Divide and Conquer
Divide and Conquer Algorithms

1. Partition task into sub-tasks which are smaller instances of the same task

2. Recursively solve the sub-tasks
   - Each sub-task is only solved outright when the bottom (threshold) of the recursion is reached

3. Appropriately combine the results

Why Divide and Conquer?
- Can be an easier way to approach solving large problems
- Can be faster - but not always
  - Critical point! Some examples coming
Depth log base depends on how split and potential overlap in splits

- $O(n)$ splits – $n-1$ for full binary tree where $n$ is # of elements in task
  - Do not always need to split all the way to one element per leaf, but often do

- $O(n)$ combines – may not need any (e.g. binary search)

- Actual problem solving work may be done at split time, at the tree leaves, at combine time, or any combination of these three

- Efficiency gains occur if the problem solving work can actually be simplified based on the split/merge paradigm
Grading $n$ exams

Non DC time required?

log $n$ depth
Grading $n$ exams

- Non DC time required? – $nG$ where $G$ is time to grade 1 exam: $O(n)$
- Divide and Conquer? – Feels more manageable, etc.
- Any overall speed-up on exam grading?
Grading $n$ exams

- Non DC time required? – $nG$ where $G$ is time to grade 1 exam: $O(n)$
- Divide and Conquer? – Feels more manageable, etc.
- Any overall speed-up on exam grading?
  - No. Although note potential parallelism
- Some overhead to split (dividing) and combine (re-stacking) the exams
  - $n$ splits each being $O(1)$ gives $O(n)$ but splits are very fast operations compared to $nG$
- Divide and conquer version still $O(n)$
Sorting $n$ Integers

- Non DC time required?
Non DC time required? – \( n^2 \) using some bubble sort variation: \( O(n^2) \)

Divide and Conquer?

Splitting is fast just like for grading exams

No work at leaves – Each just has an "ordered" list of length 1 to return
  - Now have \( n \) lists of length 1 at the leaves – Fast to do \( O(n) \) but was it worth it?
Combining requires a merge of two ordered lists which is $O(n)$ compared to the $O(n^2)$ required to sort one list directly. The key to the speedup!!

- Note that at the first combination the lists are of length one so it is a $O(1)$ merge, but there are $n/2$ of those merges at the level, which adds up to a total of $O(n)$ work at that level.
- The lists double in size each level up, but the number of merges halves, so the work stays $O(n)$

Divide and conquer version is $O(n)$ at each of the $\log(n)$ levels for total of $O(n\log(n))$
DC Multiply

- Assume $x$ and $y$ are $n$-bit numbers and $n$ is a power of 2
  - power of 2 is not essential, just makes explanation easier
- First split $x$ and $y$ into halves $n/2$ bits long: $x_L, x_R, y_L, y_R$
  
  $x \cdot y = (2^{n/2}x_L + x_R)(2^{n/2}y_L + y_R) =$
  
  $2^n x_Ly_L + 2^{n/2}(x_Ly_R + x_Ry_L) + x_Ry_R$

- The 4 multiplies dominate the complexity, shifts and adds are $O(n)$
- $T(n) = ?$
DC Multiply

- Assume $x$ and $y$ are $n$-bit numbers and $n$ is a power of 2
  - power of 2 is not essential, just makes explanation easier
- First split $x$ and $y$ into halves $n/2$ bits long: $x_L, x_R, y_L, y_R$

\[
x \cdot y = (2^{n/2}x_L + x_R)(2^{n/2}y_L + y_R) = 2^n x_L y_L + 2^{n/2}(x_Ly_R + x_Ry_L) + x_Ry_R
\]

- The 4 multiplies dominate the complexity, shifts and adds are $O(n)$
- $T(n) = 4T(n/2) + O(n)$
- Each multiply just a recursive call until leaves are reached at which level operations are $O(1)$
- Since branching factor is 4, the # of leaf nodes is

\[
4^{\text{depth}} = 4^{\log_2 n} = 4(\log_4 n)(\log_2 4) = n^{\log_2 4} = n^2
\]
  - Complexity at leaf level?
DC Multiply

- Assume $x$ and $y$ are $n$-bit numbers and $n$ is a power of 2
  - power of 2 is not essential, just makes explanation easier
- First split $x$ and $y$ into halves $n/2$ bits long: $x_L, x_R, y_L, y_R$
  \[
  x \cdot y = (2^{n/2}x_L + x_R)(2^{n/2}y_L + y_R) = 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R
  \]
- The 4 multiplies dominate the complexity, shifts and adds are $O(n)$
- $T(n) = 4T(n/2) + O(n)$
- Each multiply just a recursive call until leaves are reached at which level operations are $O(1)$
- Since branching factor is 4, the # of leaf nodes is
  \[
  4^{\text{depth}} = 4^{\log_2 n} = 4^{(\log_4 n)(\log_2 4)} = n^{\log_2 4} = n^2
  \]
  - Complexity at leaf level?
  - $n^2$ leaf nodes each with $O(1)$ complexity gives a leaf level complexity of $O(n^2)$
  - What is complexity at next level up? $n^2/2$ (adds and shifts).
Assume $x$ and $y$ are $n$-bit numbers and $n$ is a power of 2
- Power of 2 is not essential, just makes explanation easier

First split $x$ and $y$ into halves $n/2$ bits long: $x_L, x_R, y_L, y_R$

$$x \cdot y = (2^{n/2}x_L + x_R)(2^{n/2}y_L + y_R) = 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R$$

The 4 multiplies dominate the complexity, shifts and adds are $O(n)$

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

Each multiply just a recursive call until leaves are reached at which level operations are $O(1)$

Since branching factor is 4, the # of leaf nodes is

$$4^{depth} = 4^{\log_2 n} = 4^{(\log_4 n)(\log_2 4)} = n^{\log_2 4} = n^2$$

- Complexity at leaf level?
- $n^2$ leaf nodes each with $O(1)$ complexity gives a leaf level complexity of $O(n^2)$
- What is complexity at next level up? $n^2/2$ (adds and shifts). $O(n^2)$
- What is total complexity? What is top level Complexity?
Intuition: Geometric Series Review

- Why isn't complexity $n^2 \log n$? Important concept here!
- Remember geometric series (HW# 0.2) for $c > 0$
  - $f(n) = 1 + c + c^2 + \ldots + c^n = (c^{n+1} - 1)/(c-1)$
  - if $c < 1$ then each term gets smaller and $f = \Theta(1)$, the first term
    - Since, $1 < f(n) = (1-c^{n+1})/(1-c) < 1/(1-c)$
    - example: if $c=.5$, then $1 + 1/2 + 1/4 + \ldots + 1/2^n = ?$
Intuition: Geometric Series Review

- Why isn't complexity \( n^2 \log n \)? Important concept here!
- Remember geometric series (HW# 0.2) for \( c > 0 \)
  - \( f(n) = 1 + c + c^2 + \ldots + c^n = \frac{c^{n+1} - 1}{c-1} \)
    - if \( c < 1 \) then each term gets smaller and \( f = \Theta(1) \), the first term
      - Since, \( 1 < f(n) = \frac{1-c^{n+1}}{1-c} < \frac{1}{1-c} \)
      - example: if \( c=.5 \), then \( 1 + \frac{1}{2} + \frac{1}{4} + \ldots + \frac{1}{2^n} < 2 \)
    - if \( c > 1 \) then \( f = \Theta(c^n) \), which is the last term
      - Since, \( \frac{1}{(c-1)} \cdot c^n < c^n < \frac{c}{(c-1)} \cdot c^n \)
      - example: if \( c=2 \), then \( 1 + 2 + 4 + 8 + \ldots + 2^n < 2 \cdot 2^n \) and is \( \Theta(2^n) \)
  - if \( c = 1 \) then \( f = \Theta(n) \), why?
- For geometric series (\( c \) is the geometric ratio)
  - If decreasing (ratio \( < 1 \)) then complexity is \( \Theta(\text{first term}) \)
  - If increasing (ratio \( > 1 \)) then complexity is \( \Theta(\text{last term}) \)
  - If unchanging (ratio \( = 1 \)) then complexity is \( \Theta(n = \text{number of terms}) \)
- Divide and Conquer tree analogy with complexity at levels
Key Takeaway

- Assume $C(k)$ is the complexity (amount or work) at level $k$
- If starting from top level, $C(k)$ is decreasing then the total asymptotic work will be equal to that done at the top level, since all the other work done at lower levels is within a constant factor of the top level.
- If $C(k)$ increases with depth $k$ then the total asymptotic work will be equal to that done at the leaf (bottom) level, since all the other work done at higher levels is within a constant factor of the leaf level.
- If $C(k)$ is the same at each level, then the total work is $C(k)\log n$. 

CS 312 - Divide and Conquer/Recurrence Relations
The tree has a branching factor of 4 and a depth of $\log_2 n$

At any level $k$, there are $4^k$ subproblems each of size $n/2^k$. Why? (consider $k = 0, 1, 2 \ldots$)

For each subproblem in the tree

$$2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R$$

there is a linear amount of work done (splits going down and adds when combining)

Thus, the total time at each level is $4^k O(n/2^k) = (4/2)^k O(n) = 2^k O(n)$

- Geometric Series: $O(n)(1 + c + c^2 + \ldots) = O(n)(2^0 + 2^1 + 2^2 + \ldots)$

At level 0 time is $O(n)$ (just the adds upon return) and at the leaf level time is $2^{\log_2 n} O(n) = n O(n) = O(n^2)$

Thus time is a geometric series from $n$ to $n^2$ which increases by a factor of 2 at each level (ratio = 2, thus time increasing at each level) and the complexity is equal to that of the last term $O(n^2)$. 
Divide and Conquer Example

- Faster multiplication algorithm
- The product of 2 complex numbers is
  \[(a+bi)(c+di) = ac - bd + (bc+ad)i\]
  which requires 4 multiplications
- Carl Freidrich Gauss noted that this could be done with 3 multiplications (but a few more adds) because
  \[bc + ad = (a+b)(c+d) - ac - bd\]
- While this is just a constant time improvement for one multiplication, the savings becomes significant when applied recursively in a divide and conquer scheme
Faster Multiply

Now let's use Gauss's trick:

\[ xy = 2^n x_Ly_L + 2^{n/2}(x_Ly_R + x_Ry_L) + x_Ry_R \]

\[ x_Ly_R + x_Ry_L = (x_L + x_R)(y_L + y_R) - x_Ly_L - x_Ry_R \]

\[ xy = 2^n x_Ly_L + 2^{n/2}((x_L + x_R)(y_L + y_R) - x_Ly_L - x_Ry_R) + x_Ry_R \]

Now have 3 multiplies and \( T(n) = 3T(n/2) + O(n) \)

But this savings happens at every branch of the recursion tree, 4-ary vs a 3-ary tree – Key to speed-up in this case

# of leaf nodes is \( 3^{\log_2 n} = n^{\log_2 3} = n^{1.59} \)

Thus time in the tree is a geometric series from \( n \) to \( n^{1.59} \) increasing by \( 3/2 \) (ratio) at each level with the last term (leaf nodes) again dominating the complexity

Complexity is \( O(n^{1.59}) \)

Improved complexity class – Because we combined DC with Gauss trick to decrease tree arity

Can we do an even faster multiply? - Yes - Next chapter
General Divide and Conquer

function $DC(x)$: answer  // $x$ is a data item initially of size $n$
    if $||x|| < \text{threshold}$ then return $adhoc(x)$
    decompose $x$ into $a$ sub-tasks $x_1, x_2 \ldots x_a$ of size $n/b$
    for $i \leftarrow 1$ to $a$ do $y_i \leftarrow DC(x_i)$
    recombine the $y_i$’s to get a solution $y$ for $x$
    return $y$

Where:
$adhoc(x)$ = is the basic sub-algorithm for small instances
$a$ = the number of divisions at each level
$n/b$ = the fraction of the whole for a sub-instance
$\text{threshold}$ = size at which to stop dividing problem, when to stop?

What were these for the Gauss Multiply?
Master Theorem

Given: \[ t(n) = at(\lceil n/b \rceil) + O(n^d) \]

Where \( a > 0, \ b > 1, \ d \geq 0 \) and
\( a = \) number of sub-tasks that must be solved
\( n = \) original task size (variable)
\( n/b = \) size of sub-instances
\( d = \) polynomial order of work at each node (leaf/partitioning/recombining)

Then:
\[
t(n) = \begin{cases} 
O(n^d) & \text{if } \frac{a}{b^d} < 1 \iff d > \log_b a \\
O(n^d \log n) & \text{if } \frac{a}{b^d} = 1 \iff d = \log_b a \\
O(n^{\log_b a}) & \text{if } \frac{a}{b^d} > 1 \iff d < \log_b a 
\end{cases}
\]

This theorem gives big O complexity for most common DC algorithms
Proof/Intuition of the Master Theorem

- For simplicity assume $n$ is a power of $b$ (only a constant factor difference from any $n$)
- Height of the recursion tree is $\log_b n$
- Branching factor is $a$, thus there are $a^k$ subtasks at the $(k^{th})$ level of the tree, each task of size $n/b^k$
- Total work at $k^{th}$ level is thus $a^k \cdot O(n/b^k)^d$ (#tasks·(task size)$^d$)
  
  \[ = O(n^d) \cdot (a/b^d)^k \text{ Work/level} \cdot (\text{geometric ratio})^k \]
- Note that $(a/b^d)^k$ as $k$ goes from 0 (root) to leaf ($\log_b n$) is a geometric series with ratio $a/b^d$.
- The geometric ratio $a/b^d$ shows to what extent work is increasing/decreasing at each level. The big-O sum of such a series is
  - The first term of the series if the ratio is less than 1
  - $k$ (number of terms in the series) if the ratio is 1
  - The last term of the series if the ratio is greater than 1
Master Theorem Example/Intuition

- Assume \( n = 4, a = 4, b = 2 \)
- We'll consider \( d = 1, 2, \) and \( 3 \)
  - total combining/pre-partitioning work at each node is \( O(n), O(n^2), \) or \( O(n^3) \)
- Total work at \( k^{th} \) level is \( a^k \cdot O(n/b^k)^d = \#\text{tasks} \cdot (\text{task size})^d \)
  \[
  = O(n^d) \cdot (a/b^d)^k = \text{Work/level} \cdot (\text{geometric ratio})^k
  \]

<table>
<thead>
<tr>
<th>Level ( k )</th>
<th>Number of tasks ( a^k )</th>
<th>Task size ( n/b^k )</th>
<th>Total work at level ( k ) for ( d = 1 ) ( = a^k \cdot O(n/b^k)^d = #\text{tasks} \cdot (\text{task size})^d )</th>
<th>Total work at level ( k ) for ( d = 2 ) ( = a^k \cdot O(n/b^k)^d = #\text{tasks} \cdot (\text{task size})^d )</th>
<th>Total work at level ( k ) for ( d = 3 ) ( = a^k \cdot O(n/b^k)^d = #\text{tasks} \cdot (\text{task size})^d )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>4</td>
<td>( 4 = 1 \cdot 4^1 \cdot n^d )</td>
<td>( 16 = 1 \cdot 4^2 )</td>
<td>( 64 = 1 \cdot 4^3 )</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>2</td>
<td>( 8 = 4 \cdot 2^1 )</td>
<td>( 16 = 4 \cdot 2^2 )</td>
<td>( 32 = 4 \cdot 2^3 )</td>
</tr>
<tr>
<td>( 2 = \log_b n )</td>
<td>16</td>
<td>1</td>
<td>( 16 = 16 \cdot 1^1 )</td>
<td>( 16 = 16 \cdot 1^2 )</td>
<td>( 16 = 16 \cdot 1^3 )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>( O(n^{\log_b a}) = n^2 )</td>
<td>( O(n^d \log n) = n^2 \log n )</td>
<td>( O(n^d) = n^3 )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>( # \text{of leaf nodes} )</td>
<td>( \text{work/leaf} = n^d = 1^d )</td>
<td>( \text{Main complexity} )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>( \text{work/level} \cdot #\text{levels} )</td>
<td></td>
<td>( \text{at root node} )</td>
</tr>
</tbody>
</table>
Proof/Intuition of the Master Theorem

\[ t(n) = at([n/b]) + O(n^d) \]

\[ t(n) = \begin{cases} 
O(n^d) & \text{if } \frac{a}{b^d} < 1 \iff d > \log_b a \\
O(n^d \log n) & \text{if } \frac{a}{b^d} = 1 \iff d = \log_b a \\
O(n^{\log_b a}) & \text{if } \frac{a}{b^d} > 1 \iff d < \log_b a 
\end{cases} \]

- Total work at level \( k \) is \( a^k \cdot O(n/b^d)^k = O(n^d) \cdot (a/b^d)^k \)
  - If \( a/b^d < 1 \) (i.e. \( a < b^d \)) then complexity is dominated by root node: \( O(n^d) \)
    - \( a/b^d = .5: \ n^d + n^d/2 + n^d/4 \ldots n^d/2^{\log_b n} = n^d (1 + 1/2 + 1/4 + \ldots + 1/2^{\log_b n}) < 2n^d \)
  - If \( a/b^d = 1 \) then all \( \log_b n \) levels of tree take equal time \( O(n^d) \) giving a total complexity of \( O(n^d \log n) \)
  - If \( a/b^d > 1 \) then complexity is dominated by the leaf level: \( n^d \cdot (a/b^d)^{\log_b n} \)

\[ \frac{n^d (a/b^d)^{\log_b n}}{n^d} = n^d \left( \frac{a^{\log_b n} n}{\left( b^{\log_b n} \right)^d} \right) = n^d \left( \frac{a^{\log_b n}}{n^d} \right) = a^{\log_b n} = a^{(\log_a n)(\log_b a)} = n^{\log_b a} \]
Convex Hull

The convex hull of a set of $Q$ points is the smallest convex polygon $P$ for which each point is either on the boundary of $P$ or in its interior.
Convex Hull

Basic Algorithm

- $n$ points
- $n^2$ possible edges (possible parts of hull)
Convex Hull

- Basic Algorithm
  - $n$ points
  - $n^2$ possible edges (possible parts of hull)
  - Test for each edge
  - Total brute force time?
Convex Hull – Divide and Conquer

- Sort all points by $x$-coordinate – $n \log n$
- Divide and Conquer
  - Find the convex hull of the left half of points
  - Find the convex hull of the right half of points
  - Merge the two hulls into one
- All the work happens at the merge and that is where we have to be smart
Convex Hull

- If divide and conquer
  - How much work at merge
    - Can just pass back Hull and can drop internal nodes at each step, thus saving a large constant factor in time. (but not sufficient for project)
    - Hull can still have $O(n)$ points.
  - At merge can test all hull points to see which points/edges are part of the new merged hull
  - Complexity? Do relation and master theorem.
Master Theorem

Given: \[ t(n) = at(\lceil n/b \rceil) + O(n^d) \]

Where \( a > 0, \ b > 1, \ d \geq 0 \) and
- \( a \) = number of sub-tasks that must be solved
- \( n \) = original task size (variable)
- \( n/b \) = size of sub-instances
- \( d \) = polynomial order of work at each node (leaf/partitioning/recombining)

Then:

\[
\begin{align*}
  t(n) = \\
  \begin{cases}
    O(n^d) & \text{if } \frac{a}{b^d} < 1 \iff d > \log_b a \\
    O(n^d \log n) & \text{if } \frac{a}{b^d} = 1 \iff d = \log_b a \\
    O(n^{\log_b a}) & \text{if } \frac{a}{b^d} > 1 \iff d < \log_b a
  \end{cases}
\end{align*}
\]

This theorem gives big O complexity for most common DC algorithms.
Convex Hull

If divide and conquer
- How much work at merge
  - Can just pass back Hull and can drop internal nodes at each step, thus saving a large constant factor in time. (but not sufficient for project)
  - Hull can still have $O(n)$ points.
- At merge can test all hull points to see which points/edges are part of the new merged hull
- Complexity? Still $O(n^3)$. But can get a big constant factor improvement. How?
Convex Hull

If divide and conquer

- How much work at merge
  - Can just pass back Hull and can drop internal nodes at each step, thus saving a large constant factor in time. (but not sufficient for project)
  - Hull can still have $O(n)$ points.
- At merge can test all hull points to see which points/edges are part of the new merged hull
- Complexity? Still $O(n^3)$. But can get a big constant factor improvement. How?
Convex Hull

Can we merge faster?
- Note new hull will be a subset of the old hull edges plus 2 new edges
- An improved approach is to just consider current hull edges. This is $O(n)$ edges rather than testing all possible edges which is $O(n^2)$.
- All (and only) current edges still legal with opposite hull will remain
- Complexity of this step?
Convex Hull

Can we merge faster?

- Note new hull will be a subset of the old hull edges plus 2 new edges
- An improved approach is to just consider current hull edges. This is $O(n)$ edges rather than testing all possible edges which is $O(n^2)$.
- All (and only) current edges still legal with opposite hull will remain
- Complexity of this step? - $O(n^2)$
- Leaving 4 points which must be end-points of tangent lines to create the merged convex hull.
Can we merge faster?
- Then just appropriately connect the 4 points, adding the 2 needed edges for the merge – complexity of this part?
- Total complexity? Do relation and master theorem.
Convex Hull

- Can we do even a smarter merge to get even faster?
Convex Hull – Divide and Conquer

- **Hint:** Keep the hulls ordered clockwise or counter clockwise
  - Merging ordered hulls can be faster (like with merge sort)

- From one point (e.g. left-most) to each other point, clockwise order will be by decreasing slopes
Merging Hulls

- First find the edges which are upper and lower tangent
  - A common tangent of two simple convex polygons is a line segment in the exterior of both polygons intersecting each polygon at a single vertex

- Then remove hull points and edges that are cut off
  - Other internal points should already have been removed by this time
Finding Tangent Lines

- Start with the rightmost point of the left hull and the leftmost point of the right hull (maintain sorting)
- While the edge is not upper tangent to both left and right
  - While the edge is not upper tangent to the left, move counter clockwise to the next point on the left hull
    - Hint: Note that line is not upper tangent to the left if moving it up to the next point(s) on the left hull decreases slope of the tangent line
  - While the edge is not upper tangent to the right, move clockwise to the next point on the right hull
Finding Tangent Lines

- Start with the rightmost point of the left hull and the leftmost point of the right hull
- While the edge is not upper tangent to both left and right
  - While the edge is not upper tangent to the left, move counter clockwise to the next point on the left hull
    - Hint: Note that line is not upper tangent to the left if moving it up to the next point(s) on the left hull decreases slope of the tangent line
  - While the edge is not upper tangent to the right, move clockwise to the next point on the right hull
Finding Tangent Lines

- Start with the rightmost point of the left hull and the leftmost point of the right hull
- While the edge is not upper tangent to both left and right
  - While the edge is not upper tangent to the left, move counter clockwise to the next point on the left hull
    - Hint: Note that line is not upper tangent to the left if moving it up to the next point(s) on the left hull decreases slope of the tangent line
  - While the edge is not upper tangent to the right, move clockwise to the next point on the right hull
Finding Tangent Lines

- Start with the rightmost point of the left hull and the leftmost point of the right hull
- While the edge is not upper tangent to both left and right
  - While the edge is not upper tangent to the left, move counter clockwise to the next point on the left hull
    - Hint: Note that line is not upper tangent to the left if moving it up to the next point(s) on the left hull decreases slope of the tangent line
  - While the edge is not upper tangent to the right, move clockwise to the next point on the right hull
Finding Tangent Lines

- Start with the rightmost point of the left hull and the leftmost point of the right hull

- While the edge is not upper tangent to both left and right
  - While the edge is not upper tangent to the left, move counter clockwise to the next point on the left hull
    - Hint: Note that line is not upper tangent to the left if moving it up to the next point(s) on the left hull decreases slope of the tangent line
  - While the edge is not upper tangent to the right, move clockwise to the next point on the right hull
Finding Tangent Lines

- Start with the rightmost point of the left hull and the leftmost point of the right hull
- While the edge is not upper tangent to both left and right
  - While the edge is not upper tangent to the left, move counterclockwise to the next point on the left hull
    - Hint: Note that line is not upper tangent to the left if moving it up to the next point(s) on the left hull decreases slope of the tangent line
  - While the edge is not upper tangent to the right, move clockwise to the next point on the right hull – What is complexity of merge and total alg - master
Some Hints

- Maintain clockwise (or counter clockwise) ordering when merging (natural if start that way).
- Handle the base cases ($n < 4$) properly
  - Get started with appropriately ordered hulls
- Need to be careful when accessing your hull data structure since it is really a circular list. If using an array then make sure indexes properly change between the 0 element and the last element when you are moving either clockwise or counterclockwise through the array.
- Review Project Description
  - Theoretical vs Empirical graph and proportionality constant
  - Podium Demo