Divide and Conquer

- Divide the problem into parts.
- Solve each part.
- Combine the Solutions.
- Complexity is usually of form $T(n) = aT(n/b) + f(n)$.
- Chapter 5 of the book

Merge Sort

1. Divide the array into two parts.
2. Sort each of the parts recursively.
3. Merge the two sorted parts. Since the two parts are sorted, merging is easier than sorting the whole array.

Input: $A[i], \ldots, A[j]$, where $i \leq j$.
Output: Sorted $A[i], \ldots, A[j]$.

MergeSort(A, i, j)
(* Assumption: $i \leq j$. *)
1. If $i = j$, then return Endif
2. Let $k = \lfloor \frac{i+j-1}{2} \rfloor$.
3. MergeSort(A, i, k)
4. MergeSort$(A, k+1, j)$
5. Merge(A, i, k, j)

End

Merge(A, i, k, j)
(* Assumption: $i \leq k < j$; $A[i : k]$ and $A[k + 1 : j]$ are sorted. *)

1. $p1 = i$; $p2 = k + 1$; $p3 = i$
2. While $p1 \leq k$ and $p2 \leq j$ {
   2.1 If $A[p1] \leq A[p2]$, then $B[p3] = A[p1]$, $p1 = p1 + 1$
   Else $B[p3] = A[p2]$, $p2 = p2 + 1$; Endif
   2.2 $p3 = p3 + 1$;
   }
3. (* Copy the remaining elements into B *)
   While $p1 \leq k$ { $B[p3] = A[p1]$; $p1 = p1 + 1$; $p3 = p3 + 1$; }
   While $p2 \leq j$ { $B[p3] = A[p2]$; $p2 = p2 + 1$; $p3 = p3 + 1$; }
4. For $r = i$ to j { $A[r] = B[r]$ }

End

Complexity of Merge: $O(j - i + 1)$, that is the size of the two parts to be merged.

Complexity of MergeSort:
$T(n) \leq T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + cn$.
Gives $T(n) \in O(n \log n)$.
– For n being power of 2:
$T(n) \leq 2T(n/2) + cn$.
$T(n) \leq 4T(n/4) + 2(cn/2) + cn = 4T(n/4) + 2cn$.
$T(n) \leq 8T(n/8) + 4(cn/4) + 2cn = 8T(n/8) + 3cn$.
$\ldots$
$T(n) \in O(n \log n)$

For n not a power of two, it follows using monotonicity of the complexity formula.
That is, $T(n) \leq T(n') \leq T(2n)$, for n being power of 2, and $n \leq n' \leq 2n$.
Thus, $T(n') \leq c(2n \log(2n)) \leq c(4n' \log(4n')) \leq c'n' \log(n')$.

Can also be done using Master Theorem, for n being a power of 2:
$T(n) \leq 2T(n/2) + cn$.
Thus, by Master Theorem
$k = 1$, $b = 2$, $a = 2$, and $2 = 2^1$
Thus, $T(n) \in O(n^1 \log n)$

- Average Case also $\Theta(n \log n)$
- Stable.
- Uses extra space for merging: $O(n)$

Quicksort

- Choose a pivot element.
  $5, 2, 7, 5, 9, 4$
- partition the numbers into two parts, those $<$ pivot and those $\geq$ pivot.
  $4, 2, 5, 5, 9, 7$
- Sort each halves recursively.

Animation: Several webpages. May use a slightly different way to divide "less than" "greater than" parts.

Partition(A, i, j)
Intuition:
Aim is to return h such that:

- elements in $A[i : h-1]$ are smaller than pivot
- elements in $A[h+1 : j]$ are at least as large as the pivot
- $A[h]$ contains the pivot element.

P, SSSSSS, LLLLL, C, RRRRRR
(P=pivot, S=smaller, L=larger, C=current, R=remaining)

If C $\geq$ P, then leave it at the same place
If C $<$ P, then swap it with the first L element
At the end swap the pivot with the last S element
Need to be slightly careful in the beginning (when the S part or the L part may be empty).

Need to remember the first L element or the last S element.
(In the algorithm we will remember the last S element).

Partition(A, i, j)

1. Let $pivot = A[i]$.
2. Let $h = i$.
3. For $k = i + 1$ to j

   (* $A[i]$ is the pivot; $A[i + 1]$,..., $A[h]$ are smaller than pivot.
   $A[h + 1]$,..., $A[k - 1]$ are $\geq$ pivot. *)

   If $A[k] < pivot$, then

   $h = h + 1$

   $swap(A[k], A[h])$

   (that is: $t = A[k]$; $A[k] = A[h]$; $A[h] = t$;)

   Endif

   EndFor
4. $swap(A[h], A[i])$; Return h

End

At the beginning of the step 3 loop, the following invariant will be true:

$\ldots A[i], A[i+1], \ldots, A[h], A[h+1], \ldots, A[k-1], A[k], \ldots$

- $A[i+1:h]$ are smaller than $A[i]$.
- $A[h+1:k-1]$ are larger than $A[i]$.

Note that initially the invariants are satisfied and each iteration of the for loop maintains the invariant.

$5, 7, 2, 1, 9, 4$

(Next item: 7; $> pivot$, so leave it as it is.)

$5, 7, 2, 1, 9, 4$

(Next item: 2; smaller than pivot: swap with 1st larger element)

$5, 2, 7, 1, 9, 4$

(Next item: 1; smaller than pivot: swap with 1st larger element)

$5, 2, 1, 7, 9, 4$

(Next item: 9; $> pivot$, so leave it as it is.)

$5, 2, 1, 7, 9, 4$

(Next item: 4; smaller than pivot: swap with 1st larger element)

$5, 2, 1, 4, 9, 7$

End: Swap the pivot element with last smaller element.

$4, 2, 1, 5, 9, 7$

Quicksort(A, i, j)
    If $i < j$, then
        $p =$Partition(A, i, j)
        Quicksort$(A, i, p-1)$
        Quicksort$(A, p+1, j)$
End

Correctness:

1. Partition does its job correctly.
2. Assuming Partition does its job correctly, Quicksort does its job correctly.

Complexity:
Time taken is $O(n^2)$
$T(n) \leq T(n-p) + T(p-1) + cn$, for the worst possible p such that $1 \leq p \leq n$.
Guess: $T(r) \leq c_1 r^2$, and prove by induction

$$\begin{aligned}
T(n) &\leq \max_p [c_1(n-p)^2 + c_1(p-1)^2 + cn] \\
&\leq \max_p [c_1[n^2 + p^2 - 2np + p^2 + 1 - 2p] + cn] \\
&\leq \max_p [c_1[n^2 + 1 - 2p(n+1-p)] + cn] \\
&\leq c_1[n^2 - 2n + 1] + cn
\end{aligned}$$

(Here, note that $p(n+1-p)$ is smallest when $p=1$ or $p=n$)
Take c_1 such that, $c_1(-2n+1)+cn \leq 0$, then we are done.
That is, we want $cn \leq c_1(2n-1)$.
So can take $c_1 = c$.

Worst Case: Array is already sorted.
$(n-1)+(n-2)+(n-3)+\ldots+1=\Omega(n^2)$.

Best Case: Each round divides the two parts into nearly equal size.
Gives complexity $T(n)=O(n\log n)$.

Advantages: Usually quite quick. No extra array needed.

Randomized Quicksort:
By choosing a random element instead of the first element as the pivot.

RandomPartition(A, i, j)
    $k = rand(i, j)$.
    (* $rand(i, j)$ gives random r such that $i \leq r \leq j$ *)
    swap$(A[i], A[k])$
    Partition(A, i, j)
End
RandomQsort(A, i, j)
    If $i < j$,
        $p =$RandomPartition(A, i, j)
        RandomQsort$(A, i, p - 1)$
        RandomQsort$(A, p + 1, j)$
End

$$T(n) \leq \frac{1}{n} \sum_{p=1}^{n} [T(n-p) + T(p-1) + C_1 n]$$

$$T(n) \leq \frac{2}{n} \sum_{p=1}^{n-1} [T(p) + C_1 n]$$

$$nT(n) \leq C_2 n^2 + 2 \sum_{p=1}^{n-1} T(p)$$

Replacing $\leq$ by $=$ in the above line, and using H instead of T one gets,

$$nH(n) = C_2 n^2 + 2 \sum_{p=1}^{n-1} H(p)$$

where $T(n) \leq H(n)$.

$$nH(n) = C_2 n^2 + 2 \sum_{p=1}^{n-1} H(p)$$

Replacing n by $n-1$ in the above equation we get.

$$(n-1)H(n-1) = C_2 (n-1)^2 + 2 \sum_{p=1}^{n-2} H(p)$$

Taking the difference of the two equations, we get

$$nH(n) - (n-1)H(n-1) = C_2(2n-1) + 2H(n-1)$$

$$nH(n) = C_2(2n-1) + (n+1)H(n-1)$$

By dividing by $n(n+1)$, we get

$$\begin{aligned}
\frac{H(n)}{n+1} &\leq \frac{H(n-1)}{n} + C_3(\frac{1}{n}) \\
&\leq \frac{H(n-2)}{n-1} + C_3(\frac{1}{n-1} + \frac{1}{n}) \\
&\leq \frac{H(n-3)}{n-2} + C_3(\frac{1}{n-2} + \frac{1}{n-1} + \frac{1}{n}) \\
\cdots & \\
&\leq \frac{H(1)}{2} + C_3(\frac{1}{2} + \ldots + \frac{1}{n-1} + \frac{1}{n}) \\
&\leq C_4 * \log n
\end{aligned}$$

Therefore $T(n) \leq H(n) = O(n \log n)$.

Summary

- Runtime average: $\Theta(n(\log n))$
- Runtime worst case: n^2.
- Space Efficiency $O(n)$,
- Stability: no

Binary Search

- Input is Sorted.
- Want to find whether a key K is present in the input.
- Divide and conquer, but only 'one half' is relevant for continuation
- Can thus easily be implemented iteratively.

Binary Search

Binary Search
Input: A sorted array $A[0..n-1]$ and a key K

$\ell \leftarrow 0, r \leftarrow n-1$
While $\ell \leq r$ do
$m \leftarrow \lfloor \frac{\ell+r}{2} \rfloor$
If $K = A[m]$, then { "found" at m-th location}
Else If $K > A[m]$, then $\ell \leftarrow m+1$
Else $K < A[m]$, then $r \leftarrow m-1$ Endif
Endwhile
return "not found"
End

Complexity Analysis

- Each round, the size of the remaining part $r - \ell$ is halved to $\lfloor \frac{r-\ell+1}{2} \rfloor$ or from n to $\lfloor \frac{n}{2} \rfloor$.
- The recurrence for number of comparisons of array elements is: $C(n) = C(\frac{n}{2}) + 2$, for $n > 1$; $C(1) = 1$.
- $C(n) = 2\lfloor \log n \rfloor + 1$ (or $\log n + 1$ if we assume that a single comparison gives us whether K is $<$, $>$ or $=$ to the array element $A[m]$).

Tiling Problem

– Input: A n by n square board, with one of the 1 by 1 square missing, where $n = 2^k$ for some $k \geq 1$.
– Output: A tiling of the board using a tromino, a three square tile obtained by deleting the upper right 1 by 1 corner from a 2 by 2 square.
– You are allowed to rotate the tromino, for tiling the board.

Tiling Problem

Base Case: A 2 by 2 square can be tiled.

Induction:

– Divide the square into 4, $n/2$ by $n/2$ squares.

– Place the tromino at the “center”, where the tromino does not overlap the $n/2$ by $n/2$ square which was earlier missing out 1 by 1 square.

– Solve each of the four $n/2$ by $n/2$ boards inductively.

Complexity: $T(n) = 4T(n/2) + c$.
Gives, $T(n) = \Theta(n^2)$ using the master recurrence theorem.

Another way: There are $(n^2 - 1)/3$ tiles to be placed, and placing each tile takes $O(1)$ time!

Traversals on a Binary Tree

- Binary tree: Empty or (Root, left binary subtree, right binary subtree).
- Full Binary tree: Either no or two children for each node.
- Internal Node, leaf
- For Full binary tree, number of leaves ℓ and number of internal nodes i is related as follows:
  $2i = i + \ell - 1$, as every internal node as two children, and total number of children is $i + \ell - 1$.
  Thus, $i = \ell - 1$

Height of a binary tree/node

Height of single node is 0.

Height (T)
Input: A tree T
    If $T = \emptyset$, then return -1.
    Let T_L and T_R be the two children of the root of T
    return $max(Height(T_R), Height(T_L)) + 1$
End

Analysis

- Consider attaching “null children” to each of the nodes of the tree T
- The above algorithm visits every node, as well as the null children mentioned above
- n + $n + 1$ nodes/null children visited
- Total complexity is thus $C(n) = 2n + 1$
- Counting number of additions/Max: Once for each node. Total = n.

Preorder, Postorder, Inorder Traversals

Preorder(T)
(* Node is traversed before its children *)
    If $T \neq \emptyset$, then
    Visit T; $Preorder(T_L)$; $Preorder(T_R)$
End
Postorder(T)
(* Node is traversed after its children *)
    If $T \neq \emptyset$, then
    $Postorder(T_L)$; $Postorder(T_R)$; Visit T
End
Inorder(T)
(* Node is traversed in between its children *)
    If $T \neq \emptyset$, then
    $Inorder(T_L)$; Visit T; $Inorder(T_R)$
End

Multiplication of Large numbers

- Consider multiplying two n digit numbers x and y, where n is a power of 2
- $x = x_1 * 10^{n/2} + x_0$
- $y = y_1 * 10^{n/2} + y_0$
- $z = x * y = (x_1 * 10^{n/2} + x_0) * (y_1 * 10^{n/2} + y_0) = z_2 * 10^n + z_1 * 10^{n/2} + z_0$
- where $z_2 = x_1 * y_1$, $z_0 = x_0 * y_0$,
- $z_1 = x_1 * y_0 + x_0 * y_1 = (x_1 + x_0) * (y_1 + y_0) - (x_1 * y_1 + x_0 * y_0)$
- OR $z_1 = x_1 * y_0 + x_0 * y_1 = (x_1 + x_0) * (y_1 + y_0) - (z_2 + z_0)$.
- Number of multiplications reduced from 4 to 3, but additions went up!
- However, due to recurrence this helps overall.

Multiplication of Large numbers

- $M(n) = 3 * M(n/2)$, for $n > 1$. $M(1) = 1$,
- $A(n) = 3 * A(n/2) + cn$, for $n > 1$. $A(1) = 1$
- $M(n) = 3^{\log_2 n} \simeq n^{1.585}$
- $A(n) = \Theta(n^{\log_2 3})$ using the Master Theorem

Matrix Multiplication

$C = A \times B$, for $n \times n$ matrices.

$C(i,j) = \Sigma_{k=1}^{n} A(i,k) * B(k,j)$

$O(n^3)$ operations.

Strassen's algorithm

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$$

$$B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

$$AB = \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{pmatrix}$$

where $a_{11}b_{11}$ etc are smaller matrix multiplications. This however doesn't help in general.

$$q_1 = (a_{11} + a_{22}) * (b_{11} + b_{22})$$
$$q_2 = (a_{21} + a_{22}) * b_{11}$$
$$q_3 = a_{11} * (b_{12} - b_{22})$$
$$q_4 = a_{22} * (b_{21} - b_{11})$$
$$q_5 = (a_{11} + a_{12}) * b_{22}$$
$$q_6 = (a_{21} - a_{11}) * (b_{11} + b_{12})$$
$$q_7 = (a_{12} - a_{22}) * (b_{21} + b_{22})$$

$$AB = \begin{pmatrix} q_1 + q_4 - q_5 + q_7 & q_3 + q_5 \\ q_2 + q_4 & q_1 + q_3 - q_2 + q_6 \end{pmatrix}$$

- Complexity: $M(n) = 7M(n/2)$, $M(1) = 1$.
  So $M(n) = 7^{\log_2 n} = n^{\log_2 7}$.
- $A(n) = 7A(n/2) + O(n^2)$, $A(1) = 0$
  Gives, $A(n) = \Theta(n^{\log_2 7})$, using the Master Theorem.
- $\log_2 7 =$ approximately 2.807
- Best known algorithm: $O(n^{2.376})$ Copper-smith Winograd algorithm;
  $O(n^{2.374})$ algorithm by Andrew Stothers.
  $O(n^{2.3728642})$ algorithm by Virginia Williams
  $O(n^{2.3728639})$ algorithm by Francois Le Gall
- Best known lower bound: $\Omega(n^2)$: need to look at all the entries of the matrices.

Finding closest pair of points on a plane

Input: Given n points on a plane (via coordinates (a, b), where a, b are non-negative rational numbers.)
Output: A pair of points which are closest among all pairs.
Note: There could be several closest pairs. We only choose one such pair.

Closest Pair of Points

1. Find x_c, the median of the x-cordinates of the points.
2. Divide the points into two groups of (nearly) equal size based on them having x-coordinate $\leq x_c$ or $\geq x_c$ (note that several points may have same x-coordinate x_c).
3. Find closest pair among each of the two groups inductively.
4. Let the closest pair (among the two groups) have distance δ.

5. Consider all points which have x-coordinate between $x_c - \delta$ and $x_c + \delta$ and sort them according to y-coordinate
6. For each point find the distance between it and the next 7 points in the list as formed in step 5.
7. Report the shortest distance among all the distance found above (along with δ).

End

Correctness:
Note that the two points with shortest distance may be:
(a) In same group as in step 2
(b) In different groups

(a) Done in the individual group, inductively.
(b) Consider any pair of points (x, y) and (x', y') which were in different groups, but have distance $< \delta$.
Then $|x - x'| < \delta$ and $|y - y'| < \delta$.
(i) In particular, x, x' lie in the open interval $(x_c - \delta, x_c + \delta)$.

$(x_c - \delta,$ $y + \delta)$	$(x_c - \frac{\delta}{2},$ $y + \delta)$	$(x_c,$ $y + \delta)$	$(x_c + \frac{\delta}{2},$ $y + \delta)$	$(x_c + \delta,$ $y + \delta)$
$(x_c - \delta,$ $y + \frac{\delta}{2})$	$(x_c - \frac{\delta}{2},$ $y + \frac{\delta}{2})$	$(x_c,$ $y + \frac{\delta}{2})$	$(x_c + \frac{\delta}{2},$ $y + \frac{\delta}{2})$	$(x_c + \delta,$ $y + \frac{\delta}{2})$
$(x_c - \delta,$ $y)$	$(x_c - \frac{\delta}{2},$ $y)$	$(x_c,$ $y)$	$(x_c + \frac{\delta}{2},$ $y)$	$(x_c + \delta,$ $y)$

Without loss of generality, assume that in the sorting (as done in step 5) (x', y') appears after (x, y).
(ii) consider the eight squares given by the end points of the form:
$(x_c + k(\delta/2), y + k'(\delta/2))$, where $-2 \leq k \leq 2$ and $0 \leq k' \leq 2$.
Each of these squares can have at most one point (otherwise, they are both in same group (of step 2) and their distance is $< \delta$).
(Here, for the points with x-coordinate exactly x_c, we place it in the square to the left/right based on which group we placed them in step 2).
Thus, considering next seven points as done in step 6 is enough.

Complexity:
$T(n) \leq 2T(\lceil \frac{n}{2} \rceil) + cn \log n$.
Gives $T(n) = O(n(\log n)^2)$.

Smarter way: Don't need to sort every time, but only once (before the start of the algorithm). This will give
$T(n) \leq 2T(\lceil \frac{n}{2} \rceil) + cn$.