithm. Assuming that the algorithm's running time is $T(n)$, the comments show how much time each step takes. Because there are four recursive calls, each with an input of size $n/2$, we have:

$$T(n) = 4T(n/2) + O(n)$$

Here, $a = 4, b = 2, k = 1$, and given that $4 > 2^1$ we are in the "bottom heavy" case and thus plugging in these values into the bottom heavy case of the master theorem gives us:

$$T(n) = O(n^{\log_2 4}) = O(n^2).$$

Thus, after all of that hard work, we're still no better off than the grade school algorithm! Luckily, numbers and polynomials are a data set we know additional information about. In fact, we can reduce the running time by doing some mathematical tricks.

First, let's replace the $2^{n/2}$ with a variable, z:

$$x \times y = x_h * y_h z^2 + (x_h * y_l + x_l * y_h) z + x_l * y_l$$

This appears to be a quadratic formula, and we know that you only need three co-efficients or points on a graph in order to uniquely describe a quadratic formula. However, in our above algorithm we've been using four multiplications total. Let's try recasting x and y as linear functions:

$$P_x(z) = x_h \cdot z + x_l$$

$$P_y(z) = y_h \cdot z + y_l$$

Now, for $x \times y$ we just need to compute $(P_x \cdot P_y)(2^{n/2})$. We'll evaluate $P_x(z)$ and $P_y(z)$ at three points. Three convenient points to evaluate the function will be at $(P_x \cdot P_y)(1), (P_x \cdot P_y)(0), (P_x \cdot P_y)(-1)$:

[TODO: show how to make the two-parts breaking more efficient; then mention that the best multiplication uses the FFT, but don't actually cover that topic (which is saved for the advanced book)]

3.4 Base Conversion

[TODO: Convert numbers from decimal to binary quickly using DnC.]

Along with the binary, the science of computers employs bases 8 and 16 for it's very easy to convert between the three while using bases 8 and 16 shortens considerably number representations.

To represent 8 first digits in the binary system we need 3 bits. Thus we have, 0=000, 1=001, 2=010, 3=011, 4=100, 5=101, 6=110, 7=111. Assume M=$(2065)_8$. In order to obtain its binary representation, replace each of the four digits with the corresponding triple of bits: 010 000 110 101. After removing the leading zeros, binary representation is immediate: M=$(10000110101)_2$. (For the hexadecimal system conversion is quite similar, except that now one should use 4-bit representation of numbers below 16.) This fact follows from the general conversion algorithm and the observation that 8=2^3 (and, of course, 16=2^4). Thus it appears that the shortest way to convert numbers into the binary system is to first convert them into either octal or hexadecimal representation. Now let see how to implement the general algorithm programmatically.

For the sake of reference, representation of a number in a system with base (radix) N may only consist of digits that are less than N.

More accurately, if

$$(1) M = a_k N^k + a_{k-1} N^{k-1} + ... + a_1 N^1 + a_0$$

with $0 <= a_i < N$ we have a representation of M in base N system and write

$$M = (a_k a_{k-1} ... a_0) N$$

If we rewrite (1) as

$$(2) M = a_0 + N * (a_1 + N * (a_2 + N * ...))$$

the algorithm for obtaining coefficients ai becomes more obvious. For example, $a_0 = M$ *modulo* n and $a_1 = (M/N)$ *modulo* n, and so on.

3.4.1 Recursive Implementation

Let's represent the algorithm mnemonically: (result is a string or character variable where I shall accumulate the digits of the result one at a time)

```
result = ""
if M < N, result = 'M' + result. Stop.
S = M mod N, result = 'S' + result
M = M/N
goto 2
```

A few words of explanation.

"" is an empty string. You may remember it's a zero element for string concatenation. Here we check whether the conversion procedure is over. It's over if M is less than N in which case M is a digit (with some qualification for N>10) and no additional action is necessary. Just prepend it in front of all other digits obtained previously. The '+' plus sign stands for the string concatenation. If we got this far, M is not less than N. First we extract its remainder of division by N, prepend this digit to the result as described previously, and reassign M to be M/N. This says that the whole process should be repeated starting with step 2. I would like to have a function say called Conversion that takes two arguments M and N and returns representation of the number M in base N. The function might look like this

```
1 String Conversion(int M, int N) // return string, accept two
integers
2 {
3     if (M < N) // see if it's time to return
4         return new String(""+M); // ""+M makes a string out of a
digit
5     else // the time is not yet ripe
6         return Conversion(M/N, N) +
           new String(""+(M mod N)); // continue
7 }
```

This is virtually a working Java function and it would look very much the same in C++ and require only a slight modification for C. As you see, at some point the function calls itself with a different first argument. One may say that the function is defined in terms of itself. Such functions are called recursive. (The best known recursive function is factorial: n!=n*(n-1)!.) The function calls (applies) itself to its arguments, and then (naturally) applies itself to its new arguments, and then ... and so on. We can be sure that the process will eventually stop because the sequence of arguments (the first ones) is decreasing. Thus sooner or later the first argument will be less than the second and the process will start emerging from the recursion, still a step at a time.

3.4.2 Iterative Implementation

Not all programming languages allow functions to call themselves recursively. Recursive functions may also be undesirable if process interruption might be expected for whatever reason. For example, in the Tower of Hanoi puzzle, the user may want to interrupt the demonstration being eager to test his or her understanding of the solution. There are complications due to the manner in which computers execute programs when one wishes to jump out of several levels of recursive calls.

Note however that the string produced by the conversion algorithm is obtained in the wrong order: all digits are computed first and then written into the string the last digit first. Recursive implementation easily got around this difficulty. With each invocation of the Conversion function, computer creates a new environment in which passed values of M, N, and the newly computed S are stored. Completing the function call, i.e. returning from the function we find the environment as it was before the call. Recursive functions store a sequence of computations implicitly. Eliminating recursive calls implies that we must manage to store the computed digits explicitly and then retrieve them in the reversed order.

In Computer Science such a mechanism is known as LIFO - Last In First Out. It's best implemented with a stack data structure. Stack admits only two operations: push and pop. Intuitively stack can be visualized as indeed a stack of objects. Objects are stacked on top of each other so that to retrieve an object one has to remove all the objects above the needed one. Obviously the only object available for immediate removal is the top one, i.e. the one that got on the stack last.

Then iterative implementation of the Conversion function might look as the following.

```
 1 String Conversion(int M, int N) // return string, accept two
integers
 2 {
 3     Stack stack = new Stack(); // create a stack
 4     while (M >= N) // now the repetitive loop is clearly seen
 5     {
 6         stack.push(M mod N); // store a digit
 7         M = M/N; // find new M
 8     }
 9     // now it's time to collect the digits together
10     String str = new String(""+M); // create a string with a
single digit M
11     while (stack.NotEmpty())
12         str = str+stack.pop() // get from the stack next digit
13     return str;
14 }
```

The function is by far longer than its recursive counterpart; but, as I said, sometimes it's the one you want to use, and sometimes it's the only one you may actually use.

3.5 Closest Pair of Points

For a set of points on a two-dimensional plane, if you want to find the closest two points, you could compare all of them to each other, at $O(n^2)$ time, or use a divide and conquer algorithm.

[TODO: explain the algorithm, and show the n^2 algorithm]

[TODO: write the algorithm, include intuition, proof of correctness, and runtime analysis]

Use this link for the original document.

`http://www.cs.mcgill.ca/~cs251/ClosestPair/ClosestPairDQ.html`

3.6 Closest Pair: A Divide-and-Conquer Approach

3.6.1 Introduction

The brute force approach to the closest pair problem (i.e. checking every possible pair of points) takes quadratic time. We would now like to introduce a faster divide-and-conquer algorithm for solving the closest pair problem. Given a set of points in the plane S, our approach will be to split the set into two roughly equal halves (S1 and S2) for which we already have the solutions, and then to merge the halves in linear time to yield an O(nlogn) algorithm. However, the actual solution is far from obvious. It is possible that the desired pair might have one point in S1 and one in S2, does this not force us once again to check all possible pairs of points? The divide-and-conquer approach presented here generalizes directly from the one dimensional algorithm we presented in the previous section.

3.6.2 Closest Pair in the Plane

Alright, we'll generalize our 1-D algorithm as directly as possible (see figure 3.2). Given a set of points S in the plane, we partition it into two subsets S1 and S2 by a vertical line l such that the points in S1 are to the left of l and those in S2 are to the right of l.

We now recursively solve the problem on these two sets obtaining minimum distances of d1 (for S1), and d2 (for S2). We let d be the minimum of these.

Now, identical to the 1-D case, if the closes pair of the whole set consists of one point from each subset, then these two points must be within d of l. This area is represented as the two strips P1 and P2 on either side of l

Up to now, we are completely in step with the 1-D case. At this point, however, the extra dimension causes some problems. We wish to determine if some point in say P1 is less than d away from another point in P2. However, in the plane, we don't have the luxury that we had on the line when we observed that only one point in each set can be within d of the median. In fact, in two dimensions, all of the points could be in the strip! This is disastrous, because we would have to compare n2 pairs of points to merge the set, and hence our divide-and-conquer algorithm wouldn't save us anything in terms of efficiency. Thankfully, we can make another life saving observation at this point. For any particular point p in one strip, only points that meet the following constraints in the other strip need to be checked:

- those points within d of p in the direction of the other strip
- those within d of p in the positive and negative y directions

Simply because points outside of this bounding box cannot be less than d units from p (see figure 3.3). It just so happens that because every point in this box is at least d apart, there can be at most six points within it.

Now we don't need to check all n2 points. All we have to do is sort the points in the strip by their y-coordinates and scan the points in order, checking each point against a maximum of 6 of its neighbors. This means at most 6*n comparisons are required to check all candidate pairs. However, since we sorted the points in the strip by their y-coordinates the process of merging our two subsets is not linear, but in fact takes O(nlogn) time. Hence our full algorithm is not yet O(nlogn), but it is still an improvement on the quadratic performance of the brute force approach (as we shall see in the next section). In section 3.4, we will demonstrate how to make this algorithm even more efficient by strengthening our recursive sub-solution.

3.6.3 Summary and Analysis of the 2-D Algorithm

We present here a step by step summary of the algorithm presented in the previous section, followed by a performance analysis. The algorithm is simply written in list form because I find pseudo-code to be burdensome and unnecessary when trying to understand an algorithm. Note that we pre-sort the points according to their x coordinates, and maintain another structure which holds the points sorted by their y values(for step 4), which in itself takes O(nlogn) time.

ClosestPair of a set of points:

1. Divide the set into two equal sized parts by the line l, and recursively compute the minimal distance in each part.
2. Let d be the minimal of the two minimal distances.
3. Eliminate points that lie farther than d apart from l.
4. Consider the remaining points according to their y-coordinates, which we have pre-computed.
5. Scan the remaining points in the y order and compute the distances of each point to all of its neighbors that are distanced no more than d(that's why we need it sorted according to y). Note that there are no more than 5(there is no figure 3.3 , so this 5 or 6 doesnt make sense without that figure . Please include it .) such points(see previous section).
6. If any of these distances is less than d then update d.

Analysis:

- Let us note T(n) as the efficiency of out algorithm
- Step 1 takes 2T(n/2) (we apply our algorithm for both halves)
- Step 3 takes O(n) time
- Step 5 takes O(n) time (as we saw in the previous section)

so,

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

which, according the Master Theorem, result

$T(n)O(nlogn)$

Hence the merging of the sub-solutions is dominated by the sorting at step 4, and hence takes O(nlogn) time.

This must be repeated once for each level of recursion in the divide-and-conquer algorithm,

hence the whole of algorithm ClosestPair takes O(logn*nlogn) = O(nlog2n) time.

3.6.4 Improving the Algorithm

We can improve on this algorithm slightly by reducing the time it takes to achieve the y-coordinate sorting in Step 4. This is done by asking that the recursive solution computed in Step 1 returns the points in sorted order by their y coordinates. This will yield two sorted lists of points which need only be merged (a linear time operation) in Step 4 in order to yield a complete sorted list. Hence the revised algorithm involves making the following changes: Step 1: Divide the set into..., and recursively compute the distance in each part, returning the points in each set in sorted order by y-coordinate. Step 4: Merge the two sorted lists into one sorted list in O(n) time. Hence the merging process is now dominated by the linear time steps thereby yielding an O(nlogn) algorithm for finding the closest pair of a set of points in the plane.

3.7 Towers Of Hanoi Problem

[TODO: Write about the towers of hanoi algorithm and a program for it]

There are n distinct sized discs and three pegs such that discs are placed at the left peg in the order of their sizes. The smallest one is at the top while the largest one is at the bottom. This game is to move all the discs from the left peg

3.7.1 Rules

1) Only one disc can be moved in each step.

2) Only the disc at the top can be moved.

3) Any disc can only be placed on the top of a larger disc.

3.7.2 Solution

Intuitive Idea

In order to move the largest disc from the left peg to the middle peg, the smallest discs must be moved to the right peg first. After the largest one is moved. The smaller discs are then moved from the right peg to the middle peg.

Recurrence

Suppose n is the number of discs.

To move n discs from peg a to peg b,

1) If n>1 then move n-1 discs from peg a to peg c

2) Move n-th disc from peg a to peg b

3) If n>1 then move n-1 discs from peg c to peg a

Pseudocode

```
void hanoi(n,src,dst){
  if (n>1)
    hanoi(n-1,src,pegs-{src,dst});
  print "move n-th disc from src to dst";
  if (n>1)
    hanoi(n-1,pegs-{src,dst},dst);
}
```

Analysis

The analysis is trivial. $T(n) = 2T(n-1) + O(1) = O(2^n)$