CE100 Algorithms and Programming II

Matrix Multiplication / Quick Sort

Author: Asst. Prof. Dr. Uğur CORUH

## CE100 Algorithms and Programming II

## Week-3 (Matrix Multiplication/ Quick Sort)

#### Spring Semester, 2021-2022

Download [DOC](ce100-week-3-matrix.md_doc.pdf), [SLIDE](ce100-week-3-matrix.md_slide.pdf), [PPTX](ce100-week-3-matrix.md_slide.pptx)

## Matrix Multiplication / Quick Sort

## Outline

* Matrix Multiplication
  + Traditional
  + Recursive
  + Strassen

## Outline

* Quicksort
  + Hoare Partitioning
  + Lomuto Partitioning
  + Recursive Sorting

## Outline

* Quicksort Analysis
  + Randomized Quicksort
  + Randomized Selection
    - Recursive
    - Medians

## Matrix Multiplication

* **Input:**
* **Output:**

## Matrix Multiplication



alt:“alt” height:450px center

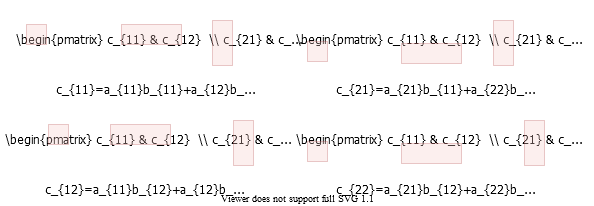
## Matrix Multiplication: Standard Algorithm

Running Time:

for i=1 to n do  
 for j=1 to n do  
 C[i,j] = 0  
 for k=1 to n do  
 C[i,j] = C[i,j] + A[i,k] + B[k,j]  
 endfor  
 endfor  
endfor

## Matrix Multiplication: Divide & Conquer

**IDEA:** Divide the matrix into matrix of submatrices.



alt:“alt” height:450px center

## Matrix Multiplication: Divide & Conquer

## Matrix Multiplication: Divide & Conquer

MATRIX-MULTIPLY(A, B)  
 // Assuming that both A and B are nxn matrices  
 if n == 1 then   
 return A \* B  
 else   
 //partition A, B, and C as shown before  
 C[1,1] = MATRIX-MULTIPLY (A[1,1], B[1,1]) +   
 MATRIX-MULTIPLY (A[1,2], B[2,1]);   
  
 C[1,2] = MATRIX-MULTIPLY (A[1,1], B[1,2]) +   
 MATRIX-MULTIPLY (A[1,2], B[2,2]);   
  
 C[2,1] = MATRIX-MULTIPLY (A[2,1], B[1,1]) +   
 MATRIX-MULTIPLY (A[2,2], B[2,1]);  
  
 C[2,2] = MATRIX-MULTIPLY (A[2,1], B[1,2]) +   
 MATRIX-MULTIPLY (A[2,2], B[2,2]);  
 endif   
  
 return C

## Matrix Multiplication: Divide & Conquer Analysis

* recursive calls
* each problem has size
* Submatrix addition

## Matrix Multiplication: Solving the Recurrence

* + ,
* Case 1:

Similar with ordinary (iterative) algorithm.

## Matrix Multiplication: Strassen’s Idea

Compute using recursive multiplications.

In normal case we need as below.

## Matrix Multiplication: Strassen’s Idea

* **Reminder:**
  + Each submatrix is of size
  + Each add/sub operation takes time
* Compute using recursive calls to matrix-multiply

$P\_1 = a\_{11} \* (b\_{12} - b\_{22} ) \\ P\_2 = (a\_{11} + a\_{12} ) \* b\_{22} \\ P\_3 = (a\_{21} + a\_{22} ) \* b\_{11} \\ P\_4 = a\_{22} \* (b\_{21} - b\_{11} ) \\ P\_5 = (a\_{11} + a\_{22} ) \* (b\_{11} + b\_{22} ) \\ P\_6 = (a\_{12} - a\_{22} ) \* (b\_{21} + b\_{22} ) \\ P\_7 = ( a\_{11} - a\_{21} ) \* (b\_{11} + b\_{12} )$

## Matrix Multiplication: Strassen’s Idea

$P\_1 = a\_{11} \* (b\_{12} - b\_{22} ) \\ P\_2 = (a\_{11} + a\_{12} ) \* b\_{22} \\ P\_3 = (a\_{21} + a\_{22} ) \* b\_{11} \\ P\_4 = a\_{22} \* (b\_{21} - b\_{11} ) \\ P\_5 = (a\_{11} + a\_{22} ) \* (b\_{11} + b\_{22} ) \\ P\_6 = (a\_{12} - a\_{22} ) \* (b\_{21} + b\_{22} ) \\ P\_7 = ( a\_{11} - a\_{21} ) \* (b\_{11} + b\_{12} )$

* How to compute using ?

$c\_{11} = P\_5 + P\_4 – P\_2 + P\_6 \\ c\_{12} = P\_1 + P\_2 \\ c\_{21} = P\_3 + P\_4 \\ c\_{22} = P\_5 + P\_1 – P\_3 – P\_7$

## Matrix Multiplication: Strassen’s Idea

* recursive multiply calls
* add/sub operations

## Matrix Multiplication: Strassen’s Idea

e.g. Show that

$c\_{12} = P\_1 + P\_2 \\ = a\_{11}(b\_{12}–b\_{22})+(a\_{11}+a\_{12})b\_{22} \\ = a\_{11}b\_{12}-a\_{11}b\_{22}+a\_{11}b\_{22}+a\_{12}b\_{22} \\ = a\_{11}b\_{12}+a\_{12}b\_{22}$

## Strassen’s Algorithm

* **Divide:** Partition and into submatrices. Form terms to be multiplied using and .
* **Conquer:** Perform multiplications of submatrices recursively.
* **Combine:** Form using and on submatrices.

**Recurrence:**

## Strassen’s Algorithm: Solving the Recurrence

* + ,
* Case 1:

so

or use https://www.omnicalculator.com/math/log

## Strassen’s Algorithm

* The number may not seem much smaller than
* But, it is significant because the difference is in the exponent.
* Strassen’s algorithm beats the ordinary algorithm on today’s machines for or so.
* Best to date: (of theoretical interest only)

## Maximum Subarray Problem

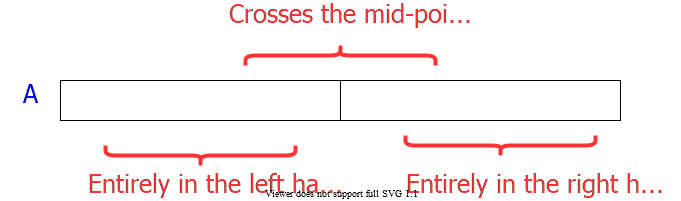
**Input:** An array of values **Output:** The contiguous subarray that has the largest sum of elements

* Input array:

## Maximum Subarray Problem: Divide & Conquer

* **Basic idea:**
* **Divide** the input array into 2 from the middle
* Pick the **best** solution among the following:
  + The max subarray of the **left half**
  + The max subarray of the **right half**
  + The max subarray **crossing the mid-point**

## Maximum Subarray Problem: Divide & Conquer



alt:“alt” height:450px center

## Maximum Subarray Problem: Divide & Conquer

* **Divide:** Trivial (divide the array from the middle)
* **Conquer:** Recursively compute the max subarrays of the left and right halves
* **Combine:** Compute the max-subarray crossing the
  + (can be done in time).
  + Return the max among the following:
    - the max subarray of the
    - the max subarray of the
    - the max subarray crossing the

TODO : detailed solution in textbook…

## Conclusion : Divide & Conquer

* Divide and conquer is just one of several powerful techniques for algorithm design.
* Divide-and-conquer algorithms can be analyzed using recurrences and the master method (so practice this math).
* Can lead to more efficient algorithms

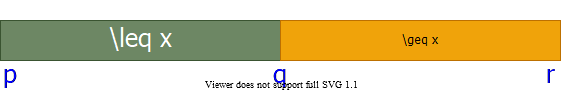
## Quicksort

* One of the most-used algorithms in practice
* Proposed by **C.A.R.** *Hoare* in 1962.
* Divide-and-conquer algorithm
* In-place algorithm
  + The additional space needed is O(1)
  + The sorted array is returned in the input array
  + *Reminder: Insertion-sort is also an in-place algorithm, but Merge-Sort is not in-place.*
* Very practical

## Quicksort

* **Divide:** Partition the array into 2 subarrays such that elements in the lower part elements in the higher part
* **Conquer:** Recursively sort 2 subarrays
* **Combine:** Trivial (because in-place)

**Key:** Linear-time partitioning algorithm



alt:“alt” height:200px center

## References

TODO