Chapter 4: Divide and Conquer

Chin Lung Lu

Department of Computer Science

National Tsing Hua University

Divide and conquer strategy

Basic idea of divide and conquer strategy:

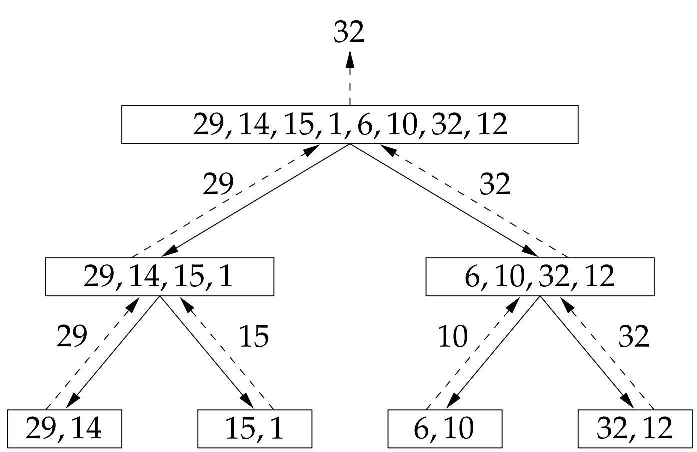
1. If the problem size is small enough, then solve the problem by some straightforward method; otherwise, divide a problem into two (or more) smaller sub-problems, preferably in equal size.

   (Note that each sub-problem is identical to its original problem, except its input size is smaller.)
2. Solve these two sub-problems by using the divide and conquer strategy again (i.e., solving them recursively).
3. Merge two sub-solutions into the final solution.

Divide and conquer strategy (cont'd)

Example (finding maximum problem):

Find the maximum among $S = \{29, 14, 15, 1, 6, 10, 32, 12\}$.



Time complexity of divide and conquer

- Let $T(n)$ be the time complexity of a problem with input size n.

Recursive formula of $T(n)$:

$$T(n) = \begin{cases} 2T(\frac{n}{2}) + S(n) + M(n) & \text{if } n \geq c \\ b & \text{if } n < c \end{cases}$$

- $S(n)$: Time to divide the problem into two sub-problems
- $M(n)$: Time to merge two sub-solutions into the final solution
- b (a constant): Time to straightforwardly solve the sub-problem of size small enough

Time complexity of divide and conquer (cont'd)

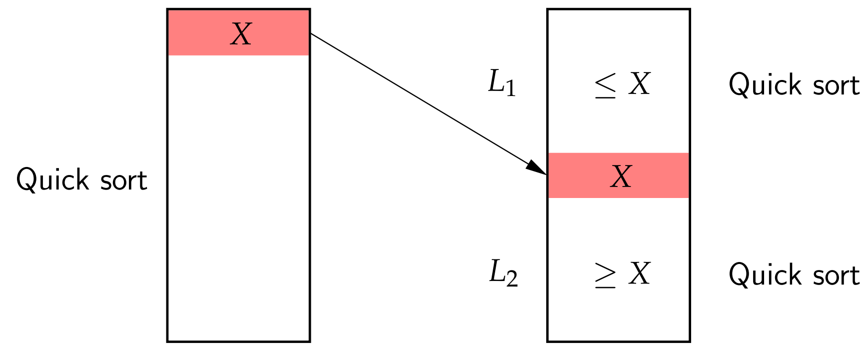
Example (finding maximum problem): let $n = 2^k$

$$T(n) = \begin{cases} 2T(\frac{n}{2}) + \mathcal{O}(1) & \text{if } n > 2 \\ 1 & \text{if } n = 2 \end{cases}$$

$$\begin{aligned} T(n) &= 2T(\tfrac{n}{2}) + 1 \\ &= 2(2T(\tfrac{n}{4}) + 1) + 1 \;=\; 2^2T(\tfrac{n}{2^2}) + 2^1 + 2^0 \\ &\;\vdots \\ &= 2^{k-1}T(2) + 2^{k-2} + \ldots + 2^0 \\ &= 2^{k-1} + 2^{k-2} + \ldots + 2^0 \\ &= 2^k - 1 \\ &= n - 1 \\ &= \mathcal{O}(n) \end{aligned}$$

Quick sort

- Given a set of n numbers $a_1, a_2, \ldots, a_n$, we choose a number X to divide $a_1, a_2, \ldots, a_n$ into two lists as shown below.



Quick sort algorithm

Algorithm: *Quicksort*(f, l)
Input: A sequence of $(l - f + 1)$ numbers $a_f, a_{f+1}, \ldots, a_l$.
Output: The sorted sequence of $a_f, a_{f+1}, \ldots, a_l$.

1. **if** $f \geq l$, **then** return
2. $X = a_f$, $i = f$, $j = l$
3. **while** $i < j$ **do**
4.   **while** $a_j \geq X$ and $i < j$ **do**
5. $j = j - 1$
6. $a_i \leftrightarrow a_j$
7.   **while** $a_i \leq X$ and $i < j$ **do**
8. $i = i + 1$
9. $a_i \leftrightarrow a_j$
10. **end while**
11. *Quicksort*$(f, j - 1)$, *Quicksort*$(j + 1, l)$

Example of quick sort

Iteration 1: Let
$a_1 = 3, a_2 = 6, a_3 = 1, a_4 = 4, a_5 = 5, a_6 = 2.$

$X = 3$	a_1	a_2	a_3	a_4	a_5	a_6
$i = 1, j = 6$	3	6	1	4	5	2
$(a_j = a_6 < X)$	$\uparrow i$					$\uparrow j$
$a_1 \leftrightarrow a_6$	2	6	1	4	5	3
$(a_i = a_1 < X)$	$\uparrow i$					$\uparrow j$
$i = i + 1 = 2$	2	6	1	4	5	3
$(a_i = a_2 > X)$		$\uparrow i$				$\uparrow j$
$a_2 \leftrightarrow a_6$	2	3	1	4	5	6
$(a_j = a_6 > X)$		$\uparrow i$				$\uparrow j$
$j = j - 1 = 5$	2	3	1	4	5	6
$(a_j = a_5 > X)$		$\uparrow i$			$\uparrow j$	

Example of quick sort (cont'd)

Iteration 1 (cont'd):

$X=3$	a_1	a_2	a_3	a_4	a_5	a_6
$j=j-1=4$	2	3	1	4	5	6
$(a_j=a_4>X)$		$\uparrow i$		$\uparrow j$		
$j=j-1=3$	2	3	1	4	5	6
$(a_j=a_3<X)$		$\uparrow i$	$\uparrow j$			
$a_2 \leftrightarrow a_3$	2	1	3	4	5	6
$(a_i=a_2<X)$		$\uparrow i$	$\uparrow j$			
$i=i+1=3$	2	1	3	4	5	6
$(i=j=3)$			$i \uparrow j$			
(end of iteration 1)	≤ 3	≤ 3	$=3$	≥ 3	≥ 3	≥ 3

Time complexity of quick sort

- Let $T(n)$ be the time complexity of quick sort with input size n.

Best case:

- The best case occurs when we can split the problem into two equal-size subproblems for each round.
- It means that $T(n) = 2T(\frac{n}{2}) + \mathcal{O}(n)$.
- In this case, we have $T(n) = \mathcal{O}(n \log n)$.

Worst case:

- The worst case occurs when the input data is already a sorted or reversely sorted sequence.
- In this case, we have $T(n) = T(n-1) + \mathcal{O}(n) = \mathcal{O}(n^2)$.

Time complexity of quick sort (cont'd)

Given $T(n) = 2T(\frac{n}{2}) + \mathcal{O}(n)$, let $n = 2^k$ (i.e., $k = \log_2 n$).

$$
\begin{aligned}
T(n) &\leq 2T(\tfrac{n}{2}) + cn \\
&\leq 2(2T(\tfrac{n}{4}) + c\tfrac{n}{2}) + cn \\
&= 4T(\tfrac{n}{4}) + 2c\tfrac{n}{2} + cn \\
&\vdots \\
&= 2^k T(\tfrac{n}{2^k}) + 2^{k-1} c \tfrac{n}{2^{k-1}} + 2^{k-2} c \tfrac{n}{2^{k-2}} + \ldots + 2^0 c \tfrac{n}{2^0} \\
&= nT(1) + cn + cn + \ldots + cn \\
&= nT(1) + cnk \\
&= n + cn \log_2 n
\end{aligned}
$$

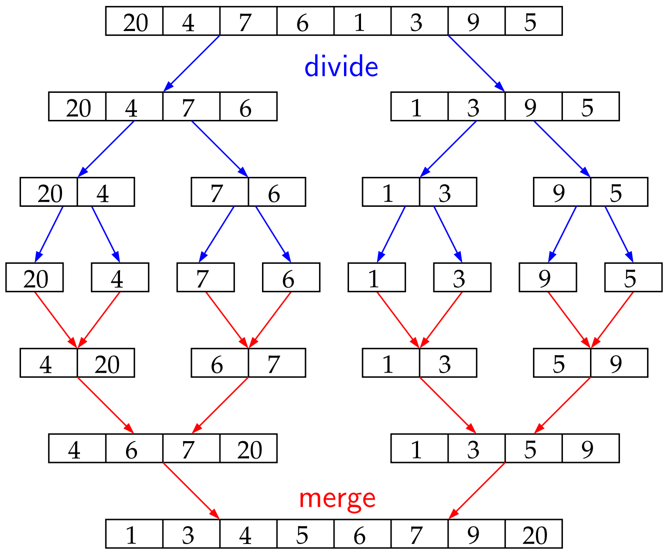
Therefore, $T(n) = \mathcal{O}(n \log n)$.

Merge sort algorithm

Algorithm: *MergeSort*(C)
Input: A list C of n elements
Output: A sorted list of these n elements

1. **if** $n = 1$ **then** return C.
2. Let C_L be the list of first $\frac{n}{2}$ elements of c.
3. Let C_R be the list of last $n - \frac{n}{2}$ elements of c.
4. $S_L = MergeSort(C_L)$.
5. $S_R = MergeSort(C_R)$.
6. $Merge(S_L, S_R)$.

Example of merge sort



Time complexity of merge sort

- Note that the cost of merging two sorted sequences is $\mathcal{O}(n)$.

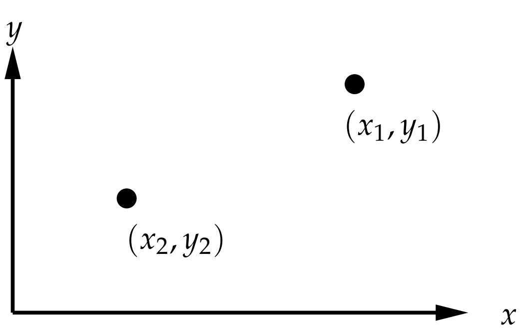
Time complexity of the merge sort $T(n)$:

$$T(n) = \begin{cases} 2T\left(\frac{n}{2}\right) + \mathcal{O}(n) & \text{if } n > 2 \\ 1 & \text{if } n = 2 \end{cases}$$

- Therefore, we have $T(n) = \mathcal{O}(n \log n)$.

Point domination

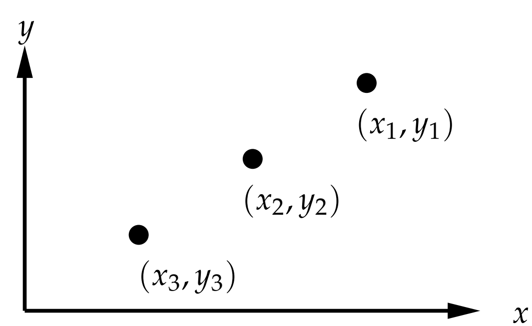
- In the 2D space, a point (x_1, y_1) dominates a point (x_2, y_2) if $x_1 > x_2$ and $y_1 > y_2$.



(x_1, y_1) dominates (x_2, y_2).

Maximal point

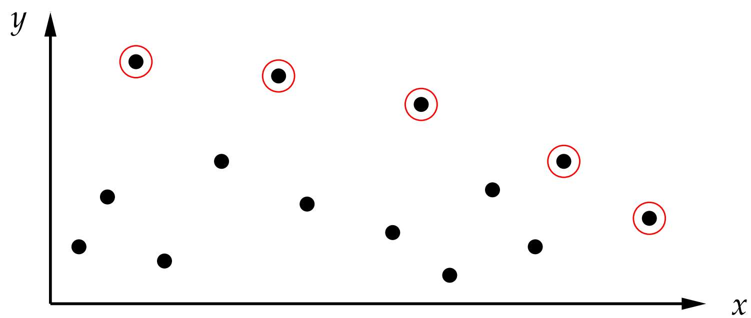
- A point is called a maximal point if no other point dominates it.



(x_1, y_1) is a maximal point, but both (x_2, y_2) and (x_3, y_3) are not.

2D maxima finding problem

- Given a set of n points, the 2D maxima finding problem is to find all of the maximal points among these n points.



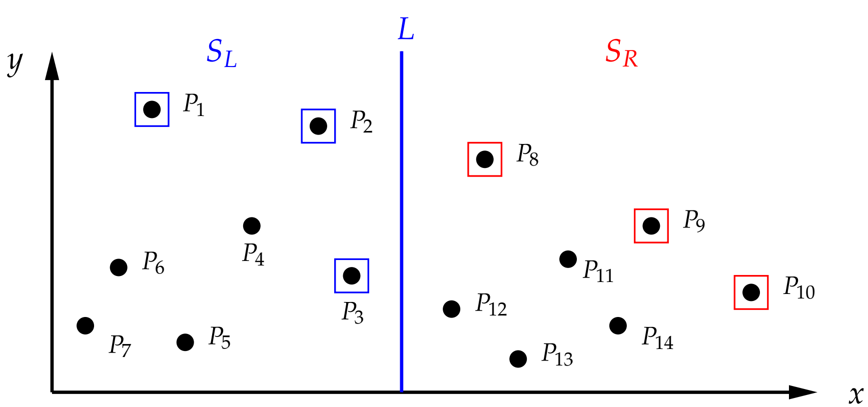
The circle points are maximal points.

2D maxima finding problem (cont'd)

- A straightforward method we can use to solve the 2D maxima finding problem is to compare every pair of points.
- This method requires $\mathcal{O}(n^2)$ comparison of points.
- In fact, the divide and conquer strategy can solve the problem in $\mathcal{O}(n \log n)$ steps.

Divide and conquer method

Dividing process of 2D maxima finding problem



Divide and conquer method

Merging process of 2D maxima finding problem

- Note that the x-value of a point in S_R is always larger than the x-value of every point in S_L.

Observation:

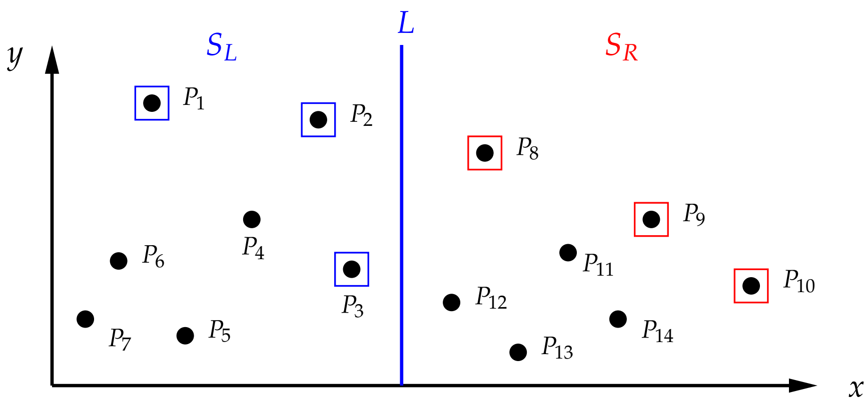
A point in S_L is a maximal point if and only if its y-value is not less than the y-value of a maximal point in S_R.

Divide and conquer method

Merging process of 2D maxima finding problem (cont'd)

Example:

P_3 is not a maximal point in S_L because its y-value is less than the y-values of P_8 and P_9 in S_R.



Divide and conquer algorithm

2D maxima finding problem

Algorithm: $FindMax(S)$
Input: A set S of n planar points.
Output: A set $MP(S)$ of all maximal points of S.

1. **if** S contains only one point **then** return S.
2. Find L perpendicular to the x-axis that separates S into S_L and S_R with $|S_L| = |S_R| = n/2$.
3. $FindMax(S_L)$ and $FindMax(S_R)$.
   `/* Recursively find maximal points of` S_L `and` S_R `*/`
4. Project the maximal points of S_L and S_R onto L and sort them by their y-values.
5. Conduct a linear scan on the projections and discard each of the maximal points of S_L if its y-value is less than the y-value of some maximal point of S_R.

Time complexity of $FindMax(S)$

- Let $T(n)$ be the time complexity of $FindMax(S)$.
- In fact, L can be found in $\mathcal{O}(n)$ time by finding the median of n numbers (will be discussed in Chapter 6 on prune and search).
- Step 4 costs $\mathcal{O}(n \log n)$ time using heap sort.

Recursive formula of $T(n)$:

$$T(n) = \begin{cases} 2T(\frac{n}{2}) + \mathcal{O}(n) + \mathcal{O}(n \log n) & \text{if } n > 1 \\ 1 & \text{if } n = 1 \end{cases}$$

- It can be proved that $T(n) = \mathcal{O}(n \log^2 n)$.

Time complexity of $FindMax(S)$ (cont'd)

Let $n = 2^k$ (i.e., $k = \log_2 n$).

$$\begin{aligned} T(n) &\leq 2T(\tfrac{n}{2}) + cn\log_2 n \\ &\leq 2(2T(\tfrac{n}{4}) + c\tfrac{n}{2}\log_2 \tfrac{n}{2}) + cn\log_2 n \\ &= 2^2T(\tfrac{n}{2^2}) + cn(\log_2 \tfrac{n}{2^1} + \log_2 \tfrac{n}{2^0}) \\ &\vdots \\ &= 2^kT(\tfrac{n}{2^k}) + cn(\log_2 \tfrac{n}{2^{k-1}} + \log_2 \tfrac{n}{2^{k-2}} + \ldots + \log_2 \tfrac{n}{2^0}) \\ &= nT(1) + cn(\log_2 2 + \log_2 4 + \ldots + \log_2 n) \\ &= nT(1) + cn\left(\tfrac{(1+\log_2 n)\log_2 n}{2}\right) \\ &= n + \tfrac{cn\log_2^2 n}{2} + \tfrac{cn\log_2 n}{2} \end{aligned}$$

Therefore, $T(n) = \mathcal{O}(n \log^2 n)$.

Time complexity of $FindMax(S)$ (cont'd)

- The divide and conquer algorithm is dominated by sorting in the merging step.
- The sorting actually can be done only once by a pre-sorting.
- Using the pre-sorting, the merging complexity is $\mathcal{O}(n)$.

Modified recursive formula of $T(n)$:

The total time complexity is $T(n) + \mathcal{O}(n \log n)$, where

$$T(n) = \begin{cases} 2T(\frac{n}{2}) + \mathcal{O}(n) + \mathcal{O}(n) & \text{if } n > 1 \\ 1 & \text{if } n = 1 \end{cases}$$

- As shown before, $T(n) = \mathcal{O}(n \log n)$.
- Hence, the time complexity of $FindMax(S)$ is $\mathcal{O}(n \log n)$.

1D closest pair problem

Definition:

Given a set S of n points in one-dimensional space, the 1D closest pair problem is to find a pair of points which are closest together.

- Example The closest pair in the set $\{4, 11, 1, 7, 3\}$ is $(3, 4)$.

A simple algorithm to solve the 1D closest pair problem:

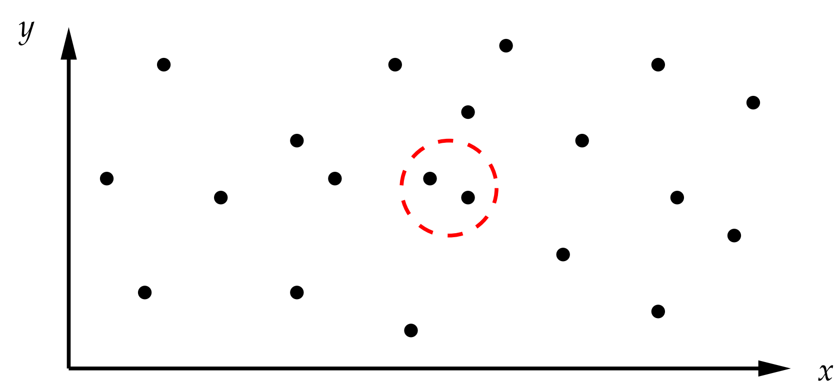
1. Sort n numbers.
2. Linearly scan the sorted n numbers and find two consecutive numbers whose distance is minimum.

- The time complexity of this simple algorithm is $\mathcal{O}(n \log n)$.

2D closest pair problem

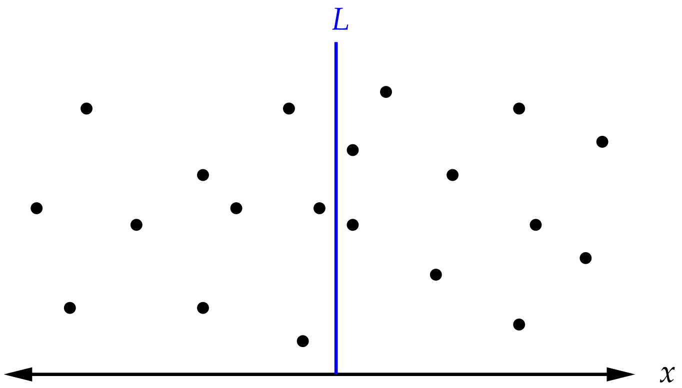
Definition:

Given a set S of n points in two-dimensional space, find a pair of points which are closest together.



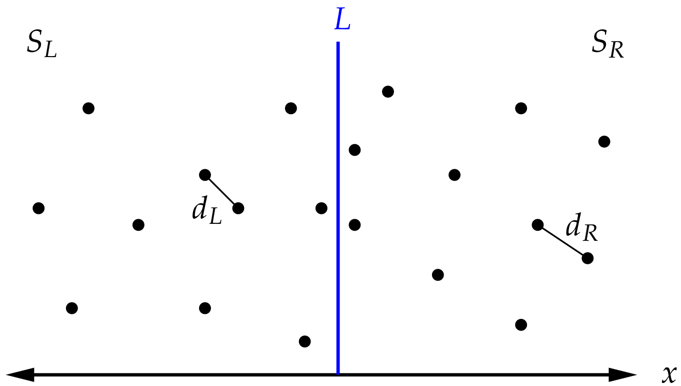
Divide and conquer for 2D closest pair

- Find a line L perpendicular to the x-axis to partition S into S_L and S_R such that $|S_L| \simeq |S_R|$.



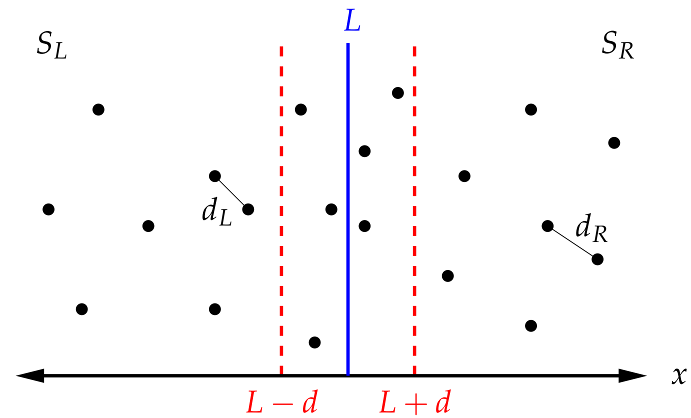
Divide and conquer for 2D closest pair (cont'd)

- By recursively solving the 2D closest pair problem in S_L and S_R, we obtain d_L and d_R, which denote the distances of the closest pairs in S_L and S_R, respectively.



Divide and conquer for 2D closest pair (cont'd)

- Let $d = \min(d_L, d_R)$.
- If the closest pair (P_a, P_b) of S consists of a point in S_L and a point in S_R, then P_a and P_b have to lie within a slab centered at line L and bounded by lines $L - d$ and $L + d$.

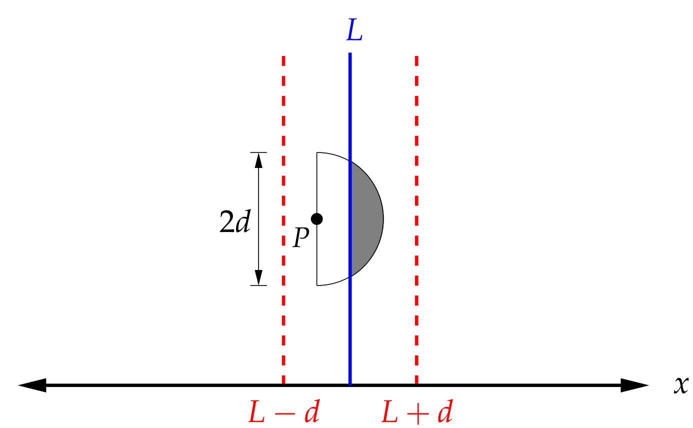


Divide and conquer for 2D closest pair (cont'd)

- In other words, we may examine only points in the slab during the merging step in the divide and conquer algorithm.
- In the worst case, there can be n points within the slab.
- Hence the brute force method to find the closest pair in this slab need to calculate $n^2/4$ distances and comparisons.
- This kind of merging step will not be good for our divide and conquer algorithm.

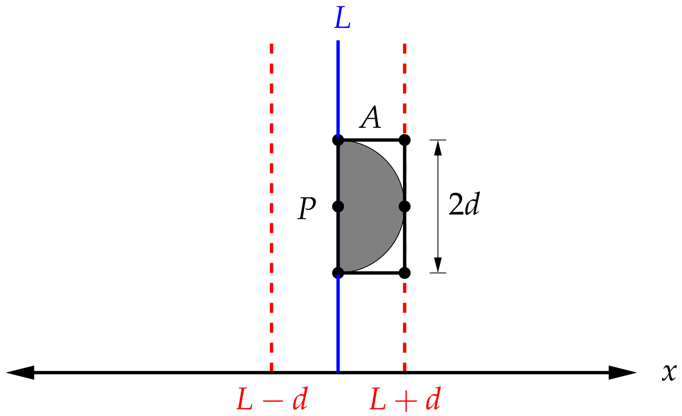
Divide and conquer for 2D closest pair (cont'd)

- If a point P in S_L and a point Q in S_R constitute a closest pair, the distance between P and Q must be less than d.
- It means that we do not consider a point too far away from P.
- Actually, we only have to examine the shaded area as follows.



Divide and conquer for 2D closest pair (cont'd)

- If P is exactly on L, then we only have to examine points within the rectangle A.



There are at most 6 points in the rectangle A.

Divide and conquer algorithm

2D closest pair problem

Algorithm: $2dClosestPair(S)$
Input: A set S of n points in the plane.
Output: The distance of the closest pair in S.
Preprocessing: Sort the points of S by their y-values and x-values, respectively.

1. **if** S contains only one point **then** return ∞ as its distance.
2. Find a median line L perpendicular to the x-axis to divide S into two equal sized subsets S_L and S_R.
3. $2dClosestPair(S_L)$ and $2dClosestPair(S_R)$.
   `/* Recursively solve the problems of` S_L `and` S_R `*/`
4. Let $d = \min\{d_L, d_R\}$.

Divide and conquer algorithm (cont'd)

2D closest pair problem

Algorithm: $2dClosestPair(S)$ (cont'd)

5. Project all points within the area bounded by $L-d$ and $L+d$ onto L.
6. **for** each point P in the half slab bounded by $L-d$ and L with descending y-value **do**
7. Let the y-value of P be y_p.
8. Find all points in the half slab bounded by L and $L+d$ whose y-values fall within y_p+d and y_p-d.
9. **if** the shortest distance d' between P and a point in the other half slab is less than d **then**
10. $d = d'$.
11. **end if**
12. **end for**

Divide and conquer algorithm (cont'd)

2D closest pair problem

Time complexity of $2dClosestPair(S)$:

The time complexity of this algorithm is $\mathcal{O}(n\log n) + T(n)$ and

$$T(n) = 2T\left(\frac{n}{2}\right) + Split(n) + Merge(n)$$

where $Split(n) = \mathcal{O}(n)$ and $Merge(n) = \mathcal{O}(n)$.

- $Split(n) = \mathcal{O}(n)$ because points are sorted by their x-values.
- $Merge(n) = \mathcal{O}(n)$ since the area bounded by $L-d$ and $L+d$ contains at most n points and for each point from the left half slab, at most 6 points need to be examined.

Divide and conquer algorithm (cont'd)

2D closest pair problem

Recursive formula of $T(n)$:

$$T(n) = \begin{cases} 2T\left(\frac{n}{2}\right) + \mathcal{O}(n) + \mathcal{O}(n) & \text{if } n > 1 \\ 1 & \text{if } n = 1 \end{cases}$$
$$= \mathcal{O}(n \log n)$$

- As a result, the time complexity of the divide and conquer algorithm for the 2D closest pair problem is $\mathcal{O}(n \log n)$.

Master theorem

- Let $T(n)$ be the recurrence defined below:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

  where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is a positive function.
- The above recurrence of $T(n)$ describes the running time of an algorithm that divides a problem of size n into a subproblems, each of size n/b.
- The a subproblems are solved recursively, each in $T\left(\frac{n}{b}\right)$ time.
- The function $f(n)$ denotes the cost of dividing the problem and combining the results of the subproblems.

Master theorem (cont'd)

Case 1:

$T(n) = \Theta(n^{\log_b a})$, if $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$.

Example 1:

Let $T(n) = 9T\left(\frac{n}{3}\right) + n$. Then $T(n) = \Theta(n^2)$.

- In this case, we have $a = 9, b = 3$ and $f(n) = n$.
- We then have $n^{\log_b a} = n^{\log_3 9} = n^2$.
- By letting $\epsilon = 1$, we have $f(n) = n = \mathcal{O}(n) = \mathcal{O}(n^{\log_b a - \epsilon})$.
- Hence, $T(n) = \Theta(n^{\log_b a}) = \Theta(n^2)$.

Master theorem (cont'd)

Case 2:

$T(n) = \Theta(n^{\log_b a} \log_2 n)$, if $f(n) = \Theta(n^{\log_b a})$.

Example 2:

Let $T(n) = T\left(\frac{2n}{3}\right) + 1$. Then $T(n) = \Theta(\log_2 n)$.

- In this case, we have $a = 1, b = \frac{3}{2}$ and $f(n) = 1$.
- Thus, we have $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$.
- Clearly, $f(n) = 1 = \Theta(1) = \Theta(n^{\log_b a})$.
- Hence, $T(n) = \Theta(n^{\log_b a} \log_2 n) = \Theta(\log_2 n)$.

Master theorem (cont'd)

Case 3:
$T(n) = \Theta(f(n))$, if $f(n) = \Omega(n^{\log_b a+\epsilon})$ for some constant $\epsilon > 0$ and $af(n/b) \le cf(n)$ for some constant $c < 1$ and all sufficiently large n.

Example 3:
Let $T(n) = 3T\left(\frac{n}{4}\right) + n\log_2 n$. Then $T(n) = \Theta(n\log_2 n)$.

- In this case, $n^{\log_b a} = n^{\log_4 3} \approx O(n^{0.793})$ and $f(n) = n\log_2 n$.
- By letting $\epsilon \approx 0.2$, we have $f(n) = n\log_2 n = \Omega(n^{\log_b a+\epsilon})$.
- Also, $af\left(\frac{n}{b}\right) = 3\left(\frac{n}{4}\right)\log_2\left(\frac{n}{4}\right) \le \left(\frac{3}{4}\right)n\log_2 n = cf(n)$ for $c = \frac{3}{4}$.

Master theorem (cont'd)

Example 4:
Let $T(n) = 2T\left(\frac{n}{2}\right) + n\log_2 n$. Then $T(n) = \Theta(n\log_2^2 n)$.

- In this case, $n^{\log_b a} = n^{\log_2 2} = n$ and $f(n) = n\log_2 n$.
- Note that case 3 of the master method cannot apply to this case, since $\log_2 n = O(n^\epsilon)$ for any positive constant ϵ.

Master theorem (cont'd)

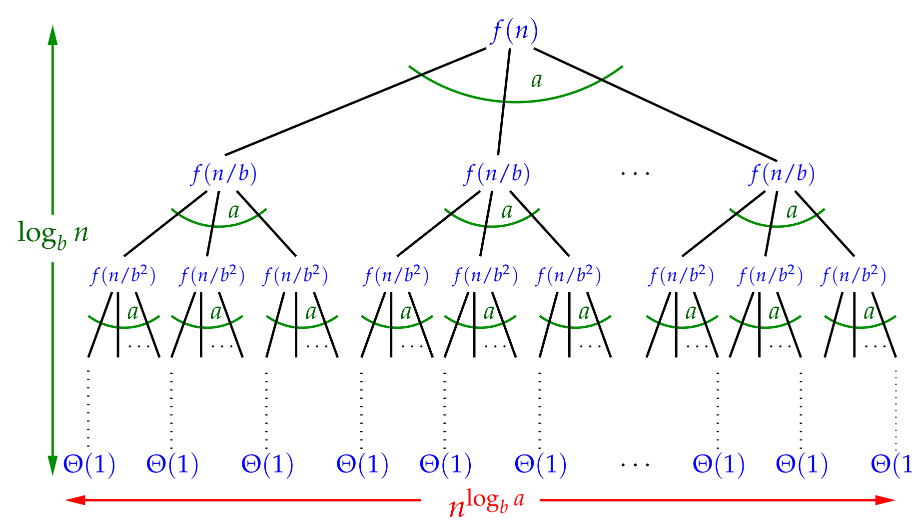
- To easily prove the correctness of the master theorem, we assume that $T(n)$ is defined on exact power of b (i.e., $n = 1, b, b^2, \ldots$):

$$T(n) = \begin{cases} aT\left(\frac{n}{b}\right) + f(n) & \text{if } n = b^i \\ \Theta(1) & \text{if } n = 1 \end{cases}$$

- Actually, the proof below can be applied to all positive integers n with a little modification.

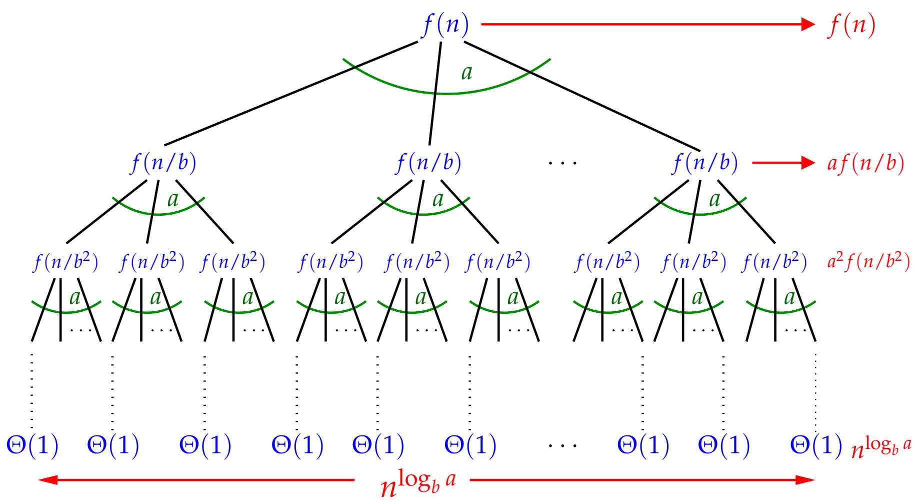
Proof of master theorem

Recursive tree $T(n) = aT(n/b) + f(n)$



Proof of master theorem (cont'd)

Recursive tree $T(n) = aT(n/b) + f(n)$



Proof of master theorem (cont'd)

Lemma:

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n - 1} a^j f\left(\frac{n}{b^j}\right).$$

- In general, there are a^j nodes at depth j (starting from 0) and each such node has the cost $f\left(\frac{n}{b^j}\right)$.
- There are $a^{\log_b n} = n^{\log_b a}$ leaves in the tree, because each leaf is at depth $\log_b n$.
- Since the cost of each leaf is $T(1) = \Theta(1)$, the total cost of all leaves is $n^{\log_b a}$.
- This is the cost of doing all $n^{\log_b a}$ subproblems of size 1.

Proof of master theorem (cont'd)

- The cost for all internal nodes at depth j is $a^j f\left(\frac{n}{b^j}\right)$.
- Therefore, the total cost of all internal nodes is:

$$\sum_{j=0}^{\log_b n - 1} a^j f\left(\frac{n}{b^j}\right)$$

- This sum is the cost of dividing problems into subproblems and then recombining the subproblems.
- As a result, we have;

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n - 1} a^j f\left(\frac{n}{b^j}\right)$$

Proof of master theorem (cont'd)

- Let $g(n) = \sum_{j=0}^{\log_b n - 1} a^j f\left(\frac{n}{b^j}\right)$.

Case 1:

If $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, $g(n) = \mathcal{O}(n^{\log_b a})$.

- Since $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$, $f\left(\frac{n}{b^j}\right) = \mathcal{O}\left(\left(\frac{n}{b^j}\right)^{\log_b a - \epsilon}\right)$.
- Then we have $g(n) = \mathcal{O}\left(\sum_{j=0}^{\log_b n - 1} a^j \left(\frac{n}{b^j}\right)^{\log_b a - \epsilon}\right)$.

Proof of master theorem (cont'd)

$$\begin{aligned}\sum_{j=0}^{\log_b n-1} a^j\left(\frac{n}{b^j}\right)^{\log_b a-\epsilon} &= n^{\log_b a-\epsilon}\sum_{j=0}^{\log_b n-1}\left(\frac{ab^{\epsilon}}{b^{\log_b a}}\right)^j \\ &= n^{\log_b a-\epsilon}\sum_{j=0}^{\log_b n-1}(b^{\epsilon})^j \\ &= n^{\log_b a-\epsilon}\left(\frac{b^{\epsilon\log_b n}-1}{b^{\epsilon}-1}\right) \\ &= n^{\log_b a-\epsilon}\left(\frac{n^{\epsilon}-1}{b^{\epsilon}-1}\right)\end{aligned}$$

Since b and ϵ are constants, $n^{\log_b a-\epsilon}\left(\frac{n^{\epsilon}-1}{b^{\epsilon}-1}\right) = \mathcal{O}(n^{\log_b a})$.

Therefore, we have $g(n) = \mathcal{O}(n^{\log_b a})$.

Proof of master theorem (cont'd)

Case 2:
If $f(n) = \Theta(n^{\log_b a})$, then $g(n) = \Theta(n^{\log_b a}\log_2 n)$.

- Assume that $f(n) = \Theta(n^{\log_b a})$.
- Then $f\left(\frac{n}{b^j}\right) = \Theta\left(\left(\frac{n}{b^j}\right)^{\log_b a}\right)$ and

$$g(n) = \sum_{j=0}^{\log_b n-1} a^j f\left(\frac{n}{b^j}\right) = \Theta\left(\sum_{j=0}^{\log_b n-1} a^j\left(\frac{n}{b^j}\right)^{\log_b a}\right)$$

Proof of master theorem (cont'd)

$$\begin{aligned}\sum_{j=0}^{\log_b n-1} a^j\left(\frac{n}{b^j}\right)^{\log_b a} &= n^{\log_b a}\sum_{j=0}^{\log_b n-1}\left(\frac{a}{b^{\log_b a}}\right)^j \\ &= n^{\log_b a}\sum_{j=0}^{\log_b n-1} 1 \\ &= n^{\log_b a}\log_b n\end{aligned}$$

Therefore, $g(n) = \Theta(n^{\log_b a}\log_b n) = \Theta(n^{\log_b a}\log_2 n)$

Proof of master theorem (cont'd)

Case 3:
If $af\left(\frac{n}{b}\right) \le cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $g(n) = \Theta(f(n))$.

- Since $f(n)$ appears in the definition of $g(n)$ and all terms of $g(n)$ are nonnegative, we have $g(n) = \Omega(f(n))$.
- Assume that $af\left(\frac{n}{b}\right) \le cf(n)$ for some constant $c < 1$ and all sufficiently large n.
- We can rewrite this assumption as $f\left(\frac{n}{b}\right) \le \left(\frac{c}{a}\right)f(n)$.

Proof of master theorem (cont'd)

- By iterating the inequality $f(\frac{n}{b}) \leq (\frac{c}{a})f(n)$ j times, we have (assume $\frac{n}{b^{j-1}}$ is still sufficiently large):

$$\begin{aligned} f\left(\frac{n}{b^j}\right) &\leq \left(\frac{c}{a}\right)^1 f\left(\frac{n}{b^{j-1}}\right) \\ &\leq \left(\frac{c}{a}\right)^2 f\left(\frac{n}{b^{j-2}}\right) \\ &\vdots \\ &\leq \left(\frac{c}{a}\right)^j f\left(\frac{n}{b^0}\right) \end{aligned}$$

- Therefore, $f\left(\frac{n}{b^j}\right) \leq (\frac{c}{a})^j f(n)$ or equivalently $a^j f\left(\frac{n}{b^j}\right) \leq c^j f(n)$.

Proof of master theorem (cont'd)

- We use $\mathcal{O}(1)$ to denote the terms in $g(n)$ that are not covered by the assumption that n is sufficiently large.

$$\begin{aligned} g(n) &= \sum_{j=0}^{\log_b n-1} a^j f\left(\frac{n}{b^j}\right) \\ &\leq \sum_{j=0}^{\log_b n-1} c^j f(n) + \mathcal{O}(1) \\ &\leq f(n)\sum_{j=0}^{\infty} c^j + \mathcal{O}(1) \\ &= f(n)\left(\frac{1}{1-c}\right) + \mathcal{O}(1) \\ &= \mathcal{O}(f(n)) \end{aligned}$$

- Therefore, $g(n) = \Theta(f(n))$.

Proof of master theorem (cont'd)

Case 1: Total cost is dominated by cost of leaves

If $f(n) = \mathcal{O}(n^{\log_b a-\epsilon})$ for some $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.

$\because\ T(n) = \Theta(n^{\log_b a}) + \mathcal{O}(n^{\log_b a}) = \Theta(n^{\log_b a})$.

Case 2: Total cost is evenly distributed among levels

If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a}\log_2 n)$.

$\because\ T(n) = \Theta(n^{\log_b a}) + \Theta(n^{\log_b a}\log n) = \Theta(n^{\log_b a}\log_2 n)$.

Proof of master theorem (cont'd)

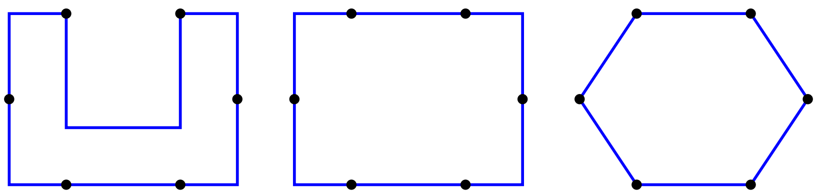
Case 3: Total cost is dominated by cost of root

If $f(n) = \Omega(n^{\log_b a+\epsilon})$ for some $\epsilon > 0$ and $af(\frac{n}{b}) \leq cf(n)$ for some constant $c < 1$ and for all sufficiently large n, $T(n) = \Theta(f(n))$.

$\because\ T(n) = \Theta(n^{\log_b a}) + \Theta(f(n)) = \Theta(f(n))$.

Convex hull

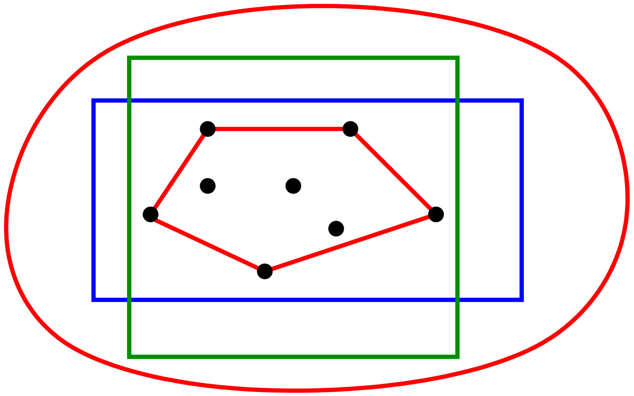
- Convex polygon is a polygon P such that any line connecting any two points inside P must lie in P.
- The convex hull of a given set of planar points is the smallest convex polygon containing all of the points



not convex polygon | convex polygon but not convex hull | convex hull

Convex hull problem

Definition:
Given a set S of planar points, find a convex hull for S, i.e., obtain the vertices of the convex hull in counterclockwise (clockwise) order.



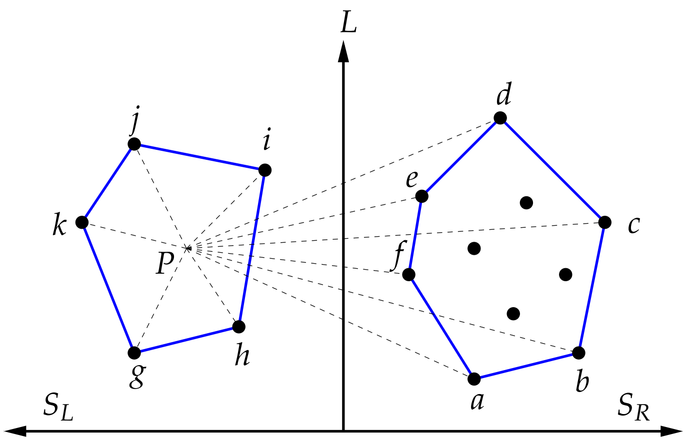
Convex hull problem

Divide and conquer algorithm

Algorithm: $CH(S)$ for computing a convex hull for S

1. **if** $|S| \leq 3$ **then** $CH(S) = S$ and return.
2. Find a median line perpendicular to the x-axis to divide S into S_L and S_R.
3. Recursively find $CH(S_L)$ and $CH(S_R)$.
4. Find an interior point P of $CH(S_L)$. `/* Merging step */`
5. Find v_1 and v_2 of $CH(S_R)$ such that $CH(S_R)$ is divided into two sequences of vertices that have increasing polar angles with respect to P, and let $y(v_1) > y(v_2)$.
   S_1: $V(CH(S_L))$ in counterclockwise direction.
   S_2: $V(CH(S_R))$ from v_2 to v_1 in counterclockwise.
   S_3: $V(CH(S_R))$ from v_2 to v_1 in clockwise order.
6. Merge S_1, S_2, S_3 and conduct the Graham scan.

Convex hull problem (cont'd)



- $CH(S_L) = (g, h, i, j, k)$ and $CH(S_R) = (a, b, c, d, e, f)$
- $v_1 = d$ and $v_2 = a$
- $S_1 = (g, h, i, j, k)$, $S_2 = (a, b, c, d)$ and $S_3 = (f, e)$

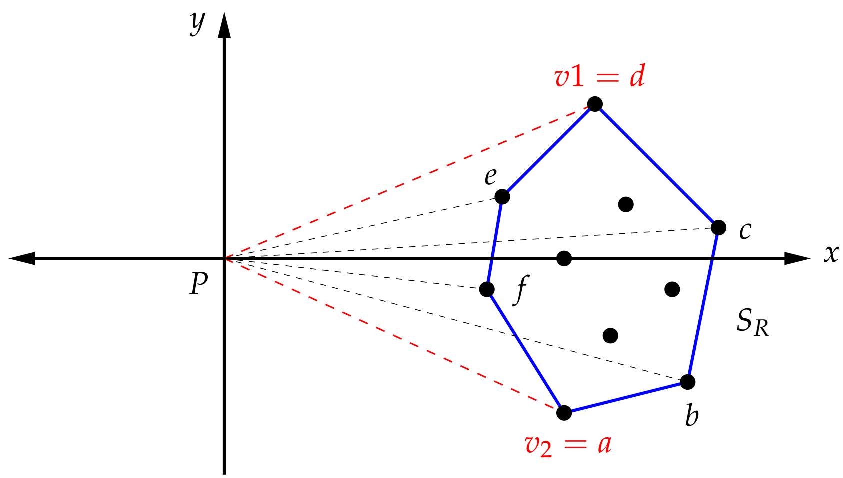
Convex hull problem (cont'd)

How to find v_1 and v_2?

- Construct a horizontal line through P.
- If this line intersects $CH(S_R)$, then v_1 is the vertex of $CH(S_R)$ with the greatest polar angle $< \frac{\pi}{2}$ and v_2 is the vertex with the least polar angle $> \frac{3\pi}{2}$.
- If this line does not intersects $CH(S_R)$, then v_1 is the vertex of $CH(S_R)$ with the largest polar angle and v_2 is the vertex with the smallest polar angle.

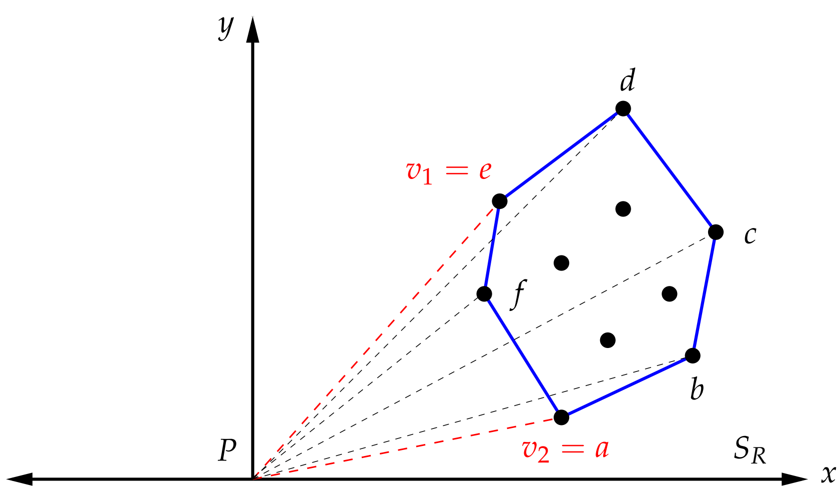
Convex hull problem (cont'd)

How to find v_1 and v_2?



Convex hull problem (cont'd)

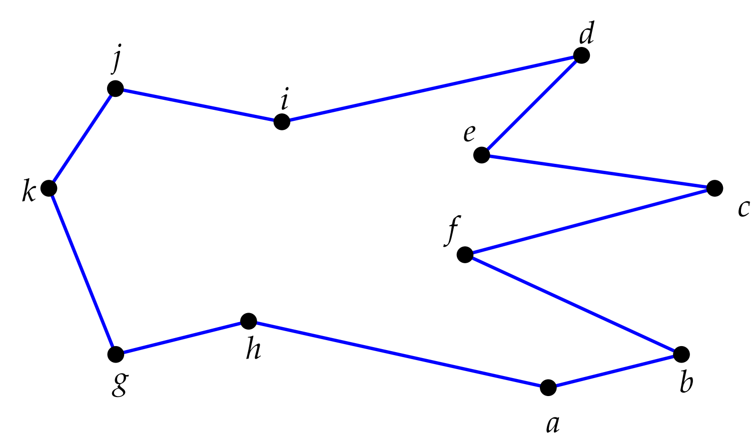
How to find v_1 and v_2?



Convex hull problem (cont'd)

Merging step in the divide and conquer algorithm

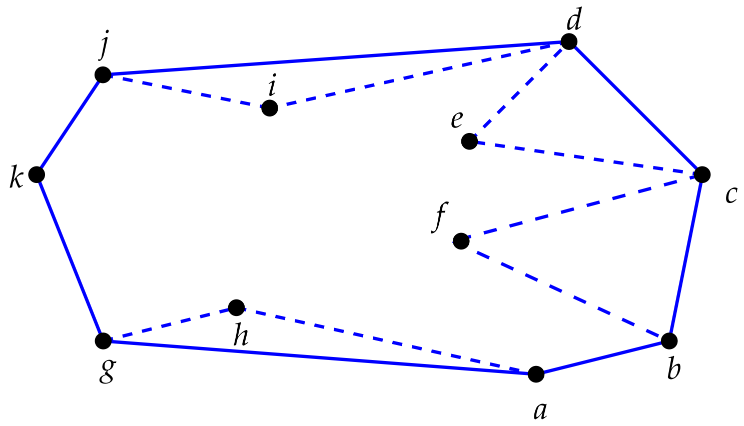
- Merge S_1, S_2, S_3 to $(g, h, a, b, f, c, e, d, i, j, k)$.



Convex hull problem (cont'd)

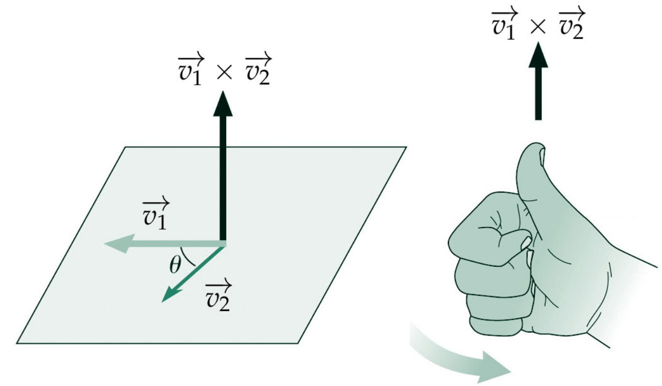
Merging step in the divide and conquer algorithm

- Graham scan examines the points of the new sequence one by one and eliminates those points h, f, e, i with reflexive angles.



Cross product

- Given two vectors $\overrightarrow{v_1} = (x_1, y_2)$ and $\overrightarrow{v_2} = (x_2, y_2)$, the cross product $\overrightarrow{v_1} \times \overrightarrow{v_2}$ is a vector, which is perpendicular to both $\overrightarrow{v_1}$ and $\overrightarrow{v_2}$, with a direction given by the right-hand rule and a magnitude equal to $|x_1y_2 - x_2y_1|$.

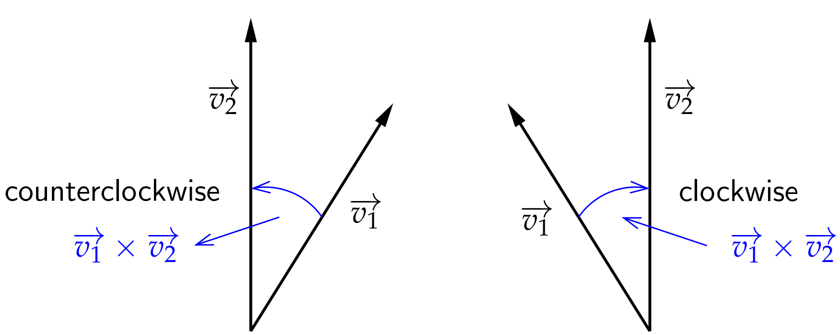


Cross product (cont'd)

- To determine the direction of $\overrightarrow{v_1} \times \overrightarrow{v_2}$ in a convenient way, we treat $\overrightarrow{v_1} \times \overrightarrow{v_2}$ simply as the value of $x_1y_2 - x_2y_1$.

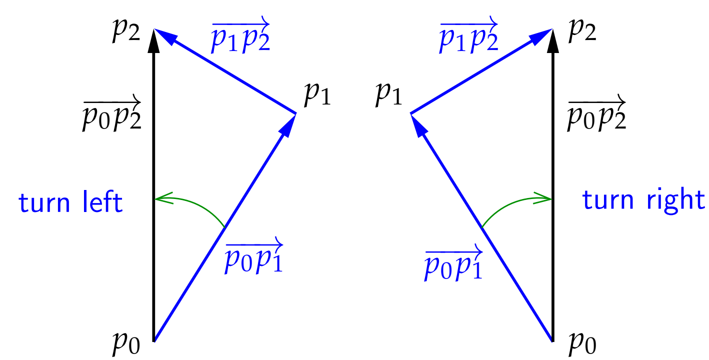
$$\overrightarrow{v_1} \times \overrightarrow{v_2} = x_1y_2 - x_2y_1$$

- If $\overrightarrow{v_1} \times \overrightarrow{v_2}$ is positive, $\overrightarrow{v_2}$ is counterclockwise from $\overrightarrow{v_1}$.
- If $\overrightarrow{v_1} \times \overrightarrow{v_2}$ is negative, $\overrightarrow{v_2}$ is clockwise from $\overrightarrow{v_1}$.



Turn left or right?

- We can use cross product to determine whether two consecutive line segments $\overline{p_0p_1}$ and $\overline{p_1p_2}$ turn left or right at point p_1.
- If the sign of $\overrightarrow{p_0p_1} \times \overrightarrow{p_0p_2}$ is positive, $\overrightarrow{p_0p_2}$ is counterclockwise from $\overrightarrow{p_0p_1}$ and hence we make a left turn at p_1.
- If the sign of $\overrightarrow{p_0p_1} \times \overrightarrow{p_0p_2}$ is negative, $\overrightarrow{p_0p_2}$ is clockwise from $\overrightarrow{p_0p_1}$ and hence we make a right turn at p_1.



Convex hull problem (cont'd)

Divide and conquer algorithm

Time complexity of divide and conquer algorithm:

$T(n) = 2T(\frac{n}{2}) + Split(n) + Merge(n)$

- $Split(n) = \mathcal{O}(n)$ for median finding process
- $Merge(n) = \mathcal{O}(n)$ for finding P, determination of v_1 and v_2, merging of S_1, S_2, S_3, and Graham scan

- In other words, we have $T(n) = 2T(\frac{n}{2}) + \mathcal{O}(n) = \mathcal{O}(n \log n)$.
- Note that it is optimal because the lower bound of the convex hull problem is $\Omega(n \log n)$.

Matrix multiplication

- Let $A = (a_{ij})$ and $B = (b_{ij})$ be two $n \times n$ matrices.
- Then the product matrix $C = A \cdot B$ is also a $n \times n$ matrix and its entry c_{ij} is computed by the following formula:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

- To obtain matrix C, the conventional method needs to compute n^2 matrix entries, each of which is the sum of n values.
- As a result, the time of this conventional method is $\Theta(n^3)$.

A simple divide and conquer algorithm

- For simplicity, we assume that n is a power of 2 (i.e., $n = 2^k$, where k is a nonnegative integer).
- If n is not a power of two, extra rows and columns of zeros can be added to both A and B so that the resulting dimensions are a power of two.
- Suppose that we partition each of A, B and C into four $\frac{n}{2} \times \frac{n}{2}$ matrices as follows:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

A simple divide and conquer algorithm (cont'd)

- Then we can rewrite the equation $C = A \cdot B$ as follows:

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

where

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}$$
$$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}$$
$$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}$$
$$C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}$$

A simple divide and conquer algorithm (cont'd)

- Hence, to compute $A \cdot B$, we need to perform 8 multiplications of $\frac{n}{2} \times \frac{n}{2}$ matrices and 4 additions of $\frac{n}{2} \times \frac{n}{2}$ matrices.
- Because two $\frac{n}{2} \times \frac{n}{2}$ matrices can be added in time $\mathcal{O}(n^2)$, the overall computing time $T(n)$ of this simple divide and conquer algorithm is as follows:

$$T(n) = \begin{cases} b & \text{if } n \leq 2 \\ 8T(\frac{n}{2}) + cn^2 & \text{if } n > 2 \end{cases}$$

  where b and c are constants.
- By the master theorem, we hvae $T(n) = O(n^3)$, which is not better than the time of the conventional method.

Strassen's matrix multiplication

- Observe that matrix multiplications are more expensive than matrix additions ($\mathcal{O}(n^3)$ versus $\mathcal{O}(n^2)$).
- If we can reformulate the equations of C_{ij} used in the simple divide and conquer algorithm such that fewer multiplications (and possibly more additions) are used, then the time of the simple divide and conquer can be improved.
- Finally, Strassen discovered a way to compute all the C_{ij} using only 7 muliplications and 18 additions or substractions.

Strassen's matrix multiplication (cont'd)

Step 1:

Divide the input matrices A and B and output matrix C into $\frac{n}{2} \times \frac{n}{2}$ submatrices:

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

- This step takes $\Theta(1)$ time by index calculation.

Strassen's matrix multiplication (cont'd)

Step 2:

Create the following 10 matrices $S_1, S_2, \ldots, S_{10}$, each of which is $\frac{n}{2} \times \frac{n}{2}$:

- $S_1 = B_{12} - B_{22}$
- $S_2 = A_{11} + A_{12}$
- $S_3 = A_{21} + A_{22}$
- $S_4 = B_{21} - B_{11}$
- $S_5 = A_{11} + A_{22}$
- $S_6 = B_{11} + B_{22}$
- $S_7 = A_{12} - A_{22}$
- $S_8 = B_{21} + B_{22}$
- $S_9 = A_{11} - A_{21}$
- $S_{10} = B_{11} + B_{12}$

- Since we have to add or substract $\frac{n}{2} \times \frac{n}{2}$ matrices 10 times, this step takes $\Theta(n^2)$ time.

Strassen's matrix multiplication (cont'd)

Step 3:

Recursively compute the following 7 matrix products $P_1, P_2, \ldots, P_7$, each of which is $\frac{n}{2} \times \frac{n}{2}$:

- $P_1 = A_{11} \cdot S_1 = A_{11} \cdot B_{12} - A_{11} \cdot B_{22}$
- $P_2 = S_2 \cdot B_{22} = A_{11} \cdot B_{22} + A_{12} \cdot B_{22}$
- $P_3 = S_3 \cdot B_{11} = A_{21} \cdot B_{11} + A_{22} \cdot B_{11}$
- $P_4 = A_{22} \cdot S_4 = A_{22} \cdot B_{21} - A_{22} \cdot B_{11}$
- $P_5 = S_5 \cdot S_6 = A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22}$
- $P_6 = S_7 \cdot S_8 = A_{12} \cdot B_{21} + A_{12} \cdot B_{22} - A_{22} \cdot B_{21} - A_{22} \cdot B_{22}$
- $P_7 = S_9 \cdot S_{10} = A_{11} \cdot B_{11} + A_{11} \cdot B_{12} - A_{21} \cdot B_{11} - A_{21} \cdot B_{12}$

Strassen's matrix multiplication (cont'd)

Step 4:

Compute $C_{11}, C_{12}, C_{21}, C_{22}$ by the following formulas:

- $C_{11} = P_5 + P_4 - P_2 + P_6$
- $C_{12} = P_1 + P_2$
- $C_{21} = P_3 + P_4$
- $C_{22} = P_5 + P_1 - P_3 - P_7$

- We add or substract $\frac{n}{2} \times \frac{n}{2}$ matrices 8 times in step 4 and hence this step takes $\Theta(n^2)$.

Strassen's matrix multiplication (cont'd)

Proof of $C_{11} = P_5 + P_4 - P_2 + P_6$:

$$
\begin{array}{lllllll}
A_{11}B_{11} & +A_{11}B_{22} & +A_{22}B_{11} & +A_{22}B_{22} & & & \\
 & & -A_{22}B_{11} & & +A_{22}B_{21} & & \\
 & -A_{11}B_{22} & & & & -A_{12}B_{22} & \\
 & & & -A_{22}B_{22} & -A_{22}B_{21} & +A_{12}B_{22} & +A_{12}B_{21} \\
\hline
A_{11}B_{11} & & & & & & +A_{12}B_{21}
\end{array}
$$

Strassen's matrix multiplication (cont'd)

Proof of $C_{12} = P_1 + P_2$:

$$
\begin{array}{lllll}
A_{11}B_{12} & - & A_{11}B_{22} & & \\
 & + & A_{11}B_{22} & + & A_{12}B_{22} \\
\hline
A_{11}B_{12} & & & & A_{12}B_{22}
\end{array}
$$

Strassen's matrix multiplication (cont'd)

Proof of $C_{21} = P_3 + P_4$:

$$\begin{array}{ccccc} A_{21}B_{11} & + & A_{22}B_{11} & & \\ & - & A_{22}B_{11} & + & A_{22}B_{21} \\ \hline A_{21}B_{11} & & & & A_{22}B_{21} \end{array}$$

Strassen's matrix multiplication (cont'd)

Proof of $C_{22} = P_5 + P_1 - P_3 - P_7$:

$$\begin{array}{cccccc} & A_{11}B_{11} + A_{11}B_{22} + A_{22}B_{11} + A_{22}B_{22} & & & & \\ & -A_{11}B_{22} & & +A_{11}B_{12} & & \\ & -A_{22}B_{11} & & & -A_{21}B_{11} & \\ -A_{11}B_{11} & & & -A_{11}B_{12} + A_{21}B_{11} + A_{21}B_{12} & & \\ \hline & +A_{22}B_{22} & & & & +A_{21}B_{12} \end{array}$$

Strassen's matrix multiplication (cont'd)

- When $n > 1$, steps 1, 2 and 4 take a total of $\Theta(n^2)$ time and step 3 requires to perform 7 multiplication of $\frac{n}{2} \times \frac{n}{2}$ matrices.
- Hence, the running time $T(n)$ of Strassen's algorithm can be expressed as follows:

$$T(n) = \begin{cases} b & \text{if } n \leq 2 \\ 7T(\frac{n}{2}) + cn^2 & \text{if } n > 2 \end{cases}$$

  where b and c are constants.
- By the master theorem, we have:

$$T(n) = \mathcal{O}(n^{\log_2 7}) \approx O(n^{2.81})$$