ALGORITHMS & ADVANCED DATA STRUCTURES (#4)

DIVIDE AND CONQUER

MERGE SORT & MAXIMUM SUBARRAY & MATRIX MULTIPLICATION

ADAPTED FROM
CS 146 SJSU (KATERINA POTIKA)

Designing algorithms

- **1st Technique: Divide and conquer**
  - **Divide** the problem into a number of sub-problems.
  - **Conquer** the sub-problems by solving them recursively.
    - ***Base case:*** If the sub-problems are small enough, just solve them by brute force.
  - **Combine** the sub-problem solutions to give a solution to the original problem

Mergesort

- Split the input into 2 parts.
- Recursively sort each of them.
- Merge the two sorted parts.

Mergesort – more details

- Each sub-problem as sorting a sub-array ***A*[*p* . . *r*]**.
  - Initially, $p = 1$ and $r = n$, but these values change as we recursively solve sub-problems.
- To sort $A[p\ ..\ r]$:
  - **Divide** by splitting into two sub-arrays
    - $A[p\ ..\ q]$
    - $A[q + 1\ ..\ r]$, where q is the halfway point of $A[p\ ..\ r]$.
  - **Conquer** by *recursively* sorting the two sub-arrays $A[p\ ..\ q]$ and $A[q + 1\ ..\ r]$.
  - **Combine** by merging the two sorted sub-arrays $A[p\ ..\ q]$ and $A[q + 1\ ..\ r]$ to produce a single sorted sub-array $A[p\ ..\ r]$.
    - MERGE*(A, p, q, r)* // basic "sort" operation
- The recursion ends when the sub-array has just 1 element, so that it's trivially sorted.

How do we merge?

- Input: 2 sorted sub-array *A[p..q]* and *A[q+1..r]*
- Output: A sorted sub-array *A[p..r]* which contains all the elements.

- *Merge*(*A*,*p*,*q*,*r*)
  - **while** there are still elements in the 2 sub-arrays **do**
  - Compare the 1st elements of the sorted 2 sub-arrays.
  - Move the minimum of them from its corresponding list to the end of output sub-array.

MERGE-SORT(A, p, r)

if $p < r$ // Check for base case

then $q \leftarrow (p + r)/2$ //Divide

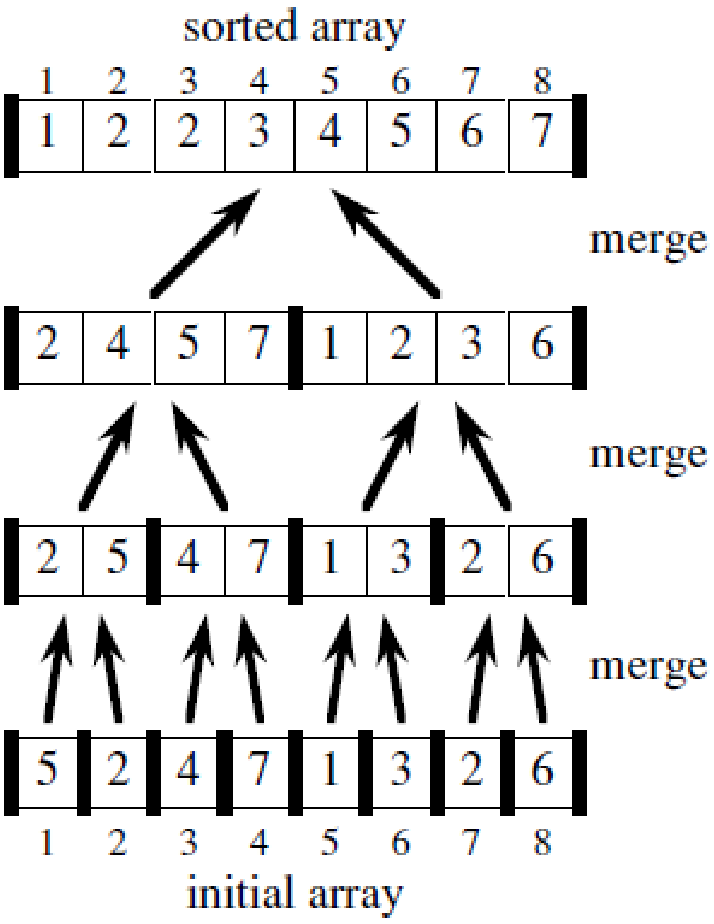
MERGE-SORT(A, p, q) // Conquer

MERGE-SORT$(A, q + 1, r)$ // Conquer

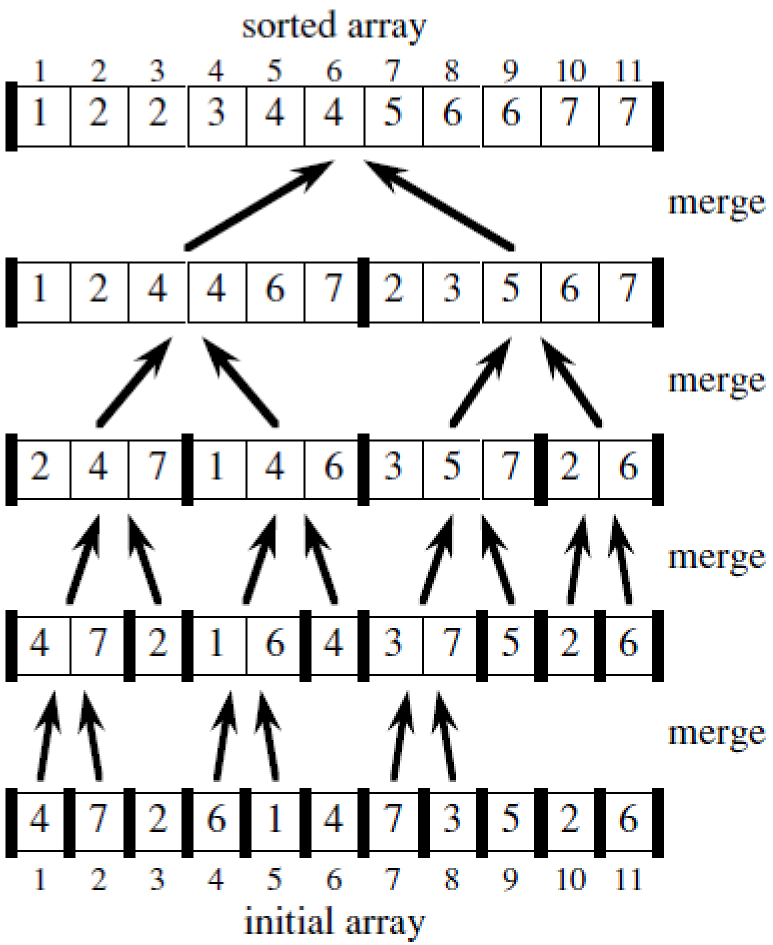
MERGE(A, p, q, r) // Combine

- ***Initial call:*** MERGE-SORT$(A, 1, n)$

Example with n=8



Example with n=11



Analysis of Merge Sort

Merge-Sort(A, p, r)	**//T(n)**
if (p < r)	**//Θ(1)**
q = $\lfloor (\boldsymbol{p} + \boldsymbol{r})/\mathbf{2} \rfloor$	**//Θ(1)**
Merge-Sort(A, p, q);	**//T(n/2)**
Merge-Sort(A, q+1, r);	**//T(n/2)**
Merge(A, p, q, r);	**//Θ(n)**

Time analysis

- If the problem size is small, say c for some constant c, we can solve the problem in constant, i.e., $\Theta(1)$ time.
- Let $T(n)$ be the time needed to sort for input of size n.
- Let cn be the time needed to merge 2 lists of total size n. We know that $cn = \Theta(n)$.
- Assume that the problem can be split into 2 subproblems in constant time and that $c = 1$.

Recurrences

- The expression:

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

- is a *recurrence*.
  - Recurrence: an equation that describes a function in terms of its value on smaller functions

Recurrence Examples

$$s(n) = \begin{cases} 0 & n = 0 \\ s(n-1) + c & n > 0 \end{cases}$$

$$s(n) = \begin{cases} 0 & n = 0 \\ s(n-1) + n & n > 0 \end{cases}$$

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

$$T(n) = \begin{cases} c & n = 1 \\ aT\left(\frac{n}{b}\right) + cn & n > 1 \end{cases}$$

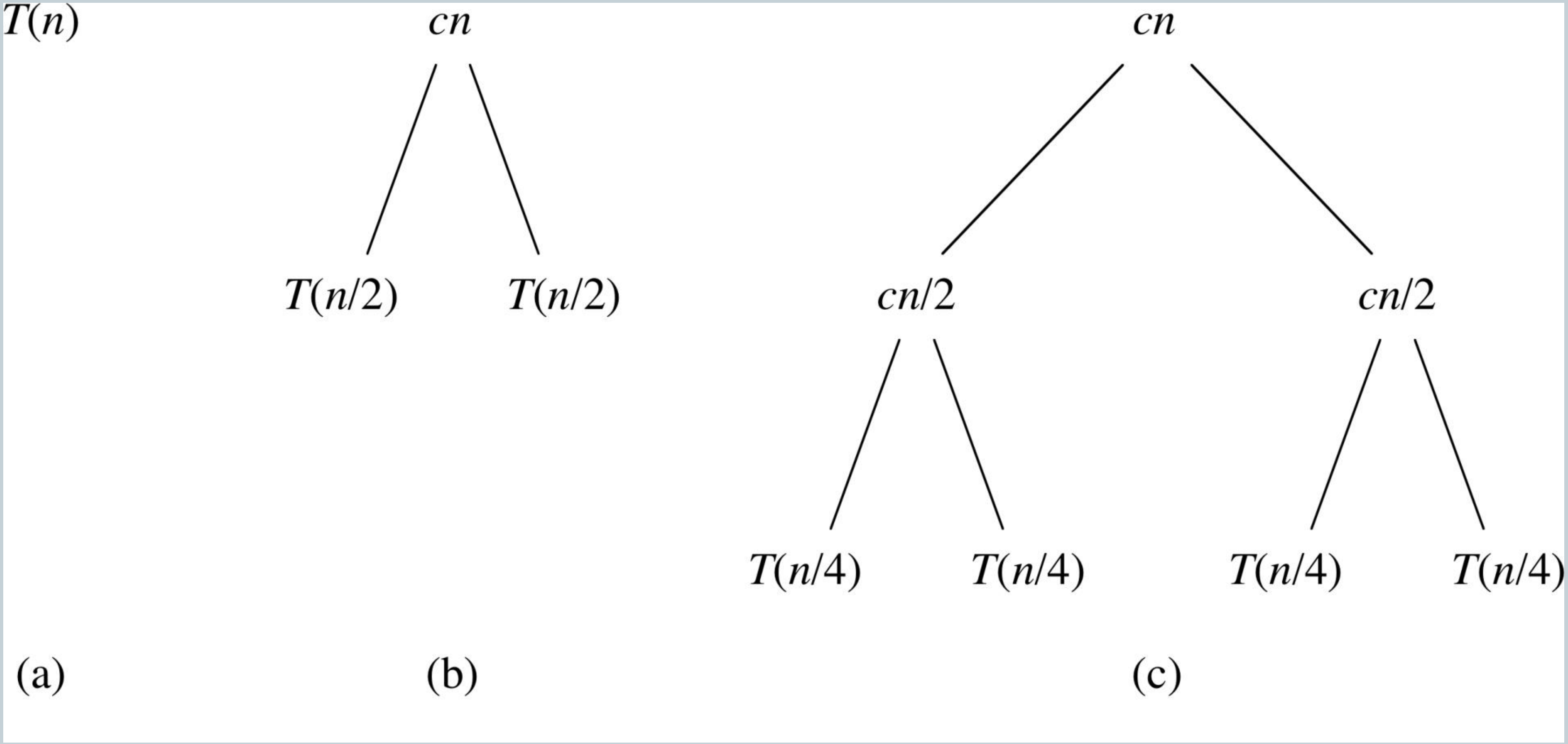
How to we find T(n)

- Question: Is there a closed form for $T(n)$?
- W.l.o.g., assume $n = 2^k$ (or, lg $n = k$).

- *Note:* $lgn = log_2 n$

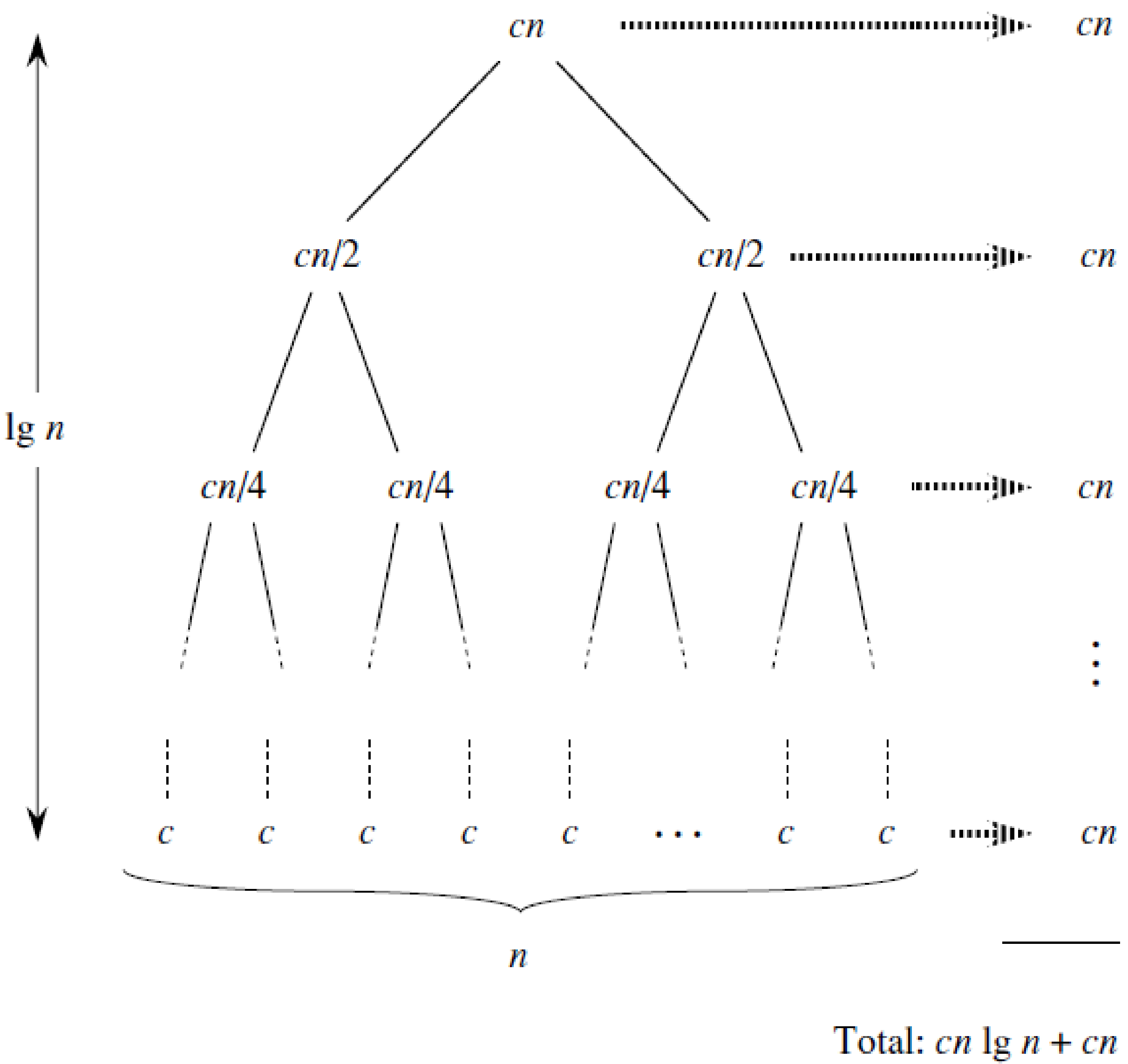
Recursion tree

- For the original problem, cost c*n+2 subproblems, each of them c*n/2 + subproblems



Recursion tree – cont.

Continue expanding until the problem sizes get down to 1:



Cost of each level

- Top level: cn
- Next level: c(n/2)+c(n/2)=cn
- Next next level: 4c(n/4)=cn

- General:
- i-th level from top has 2^i nodes
- each with cost $c(n/\ 2^i)$
- Total cost of this level: cn
- Bottom level: n nodes, each cost c

Total number of levels

- is **lgn+1**, where n: input size (number of leaves)
- Use induction to prove this
- Base case: n=1, only one level lg1 =0
- Inductive Hypothesis: number of levels with 2^i *leaves* is $\lg 2^i + 1 = i + 1$
- Prove that for $n = 2^{i+1}$ *leaves* (*power of* 2) one more level than with 2^i leaves, i.e. (i+1)+1=lg $2^{i+1} + 1$

Running time of Merge-sort

- lgn+1 levels each with cost cn ➔ cn(lgn+1)
- Ignore lower order term and c
- Θ(nlgn)

Practice

merge sort on the array

3, 41, 52, 26, 38, 57, 9, 49

Maximum Subarray Problem

- Suppose you are a freelancer and that you plan to work at a Mykonos resort for some part of the n-day summer season next year.
- Unfortunately, there isn't enough work for you to be paid every day and you need to cover your own expenses (but you want to go).
- Fortunately, you know in advance that if you are at the resort on the i^{th} day of the season, you'll make p_i euros where p_i could be negative (if expenses are more than earnings) or positive (if expenses are less than earnings).
-

Maximum Subarray Problem Example

- To maximize your earning you should choose carefully which day you arrive and which day you leave; the days you work should be consecutive and you don't need to work all season. For example, if $n = 8$ and $p_1 = -9$, $p_2 = 10$, $p_3 = -8$, $p_4 = 10$, $p_5 = 5$, $p_6 = -4$, $p_7 = -2$, $p_8 = 5$ then if you worked from day 2 to day 5, you would earn $10 - 8 + 10 + 5 = 17$ euros in total. Assume the resort pays your airtickets.

Brute force again

- Trivial if only positive numbers (assume not)
- Need to check $O(n^2)$ pairs
- For each pair, find the sum
- Thus total time is (see next)

Brute force $O(n^3)$

- Calculate the value val(i,j) for each pair i<j and return max

```
FIND-MAXIMUM-SUBARRAY-BF1(A, 1, n)
    max=A[1]
    for i=1 to n
        for j=i to n
            val=0
            for x=i to j
                val=val+A[x]
            if val>max
                max=val
    return max
```

A= [-2, -5, **6, -2, -3, 1, 5**, -6] the maximum subarray sum is 6-2-3+1+5

Can I do better?

- Save on one for loop?

Brute force O(n²) - Reuse

- Reuse previous values

```
FIND-MAXIMUM-SUBARRAY-BF2(A, 1, n)
    max=A[1]
    for i=1 to n
        val=0
        for j=i to n
            val=val+A[j]
            if val>max
                max=val
    return max
```

A= [-2, -5, **6, -2, -3, 1, 5**, -6] the maximum subarray sum is 6-2-3+1+5

Divide-and-Conquer

- A[low..high]
- Divide in the middle:
  - A[low,mid], A[mid+1,high]
- Any subarray A[i,..j] is

  (1) Entirely in A[low,mid]

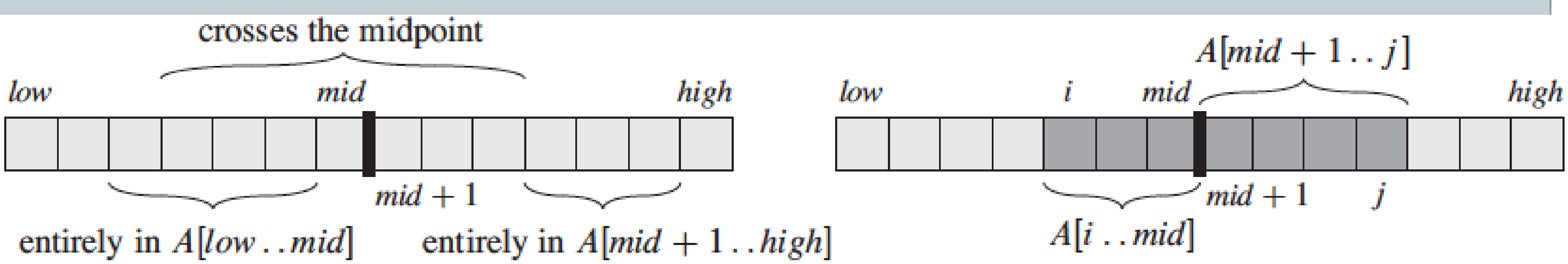
  (2) Entirely in A[mid+1,high]

  (3) In both
- (1) and (2) can be found recursively

Divide-and-Conquer (cont.)

- (3) find maximum subarray that crosses midpoint
  - Need to find maximum subarrays of the form

    A[i..mid], A[mid+1..j], low <= i, j <= high
- Take subarray with largest sum of (1), (2), (3)



(a) (b)

Divide-and-Conquer (cont.)


```
Find-Max-Cross-Subarray(A,low,mid,high)
    left-sum = -∞
    sum = 0
    for i = mid downto low
            sum = sum + A[i]
            if sum > left-sum then
                    left-sum = sum
                    max-left = i
    right-sum = -∞
    sum = 0
    for j = mid+1 to high
            sum = sum + A[j]
            if sum > right-sum then
                    right-sum = sum
                    max-right = j
return (max-left, max-right, left-sum + right-sum)
```

A= [-2, -5, **6, -2, -3, 1, 5**, -6] the maximum subarray sum is 6-2-3+1

Maximum Subarray


```
FIND-MAXIMUM-SUBARRAY(A, low, high)
  if high == low
    return (low, high, A[low])/ // base case: only one element
  else
    mid =(low + high)/2
    (left-low, left-high, left-sum)=FIND-MAXIMUM-SUBARRAY(A, low, mid)
    (right-low, right-high, right-sum)=FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
    (cross-low, cross-high, cross-sum)=FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
    if left-sum >= right-sum and left-sum >= cross-sum
      return (left-low, left-high, left-sum)
    elseif right-sum >= left-sum and right-sum >= cross-sum
      return (right-low, right-high, right-sum)
    else
      return (cross-low, cross-high, cross-sum)
```

Time analysis

- Find-Max-Cross-Subarray: $O(n)$ time
- Two recursive calls on input size $n/2$
- Thus:

  $T(n) = 2T(n/2) + O(n)$

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

Matrix Multiplication (Strassen's Algorithm)

- Another Divide and Conquer Algorithm
- Matrix Multiplication: If A =(aij) and B = (bij) are square nxn matrices, then in the product C= A *B, we define the entry c(ij) , for i,j =1,2,…n:

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

Basic Matrix Multiplication


```
for i = 1 to n
    for j = 1 to n
        for k = 1 to n
            C(i,j) = C(i,j) + A(i,k) * B(k,j)
```

algorithm

Time analysis

$$C_{i,j} = \sum_{k=1}^{n} a_{i,k} b_{k,j}$$

$$\text{Thus } T(N) = \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{n} c = cn^3 = O(n^3)$$

Basic Divide and Conquer Matrix Multiplication

Suppose we want to multiply two matrices of size *n*xn: for example $A * B = C$.

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

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

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

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

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

2x2 matrix multiplication can be accomplished in 8 multiplication.**($2^{\log_2 8} = 2^3$)**

Recurrence for the running time of the basic D&C algorithm

- Why?

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1\,, \\ 8T(n/2) + \Theta(n^2) & \text{if } n > 1\,. \end{cases}$$

Strassen's Matrix Multiplication

Strassen observed [1969] that the product of two matrices can be computed in general as follows:

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

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

Formulas for Strassen's Algorithm

$P_1 = A_{11} * (B_{12} - B_{22})$

$P_2 = (A_{11} + A_{12}) * B_{22}$

$P_3 = (A_{21} + A_{22}) * B_{11}$

$P_4 = A_{22} * (B_{21} - B_{11})$

$P_5 = (A_{11} + A_{22}) * (B_{11} + B_{22})$

$P_6 = (A_{12} - A_{22}) * (B_{21} + B_{22})$

$P_7 = (A_{11} - A_{21}) * (B_{11} + B_{12})$

How much time for computing each parenthesis (10 total): **$\Theta(n^2)$**

7 multiplications

18 additions

Analysis of Strassen's Algorithm

If n is not a power of 2, matrices can be padded with zeros.

What if we count both multiplications and additions?

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

Solution: T(n) = $n^{\log_2 7} \approx n^{2.807}$ vs. n^3 of brute-force and basic D&C alg.

(see next how to find running time easy)

Algorithms with better asymptotic efficiency are known but they are even more complex and not used in practice.

Practice: Find the maximum element of an array

- Complete the missing statements at the end and then find the function that describes the running time of this algorithm and solve it (using O-notation). Use Divide and Conquer.

```
int maxValue(A, left, right)
        if (left==right)
                return A[left]
        mid=(left+right)/2
        ans1=maxValue(left,mid)
        ans2=maxValue(mid+1,right)
        ______
        ______
```