Heapsort

Construct a heap for a given list of *n* keys

Repeat operation of root removal *n*-1 times:

- Exchange keys in the root and in the last (rightmost) leaf
- Decrease heap size by 1
- If necessary, swap new root with larger child until the heap condition holds (Drift Down)

Sort the list 2, 9, 7, 6, 5, 8 by heapsort

Stage 1 (heap construction)

2 9 7 6 5 8 (7↔8)
2 9 8 6 5 7 (9 ok)
2 9 8 6 5 7 (2↔9)
9 2 8 6 5 7 (2↔6)
9 6 8 2 5 7 (heap)

Stage 2 (root/max removal)

9 6 8 2 5 7
7 6 8 2 5 | 9
8 6 7 2 5 | 9
5 6 7 2 | 8 9
7 6 5 2 | 8 9
2 6 5 | 7 8 9
6 2 5 | 7 8 9
5 2 | 6 7 8 9
5 2 | 6 7 8 9
2 | 5 6 7 8 9

Analysis of Heapsort

Stage 1: Build heap for a given list of n keys worst-case

$C(n) = \sum_{i=0}^{h-1} 2(h-i)2^i = 2(n - log_2(n+1)) \in \Theta(n)$

Stage 2: Repeat operation of root removal n-1 times (fix heap) worst-case

$C(n) = \sum_{i=0}^{n-1} 2(log_2(i)) \in \Theta(n\ log\ n)$

- Both worst-case and average-case efficiency: Θ(n*logn)

Performance Analysis of Build Heap

- How do binary heaps grow ???
  A binary heap of height k contains between 2^k and 2^{k+1} -1 keys

Two Partitioning Algorithms

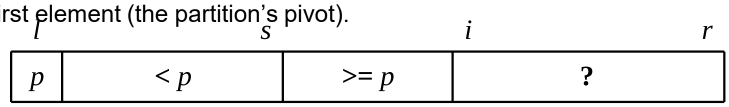
There are two principal ways to partition an array:

- One-directional scan (Lomuto's partitioning algorithm)

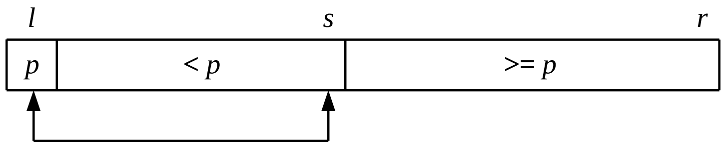
- Two-directional scan (Hoare's partitioning algorithm)

Lomuto's Partitioning Algorithm

- Scans the array left to right maintaining the array's partition into three contiguous sections: < p, ≥ p, and unknown, where p is the value of the first element (the partition's pivot).



- On each iteration the unknown section is decreased by one element until it's empty
- The partition is achieved by exchanging the pivot with the element in the split position *s*



Lomuto Partition algorithm

LomutoPartition(a[left..right]) // note that this works on subarrays

```
p = a[left]
s = left // elements in slots from left+1 to s are < p
For i = left+1 to right
  if a[i] < p
      s=s+1
      swap(a[s] and a[i])
swap(a[left] and a[s])
Return s   //  returns left ≤ s ≤ right
```

l	*s*	*i*	*r*
p	< *p*	>= *p*	**?**

Tracing Lomuto's Partioning Algorithm

s	*i*							
4	1	10	8	7	12	9	2	15
	s	*i*						
4	1	10	8	7	12	9	2	15
	s						*i*	
4	1	10	8	7	12	9	2	15
		s						*i*
4	1	2	8	7	12	9	10	15
		s						
4	1	2	8	7	12	9	10	15
2	1	**4**	8	7	12	9	10	15

Tracing Lomuto's Partioning Algorithm - 2

			s	*i*				
			8	7	12	9	10	15
				s	i			
			8	7	12	9	10	15
				s				*i*
			8	7	12	9	10	15
				s				*i*
			7	**8**	12	9	10	15

9

Partitioning Algorithm Idea (contd.)

- **`While (i < j)`**
  - Move **`j`** to the left till we find a number ≤ than **`pivot`**
  - Move **`i`** to the right till we find a number ≥ than **`pivot`**
  - **`If (i < j) swap(S[i], S[j])`**
- Swap the **`pivot`** with **`S[i]`**

Hoare's Partitioning Algorithm

Algorithm *Partition*($A[l..r]$)
//Partitions a subarray by using its first element as a pivot
//Input: A subarray $A[l..r]$ of $A[0..n-1]$, defined by its left and right
// indices l and r ($l < r$)
//Output: A partition of $A[l..r]$, with the split position returned as
// this function's value
$p \leftarrow A[l]$
$i \leftarrow l$; $j \leftarrow r + 1$
repeat
  repeat $i \leftarrow i + 1$ **until** $A[i] \geq p$
  repeat $j \leftarrow j - 1$ **until** $A[j] \cdot p \leq$
  swap($A[i]$, $A[j]$)
until $i \geq j$
swap($A[i]$, $A[j]$) //undo last swap when $i \geq j$
swap($A[l]$, $A[j]$)
return j

Correct version of 2 directional partitioning using median of three

```
pivot = median3( a, left, right )
    // puts smallest in left, pivot in right -1 and largest in right

int i = left, j = right - 1
while (i<j)        {
    repeat  i=i+1 until a[i] >= pivot
    repeat  j=j+1 until a[j] <= pivot
    if( i < j )        swap ( a[i], a[j] )

 swap (a[i], a[right - 1]);   // Restore pivot
```

Note: This is difficult to get correct - problem is when there are many identical keys

Quickselect (this is corrected text version)

Algorithm QuickSelect (A[left..right], k)
// Input: A[left..right] a subarray of A[0..n-1] and k an integer between 1 and
// right-left+1
// Output: The value of the k-th smallest element in A[left..right]

```
s ← LomutoPartition (A[left..right])  // note that s in left..right
if s- left +1 = k
    return A[s]                        // found it
else if s- left +1 > k
    return QuickSelect (A[left..s – 1], k)        //look in front
else
    return QuickSelect (A[s+1..right], k-(s-left+1))      //look in back,
reduce k
```

NOTE: There are easier/better ways to structure this.

Quickselect

Algorithm QuickSelect (A[left..right], k)
// Input: A[left..right] a subarray of A[0..n-1] and
// integer k such that <u>1 ≤ k ≤ right</u>
// Note: this code assumes first call to Quickselect had left = 0
// Output: The value of the k-th smallest element in A[left..right]

```
s ← LomutoPartition (A[left..right])  // note that s in left..right
if s = k-1
    return A[s]                        // found it
else if s > k-1
    QuickSelect (A[left..s – 1], k)        //look in front
else
    QuickSelect (A[s+1..r], k))        //look in back
```

Tracing Quickselect (Partition-based Algorithm)

Find the median of 4, 1, 10, 9, 7, 12, 8, 2, 15

Here: $n = 9$, $k = \lceil 9/2 \rceil = 5$, $k - 1 = 4$ ⇦

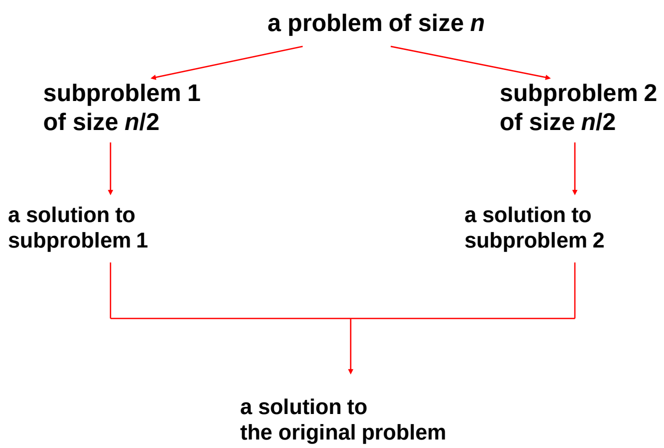
	0	1	2	3	**4**	5	6	7	8
	4	1	10	8	7	12	9	2	15
after 1st partitioning: s=2<k-1=4	2	1	**4**	8	7	12	9	10	15
				8	7	12	9	10	15
after 2nd partitioning: s=4=k-1				7	**8**	12	9	10	15

The median is A[4]= 8

Efficiency of Quickselect

- Average case (average split in the middle):
  $C(n) = C(n/2)+(n+1)$ $C(n) \in \Theta(n)$
- Worst case (degenerate split): $C(n) \in \Theta(n^2)$
- A more sophisticated choice of the pivot leads to a complicated algorithm with $\Theta(n)$ worst-case efficiency.
- Similar to Quicksort but Quicksort "divides" problem into two parts and then recombines the two parts – hence the author considers this "divide and conquer"

Divide-and-Conquer Technique



Divide-and-Conquer

- The most-well known algorithm design strategy:
- Divide instance of problem into two or more smaller instances
- Solve smaller instances recursively (can implement iteratively)
- Obtain solution to original (larger) instance by combining these solutions

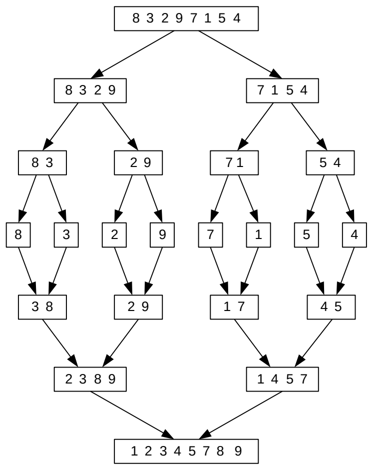
Divide-and-Conquer Examples

- Sorting: mergesort and quicksort
- Binary tree traversals
- Multiplication of large integers
- Matrix multiplication: Strassen's algorithm
- Polynomial multiplication
- Fast Fourier Transform
- Closest-pair algorithms
- Convex-hull algorithms

Mergesort

- Split array A[0..n-1] in two about equal halves and make copies of each half in arrays B and C
- Sort arrays B and C recursively
- Merge sorted arrays B and C into array A as follows:
  - Repeat the following until no elements remain in one of the arrays:
    - » compare the first elements in the remaining unprocessed portions of the arrays
    - » copy the smaller of the two into A, while incrementing the index indicating the unprocessed portion of that array
  - Once all elements in one of the arrays are processed, copy the remaining unprocessed elements from the other array into A.

Mergesort Example



CAL POLY
SAN LUIS OBISPO

Computer Science Department

Pseudocode of Mergesort

ALGORITHM $Mergesort(A[0..n-1])$

//Sorts array $A[0..n-1]$ by recursive mergesort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
if $n > 1$
  copy $A[0..\lfloor n/2\rfloor - 1]$ to $B[0..\lfloor n/2\rfloor - 1]$
  copy $A[\lfloor n/2\rfloor..n-1]$ to $C[0..\lceil n/2\rceil - 1]$
  $Mergesort(B[0..\lfloor n/2\rfloor - 1])$
  $Mergesort(C[0..\lceil n/2\rceil - 1])$
  $Merge(B, C, A)$

CAL POLY
SAN LUIS OBISPO

Computer Science Department

Pseudocode of Merge

ALGORITHM $Merge(B[0..p-1], C[0..q-1], A[0..p+q-1])$

```
//Merges two sorted arrays into one sorted array
//Input: Arrays B[0..p − 1] and C[0..q − 1] both sorted
//Output: Sorted array A[0..p + q − 1] of the elements of B and C
i ← 0; j ← 0; k ← 0
while i < p and j < q do
    if B[i] ≤ C[j]
        A[k] ← B[i]; i ← i + 1
    else A[k] ← C[j]; j ← j + 1
    k ← k + 1
if i = p
    copy C[j..q − 1] to A[k..p + q − 1]
else copy B[i..p − 1] to A[k..p + q − 1]
```

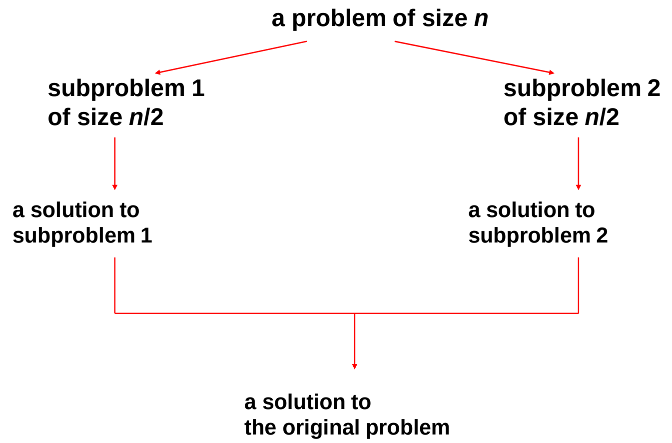

Analysis of Mergesort

- All cases have same efficiency: Θ(n log n)
- Number of comparisons in the worst case is close to theoretical minimum for comparison-based sorting:
  - ⌈log2 n!⌉ ≈ n log2 n - 1.44n
- Space requirement: Θ(n) (not in-place)
- Can be implemented without recursion (bottom-up)

Divide-and-Conquer

- The most-well known algorithm design strategy:
- Divide instance of problem into two or more smaller instances
- Solve smaller instances recursively (can implement iteratively)
- Obtain solution to original (larger) instance by combining these solutions

Divide-and-Conquer Technique

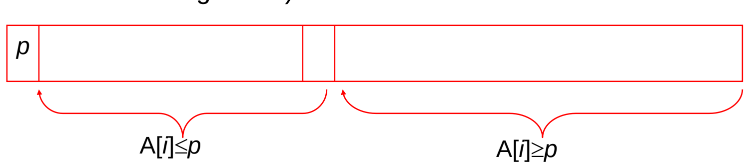


Divide-and-Conquer Examples

- Sorting: mergesort and quicksort
- Binary tree traversals
- Multiplication of large integers
- Matrix multiplication: Strassen's algorithm
- Polynomial multiplication
- Fast Fourier Transform
- Closest-pair algorithms
- Convex-hull algorithms

Quicksort

- Select a pivot (partitioning element) – here, the first element
- Rearrange the list so that all the elements in the first s positions are smaller than or equal to the pivot and all the elements in the remaining n-s positions are larger than or equal to the pivot (see next slide for an algorithm)



- Exchange the pivot with the last element in the first (i.e., ≤) subarray — the pivot is now in its final position
- Sort the two subarrays recursively

Quicksort Example

- 5 3 1 9 8 2 4 7

QuickSort

```
public int quickSort(int a[], int left, int right)   {
          // note that this works on subarray
          // defined by left and right

        if ( l < r)
                    s = Partition(a, left, right);
                    quickSort(a, left, s-1);
                    quickSort(a, s+1, right);

  }
```

Quicksort with 2 directional partitioning

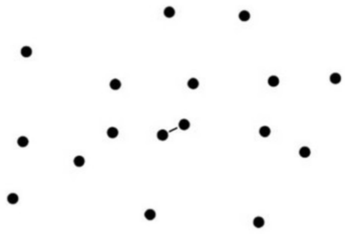
```
void quicksort( AnyType [ ] a, int left, int right )    {
      if( left + CUTOFF <= right )       {                    // usually  CUTOFF < 20
         AnyType pivot = median3( a, left, right );
         int i = left, j = right - 1;
         while (I<j)          {
            while( a[ ++i ].compareTo( pivot ) < 0 ) { }
            while( a[ --j ].compareTo( pivot ) > 0 ) { }
            if( i < j ) swapReferences( a, i, j );
         }
         swapReferences( a, i, right - 1 );   // Restore pivot
         quicksort( a, left, i - 1 );      // Sort small elements
         quicksort( a, i + 1, right );   // Sort large elements
      }
      else                                    // Do an insertion sort on the subarray
         insertionSort( a, left, right );
 }
```


Analysis of Quicksort

- Best case: split in the middle — Θ(n log n)
- Worst case: sorted array! — Θ(n^2)
- Average case: random arrays — Θ(n log n)
- Improvements:
  - better pivot selection: median of three partitioning
  - switch to insertion sort on small subfiles
  - elimination of recursion
  - These combine to 20-25% improvement
- Considered the method of choice for internal sorting of large files (n ≥ 10000)

Closest Pair Problem

- Naïve approach – compute distance between all pairs $\Theta(n^2)$ -- Can we do better?



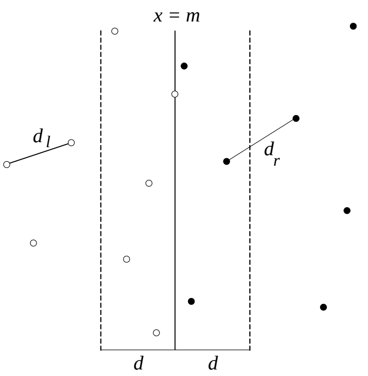
Closest-Pair Problem by Divide-and-Conquer

Step 1a: Sort the points by both x and y coordinates. Need to keep separate arrays to access the points in sorted order.

- Some algorithms only sort by x-coordinate initially, then sort by y-coordinate in the combine step

Closest-Pair Problem by Divide-and-Conquer

- Step 2: Divide the points given into two subsets P_{left} and P_{right} by a vertical line x = m so that half the points lie to the left or on the line and half the points lie to the right or on the line.



Closest Pair by Divide-and-Conquer (cont.)

- Step 3: Find recursively the closest pairs for the left and right subsets.
- Step 4: Set d = min { d_l , d_r }

  We can limit our attention to the points in the symmetric vertical strip S of width 2d as potentially closest pair.
  (The points are stored and processed in increasing order of their y coordinates.)
- Step 5: Scan the points in the vertical strip S from the lowest up to highest. For every point p(x, y) in the strip, inspect points in the strip that may be closer to p than d. There can be no more than 5 such points following p on the strip list!

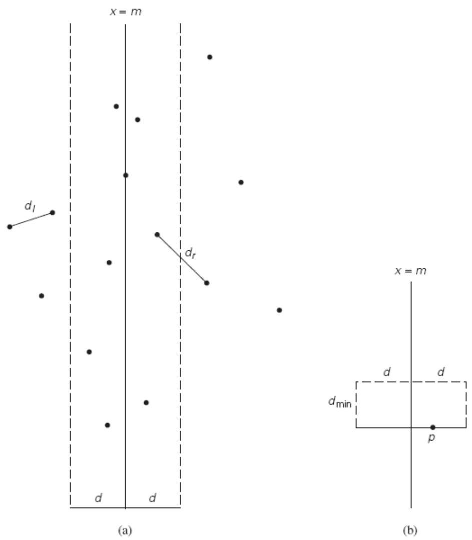


FIGURE 5.7 (a) Idea of the divide-and-conquer algorithm for the closest-pair problem.
(b) Rectangle that may contain points closer than d_{min} to point p.

Divide and Conquer Closest-Pair Algorithm

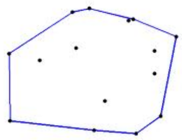
P: Array of points in non-decreasing order of x-coordinate
Q: Array of same points in non-decreasing order of y-coordinate
If n ≤ 3
    return minimum distance using brute force
else
    copy first half of points into P_{left} , second half into P_{right}
    copy same points from Q into Q_{left} , second half into Q_{right}
        // points ordered both by x and y coordinates
    d_{left} = closestPair (P_{left} , Q_{left})
    d_{right} = closestPair (P_{right} , Q_{right})
    d_{min} = min {d_{left} , d_{right} }
    m = middle x-coordinate = P[n/2].x
    copy all the points within $2d_{min}$ band around m into a temp array sorted by y coord
    loop over the array, for each point - p
        loop over points q that are within d_{min} vertically (|p.y – q.y| < d_{min})
            check if dist(p,q) smaller than d_{min},
                if yes replace d_{min} with dist(p,q)
    return d_{min}

Efficiency of the Closest-Pair Algorithm

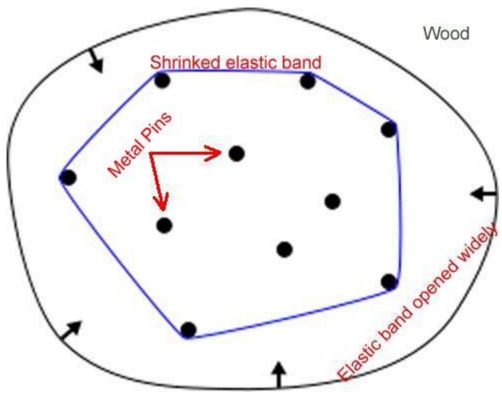
- Running time of the algorithm is described by
- T(n) = 2T(n/2) + M(n), where M(n) $\in$ O(n) just like Merge sort
- By the Master Theorem (with a = 2, b = 2, d = 1)
  T(n) $\in$ O(n log n)

Convex Hull Problem: given a set of points, find the smallest convex set containing the points, Convex Hull

- A convex combination of two distinct points is any point on the line segment between them.
- Convex set: A set of points in the plane is called convex if, for any two points p and q in the set, the entire line segment from p to q including the endpoints belongs to the set.
- Convex hull of a set S is the smallest convex set that includes S
- To "solve" the convex hull problem we will find the extreme points of the convex set – that is the corners of the convex hull.



Convex Hull – Physical determination

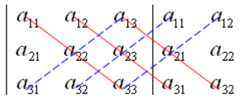


Some necessary computational facts

Two key steps: Determine

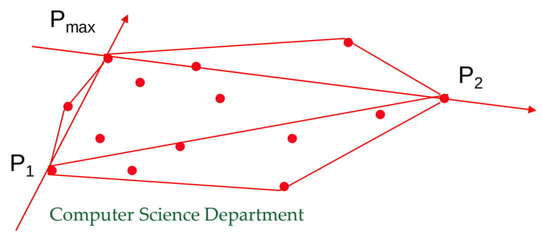
- Where a point, p_3, is to the left or right of a line (with direction), e.g. line from p_1 to p_2 .
- The distance of a point p from a line.

- Great news – both of these are solved by one calculation
  - Det of $\begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix}$
  - Det > 0, then p_3 is to the left of $\overrightarrow{p_1p_2}$; < 0, to the right ; = 0, on the line
  - Finally normalizing by the length of the segment $\overline{p_1p_2}$ gives the distance



Quickhull Algorithm

- Convex hull: smallest convex set that includes given points
- Assume points are sorted by x-coordinate values
- Identify extreme points P_1 and P_2 (leftmost and rightmost)
- Compute upper hull recursively:
  - find point P_{max} that is farthest away from line P_1P_2
  - compute the upper hull of the points to the left of line P_1P_{max}
  - compute the upper hull of the points to the left of line $P_{max}P_2$
- Compute lower hull in a similar manner



Quickhull Algorithm

Input = a set S of n points
Assume that there are at least 2 points in the input set S of points
QuickHull (S)
// Find convex hull from the set S of n points
Convex Hull := {}
Find left and right most points, say A & B, and add A & B to convex hull
Segment AB divides the remaining (n-2) points into 2 groups S_1 and S_2
where S_1 are points in S that are on the left side of the oriented line from A to B,
and S_2 are points in S that are on the left side of the oriented line from B to A
FindHull (S_1, A, B)
FindHull (S_2, B, A)

Quickhull Algorithm

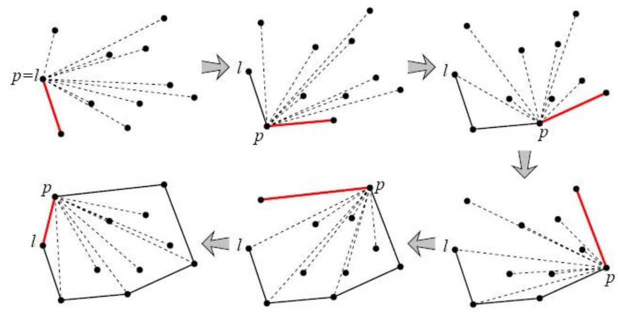
FindHull (S_k, P, Q)
// Find points on convex hull from the set S_k of points
// that are on the left side of the oriented line from P to Q
If S_k has no point then return
From the given set of points in S_k, find farthest point, say C, from segment PQ
Add point C to convex hull at the location between P and Q
Three points P, Q, and C partition the remaining points of S_k into 3 subsets:
S_0 are points inside triangle PCQ,
S_1 are points on the left side of the oriented line from P to C, and
S_2 are points on the left side of the oriented line from C to Q.
FindHull(S_1, P, C)
FindHull(S_2, C, Q)
Output = convex hull

Efficiency of Quickhull Algorithm

- Finding point farthest away from line P_1P_2 can be done in linear time
- Time efficiency:
  - worst case: $\Theta(n^2)$ (as quicksort)
  - average case: $\Theta(n)$ (under reasonable assumptions about distribution of points given)
- If points are not initially sorted by x-coordinate value, this can be accomplished in O(n log n) time
- Several O(n log n) algorithms for convex hull are known

Convex Hull Problem: Jarvis March (Wrapping algorithm)

Algorithm finds the points on the convex hull in the order in which they appear. It is quick if there are only a few points on the convex hull, but slow if there are many. Let x_0 be the leftmost point. Let x_1 be the first point counterclockwise when viewed from x_0. etc. (O(nh)) h: #pts in CH)



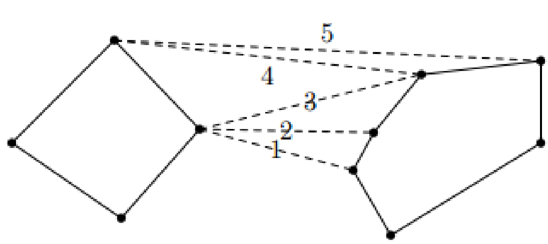
The execution of Jarvis's March.

Convex Hull Problem: (Pure) Divide and Conquer

Divide and conquer

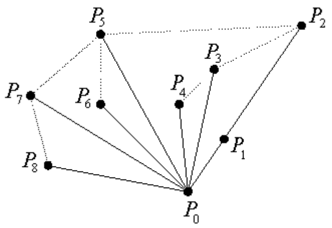
1. Divide the n points into two halves.
2. Find convex hull of each subset.
3. Combine the two hulls into overall convex hull.

Combine! (march up/down until upper/lower tangent)



Convex Hull Problem: Graham Scan

The idea is to identify one vertex of the convex hull and sort the other points as viewed from that vertex. Then the points are scanned in order. Let x_0 be the leftmost point and number the remaining points by angle from x_0 going counterclockwise: x_1; x_2; : : : ; x_{n-1}. Let $x_n = x_0$, the chosen point.



General Divide-and-Conquer Recurrence

$T(n) = aT(n/b) + f(n)$ where $f(n) \in \Theta(n^d)$, $d \geq 0$

Examples: $T(n) = 2T(n/2) + C \Rightarrow T(n) \in$?
What if a = 1, 4; what term dominates?

$T(n) = 2T(n/2) + n \Rightarrow T(n) \in$?
What if a = 4, 8; what term dominates?

$T(n) = 2T(n/2) + n^2 \Rightarrow T(n) \in$?
What if a = 4, 8; what term dominates?

General Divide-and-Conquer Recurrence

$T(n) = aT(n/b) + f(n)$ where $f(n) \in \Theta(n^d)$, $d \geq 0$

<u>Master Theorem</u>: If $a < b^d$ or $\log_b(a) < d$, $T(n) \in \Theta(n^d)$
If $a = b^d$ or $\log_b(a) = d$, $T(n) \in \Theta(n^d \log n)$
If $a > b^d$ or $\log_b(a) > d$, $T(n) \in \Theta(n^{\log_b a})$

Note: The same results hold with O instead of Θ.

End Spring 2018

Multiplication of Large Integers

- Consider the problem of multiplying two (large) n-digit integers represented by arrays of their digits such as:

A = 12345678901357986429

B = 87654321284820912836

The grade-school algorithm:

```
                    a1  a2 …  an
                    b1  b2 …  bn
             (d10) d11d12 … d1n
        (d20) d21d22 … d2n
        … … … … … … …
(dn0) dn1dn2 … dnn
```

- Efficiency: n^2 one-digit multiplications

First Divide-and-Conquer Algorithm

A small example: A ∗ B where A = 2135 and B = 4014

A = (21·102 + 35), B = (40 ·102 + 14)

So, A ∗ B = (21 ·102 + 35) ∗ (40 ·102 + 14)

= 21 ∗ 40 ·104 + (21 ∗ 14 + 35 ∗ 40) ·102 + 35 ∗ 14

- In general, if A = A1A2 and B = B1B2 (where A and B are n-digit, A1, A2, B1, B2 are n/2-digit numbers)
- A ∗ B = A1 ∗ B1·10n + (A1 ∗ B2 + A2 ∗ B1) ·10n/2 + A2 ∗ B2

- Recurrence for the number of one-digit multiplications M(n): →
M(n) = 4M(n/2), M(1) = 1
Solution: M(n) = n^2

First Divide-and-Conquer Algorithm

- A small example: $A * B$ where $A = 23$ and $B = 54$
  - $A = (2 \cdot 10 + 3)$, $B = (5 \cdot 10 + 4)$
- So, $A * B = (2 \cdot 10 + 3) * (5 \cdot 10 + 4)$
  $= 20 * 50 + (20 * 4 + 50 * 3) + 3 * 4$

- Can easily generalize with any split of the numbers
- Recurrence for the number of one-digit multiplications M(n): → $M(n) = 4M(n/2)$, $M(1) = 1$
  Solution: $M(n) = n2$

Second Divide-and-Conquer Algorithm

$A * B = A1 * B1 \cdot 10n + (A1 * B2 + A2 * B1) \cdot 10n/2 + A2 * B2$
The idea is to decrease the number of multiplications from 4 to 3:

$(A1 + A2) * (B1 + B2) = A1 * B1 + (A1 * B2 + A2 * B1) + A2 * B2$,
→
$(A1 * B2 + A2 * B1) = (A1 + A2) * (B1 + B2) - A1 * B1 - A2 * B2$,

which requires only 3 multiplications at the expense of (4-1) extra add/sub.

- Recurrence for the number of multiplications M(n):
  $M(n) = 3M(n/2)$, $M(1) = 1$
  Solution: $M(n) = 3^{\log 2\ n} = n^{\log 2\ 3} \approx n^{1.585}$

Why is this important?

- Showed that more efficient algorithms existed for mathematical problems.
- Basic idea can be extended to matrices
- Mathematical algorithms that use similar ideas
  - Polynomial multiplication
  - Fast Fourier Transform