

1. Consider the following divide and conquer algorithm that claims to return a hamiltonian path (sequence of vertices) in an undirected graph  $G$  or will return FALSE if none exists:

HAMDC( $G$ )

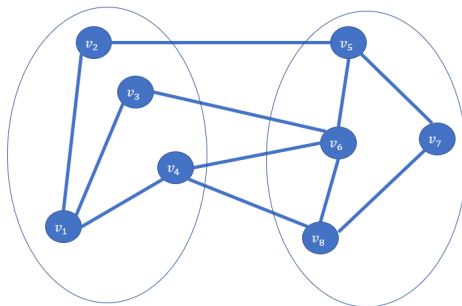
1. If  $G$  has one vertex  $v$  then return  $(v)$
2. Split the vertices of  $G$  into two halves (randomly)  $G_1, G_2$ .
3.  $S_1 = \text{HAMDC}(G_1)$
4.  $S_2 = \text{HAMDC}(G_2)$
5. If either  $S_1, S_2$  is FALSE then return FALSE.
6. Otherwise  $S_1 = (v_1, \dots, v_k), \quad S_2 = (u_1, \dots, u_\ell)$
7. If  $\{v_k, u_1\} \in E$ :
8.     return  $(v_1, \dots, v_k, u_1, \dots, u_\ell)$
9. If  $\{v_1, u_1\} \in E$ :
10.    return  $(v_k, v_{k-1}, \dots, v_1, u_1, \dots, u_\ell)$
11. If  $\{v_k, u_\ell\} \in E$ :
12.    return  $(v_1, \dots, v_k, u_\ell, u_{\ell-1}, \dots, u_1)$
13. If  $\{v_1, u_\ell\} \in E$ :
14.    return  $(v_k, v_{k-1}, \dots, v_1, u_\ell, u_{\ell-1}, \dots, u_1)$
15. Otherwise, return FALSE

(The high level description is: split the vertex set into two halves. Recursively find a Hamiltonian path in each half (if possible). Then if you found two Hamiltonian paths, test to see if you can connect them together with a single edge.)

- (a) (5 points) Provide a counterexample to show this algorithm is not correct. (please include justification.)

**Solution:**

Consider this graph and consider the question if there is a Hamiltonian path.



Then, there indeed is a Hamiltonian path starting at  $v_1$ , namely,  $(v_1, v_2, v_5, v_7, v_8, v_4, v_6, v_3)$ . But if you split the set of vertices as is done in the picture, notice that the vertices  $\{v_1, v_2, v_3, v_4\}$  do not have a Hamiltonian path and so the recursive call on this half of the vertices should return FALSE. This brings us to the next part which addresses the fact that you have to be very lucky for this algorithm to actually find a Hamiltonian Path.

- (b) (5 points) Suppose that  $G$  is a graph with  $n$  vertices that is just a single path consisting of all vertices. Let  $P(n)$  be the probability that this algorithm correctly outputs TRUE. Explain why  $P(n)$  satisfies the following recurrence (you can assume  $n$  is a power of 2):

$$P(n) = P(n/2)^2 * \left( \frac{2}{\binom{n}{n/2}} \right)$$

**Solution:** If the graph is just a single path, then there is only one Hamiltonian path (well, really two if you count the two different directions as different.) In order for the algorithm to detect the Hamiltonian path, it must divide the vertices perfectly so that each half is actually the path cut in half. There are  $\binom{n}{n/2}$  ways to divide the vertices in half. There are only 2 ways that split the path in half. Therefore, the probability that you split the path in half correctly during the first recursive call is  $\frac{2}{\binom{n}{n/2}}$ . Then you must correctly identify each half as a hamiltonian path with probability  $P(n/2)$  for each side. This gives the recursion:

$$P(n) = P(n/2)^2 * \left( \frac{2}{\binom{n}{n/2}} \right)$$

Just to see how bad this is, we start with  $P(1) = 1$  since if there is only one vertex, the algorithm will find the trivial path.

Notice that  $P(2) = P(1)^2 * \left( \frac{2}{\binom{2}{1}} \right) = 1^2 * 1 = 1$ . So, if your graph is just two vertices connected by a single edge, then it is certain that the algorithm will find the path. Then,  $P(4) = 1/3, P(8) = 1/315, P(16) = 1/638512875$ . So, you can see that the probabilities get really small.

- Let's say you have a stack of (rectangular) windows open on your computer and you have a setting that tethers each program window so that the lower left corner of each window is in the lower left corner of the screen. In order for your computer to know what part of the screen you are using, it needs to know the general outline of all the windows put together.

Notice that since all windows are tethered to the lower left corner, each window can be identified by the upper right corner.

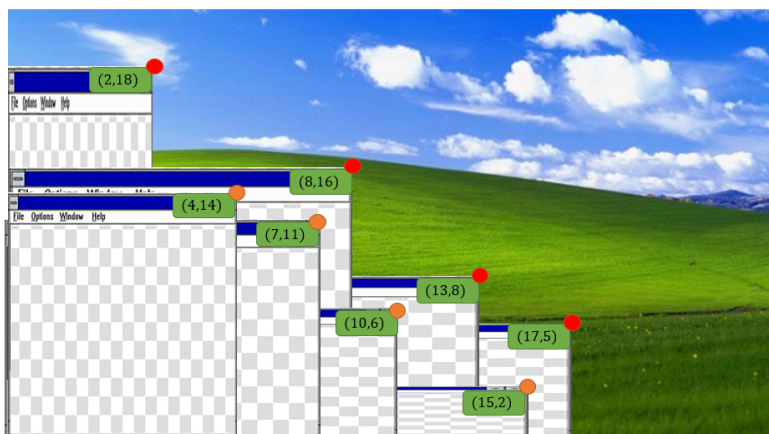
Given a list of  $n$  upper right corners of windows:  $[(x_1, y_1), \dots, (x_n, y_n)]$ , find the set of corners that identify the outline of the stack of windows.

Example: Suppose that you had windows with upper right coordinates at:

$$[(4, 14), (7, 11), (10, 6), (8, 16), (15, 2), (2, 18), (13, 8), (17, 5)]$$

Then the outline of the collection of windows is identified by the red points:

$$[(2, 18), (8, 16), (13, 8), (17, 5)]$$



Design a *divide and conquer* algorithm to solve this problem. Your algorithm should run in  $O(n \log n)$  time.

(10 points for a reasonably efficient correct algorithm high level and implementation level (no proof of correctness necessary), 5 points for correct time analysis.)

**Algorithm Description** Sort the points in increasing order by  $x$  coordinate. If there is only one window, then return the top right corner of that window. Otherwise split the list of points in half to get the left half  $L$  and the right half  $R$ . Recursively find the outline of  $L$ :  $OL$  and the outline of  $R$ :  $OR$ .

Notice that the points that define the outline of  $R$  will be part of the outline of the original set. The same cannot be said about the points that define the outline of  $OL$ . There could be a point in  $OR$  that is higher than a point in  $OL$ . So, only keep the points from  $OL$  that are higher than the highest point of  $OR$ .

```

Windows[( $x_1, y_1$ ), ..., ( $x_n, y_n$ )] (sorted by  $x$  coordinates.)
1. if  $n == 1$ :
2.     return ( $x_1, y_1$ )
3.  $m = \lfloor n/2 \rfloor$ 
4.  $L = [(x_1, y_1), \dots, (x_m, y_m)]$ 
5.  $R = [(x_{m+1}, y_{m+1}), \dots, (x_n, y_n)]$ 
6.  $OL = \text{Windows}(L)$ 
7.  $OR = \text{Windows}(R)$ 
8.  $\text{output} = [ ]$                                      #(initialize the output to be the empty list)
9. Set  $Y$  to be the maximum  $y$  value of  $OR$ 
10. for ( $x, y$ )  $\in OL$ :
11.     if  $y > Y$ :
12.          $\text{output} = \text{output} \circ (x, y)$            #(add ( $x, y$ ) as the last element of output)
13. for ( $x, y$ )  $\in OR$ :
14.      $\text{output} = \text{output} \circ (x, y)$ 
15. return output

```

**Runtime Analysis:** There are two recursive calls, each on a problem of half the size. Then, finding the maximum  $y$  coordinate will take  $O(n)$  time and adding in the coordinates to output will take  $O(n)$  time. All in all, the non-recursive part takes  $O(n)$  time. So if  $T(n)$  is the runtime of the algorithm on an input of length  $n$ , then:

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

By the master theorem with  $a = 2, b = 2, d = 1$ , we get a steady state recursion tree and  $T(n) = O(n \log n)$ .

Don't forget that we sorted the points before we ran the algorithm. Sorting takes  $O(n \log n)$ . So, all in all, the entire process takes  $O(n \log n)$ .

3. Given a sorted array of 0's, 1's and 2's, design an algorithm that returns the total sum of the array. (Your algorithm should run in  $O(\log n)$  time. You can use high and low pointers akin to Binary Search. Note that your algorithm returns an index.)

Example: if the input is (0, 0, 0, 0, 0, 1, 1, 1, 2, 2, 2, 2) then the answer should be 13.

(10 points for  $O(\log n)$  correct algorithm high-level and implementation level (no proof of correctness necessary), 5 points for correct time analysis.)

**Algorithm Description:** The idea behind the algorithm is to find the index  $z$  of the last 0 and index  $t$  of the first 2. (if there are no 0's then set  $z$  to be 0 and if there are no 2's, set  $t$  to be  $n + 1$ .) Then there are  $t - 1 - z$  ones and  $(n + 1 - t)$  twos. So we will want to return the value:  $(t - 1 - z) + 2 * (n + 1 - t)$ .

Now, how do we find those indices? We can use two calls to a Binary search type of algorithm to find the last zero and the first 2.

To find the last zero.

```

LastZero( $x_1, \dots, x_n$ ):
1. if  $x_1 \neq 0$ :
2.   return 0.
3.  $LOW = 1$ 
4.  $HIGH = n$ 
5. while  $HIGH - LOW > 0$ :
6.    $m = \lceil (LOW + HIGH)/2 \rceil$ 
7.   if  $x_m > 0$ :
8.      $HIGH = m - 1$ 
9.   else:
10.     $LOW = m$ 
11. return  $m$ 

```

To find the first two:

```

FirstTwo( $x_1, \dots, x_n$ ):
1. if  $x_n \neq 2$ :
2.   return  $n + 1$ .
3.  $LOW = 1$ 
4.  $HIGH = n$ 
5. while  $HIGH - LOW > 0$ :
6.    $m = \lfloor (LOW + HIGH)/2 \rfloor$ 
7.   if  $x_m > 0$ :
8.      $HIGH = m$ 
9.   else:
10.     $LOW = m + 1$ 
11. return  $m$ 

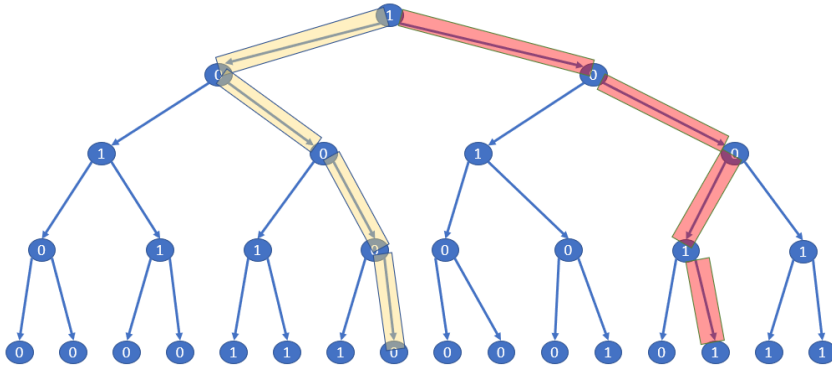
```

**Runtime:** Each of the above algorithms essentially split the input in half and recursively call one of the halves. The non-recursive part is constant time since it just requires a comparison. So if  $T(n)$  is the runtime of this algorithm,  $T(n) = T(n/2) + O(1)$ . By the master theorem with  $a = 1, b = 2, d = 0$ ,  $T(n) = O(\log n)$ .

So, there are two algorithms each running in  $O(\log n)$  time and then a constant time arithmetic operation. So all in all the entire process takes  $O(\log n)$  time.

4. Consider the following problem: You are given a pointer to the root  $r$  of a binary tree, where each vertex  $v$  has pointers  $v.lc$  and  $v.rc$  to the left and right child, and a value  $Val(v) \in \{0, 1\}$ . The value NIL represents a null pointer, showing that  $v$  has no child of that type. Design an algorithm that returns an ordered pair  $(a, b)$  where  $a$  is the maximum number of 0's in a path from root to leaf and  $b$  is the maximum number of 1's in a path from root to leaf.

For example: In this example, the output should be  $(4, 3)$  because the yellow highlighted path has the maximum number of 0's: 4 zeros and the red highlighted path has the maximum number of 1's: 3 ones.



(you can assume that the tree is a full balanced binary tree with  $n$  vertices where  $n = 2^k - 1$  for some  $k \geq 1$ .)

(10 points correct reasonably fast algorithm high-level and implementation level no justification necessary) (5 points for efficiency and time analysis.)

### Algorithm Description:

Run the recursive algorithm on the root vertex. First thing is to recursively call the algorithm on the left and right children of the root. Suppose that  $(a_L, b_L), (a_R, b_R)$  are the results of the recursive calls.

Then if  $Val(r) = 0$  then take the max of  $(a_L, a_R)$  and add 1 for the root to get the maximum number of zeros in a path from root to leaf. And then take the max of  $(b_L, b_R)$  to get the maximum number of ones in a path from root to leaf.

Similarly, if  $Val(r) = 1$  then take the max of  $(b_L, b_R)$  to get the maximum number of zeros in a path from root to leaf. And then take the max of  $(a_L, a_R)$  and add 1 for the root to get the maximum number of ones in a path from root to leaf.

**ZerosOnes( $r$ ):**

1. **if**  $r = NIL$ :
2.     **return**  $(0, 0)$ .
3.  $(a_L, b_L) = \text{ZerosOnes}(r.lc)$
4.  $(a_R, b_R) = \text{ZerosOnes}(r.rc)$
5. **if**  $Val(r) == 0$  :
6.     **return**  $(1 + \max(a_L, a_R), \max(b_L, b_R))$
7. **else**:
8.     **return**  $(\max(a_L, a_R), 1 + \max(b_L, b_R))$

### Runtime:

This algorithm splits the tree into roughly half and recursively calls on each half. The non-recursive part is constant time since it is a simple maximum and addition. Therefore if  $T(n)$  is the runtime of the algorithm then it satisfies the recurrence  $T(n) = 2T(n/2) + O(1)$ . Then by the master theorem with  $a = 2, b = 2, d = 0$ ,  $T(n) = O(n)$ .