Divide & Conquer

- Idea:
  - Break problem into pieces
  - Solve the pieces (recursively)
  - Put the pieces back together
- Canonical example:
  - MergeSort
- Recursive algorithm that is relatively easy to design
- Running time that is relatively easy to analyze
  - Time is given by a *recurrence relation*
  - Recurrence relation is typically not too difficult to solve

Canonical Example: MergeSort

MergeSort (A,low,high):
```
/* A[low..high] contains items
  to be sorted. */
if (low < high) {
    mid = ⌊(low+high)/2⌋;
    MergeSort(A,low,mid);
    MergeSort(A,mid+1,high);
    Merge(A,low,mid,high);
}
```

Merge (A,i,j,k):
```
/* Merge A[i..j] with A[j+1..k]
  leaving the result back in A. */
B = A[i..j];
C = A[j+1,k];
while (B nonempty or C nonempty){
    Choose smallest of B and C;
    Move it into D;
}
Copy D back into A[i..k];
```


Analysis of MergeSort

- Time for Merge is O(n) where n is the number of elements being merged
- Time for MergeSort
  T(n) = 2T(n/2) + O(n) and T(1) = O(1)

  Recurrence can be simplified to T(n) = 2T(n/2) + n where T(1) = 1

  Solution is T(n) = O(n log n)
- One solution method for recurrence
  Can divide by n to get T(n)/n = T(n/2)/(n/2) + 1

  Define S(n) = T(n)/n

  S(n) = S(n/2) + 1 and S(0) = 1

  Easy to see that S(n) = 2 + log n

  Thus T(n) = n(2 + log n) or T(n) = O(n log n)

Solving Recurrence Relations

Solution techniques:
- Can sometimes change variables to make it into a simpler recurrence
- Make a guess then prove the guess correct by induction
- Build a recursion tree and use it to determine solution
- The *Master Method*
  - A "cookbook" scheme that handles many common recurrences

To solve T(n) = aT(n/b) + f(n) compare f(n) with $n^{\log_b a}$

Solution is T(n) = f(n) if f(n) grows more rapidly
Solution is T(n) = $n^{\log_b a}$ if $n^{\log_b a}$ grows more rapidly
Solution is T(n) = f(n) log n if both grow at same rate

Not an exact statement of the theorem (see CLR)

A Familiar Example: Binary Search

BinarySearch (query,A,low,high):
```
/* A[low..high] is assumed to be
    sorted */
if (low > high) return NotFound;
else {
    mid = ⌊(low+high)/2⌋;
    if (query < A[mid]) return
        BinarySearch(A,low,mid-1);
    else if (query > A[mid]) return
        BinarySearch(A,mid+1,high);
    else return mid;
}
```

- Recurrence is
  T(n) = T(n/2) + 1 and T(1) = 1
- Solution is
  T(n) = O(log n)

Example: Integer Multiplication

- Want to multiply two large numbers, each of size n
- Using traditional algorithm, $T(n) = O(n^2)$
- Idea: break each number into two pieces
  $A2^{(n/2)} + B$ and
  $C2^{(n/2)} + D$
- The result of multiplication is
  $AC2^n + (AD+BC)2^{(n/2)} + BD$
- Now we accomplish our multiply of size n by doing 4 multiplies of size (n/2)
- Recurrence
  T(n) = 4T(n/2) + n and T(1) = 1
- This can be solved to show $T(n) = O(n^2)$
  Wait a minute! This isn't any better than the method we learned in 3rd grade

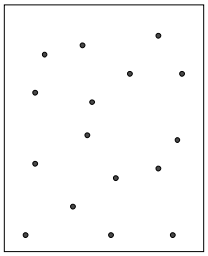
Improved Integer Multiplication

- The difficulty is that we have 4 multiplies
- Idea: Rewrite to use just 3 multiplies
- Compute
  AC, BD, and (A+B)(C+D) = AC + (AD+BC) + BD
- The coefficient of the middle term of
  $AC2^n + (AD+BC)2^{(n/2)} + BD$ is (A+B)(C+D)-AC-BD
- The resulting recurrence is
  $T(n) = 3T(n/2) + n$ where $T(1) = 1$
- Its solution is
  $T(n) = O(n^{\log 3}) \approx O(n^{1.585})$
- Wow! Why didn't my 3rd grade teacher show me this?
- Can actually do better than this using the Fast Fourier Transform

Example: Matrix Multiplication

- We can compute C=AB recursively by dividing each matrix into 4 submatrices
- The submatrices of A and B can be multiplied (using 8 multiplies of matrices of size n/2)
- Resulting recurrence is $T(n) = 8T(n/2) + O(n^2)$
- Solution is $T(n) = O(n^3)$
- Strassen found a way to use just 7 multiplies of matrices of size n/2
- Recurrence is $T(n) = 7T(n/2) + O(n^2)$
- Solution is $T(n) = O(n^{\log 7}) \approx O(n^{2.81})$
- The naive algorithm is better up to about n = 45
- Strassen's Algorithm becomes faster than naive algorithm at about n = 120
- Current best bound for matrix multiply is $O(n^{2.376})$

Example: Closest Pair of Points



- Given a set of points in the plane, determine the two points that are closest to each other
- An attempt at a simple solution
  - Project the points onto a line
  - Sort the points on along the line to find the smallest distance
  - Problem: Projection changes the distances

A Divide & Conquer Algorithm for Closest Pair of Points

ClosestPair (pointSet):

Split pointSet in half with a vertical line so that half are on left and half are on right;

Recursively determine closest pair in each half;

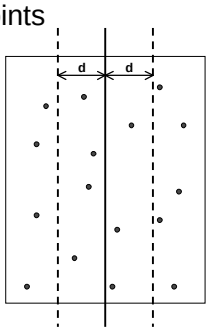
Let d be smallest of these two distances;

Search along the boundary between the two halves to see if there are any pairs closer than d;

- Analysis
  - The last step appears to require time $O(n^2)$
  - Recurrence for total time is $T(n) = 2T(n/2) + n^2$ and $T(1) = 1$
  - Solution: $T(n) = O(n^2)$
- The algorithm's last step is the problem
- How can we do the last step in linear time?

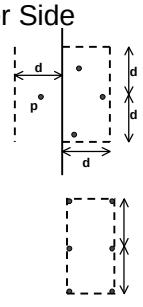
We Don't Have to Examine All the Points

- When we search along the boundary, we don't have to look at all the points in each half
  - We can ignore any point farther than d from the boundary line
  - Why?
- But each side can still have Θ(n) point within d of the boundary



There are Just a Few Points To Check on the Other Side

- For a point p on the left to be close to a point q on the right
  - q must be within distance d of p
  - q must fall within a rectangle of size d by 2d
- How many points (on the right) can fit into such a rectangle?
  - Any two points on the right are distance d or more apart
  - Thus there are at most 6 points in the rectangle



Closest Pair Algorithm (Expanded)

ClosestPair (pointSet):

O(n)	Split pointSet in half with a vertical line so that half are on left and half are on right;
2T(n/2)	Recursively determine closest pair in each half and let d be smallest of these two distances;
O(n)	Let L (on the left) and R (on the right) be the sets of points that are within distance d of the dividing line;
O(n log n)	Sort L and R by y-coordinate;
O(n)	For each point p of L, inspect the points of R with y-coordinate within distance d of p's y-coordinate to determine if there is a point within distance d of p; /* The L pointer always advances */ /* The R pointer may oscillate, but never by more than 6 */
O(1)	return the shortest distance found;

Analysis

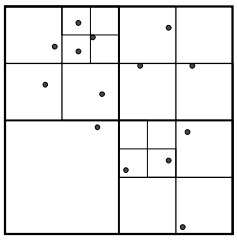
- Running time recurrence
  T(n) = 2T(n/2) + n log n and T(1) = 1
  This does *not* solve to T(n) = O(n log n)
- Final trick: Presorting
  - Sort the set of points by y-coordinate before we start
  - Whenever we split a point set, we can run through the list sorted by y-coordinate and create a new list for each part, sorted by y-coordinate
- Recurrence becomes
  T(n) = 2T(n/2) + n and T(1) = 1
- Solution: T(n) = O(n log n)
- It's possible to show that the closest pair can be found in O(n log n) time for any dimension

Example: Gravitational N-Body Problem

- Goal: Model a large number of "self-gravitating" particles
  - To simulate evolution of a galaxy, for instance
- Straightforward solution
  - Compute all n(n-1)/2 pairwise forces
  - For each particle, we sum the forces and compute its motion for one time-step
  - Recompute pairwise forces and repeat
- Straightforward solution is too slow for large n
- Idea: Use hierarchical grouping to speed up the calculation
  - Calculations for nearby particles are exact
  - Distant particles are grouped and act as if their total mass is concentrated at their center-of-mass

Use an Octree

- We start with a cube containing all the particles
- Subdivide this cube into 8 equal cubes each with side-length of half the original cube's side-length
- Continue this process until each cube contains at most one particle
- The 2D version is called a *quadtree*



To Find the Force on Particle p Due to Box B

force(p,B):
  if (B contains just one particle) return force due to that particle;
  L = side-length of B;
  c = B's center-of-mass;
  D = distance(p,c);
  if (L < D/2)
    Calculate force as if all particles in B are located at c;
  else
    force = Combine(force(p,B_1), force(p,B_2),...,force(p,B_8));
    /* B_1 to B_8 are the 8 subboxes of B */
  return force

Requirements

- Each box needs to know
  - The total mass of all the particles that it contains
  - The center-of-mass of all the particles it contains
  - This can be computed recursively (i.e., we can compute the values for box B if we know the values for each of B's subboxes)
- Building the octree must be efficient
  - The tree is rebuilt after every time step
  - It is possible for the tree to be badly balanced, requiring $O(n^2)$ time to build
  - In practice, tree has height of about O(log n)
  - Thus, tree building takes time O(n log n)

Results

- Time to compute the force on a particle p behaves like O(log n) instead of O(n)
- Experimental results: Number of 2-body interactions calculated for a 5000 particle system is about 1/10 the number needed for the straightforward algorithm
- Reference: [Barnes & Hut, Nature 1986]
- Note that this is still an expensive calculation
  - O(n log n) time needed for force calculations for *one* time step
  - Ideally, a galaxy simulation should use millions of stars