Divide and Conquer

Binary Search, MergeSort, Max_Subarray, Long Integer Multiplication, Matrix Multiplication, Convex Hull, Recurrence Relations and Master theorem

What is Divide and Conquer?

- The divide-and-conquer strategy solves a problem by:
  1. Breaking (Divide) it into subproblems that are themselves smaller instances of the same type of problem
  2. Recursively solving (Conquer) these subproblems
  3. Appropriately combining (Combine) the subproblem solutions to a solution of the original problem.
- The real work is done piecemeal, in three different places: in the partitioning of problems into subproblems; at the very tail end of the recursion, when the subproblems are so small that they are solved outright; and in the gluing together of partial answers.
- A very easy problem to illustrate the paradigm: **Binary Search**, Search for a key (x) in a sorted array A[1…n].
  - **Algorithm:** Compare x with A[$\lceil n/2 \rceil$; if x = A[$\lceil n/2 \rceil$] then done; if x < A[$\lceil n/2 \rceil$] then search A[1..$\lceil n/2 \rceil$-1]; if x > A[$\lceil n/2 \rceil$] then search A[$\lceil n/2 \rceil$+1, … n];
  - **Analysis:** Let T(n) denotes the execution time (# of elem comparison), in the worst case we have the recurrence $T(n) = T(n/2) + \Theta(1)$ for $n > 1$ and $T(n) = \Theta(1)$ for $n = 1$. By simple telescoping, $T(n) = \Theta(\log_2 n)$.

Recurrence Solution

Recurrence Relation

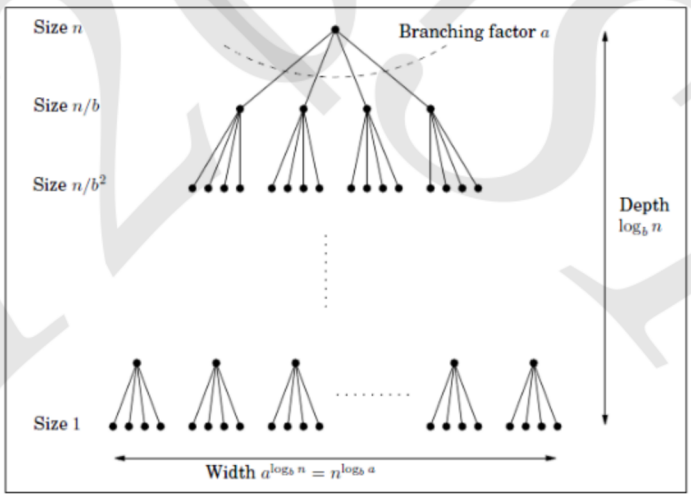
Master Theorem of Recurrence Relations

- Divide-and-conquer algorithms often follow a generic pattern: they tackle a problem of size n by recursively solving, say, a subproblems of size n/b and then combining these answers in $O(n^d)$ time, for some constants, $a > 0$, $b > 1$, $d \geq 0$ (in binary search a = 1, b = 2, d = 0). Their running time can therefore be captured by the equation $T(n) = aT(\lceil n/b \rceil) + O(n^d)$. **Master theorem** gives a closed form solution to this general recurrence so that we do not need to solve it in each new instance.

$$\text{If } T(n) = aT(\lceil n/b \rceil) + O(n^d), \text{ for some constants } a > 0,\ b > 1,\ d \geq 0,$$
$$\text{then } T(n) = \begin{cases} O(n^d), & \text{if } d > \log_b a \\ O(n^d \log n), & \text{if } d = \log_b a \\ O(n^{\log_b a}), & \text{if } d < \log_b a \end{cases}$$

- **Proof:** Let us assume n is some power of b – this is for convenience only, does not change the result since n is at most a multiplicative factor of b away from some power of b. Observe that the size of the subproblems decreases by a factor of b with each level of recursion, and therefore reaches the base case after $\log_b n$ levels. Thus, the height of the recursion tree is $\log_b n$. Its branching factor is a, so the kth level of the tree is made up of a^k subproblems, each of size n/b^k.

Master Theorem of Recurrence Relations



Total work done at the k^{th} level is # of nodes multiplied by work at each node. When k = 0 (root), the work is n^d.

$$a^k \times O\left(\frac{n}{b^k}\right)^d = O(n^d) \times \left(\frac{a}{b^d}\right)^k \leftarrow \text{Work at level k}$$

Master Theorem of Recurrence Relations

- As k goes from 0 (the root) to $\log_b n$ (the leaves), these numbers form a geometric series with ratio a/b^d. Finding the sum of such a series in big-O notation is easy and comes down to three cases.
  - The ratio is less than 1 ⇒ the series is decreasing, and its sum is just given by its first term, $O(n^d)$, in order-algebra.
  - The ratio is exactly 1 ⇒ all O(log n) terms of the series are equal to $O(n^d)$.
  - The ratio is greater than 1 ⇒ the series is increasing and its sum is given by its O(last term) which is

$$n^d\left(\frac{a}{b^d}\right)^{\log_b a} = n^d\left(\frac{a^{\log_b n}}{\left(b^{\log_b n}\right)^d}\right) = a^{\log_b n} = a^{(\log_a n)(\log_b n)} = n^{\log_b a}$$

  - These cases translate directly into the three contingencies in the theorem statement.

Consider a finite GP series, y, yr, yr^2, … yr^{m-1}, where y is the first term, r = common ratio, m = the number of terms. Its sum is S = my when r = 1, and $\frac{y - yr^m}{y - r}$ otherwise.

Long Integer Multiplication

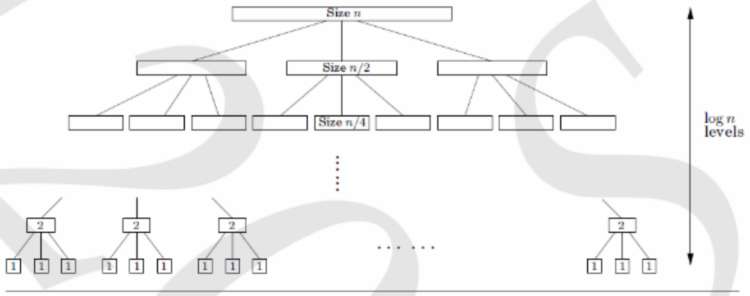
- Suppose x and y are two n-bit integers, and assume for convenience that n is a power of 2 (the more general case is hardly any different). As a first step toward multiplying x and y, split each of them into their left and right halves, which are n/2 bits long:

$$x = \boxed{x_L}\boxed{x_R} = 2^{n/2}x_L + x_R$$
$$y = \boxed{y_L}\boxed{y_R} = 2^{n/2}y_L + y_R.$$

- For instance, if $x = 10110110_2$ (the subscript 2 means binary) then $x_L = 1011_2$, $x_R = 0110_2$, and $x = 1011_2 \times 2^4 + 0110_2$. The product of x and y can then be rewritten as $xy = (2^{n/2}x_L + x_R)(2^{n/2}y_L + y_R) = 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R$
- We compute xy via the expression on the right. The additions take linear time, as do the multiplications by powers of 2 (which are merely left-shifts). The significant operations are the four n/2-bit multiplications, $x_L y_L$; $x_L y_R$; $x_R y_L$; $x_R y_R$; these we can handle by four recursive calls. Thus our method for multiplying n-bit numbers starts by making recursive calls to multiply these four pairs of n/2-bit numbers (four subproblems of half the size), and then evaluates the preceding expression in O(n) time. Writing T(n) for the overall running time on n-bit inputs, we get the *recurrence relation* $T(n) = 4T(n/2) + O(n)$ which, by Master's theorem has a solution $O(n^2)$ [Check!!]
- Now, consider the identity for 4 real numbers a, b, c, d, $bc + ad = (a+b)(c+d) - ac - bd$. Can this help to reduce multiplication time?? [The mathematician Carl Friedrich Gauss (1777 – 1855) noticed that although the product of two complex numbers $(a + bi)(c + di) = ac - bd + (bc + ad)i$ seems to involve four real-number multiplications, it can in fact be done with just three: ac, bd, and $(a + b)(c + d)$]; that observation helps us here too!

Long Integer Multiplication

- Although the expression for xy seems to demand four n/2-bit multiplications, as before just three will do: $x_L y_L$, $x_R y_R$ and $(x_L+x_R)(y_L+y_R)$. The resulting algorithm has an improved running time of T(n) = 3T(n/2) +O(n). We apply Master's theorem to obtain the solution as $O(n^{\log_2 3})$ or $O(n^{1.59})$ instead of $O(n^2)$.
- In divide-and-conquer algorithms, the number of sub problems translates into the branching factor of the recursion tree; small changes in this coefficient can have a big impact on running time. Write the pseudocode and implement.
- Can we do better? YES, even faster algorithms for multiplying numbers exist, based on another important divide-and-conquer algorithm, the Fast Fourier Transform.



MergeSort

- The problem of sorting a list of numbers lends itself immediately to a divide-and-conquer strategy: split the list into two halves, recursively sort each half, and then merge the two sorted sublists.

  function mergesort (a[1 … n])

  Input: An array of numbers a[1 … n]

  Output: A sorted version of this array

  if n > 1:

  return merge(mergesort(a[1 … ⌊n/2⌋]), mergesort(a[⌊n/2⌋ + 1 … n]))

  else: return a
- If we are given two sorted arrays x[1 … k_1] and y[1 … k_2], how do we efficiently merge them into a single sorted array z[1 … $k_1 + k_2$]? Well, the very first element of z is either x[1] or y[1], whichever is smaller. The rest of z [.] can then be constructed recursively.

function merge (x[1…k_1], y[1…k_2])
if $k_1 = 0$ then return y[1…k_2];
if $k_2 = 0$ then return x[1…k_1];
if x[1] ≤ y[1] then
  return x[1] ° merge (x[2…k_1], y[1…k_2])
else
  return y[1] ° merge (x[1…k_1], y[2…k_2])

- Here ° denotes concatenation. This merge procedure does a constant amount of work per recursive call (provided the required array space is allocated in advance), for a total running time of $O(k_1 + k_2)$. Thus merge's are linear, and the overall time taken by mergesort is T(n) = 2T(n/2) + O(n) or O(n log n)
- Actually, it's more efficient to write an iterative program for merge and mergesort, as is the case more often.

Maximum Sum SubArray Problem

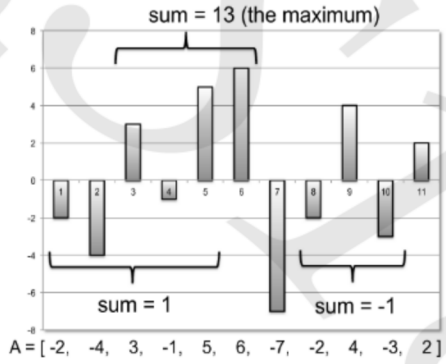
- **Problem Statement**: Given an array A = [a_1, a_2, ..., a_n] of n integers, positive and negative, compute the subarray whose elements have the largest sum, i.e., find indices j and k that maximize the sum $\sum_{i=j \text{ to } k} a_i$. **Note**: when all inputs are negative the maximum sum subset is the empty set, which has sum zero.
- Stock Market: You can buy a unit of stock, only *one* time, then sell it at a later date (e.g., Buy/sell at end of day). Strategy: buy low, sell high, but the lowest price may appear after the highest price. Assume you know future prices (make your own forecast). Question: Can you maximize profit by buying at lowest price and selling at highest price?
  - Find sequence of days so that the net change from last to first is maximized
  - Look at the daily change in price: Change on day i: price day i minus price day i-1
  - We now have an array of changes (numbers), e.g. 12,-3,-24,20,-3,-16,-23,18,20,-7,12,-5,-22,14,-4,6
  - Find contiguous subarray with largest sum maximum subarray e.g.: buy after day 7, sell after day 11.
- In practice, a bitmap image has all non-negative pixel values. When the average is subtracted from each pixel, we can apply the maximum subarray algorithm to find the brightest area within the image.

*Simple Solution**

- Easiest way to solve the problem is to systematically explore each possible subarray of all sizes, compute their sums and output the maximum.

Algorithm MaxSub1 (A):
Input: A[1…n]
Output: max, j, k
JJ ← 0, KK ← 0 JJ is starting point, KK is end point
for j ← 1 to n do
  max ← 0 the maximum found so far
  for k ← j to n do
    s ← 0 the next partial sum we need
    for i ← j to k do
      s ← s + A[i]
    if s > max then
      max ← s, JJ ← j, KK ← k
return max, JJ, KK

Note: This is another example of using the **accumulator** design pattern, where we incrementally accumulate values into a single variable to compute a sum or maximum (or minimum). This is a pattern that is used in a lot of algorithms.



It is easy to show that the algorithm is correct and the running time is $O(n^3)$. How to improve? Specifically, where are we making redundant or unnecessary computations?

* one of the most often asked questions in job interviews, used in pattern analysis in digitized images

Improvement

- Consider the Prefix_sum PS[1…n] of an integer array A[1…n] such that PS[i] = PS[i – 1] + A[i] assuming PS[0] = 0; it is easy to write a program to compute the array PS[1…n] in O(n) time. Also, observe that PS[k] – PS[j – 1], where $k \geq 1$, gives the sum of the subarray A[j … k]. We get an improved algorithm for the MaxSub problem.

Algorithm MaxSub2 (A):
Input: A[1…n]
Output: max, j, k

```
PS[0] ← 0  the initial prefix sum
for i ← 1 to n do
  PS[i] ← PS[i–1] + A[i]
for j ← 1 to n do
  max ← 0  the maximum found so far
  for k ← j to n do
    if PS[k] – PS[j–1] > max then
      max ← PS[k] – PS[j–1], JJ ← j, KK ← k
return max, JJ, KK
```

The correctness of the Maxsub2 algorithm follows along the same arguments as for the Maxsub1 algorithm, but it is much faster. In particular, the outer loop, for index j, will iterate n times, its inner loop, for index k, will iterate *at most n times*, and the steps inside that loop will only take O(1) time in each iteration.

Thus, the total running time of the Maxsub2 algorithm is $O(n^2)$, which improves the running time of the Maxsub1 algorithm by a linear factor.

Can we do an even better job? Consider:

- Why did prefix sum idea help?
- Can we use something else instead of prefix_sum?
- **Remember that if all A[i]'s are negative, the maxsubarray sum is 0 (empty subset).**

Divide and Conquer Solution

- We can use divide and conquer to solve the problem in O(n log n) time. To see how, we observe that the maximum subarray $A[k_1 \ldots k_2]$ of an array A[1 … n] can be either
  - contained entirely in the first half , i.e. $k_2 \leq n/2$
  - or contained entirely in the second half , i.e. $k_1 \geq n/2$
  - or overlaps both halves, $k_1 \leq n/2 \leq k_2$.
- We can compute the best subarray of the first two types with recursive calls on the left and right half.
- The best subarray of the third type consists of the best subarray that ends at n/2 and the best subarray that starts at n/2. We can compute these in O(n) time [two linear prefix scans starting from the midpoint – one to the left and one to the right and then combine the two]
- The recurrence for the entire algorithm is then $T(n) = 2T(n/2) + O(n)$ with a solution O(n log n).
- Can we do better? Yes, we can solve the problem in linear time (not using divide and conquer) using an interesting strategy.

Algorithm MaxSub3 (Linear)

```
Algorithm MaxSub3(A): [Kadane]
  Input: An array A[1…n] of numbers.
  Output: The subarray sum max s.t. A[j]
  + …+ A[k] is maximized.
M_0 ← 0 // the initial prefix maximum
for t ← 1 to n do
   M_t ← max{0, M_{t-1} + A[t]}
max ← 0 //maximum found so far
for t ← 1 to n do
   max ← max{max, M_t}
return max
```

The MaxSub3 algorithm consists of two loops, which each iterate exactly n times and take O(1) time in each iteration. Thus, the total running time of the MaxSub3 algorithm is O(n). It is also known as **Kadane's Algorithm**.

Exercise: Modify the description of the algorithm so that, in addition to the value of the maximum subarray summation, it also outputs the indices j and k that identify the maximum subarray A[j : k].

Solution to the Exercise

```
int new_2 (int ar[], int a_s, int *p, int *q){
int i, start_index=0; int tmp_sum = 0, end_index=0;
int temp_start_index = 0, maxsum = 0;
for (i = 0; i < a_s; i++){
   tmp_sum = tmp_sum + ar[i];
   if (tmp_sum > maxsum){
   maxsum = tmp_sum;
   end_index = i;
   start_index = temp_start_index;}
   if (tmp_sum < 0){tmp_sum = 0;
temp_start_index=i+1;} }
*p = start_index; *q = end_index;
return maxsum; }
```

If you look at this without first trying and thinking about how to solve the problem, you haven't learned much. One of the important idea behind writing efficient code is to think about the problem and its possible solution; when you learn something new, think about where and how you can use the approach albeit in a slightly different way.

Variation

- Given an arbitrary array A[1…n], compute the maximum difference between two elements such that the larger element appears after the smaller one in A[]. Example: for A = [2, 4, -1, 9, 8, 4, 10, 3, 1], the maximum difference is $8 = a_7 - a_3$. Assume there are at least 2 elements in the array.
- Simple Algorithm: Select elements one at a time in an outer loop and check with each element on the right side in an inner loop; time complexity = $O(n^2)$ and auxiliary space = O(1).
- Efficient Algorithm: Keep track of maximum difference found so far and minimum number visited so far; time complexity = O(n) and auxiliary space = O(1)

```
int maxDiff1 (int A[], int size)
{
  int max_diff = A[1] - A[0];
  int i, j;
  for(i = 0; i < size; i++)
  {
    for(j = i+1; j < size; j++)
    {
      if(A[j] - A[i] > max_diff)
        max_diff = A[j] - A[i];
    }   }
  return max_diff;
}
```

```
int maxDiff2 (int A[], int size)
{
  int max_diff = A[1] - A[0];
  int min_element = A[0];
  int i;
  for(i = 1; i < size; i++)
  {
    if (A[i] - min_element > max_diff)
      max_diff = A[i] - min_element;
    if (A[i] < min_element)
      min_element = A[i];
  }
  return max_diff;
}
```

Note: In maxDiff2 we can keep track of max elem from the right instead of minimum from the left. Try.

Maximum Subarray Problem (2D Array) Watch

- Given a two-dimensional array a[1..m, 1..n] as an input data set, the maximum subarray problem is to find a rectangular portion $a[r_1..r_2, c_1..c_2]$ such that the sum of contained elements should be greater than or equal to the sum of any other rectangular portions of the dataset.

$$a = \begin{bmatrix} -1 & 2 & -3 & 5 & -4 & -8 & 3 & -3 \\ 2 & -4 & -6 & -8 & 2 & -5 & 4 & 1 \\ 3 & -2 & 9 & -9 & 1 & 10 & 5 & 2 \\ 1 & -3 & 5 & -7 & 8 & -2 & 2 & -6 \end{bmatrix}$$

The maximum subarray is the array portion a[2..3, 4..5] surrounded by inner brackets, whose sum is 15.

1	2	-1	4	-20
-8	-3	4	2	1
3	8	10	-8	3
-4	-1	1	7	-6

The simple minded algorithm would be $O(mn)^2$. Can we do it in less time? Kadane's algorithm for 1D array (O(n) runtime) can be used to reduce the time complexity to $O(n^3)$. The idea is if can somehow create n^2 sub problems such that applying the 1D algorithm can solve the problem in $O(n^3)$ time. to fix the left and right columns one by one and find the maximum sum contiguous rows for every left and right column pair. We basically find top and bottom row numbers (which have maximum sum) for every fixed left and right column pair. To find the top and bottom row numbers, calculate sun of elements in every row from left to right and store these sums in an array say temp[].

So temp[i] indicates sum of elements from left to right in row i. If we apply Kadane's 1D algorithm on temp[], and get the maximum sum subarray of temp, this maximum sum would be the maximum possible sum with left and right as boundary columns. To get the overall maximum sum, we compare this sum with the maximum sum so far. **Write pseudocode, implement, test and prove the complexity. In the above example, then closed subarray is the maximum subarray, or is it?**

Matrix Multiplication

- Consider two matrices $A = [a_{ij}]$ of order $n \times m$ and $B = [B_{ij}]$ of order $y \times z$. The matrix is square of order n iff $n = m$, otherwise is called a rectangular one. The square matrices have many applications in computing algorithms like shortest path computation in networks, reachability analyses, linear system solutions and many others. We will restrict ourselves to square matrices unless stated otherwise.
  - A and B can be added only when $n = y$ and $m = z$; addition (subtraction) algorithm is then evidently $O(n*m)$.
  - A and B can be multiplied to generate the product matrix C in $O(n^3)$ time using the formula $C_{i,j} = \sum_{k=1}^{n} A_{i,k} \times B_{k,j}$ Note that $C_{i,j}$ is the dot product of the i^{th} row of X with the j^{th} column of Y. In general, XY is not the same as YX; matrix multiplication is not commutative.
  - [**for** i = 1 **to** n {**for** j = 1 **to** n {$C_{ij} = 0$; {**for** k = 1 **to** n {$C_{ij} = C_{ij} + A_{ik} \times B_{jk}$}}}. Each of the triply nested **for** loops runs exactly n iterations and each execution takes constant time (one element multiplication and one addition), disregarding the time for indexing and fetching the elements from memory.
- Can we improve the running time by applying Divide and Conquer strategy? To make things simple let us assume $n = 2^k$ for some k. Partition each of A, B, C into four $n/2 \times n/2$ submatrices to get

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

$$\begin{aligned} C_{11} &= A_{11} \cdot B_{11} + A_{12} \cdot B_{21}, \\ C_{12} &= A_{11} \cdot B_{12} + A_{12} \cdot B_{22}, \\ C_{21} &= A_{21} \cdot B_{11} + A_{22} \cdot B_{21}, \\ C_{22} &= A_{21} \cdot B_{12} + A_{22} \cdot B_{22}. \end{aligned}$$

Matrix Multiplication

- Each of those four equations specifies two multiplications of $n/2 \times n/2$ matrices and the addition of their $n/2 \times n/2$ products. We can use these equations to create a straightforward, recursive, **divide-and-conquer** algorithm:

```
SQUARE-MATRIX-MULTIPLY-RECURSIVE(A, B)
 1  n = A.rows
 2  let C be a new n × n matrix
 3  if n == 1
 4      c11 = a11 · b11
 5  else partition A, B, and C as in equations
 6      C11 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B11)
             + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B21)
 7      C12 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B12)
             + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B22)
 8      C21 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B11)
             + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B21)
 9      C22 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B12)
             + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B22)
10  return C
```

Note: To implement Line 5 efficiently we do not create 12 new $n/2 \times n/2$ matrices; we rather use index calculations – identify submatrices by a range of row and column indices of the original matrices. Try to write your program using that concept – you gain insight in good programming. The advantage is Line 5 now takes $\Theta(1)$ time instead of $\Theta(n^2)$ time.

Matrix Multiplication

- The total time for the recursive case, therefore, is the sum of the partitioning time, the time for all the recursive calls, and the time to add the matrices resulting from the recursive calls. If T(n) is the time needed to multiply two matrices of order n, then

$$T(n) = \Theta(1) + 8T(n/2) + \Theta(n^2) = 2^3 T(n/2) + \Theta(n^2), \text{ given } T(1) = \Theta(1)$$

- We can solve this equation as $T(n) = \Theta(n^3)$ [we will see how a bit later]. Thus, this simple divide-and-conquer approach is no faster than the straightforward SQUARE-MATRIX-MULTIPLY procedure.
- **Note:**
  - Partitioning each n × n matrix by index calculation takes Θ (1) time, but we have two matrices to partition. Although you could say that partitioning the two matrices takes Θ(2) time, the constant of 2 is subsumed by the Θ -notation.
  - Adding two matrices, each with, say, k entries, takes Θ(k) time. Since the matrices we add each have $n^2/4$ entries, we write it takes time is $\Theta(n^2)$ instead of $\Theta(n^2/4)$.
  - The point here is that, Θ -notation subsumes constant factors, whatever they are.)
  - But in case of recursive calls, we cannot just subsume the constant factor of 8. We must say that together they take 8T(n/2) time, rather than just T(n/2) time. You can get a feel for why by looking at the recursion tree. The factor of 2 determined how many children each tree node had, which in turn determined how many terms contributed to the sum at each level of the tree.
  - If we ignore the factor of 8 in recurrence, the recursion tree would just be linear, rather than "bushy," and each level would contribute only one term to the sum.

Matrix Multiplication

- Remember that although asymptotic notation subsumes constant multiplicative factors, recursive notation such as T (n/2) does not.
- The key to Strassen's method is to make the recursion tree slightly less bushy. That is, instead of performing eight recursive multiplications of n/2 × n/2 matrices, it performs only seven. The cost of eliminating one matrix multiplication will be several new additions of n/2 × n/2 matrices, but still only a constant number of additions. As before, the constant number of matrix additions will be subsumed by Θ-notation when we set up the recurrence equation to characterize the running time.
- It turns out XY can be computed from just seven n/2 × n/2 sub problems, via a decomposition so *tricky and intricate* that one wonders how Strassen was ever able to discover it! We change the notations to avoid subscripts:

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix}, \quad Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}, \quad XY = \begin{bmatrix} A & B \\ C & D \end{bmatrix}\begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE+BG & AF+BH \\ CE+DG & CF+DH \end{bmatrix}$$

$$XY = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix}$$

- where

$$\begin{aligned} P_1 &= A(F-H) \\ P_2 &= (A+B)H \\ P_3 &= (C+D)E \\ P_4 &= D(G-E) \end{aligned} \qquad \begin{aligned} P_5 &= (A+D)(E+H) \\ P_6 &= (B-D)(G+H) \\ P_7 &= (A-C)(E+F) \end{aligned}$$

Matrix Multiplication – Strassen's Algorithm

- Hence, we obtain the following recurrence for the running time T (n) of Strassen's algorithm:

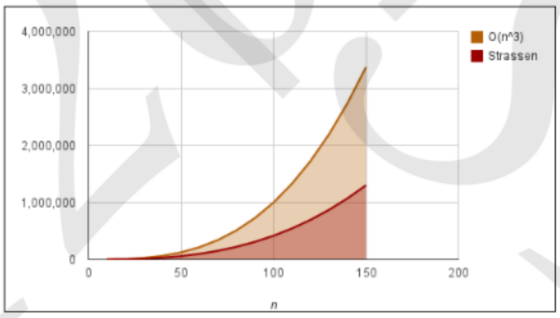
$$T(n) = \begin{cases} \Theta(1), & n = 1 \\ 7T(n/2 + \Theta(n^2), & n > 1 \end{cases}$$

- The new running time is $T(n) = 7T(n/2) + O(n^2)$ which by Master's theorem works out to

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

- It's easy to write the code as well. From a practical point of view, Strassen's algorithm is often not the method of choice for matrix multiplication:
  - The constant factor hidden on $O(n^{2.81})$ is much larger than that in the regular algorithm.
  - When the matrices are sparse, methods tailored for sparse matrices are faster.
  - Strassen's algorithm is not quite as numerically stable as the regular algorithm (this a bit overhype), i.e., because of the limited precision of computer arithmetic on non-integer values, larger errors accumulate in Strassen's algorithm than in the regular algorithm.
  - The submatrices formed at the levels of recursion consume space.
- In practice, fast matrix-multiplication implementations for dense matrices use Strassen's algorithm for matrix sizes above a "crossover point," and they switch to a simpler method once the subproblem size reduces to below the crossover point. The exact value of the crossover point is highly system dependent.

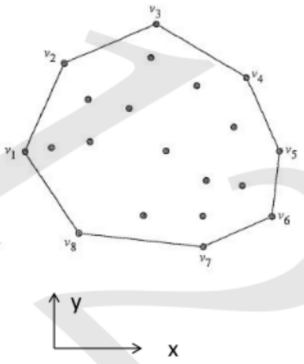
Larger Matrix Multiplication



- **Note [way outside our scope]:** Until a few years ago, the fastest known matrix multiplication algorithm, due to Coppersmith and Winograd (1990), ran in time $O(n^{2.376})$. Recently, a surge of activity by Stothers, Vassilevska-Williams and Le Gall has led to an improved algorithm running in time $O(n^{2.3728639})$, due to Le Gall (2014). These algorithms are obtained by analyzing higher and higher tensor powers of a certain identity of Coppersmith and Winograd. These algorithms start to pay dividend only when order of the matrices are higher than at least 250 or more because of large constants involved. The holy grail is to prove the **bold conjecture that for every $\epsilon > 0$, matrices can be multiplied in time $O(n^{2+\epsilon})$.**

Convex Hull Problem

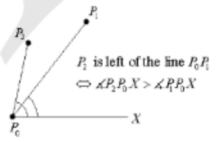
- Given a set S = {P_1,P_2, ... ,P_n} of n points in the plane, each represented by its (x, y) coordinates, the **planar convex hull** of S is the smallest convex polygon containing all the n points of S. A polygon Q is convex if, given any two points p and q in Q, the line segment whose endpoints are p and q lies entirely in Q. The convex-hull problem is to determine the ordered (say, clockwise) list CH(S) of the points of S defining the boundary of the convex hull of S. Also, this convex hull has the smallest area and the smallest perimeter of all convex polygons that contain S.



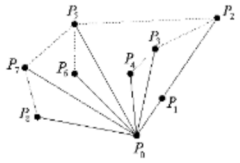
- Consider the set S of points shown in the example; the convex hull of S is given by CH(S) = (v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8).
- The convex-hull problem is an important basic problem in computational geometry that arises in a large variety of contexts.
- Simple sorting can be reduced to convex hull problem; so *sequential* complexity [optimal work] of this problem is $T_S(n) = \Omega(n \log n)$. This algorithm and its implementation has been covered in great detail by [O'Rourke, 1998, Sect 3.5, 72-86]
- **Note:** Let p and q be the points of S with the smallest and the largest x coordinates, respectively. Clearly, p and q belong to CH(S) and partition CH(S) into an upper hull UH(S) consisting of all points from p to q of CH(S) (clockwise), and a lower hull LH(S) defined similarly from q to p.

Notes on Sequential Implementation*

- We briefly describe the concepts behind the well known Graham's scan algorithm. The O(n log n) time for the Graham scan is spent doing an initial radial sort of the input set points. After that, the algorithm employs a stack-based method which runs in just O(n) time. In fact, the method performs at most 2n simple stack push and pop operations. Thus, it executes very rapidly, bounded only by the speed of sorting.
- Let S be the set of n given planar points (x-y coordinates). Select a point with minimum y-coordinate (resolve ties by taking the maximum x-coordinate) – takes O(n) time. Call this point P_0, the starting point. Then, sort the other points P in S *radially* by the increasing counter-clockwise (ccw) angle the line segment P_0P makes with the x-axis. If there is a tie and two points have the same angle, discard the one that is closest to P_0 – will take O(n log n) time. [Actual computations of angles using 'arctan' utilities is not really necessary – observe that P_2 would make a greater angle than P_1 if (and only if) P_2 lies on the left side of the directed line segment P0P1 as shown in the following diagram. This condition can be tested by a fast accurate computation that uses only 5 additions and 2 multiplications.]




```
// isLeft(): test if a point is Left|On|Right of an infinite 2D line.
//   Input:  three points P0, P1, and P2, Return: >0 for P2 left of the line through P0 to P1,
//           =0 for P2 on the line, <0 for P2 right of the line
isLeft( Point P0, Point P1, Point P2 ) {
   return ( (P1.x - P0.x) * (P2.y - P0.y) - (P2.x - P0.x) * (P1.y - P0.y) );}
```

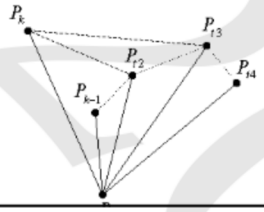


- After sorting, we get a 'fan' with a pivot at P_0

*Taken from: http://geomalgorithms.com/a10-_hull-1.html

Sequential Convex Hull

- Loop through the points of S one-by-one testing for convex hull vertices, using an inductive incremental procedure using a stack of points. At each stage, we save (on the stack) the vertex points for the convex hull of all points already processed.
- We start with P_0 and P_1 on the stack. Then at the k-th stage, we add the next point P_k, and compute how it alters the prior convex hull. Because of the way S was sorted, P_k is outside the hull of the prior points P_i with $i < k$, and it must be added as a new hull vertex on the stack. But it's addition may cause previous stack points to no longer be a hull vertices. If this happens, the previous points must be popped off the stack and discarded. One tests for this by checking if the new point P_k is to the left or the right of the line joining the top two points of the stack. Again, we use the routine isLeft() to quickly make this test.
  - If P_k is on the left of the top segment, then prior hull vertices remain intact, and P_k gets pushed onto the stack. But, if it is on the right side of the top segment, then the prior point at the stack top will get absorbed inside the new hull, and that prior point must be popped off the stack. This test against the line segment at the stack top continues until either P_k is left of that line or the stack is reduced to the single base point P0. In either case, P_k gets pushed onto the stack, and the algorithm proceeds to the next point P_{k+1} in the set. The different possibilities involved are illustrated in the following diagram.



Old Stack $= S_{k-1} = \{P_0, \ldots, P_{t3}, P_{t2}, P_{k-1}\}$
P_k right of line $P_{t2}P_{k-1} \Rightarrow$ pop P_{k-1} off stack
P_k right of line $P_{t3}P_{t2} \Rightarrow$ pop P_{t2} off stack
P_k left of line $P_{t4}P_{t3} \Rightarrow$ push P_k onto stack
New Stack $= S_k = \{P_0, \ldots, P_{t3}, P_k\}$

Sequential Convex Hull

- It is easy to understand why this works by viewing it as an incremental algorithm. The old stack $S_{k-1}=\{P_0,\ldots,P_{t2},P_{k-1}\}$, with P_{k-1} at the top, is the convex hull of all points P_i with $i < k$. The next point P_k is outside this hull since it is left of the line P_0P_{k-1} which is an edge of the S_{k-1} hull. To incrementally extend S_{k-1} to include P_k, we need to find the two tangents from P_k to S_{k-1}. One tangent is clearly the line P_kP_0. The other is a line P_kP_t such that P_k is left of the segment in S_{k-1} preceding Pt and is right of the segment following Pt (when it exists). This uniquely characterizes the second tangent since S_{k-1} is a convex polygon. The way to find P_t is simply to search from the top of the stack down until the point with the property is found. The points above P_t in S_{k-1} are easily seen to be contained inside the triangle $\Delta P_0P_tP_k$, and are thus no longer on the hull extended to include P_k. So, they can be discarded by popping them off the stack during the search for P_t. Then, the k-th convex hull is the new stack $S_k=\{P_0,\ldots,P_t,P_k\}$.

- At the end, when $k = n-1$, the points remaining on the stack are precisely the ordered vertices of the convex hull's polygon boundary. Note that for each point of S there is one push and at most one pop operation, giving at most 2n stack operations for the whole algorithm.

Pseudo-Code: Graham Scan Algorithm

Input: a set of points S = {P = (P.x,P.y)}

```
Select the rightmost lowest point P0 in S
Sort S radially (ccw) about P0 as a center {
   Use isLeft() comparisons
   For ties, discard the closer points   }
Let P[N] be the sorted array of points with P[0]=P0

Push P[0] and P[1] onto a stack Ω
while i < N   {
   Let P_T1 = the top point on Ω
   If (P_T1 == P[0]) {
      Push P[i] onto Ω;   i++}
   Let P_T2 = the second top point on OMEGA
   If (P[i] is strictly left of the line  P_T2 to P_T1) {
      Push P[i] onto OMEGA
      i++ }
   else
      Pop the top point P_T1 off the stack   }
```

Output: Ω = the convex hull of S.