Heap Trees (Priority Queues) – Min and Max Heaps, Operations and Applications
Graphs – Terminology, Representations, Basic Search and Traversals, Connected Components and Biconnected Components, applications
Divide and Conquer: The General Method, Quick Sort, Merge Sort, Strassen's matrix multiplication, Convex Hull

DIVIDE AND CONQUER

General Method:

- Given a function to compute on *'n'* inputs, the divide and conquer strategy
  - Splits the input into k subsets, $1<k\leq n$, it yields k subproblems.
  - These subproblems must be solved.
  - A method must be found to combine subsolutions into a solution of the whole.
- The Divide and Conquer strategy is reapplied, if the subproblems are large.
- Often these subproblems are of the same type of the original problem. For this reason the divide-and-conquer principle is expressed by a *recursive algorithm*.
- Splitting the problem into subproblems is continued until the subproblems become small enough to be solved without splitting.

Control Abstraction:

- Control abstraction is a procedure whose flow of control is clear but whose primary operations are specified by other procedures whose precise meanings are left undefined.
- Control abstraction that mirrors the way an algorithm is based on divide-and-conquer is shown below.

```
Algorithm DAndC(P)
{
   if Small(P) then return S(P);
   else
   {
        divide P into smaller instances P1, P2, . . Pk   k>1
        Apply DAndC to each of these problems;
        return combine(DAndC(P1), DAndC(P2) . . . DAndC(Pk));
   }
}
```

- Initially algorithm is invoked as *DAndC(P),* where P is the problem to be solved.
- *Small(P)* is a boolean-valued function, it determines whether the input size is small enough that the answer can be computed without splitting.
- *S(P)* is invoked if *Small(P)* returns *true*.
- If *Small(P)* returns false, then the problem P is divided into subproblems $P_1, P_2, \ldots P_k$. These subproblems are solved by recursive application of *DAndC*.
- Combine function combines the solutions of the k subproblems to determine the solution to problem P.
- If the size of P is n, and the sizes of k subproblmes are $n_1, n_2, \ldots n_k$, then computing time of the *DAndC* is described by the recurrence relation.

$$T(n) = \begin{cases} g(n) & \text{n small} \\ T(n_1) + T(n_2) + \ldots\ldots + T(n_k) + f(n) & \text{otherwise} \end{cases} \qquad ----(1)$$

Where

T(n) – time for DAndC on any input of size n.
g(n) – time to compute answer directly for small inputs.
f(n) – time for dividing P into subproblems and combing solutions of subproblems.

DAndC strategy produces subproblems of type original problem, Therefore it is convenient to write these algorithms using recursion.

The complexity of many *divide-and-conquer* algorithms is given by recurrences of the form

$$T(n) = \begin{cases} T(1) & n = 1 \\ a.T\left(\frac{n}{b}\right) + f(n) & n > 1 \end{cases} \qquad ----(2)$$

Where a and b are known as constants.

To solve this recurrence relation, we assume that *T(1)* is known, n is a power of b (i.e., $n = b^k, \quad \log_b n = k$)
Substitution method of solving Recurrence Relation repeatedly makes substitution for each occurrence of the function in right hand side until all such occurrences disappear.

Example: consider the case in which *a=2* and *b=2*, *T(1)=2* and *f(n)=n*.

$$T(n) = \begin{cases} T(1) & n = 1 \\ a.T\left(\frac{n}{b}\right) + f(n) & n > 1 \end{cases}$$

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

$$= 2\left[2.T\left(\frac{n}{4}\right) + \frac{n}{2}\right] + n$$

$$= 4.T\left(\frac{n}{4}\right) + 2.n$$

$$= 4\left[2.T\left(\frac{n}{8}\right) + \frac{n}{4}\right] + 2n$$

$$= 8.T\left(\frac{n}{8}\right) + 3.n$$

$$\vdots$$

In general $$= 2^i.T\left(\frac{n}{2^i}\right) + i.n \qquad \text{for any } \log_2 n \ge i \ge 1$$

In particular $$T(n) = 2^{\log_2 n}.T\left(\frac{n}{2^{\log_2 n}}\right) + n.\log_2 n$$

$$T(n) = n.T(1) + n.\log_2 n$$
$$T(n) = n.\log_2 n + 2n$$

Solving recurrence Relation (2) using substitution method, we get

$$T(n) = a^{\log_b n}.T(1) + \log_b n.f(n)$$
$$= n^{\log_b a}.T(1) + \log_b n.f(n)$$
$$= n^{\log_b a}\left[T(1) + u(n)\right]$$

$$\textit{where } u(n) = \sum_{j=1}^{k} h(b^j) \quad \text{and } h(n) = \frac{f(n)}{n^{\log_b a}}$$

Merge Sort:

- Merge Sort is a sorting algorithm with the nice property that its worst case complexity is O(n log n).
- Given a sequence of *'n'* elements *a[1],…,a[n]* the general idea is to imagine them split into two sets $a[1],.....,a[\left\lfloor \frac{n}{2} \right\rfloor]$ and $a[\left\lfloor \frac{n}{2} \right\rfloor+1],....a[n]$.
- Each set is individually sorted, and the resulting sorted sequences are merged to produce a single sorted sequence of 'n' elements.

- It is an ideal example of divide-and-conquer strategy, in which
  - Splitting is into two equal sized sets
  - Combining operation is merging of two sorted sets into one.
- *MergeSort* algorithm describes this process using recursion and *Merge* algorithm merges two sorted sets.
- *'n'* elements should be placed in *a[1:n]* before executing *MergeSort*. Then *MergeSort(1,n)* causes the keys to be rearranged into nondecreasing order in *a*.

Algorithm for MergeSort:

```
Algorithm MergeSort(low, high)
//a[low:high] is a global array to be sorted. Small(P) is true if there is only one element
{
   if (low < high ) then   //if there are more than one element
   {
       mid := ⌊(low+high)/2⌋;         //Divide P into sub probelms
       //Solve sub problems
       MergeSort(low, mid);
       MergeSort(mid+1, high);
       Merge(low, mid, high);
   }
}
```

Algorithm for merging two sorted subarrays.

```
Algorithm Merge(low, mid, high)
//a[low:high] is a global array containing two sorted subsets.The goal is to merge these two
//sets into a single set. b[] is an auxiliary array.
{
   h := low; i := low; j := mid+1;
   while ((h≤mid) and (j≤high)) do
   {
       if (a[h]≤a[j]) then
       {
           b[i] := a[h]; h := h+1;
       }
       else
       {
           b[i] := a[j];  j := j+1;
       }
       i := i+1;
```

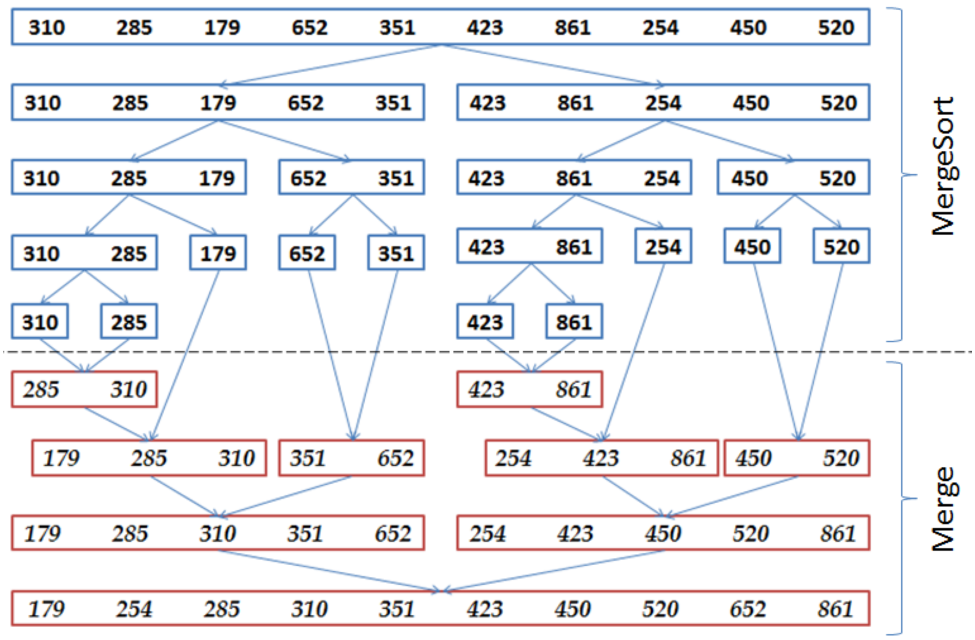
```
    }
    if (h>mid) then
       for k := j to high do
       {
          b[i]:=a[k]; i:=i+1;
       }
   else
      for k:=h to mid do
      {
         b[i]:=a[k]; i:=i+1;
      }
   for k:=low to high do
      a[k]:=b[k];
}
```

Example :

Consider an array of 10 elements.

$$a[1:10] = (310, 285, 179, 652, 351, 423, 861, 254, 450, 520)$$

Algorithm MergeSort begins by splitting *a[]* until they become one-element subarrays. Now merging begins. This division and merging is shown in the below figure.



- Following figure is a tree that represents the sequence of recursive calls that are produced by **MergeSort** when it is applied to *10* elements.
- The pair of values in each node is the values of the parameters *low* and *high*.

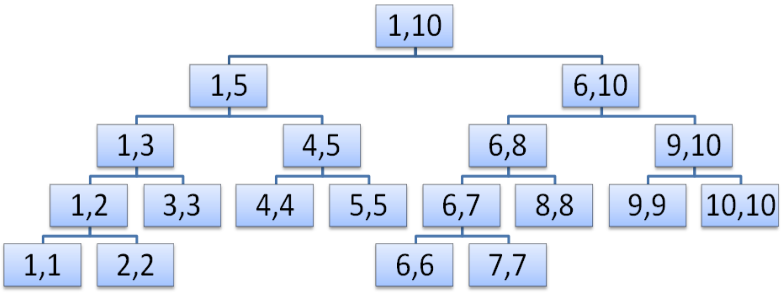


Figure: Tree of calls of MergeSort(1,10)

- Following figure is a tree representing the calls to procedure Merge. For example, the node containing 1, 2, and 3 represents the merging of a[1:2] with a[3].

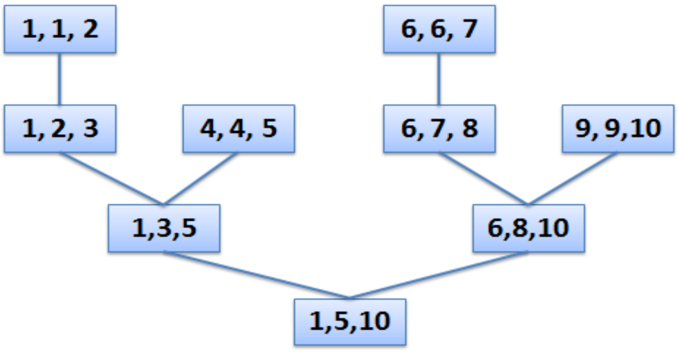


Figure: Tree of calls of Merge

If the time for the merging operation is proportional to *n,*

If the time for the merging operation is proportional to n, then computing time for merge sort is described by the recurrence relation.

$$T(n) = \begin{cases} a & n = 1 \\ a.T\left(\frac{n}{2}\right) + c.n & n > 1 \end{cases}$$

Where *a* and *c* are constants.

We can solve this equation by successive substitutions.
Assume n is a power of *2*, $n=2^k$, i.e., $\log n = k$.

$$
\begin{aligned}
T(n) &= 2.T\left(\frac{n}{2}\right) + c.n \\
&= 2\left[2.T\left(\frac{n}{4}\right) + c.\frac{n}{2}\right] + cn. \\
&= 2^2.T\left(\frac{n}{4}\right) + 2cn \\
&= 2^2\left[2.T\left(\frac{n}{8}\right) + c.\frac{n}{4}\right] + 2cn \\
&= 2^3.T\left(\frac{n}{8}\right) + 3cn \\
&\text{after k substitutions} \\
&= 2^k.T(1) + kcn \\
&= an + cn.\log n
\end{aligned}
$$

It is easy to see that $if\ 2^k < n \le 2^{k+1}$, then $T(n) \le T(2^{k+1})$

$$\therefore T(n) = O(n \log n)$$

Quick Sort:

- In Quick sort, the division into two subarrays is made in such a way that, the sorted subarrays do not need to be merged later.
- This is achieved by rearranging the elements in $a[1:n]$ such that $a[i] \le a[j]$ for all i between 1 and m and all j between $m+1$ and n for some m, $1 \le m \le n$.
- Thus, the elements in $a[1:m]$ and $a[m+1:n]$ can be independently sorted. No merge is needed.
- The rearrangement of the elements is accomplished
  - By picking some element of *a[]*, say $t=a[s]$.
  - And then reordering the elements so that all elements appearing before t in $a[1:n]$ are less than or equal to t and all elements appearing after t are greater than or equal to t.
  - The rearranging is referred to *<u>partitioning</u>*.

Following algorithm accomplishes partitioning of elements of a[m:p]. It is assumed that a[m] is the partitioning element.

```
Algorithm Partition(a, low, high)
{
   pivot:=a[low], i:=low+1, j:=high;
   while (i<j) do
   {
        while (a[i]≤pivot and i≤j) do
                 i:=i+1;
        while (a[j]≥pivot and j≥i) do
                 j:=j-1;
        if (i<j) then
                 interchange(a, i, j);
   }
```

```
    interchange(a, low, j);
    return j;
}
```

The function interchange(a,i,j) exchanges a[i] with a[j].

```
Algorithm interchange(a, i, j)
{
    temp:=a[i];
    a[i]:=a[j];
    a[j]:=temp;
}
```

Example: working of partition.
Consider the following array of 9 elements.
The partition function is initially invoked as **Partition**(*a, 1, 9*).
The element a[1] i.e., 65 is the partitioning element

a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	i	j	
65	70	75	80	85	60	55	50	45	2	9	i<j, swap a[i],a[j]
65	45	75	80	85	60	55	50	70	3	8	i<j, swap a[i],a[j]
65	45	50	80	85	60	55	75	70	4	7	
65	45	50	55	85	60	80	75	70	5	6	
65	45	50	55	60	85	80	75	70	6	5	i>j, swap pivot with a[j]
60	45	50	55	**65**	85	80	75	70			Now elements are partitioned about pivot i.e., 65

Now the elements are partitioned about pivot element and the remaining elements are unsorted.

- Using this method of partitioning, we can directly devise a divide and conquer method for completely sorting *n* elements.
- Two sets S1 and S2 are produced after calling partition. Each set can be sorted independently by reusing the function partition.

Following algorithm describes the complete process.

```
Algorithm QuickSort(low, high)
{
    if (low<high) then          //if there are more than one element.
    {
        j:=partition(a,low,high);       //Partitioning into subproblems
        // Solve the subproblems
        QuickSort(a, low, j-1);
        QuickSort(a, j+1,high);
    }
}
```

Analysis:

- In quicksort, the pivot element we chose divides the array into 2 parts.
    - One of size *k*.
    - Other of size *n-k*.
- Both these parts still need to be sorted.
- This gives us the following relation.

$$T(n) = T(k)+T(n-k)+c.n$$

Where T(n) refers to the time taken by the algorithm to sort n elements.

Worst-case Analysis:

Worst case happens when pivot is the least element in the array.

Then we have *k=1* and *n-k=n-1*

$$\Rightarrow T(n) = T(1) + T(n-1) + c.n$$
$$= T(1) + [\ T(1) + T(n-2) + c.(n-1)\ ] + c.n$$
$$= T(n-2)+2.T(1)+c.(n-1+n)$$
$$= [\ T(1) + T(n-3) + c.(n-2)\ ] + 2.T(1)+c.(n-1+n)$$
$$= T(n-3)+3.T(1)+c.(n-2+n-1+n)$$

:

Continuing likewise till ith step

$$= T(n-i)+i.T(1)+c.(n-i-1+\ ..\ +n-2+n-1+n$$
$$= T(n-i) + i.T(1) + c.\sum_{j=0}^{i-1}(n-j)$$

This recurrence can go until i=n-1. Substitute i=n-1

$$T(n) = T(1) + (n-1).T(1) + c.\sum_{j=0}^{n-2}(n-j)$$
$$= n.T(1) + c.\sum_{j=0}^{n-2}(n-j)$$
$$= O(n^2)$$

Best-case Analysis:

Best case of Quick Sort occurs when pivot we pick divide the array into two equal parts, in every step.

$$\therefore k = \frac{n}{2},\quad n-k = \frac{n}{2},\quad \text{for array of size n}$$

$$\text{We have } T(n) = T(k) + T(n-k) + c.n$$
$$= 2.T\left(\frac{n}{2}\right) + c.n$$

Solving this gives ***O(n log n).***

Randomized Quick Sort:

- Algorithm Quick Sort has an average time of *O(n log n)* and worst case of $O(n^2)$ on *'n'* elements.
- It does not make use of any additional memory like Merge Sort.
- Quick Sort can be modified by using randomizer, so that its performance will be improved.
- While sorting the array *a[p:q],* pick a random element (from *a[p]...a[q])* as the partition element.
- The randomized algorithm works on any input and runs in an expected *O(n log n)* time, where the expectation is over the space of all possible outcomes of the randomizer.

- The code of randomized quick sort is given below. It is a *Las Vegas algorithm* since it always outputs the correct answer.

```
Algorithm RQuickSort(p, q)
{
    if (p<q) then
    {
        if ((q-p)>5) then
                interchange(a, Random() mod (q-p+1)+p,p);
        j := partition(a, p, q+1);
        RQuickSort(p, j-1);
        RQuickSort(j+1, q);
    }
}
```

- Reason for invoking randomizer only if *(q – p)>5* is
  - Every call to randomizer Random takes a certain amount of time.
  - If there are only a few elements to sort, the time taken by the randomizer may be comparable to the rest of the computation.

Strassen's Matrix Multiplication

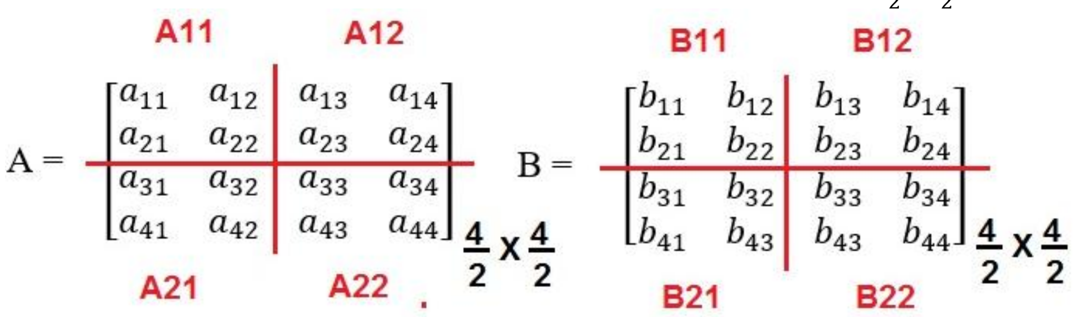
- Let A and B two $n \times n$ matrices.
- The product $C = AB$ is also a $n \times n$ matrix, where

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

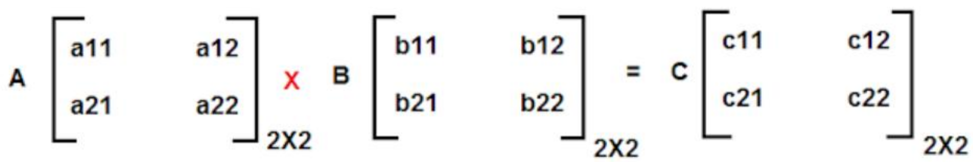
- Here we need n multiplications to compute C_{ij}.
- To compute all n^2 elements of C, we require n^3 multiplications, therefore, Time Complexity of conventional method is $\theta(n^3)$.

<u>Divide and Conquer Strategy</u>

- Assume that n is a power of 2, i.e., $n = 2^k$.
- In case n is not a power of 2, then enough rows and columns of zeros can be added to both A and B so that the resulting dimensions are a power of two.
- Divide A and B into four submatrices, each submatrix having dimensions $\frac{n}{2} \times \frac{n}{2}$.



If $n = 2$, the matrix product is computed as



Where,

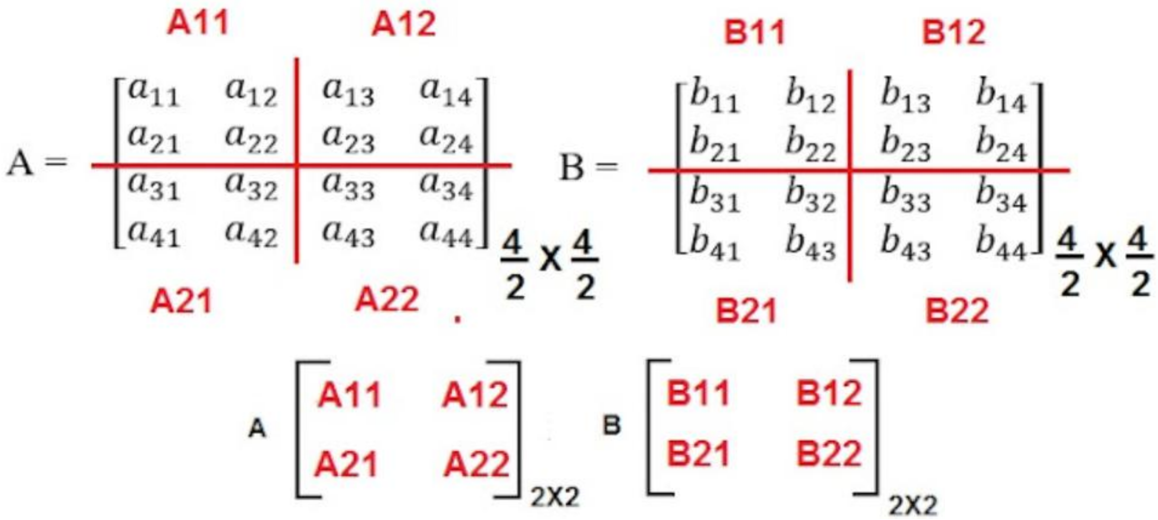
C11 = a11 * b11 + a12 * b21

C12 = a11 * b12 + a12 * b22

C21 = a21 * b11 + a22 * b22

C12 = a21 * b12 + a22 * b22

- If $n > 2$, the elements of C can be computed using matrix multiplication and addition operations applied to matrices of size $\frac{n}{2} \times \frac{n}{2}$.
- This algorithm will continue applying itself to smaller-sized submatrices until n becomes small ($n = 2$) so that the product is computed directly.
- To compute AB, 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.



The overall computing time is ,

$$T(n) = \begin{cases} b & n \leq 2 \\ 8\,T\left(\frac{n}{2}\right) + n^2 & n > 2 \end{cases}$$

- By solving this, we get $O(n^3)$. There is no improvement over the conventional method.
- Even after applying divide and conquer, the time complexity remains the same. To improve the algorithm, reformulate the equations for C_{ij} with fewer multiplications.
- Strassen discovered a way to compute C_{ij} with 7 multiplications and 18 additions or subtractions.
- The formulas given by Strassen are

$$P = (A_{11}+A_{22}) * (B_{11}+B_{22})$$

$$Q = (A_{21}+A_{22}) * B_{11}$$

$$R = A_{11} * (B_{12} - B_{22})$$

$$S = A_{22} * (B_{21} - B_{11})$$

$$T = (A_{11}+A_{12}) * B_{22}$$

$$U = (A_{21} - A_{11}) * (B_{11} + B_{12})$$

$$V = (A_{12} - A_{22}) * (B_{21} + B_{22})$$

The C matrix is computed by using the above 7 matrices

$$C_{11} = P+S-T+V$$

$$C_{12} = R+T$$

$$C_{21} = Q+S$$

$$C_{22} = P+R-Q+U$$

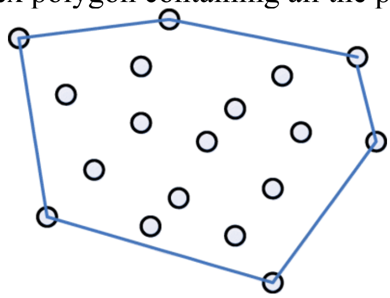
Now the time complexity is

$$T(n) = \begin{cases} b & n \le 2 \\ 7\,T\left(\frac{n}{2}\right) + n^2 & n > 2 \end{cases}$$

i., $T(n) = O\left(n^{\log_2 7}\right) = O(n^{2.81})$

Convex Hull

- A convex hull is an important structure in geometry that can be used in the construction of many other geometric structures.
- A convex hull is a polygon that encloses all of the points. The vertices maximize the area, while minimizing the circumference. In other words, The convex hull of a set S of points in the plane is defined to be the smallest convex polygon containing all the points of S.



Geometric primitives

- Let (p_1, p_2) is a line segment from $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$.
- $q = (x_3, y_3)$ is another point.
- If the angle $p_1 p_2 q$ is a left turn, then we say that q is to the left of (p_1, p_2). An angle is said to be a left turn if it is less than or equal to 180^o.
- To check whether q is to the left of (p_1, p_2), evaluate the determinant of the following matrix. If the det is positive, then q is on the left side. If q=0, then the three points are colinear.

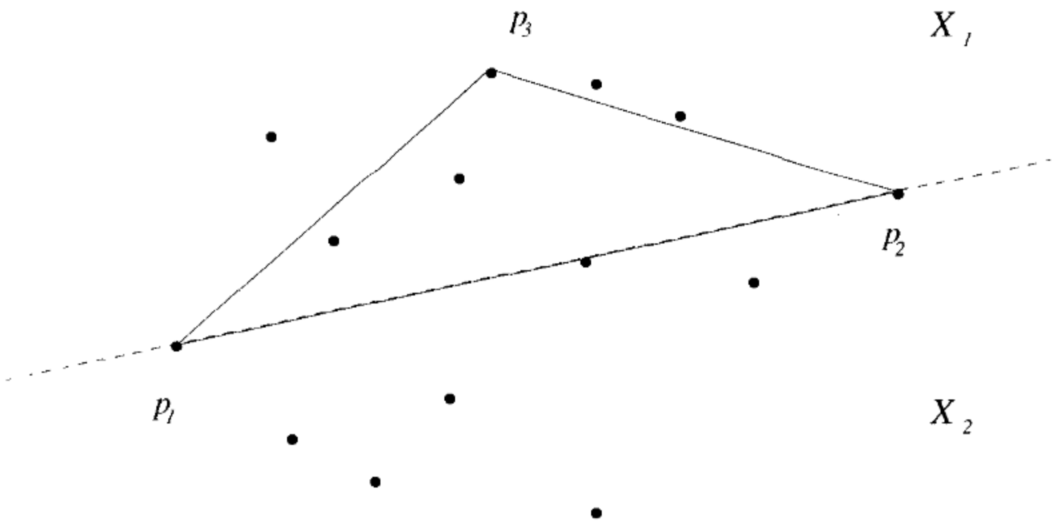
$$\begin{bmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ 1 & 1 & 1 \end{bmatrix}$$

The QuickHull algorithm

- This algorithm is used to compute the convex hull of a set X of n inputs in the plane.

- The algorithm first identifies the two points (p1 and p2) of X with the smallest and largest x-coordinate values. Both p1 and p2 are extreme points and part of the convex hull.
- The set X is divided into X_1 and X_2 so that X_1 has all the points to the left of the line segment $\langle p_1, p_2 \rangle$ and X_2 has all the points to the right of the line segment $\langle p_1, p_2 \rangle$. Both X_1 and X_2 include the two points $p_1$and p_2.

- Then the convex hulls of X_1 and X_2 (called the upper hull and lower hull) are computed using divide and conquer algorithm.
- The union of these two hulls is the overall convex hull.
- To compute the convex hull of X_1,
  - Determine a point of X_1 that belongs to the convex hull of X_1 and use it to partition the problem into two independent subproblems. The point is obtained by computing the area formed by $p_1, p \text{ and } p_2$ for each p in $X_1$and picking the one with the largest area. Let p3 be the point.



- Now X_1 is divided into two parts; the first part contains all the points of X_1 that are to the left of $< p_1, p_3 >$ (including p_1 and p_3) and the second part contains all the points of X_1 that are to the left of $< p_3, p_2 >$ (including p_3 and p_2). The remaining points are interior points and can be dropped from consideration.
- The convex hull of each part is computed recursively, and the two convex hulls are merged easily by placing one next to the other in the right order.