Chapter – 2 – Divide and Conquer Algorithm

- Introduction
- Recurrence relations
- Multiplying large Integers Problem
- Binary Search
- Merge Sort
- Quick Sort
- Matrix Multiplication
- Exponentiation

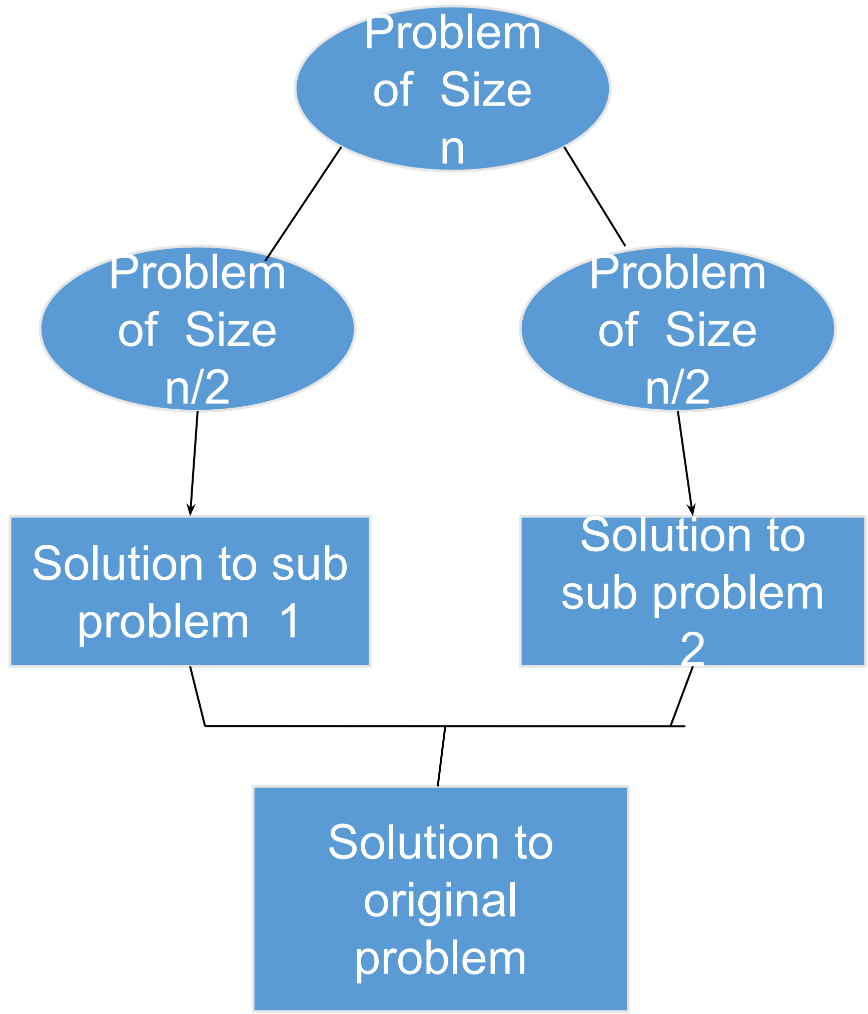
Introduction

Divide & conquer is a general algorithm design strategy with a general plan as follows:

- 1. DIVIDE: A problem's instance is divided into several smaller instances of the same problem, ideally of about the same size.
- 2. RECUR: Solve the sub-problem recursively.
- 3. CONQUER: If necessary, the solutions obtained for the smaller instances are combined to get a solution to the original instance.

Introduction

- Diagram shows the general divide & conquer plan



General Algorithm for Divide & Conquer

Divide&Conquer(I, S)
pre: I = instance of a problem
post: S = feasible solution for I

partion I into $I_1, \ldots, I_k$
for each Ik
    if I_k can be solved optimally do so
        and let S_k be this solution
    otherwise Divide&Conquer(I_k, S_k)
combine $S_1, \ldots, S_k$ into S

- The computing time of above procedure of divide and conquer is given by the recurrence relation

- $$T(n) = \begin{cases} g(n) & \text{,if n is small} \\ T(n1)+T(n2)+\ldots+T(nr)+f(n) & \text{,if n is sufficiently large} \end{cases}$$

Introduction

- The real work is done piecemeal, in three different places: in the partitioning of problems into sub problems; at the very tail end of the recursion, when the sub problems are so small that they are solved outright; and in the gluing together of partial answers.

- These are held together and coordinated by the algorithm's core recursive structure.

- As an introductory example, we'll see how this technique yields a new algorithm for multiplying numbers, one that is much more efficient than the method we all learned in elementary school!

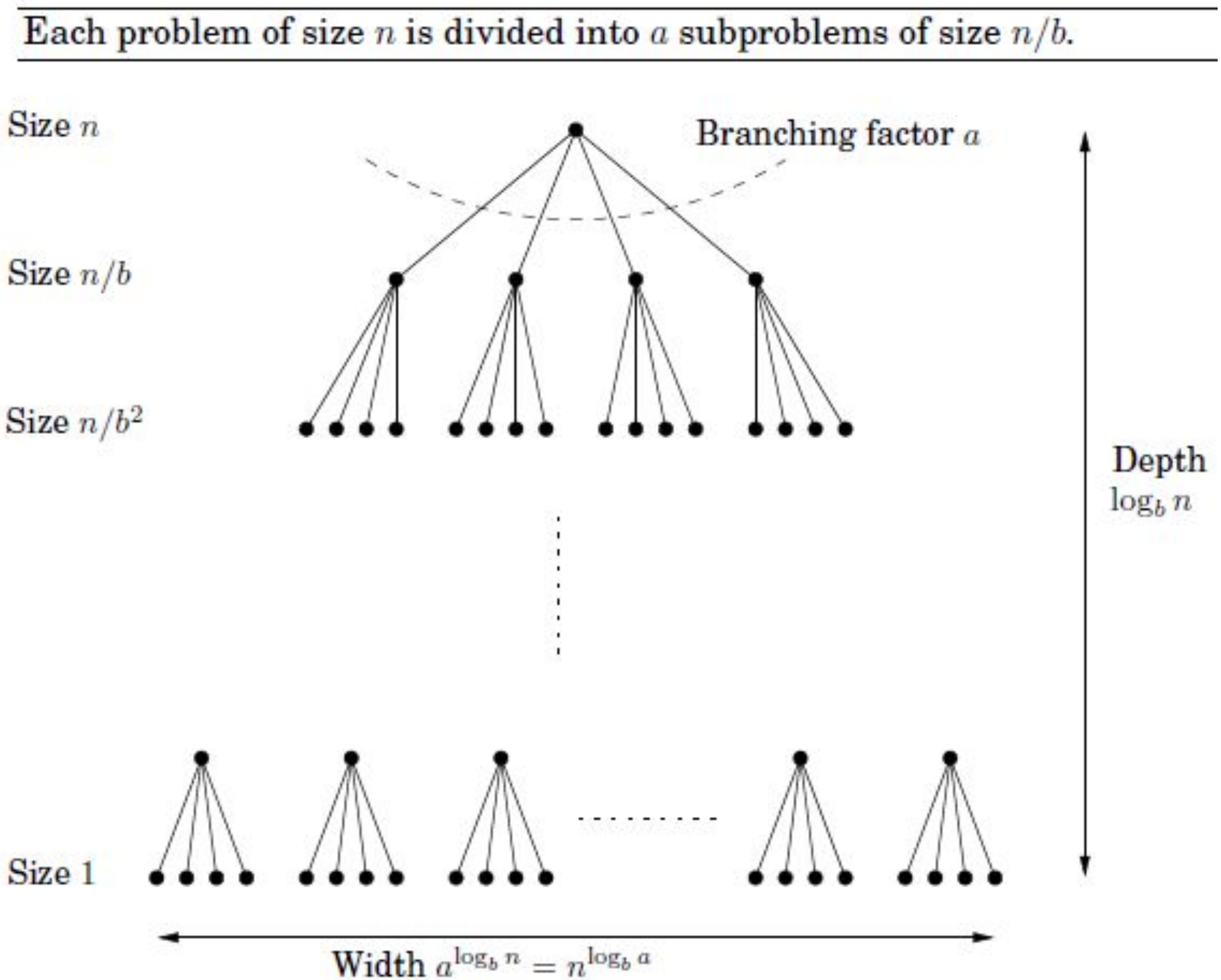
Recurrence relations

- Divide-and-conquer algorithms often follow a generic pattern: they tackle a problem of size n by recursively solving, say, a sub problems of size n/b and then combining these answers in O(n^d) time, for some a; b; d > 0 (in the multiplication algorithm, a = 3, b = 2, and d = 1).

- Their running time can therefore be captured by the equation T(n) = aT ([n/b]) + O(n^d).

- We next derive a closed-form solution to this general recurrence so that we no longer have to solve it explicitly in each new instance.

Recurrence relations



Recurrence relations

Master theorem[2] *If* $T(n) = aT(\lceil n/b \rceil) + O(n^d)$ *for some constants* $a > 0$, $b > 1$, *and* $d \geq 0$, *then*

$$T(n) = \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a\,. \end{cases}$$

subproblems, each of size n/b^k The total work done at this level is

$$a^k \times O\left(\frac{n}{b^k}\right)^d = O(n^d) \times \left(\frac{a}{b^d}\right)^k.$$

Multiplying large Integers Problem

- Consider that we want to perform operation 1234*981.
- We need to divide the integer in two portion left and right. (12 and 34)
- $1234 = 10^2 * 12 + 34 = 1234$
- For 0981 * 1234 consider following things.
  - w = 09, x = 81,y=12 and z = 34
- $0981 * 1234 = (10^2w + x)*(10^2y + z)$

$$=10^4 wy + 10^2(wz + xy) + xz$$
$$=1080000 + 1278000 + 2754$$
$$= 1210554$$

Multiplying large Integers Problem

- The above procedure still needs four half-size multiplications; wy, wz, xy and xz.

- The key observation is that there is no need to compute both wz and xy; all we really need is the sum of these two terms.

- Is it possible to obtain wz + xy at the cost of single multiplication? Our equeation is like

- Result = $(w + x) * (y + z) = wy + (wz + xy) + xz$

Multiplying large Integers Problem

- There is one mathematical formula that we are going to apply here
    - $(wz + xy) = (w + x)*(y + z) - wy - xz$

- $p = wy = 09 * 12 = 108$
- $q = xz \quad = 81 * 34 = 2754$
- $r = (w + x) * (y + z) = 90 * 46 = 4140$

- $0981 * 1234 = 10^4 p + 10^2 (r-p-q) + q$
- $= 1080000 + 127800 + 2754$
- $= 1210554$

Multiplying large Integers Problem

- If each of the three half – size multiplications is carried out by the classic algorithm, the time needed to multiply two n – figure numbers is f(n). For some constants c, g(n) is the time needed for additions, shifts, and overhead operations. So the total time required is,
  - $T(n)=f(n)+c.g(n)$

- So, the recurrence relation to multiply large integers problem is
  - $T(n)=3T(n/2)+n$

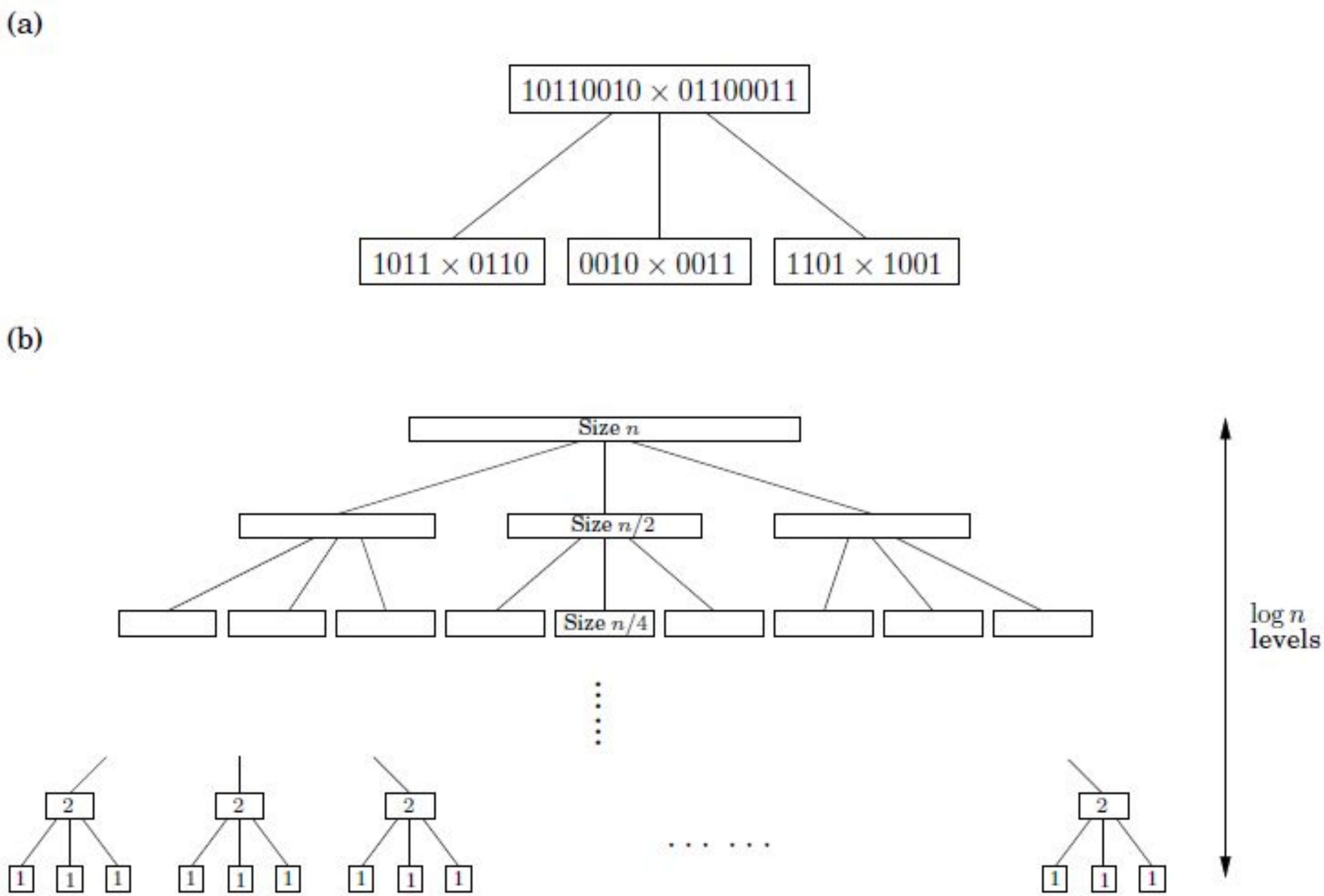
Multiplying large Integers Problem

- Calculate multiplication of following number using divide and conquer

- 1234*4321 = ?

Multiplying large Integers Problem

Divide-and-conquer integer multiplication. (a) Each problem is divided into three subproblems. (b) The levels of recursion.



Multiplying large Integers Problem

- At each successive level of recursion the sub problems get halved in size.
- At the $(\log_2 n)^{th}$ level, the sub problems get down to size 1, and so the recursion ends.
- Therefore, the height of the tree is $\log_2 n$.
- The branching factor is 3. Each problem recursively produces three smaller ones. With the result that at depth k in the tree there are 3^k sub problems, each of size $n/2^k$.

Multiplying large Integers Problem

- For each sub problem, a linear amount of work is done in identifying further sub problems and combining their answers. Therefore the total time spent at depth k in the tree is

$$3^k \times O\left(\frac{n}{2^k}\right) = \left(\frac{3}{2}\right)^k \times O(n)$$

Multiplying large Integers Problem

At the very top level, when $k = 0$, this works out to $O(n)$. At the bottom, when $k = \log_2 n$, it is $O(3^{\log_2 n})$, which can be rewritten as $O(n^{\log_2 3})$ (do you see why?). Between these two endpoints, the work done increases *geometrically* from $O(n)$ to $O(n^{\log_2 3})$, by a factor of $3/2$ per level. The sum of any increasing geometric series is, within a constant factor, simply the last term of the series: such is the rapidity of the increase (Exercise 0.2). Therefore the overall running time is $O(n^{\log_2 3})$, which is about $O(n^{1.59})$.

Multiplying large Integers Problem

A divide-and-conquer algorithm for integer multiplication.

```
function multiply(x, y)
Input:  Positive integers x and y, in binary
Output:  Their product
```

$n =$ `max(size of` x, `size of` y`)`
`if` $n = 1$: `return` xy

x_L, $x_R =$ `leftmost` $\lceil n/2 \rceil$, `rightmost` $\lfloor n/2 \rfloor$ `bits of` x
y_L, $y_R =$ `leftmost` $\lceil n/2 \rceil$, `rightmost` $\lfloor n/2 \rfloor$ `bits of` y

$P_1 = \texttt{multiply}(x_L, y_L)$
$P_2 = \texttt{multiply}(x_R, y_R)$
$P_3 = \texttt{multiply}(x_L + x_R, y_L + y_R)$
`return` $P_1 \times 2^n + (P_3 - P_1 - P_2) \times 2^{n/2} + P_2$

Binary Search

```
Function Binary_Search (T[1…n], x)
{
        i □ 1; j □ n
        while i < j do
        {
            k □ (i+j)/2;
            if (x < T[k])
                j □ k – 1;
            if(x == T[k])
                i, j □ k;
                return k
            if(x > T[k])
                i □ k + 1;
        }
}
```

Binary Search

```
Function binrec(T[i…j], x)
{
        if(j<i)
            return -1
        else
        {
            k  (i + j) /2
            if(x==t[k]) then
                return k
            else if(x <= t[k]) then
                return binrec(T[i..k-1], x)
            else
                return binrec(T[k+1…j], x)
        }
}
```

Complexity of Binary Search Method

- When n= 64 BinarySearch is called to reduce size to n=32 When n= 32 BinarySearch is called to reduce size to n=16 When n= 16 BinarySearch is called to reduce size to n=8 When n= 8 BinarySearch is called to reduce size to n=4 When n= 4 BinarySearch is called to reduce size to n=2 When n= 2 BinarySearch is called to reduce size to n=1
- Let us consider a more general case where n is a power of 2. Let us say $n = 2^k$.
- 2k = n Taking log of both sides

  $k = \log n$

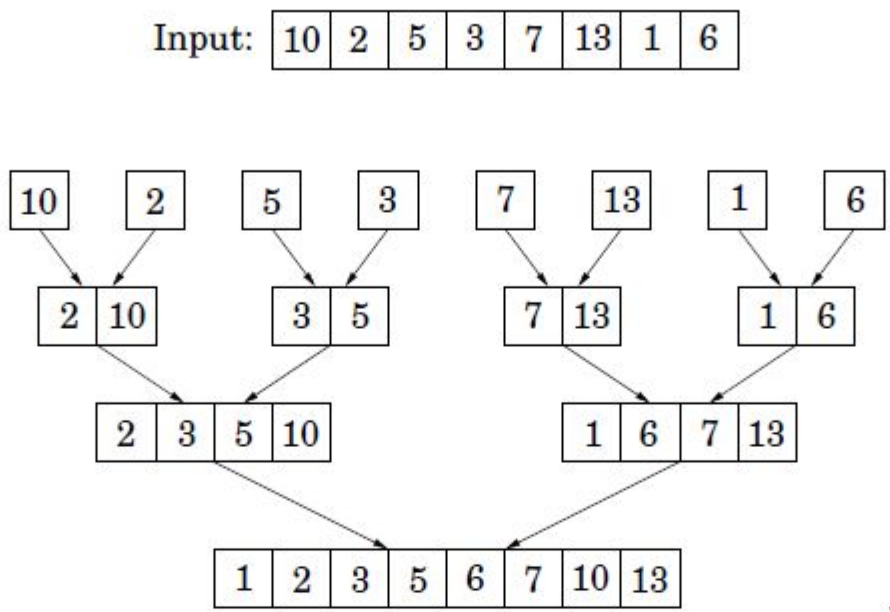
  - Recurrence for binary search is $T(n)=T(n/2)+1$ (the time to search in an array of 1 element is constant)

  We conclude from there that the time complexity of the Binary search method is **O(log n)**.

Merge Sort

- 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 sub lists.

The sequence of merge operations in `mergesort`.



Merge Sort

- The correctness of this algorithm is self-evident, as long as a correct merge subroutine is specified.

- If we are given two sorted arrays $x[1\ldots k]$ and $y[1\ldots l]$, how do we efficiently merge them into a single sorted array $z[1 \ldots k + l]$?

- 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.

Merge Sort

- MergeSort(A,p,r)

```
if p>r
    return
    q=(p+r)/2
    MergeSort(A,p,q)
    MergeSort(A,q+1,r)
    MERGE(A,p,q,r)
```

Merge Sort

- MERGE (A, p, q, r)

$n_1 \leftarrow q - p + 1$

$n_2 \leftarrow r - q$

Create arrays L[1 . . n_1 + 1] and R[1 . . n_2 + 1]

for $i \leftarrow 1$ **to** n_1 then

do L[i] $\leftarrow$ A[$p + i - 1$]

for $j \leftarrow 1$ **to** n_2 then

do R[j] $\leftarrow$ A[$q + j$]

L[n_1 + 1] $\leftarrow \infty$

R[n_2 + 1] $\leftarrow \infty$

$i \leftarrow 1$

$j \leftarrow 1$

FOR $k \leftarrow p$ **TO** r

DO IF L[i] $\leq$ R[j]

THEN A[k] $\leftarrow$ L[i]

$i \leftarrow i + 1$

ELSE A[k] $\leftarrow$ R[j]

$j \leftarrow j + 1$

Merge Sort

- Time taken for divide is $f(n)=2T(n/2)$ because of two half size partitions.
- For merge linear amount of work is done, so time taken is $g(n)=O(n)$.
- Thus merge's are linear, and the overall time taken by merge sort is
  - $T(n) = 2T(n/2) + O(n)$; or $O(n \log n)$

Quick Sort

- Given an array of n elements (e.g., integers):

- If array only contains one element, return array

  Else

  pick one element to use as pivot.

  Partition elements into two sub-arrays:

  Elements less than or equal to pivot

  Elements greater than pivot

  Quick sort two sub-arrays

  Return results

Quick Sort

40	20	10	80	60	50	7	30	100

[redacted]

[redacted]

[redacted]

Quick Sort

- Procedure quicksort(A,p,r)

```
{
        if(p&lt;r) then
            //split the array into two sub arrays
        q<-partition(A,p,r)
        quicksort(A,p,q-1)
        quicksort(A,q+1,r)
}
```

Quick Sort

- Procedure partition(A,p,r)

```
{
    x<-A[r]
    i<-p-1

        for j<-p to r-1
        {
        if A[j]<=x
            then i<-i+1
            exchange A[i]<->A[j]
        }
        exchange A[i+1]<->A[r]
        return i+1
    }
```

Quick Sort

- Quick Sort analysis
  - Assume that keys are random, uniformly distributed.
  - Best case running time: O(n log n)
  - Worst case running time? O(n∧2)
    - Recursion:
    - Partition splits array in two sub-arrays:
    - one sub-array of size 0
    - the other sub-array of size n-1
    - Quicksort each sub-array
    - Depth of recursion tree? O(n)
    - Number of accesses per partition? O(n)

Quick Sort

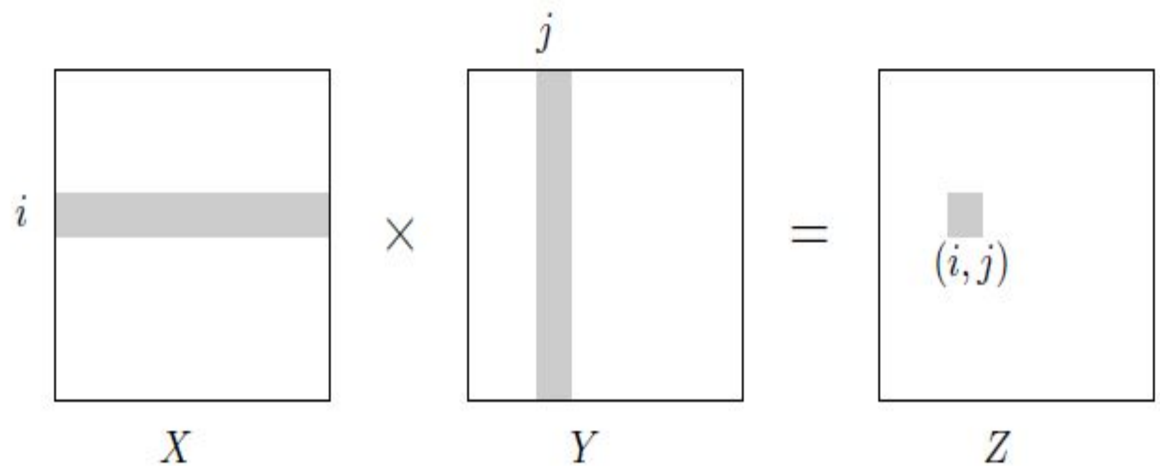
- Quick Sort analysis
  - Assume that keys are random, uniformly distributed.
  - Best case running time: O(n log n)
    - $T(n) <= 2T(n/2) + \varnothing(n)$
  - Worst case running time? O(n $\wedge$ 2)
    - $T(n) = T(n-1) + T(0) + \varnothing(n) = T(n-1) + \varnothing(n)$
      - Recursion:
      - Partition splits array in two sub-arrays:
        - one sub-array of size 0
        - the other sub-array of size n-1
        - Quicksort each sub-array
      - Number of accesses per partition? $\varnothing(n)$

Matrix Multiplication

The product of two $n \times n$ matrices X and Y is a third $n \times n$ matrix $Z = XY$, with (i, j)th entry

$$Z_{ij} = \sum_{k=1}^{n} X_{ik} Y_{kj}.$$

To make it more visual, Z_{ij} is the dot product of the ith row of X with the jth column of Y:



In general, XY is not the same as YX; matrix multiplication is not commutative.

Iterative Matrix Multiplication

Input: matrices A and B

Let C be a new matrix of the appropriate size

For i from 1 to n:

  For j from 1 to p:

    Let sum = 0

    For k from 1 to m:

      Set sum $\leftarrow$ sum + $A_{ik} \times B_{kj}$

    Set $C_{ij} \leftarrow$ sum

Return C

Recursive Matrix Multiplication

```
Function MMult(A, B, n)
If n = 1
        Output A × B
Else
        Compute A11, B11, . . ., A22, B22 % by computing m = n/2
        X1 ← MMult(A11, B11, n/2)
        X2 ← MMult(A12, B21, n/2)
        X3 ← MMult(A11, B12, n/2)
        X4 ← MMult(A12, B22, n/2)
        X5 ← MMult(A21, B11, n/2)
        X6 ← MMult(A22, B21, n/2)
        X7 ← MMult(A21, B12, n/2)
        X8 ← MMult(A22, B22, n/2)
        C 11 ← X1 + X2
        C 12 ← X3 + X4
        C 21 ← X5 + X6
        C 22 ← X7 + X8
        Output C
End If
```

Matrix Multiplication

- The preceding formula implies an O(n∧3) algorithm for matrix multiplication: there are n∧2 entries to be computed, and each takes O(n) time.

- For quite a while, this was widely believed to be the best running time possible, and it was even proved that in certain models of computation no algorithm could do better.

- It was therefore a source of great excitement when in1969, the German mathematician Volker Strassen announced a signicantly more effcient algorithm, based upon divide-and-conquer.

Matrix Multiplication

Matrix multiplication is particularly easy to break into subproblems, because it can be performed *blockwise*. To see what this means, carve X into four $n/2 \times n/2$ blocks, and also Y:

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix}, \quad Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}.$$

Then their product can be expressed in terms of these blocks and is exactly as if the blocks were single elements (Exercise 2.11).

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

We now have a divide-and-conquer strategy: to compute the size-n product XY, recursively compute eight size-$n/2$ products $AE, BG, AF, BH, CE, DG, CF, DH$, and then do a few $O(n^2)$-time additions. The total running time is described by the recurrence relation

$$T(n) = 8T(n/2) + O(n^2).$$

Optimized Matrix Multiplication Algorithm

Strassen(A, B)

If n = 1

A=[1 2
3 4]

B=[1 2
3 4]

Output A × B

Else

Compute A11, B11, . . ., A22, B22 % by computing m = n/2

P1 ← Strassen(A11, B12 − B22)

P2 ← Strassen(A11 + A12, B22)

P3 ← Strassen(A21 + A22, B11)

P4 ← Strassen(A22, B21 − B11)

P5 ← Strassen(A11 + A22, B11 + B22)

P6 ← Strassen(A12 − A22, B21 + B22)

P7 ← Strassen(A11 − A21, B11 + B12)

C 11 ← P5 + P4 − P2 + P6

C 12 ← P1 + P2

C 21 ← P3 + P4

C 22 ← P1 + P5 − P3 − P7

Output C

Matrix Multiplication

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

where

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

The new running time is

$$T(n) = 7T(n/2) + O(n^2),$$

which by the master theorem works out to $O(n^{\log_2 7}) \approx O(n^{2.81})$.

Exponentiation

- Let a and n be two integers. We wish to compute the exponentiation x = a^n.
- For simplicity, we shall assume throughout this section that n > 0.
- If n is small, the obvious algorithm is adequate.
  - Function exposeq(a, n)

```
{
    r  a
    for i  1 to n-1 do r  a*r
    return r
}
```

Complexity of this algorithm is O(n)

Exponentiation

- a ∧ n = return a if n = 1

  return (a ∧ n/2) ∧ 2 if n is even

  return a * a ∧ n-1 otherwise

a ∧ 29 = a * a ∧ 28 = a(a ∧ 14) ∧ 2 …….

Which involves only three multiplications and four squaring instead of the 28 multiplication.

Exponentiation

- Function expoDC(a, n)

```
{
        if n = 1
        then
            return a
        else
        return a * expoDC(a, n - 1)
}
```

Recurrence relation for recursive exponentiation problem

T(n)=T(n-1)+1

Now the complexity of above algorithm is **O(log n)**

Any Question ?