DIVIDE & CONQUER

Athasit Surarerks

Divide & conquer

Introduction

Divide and conquer is a technique for designing of algorithms by divide a problem (large size) into many small problems that is easier to be solved. The whole solution can be obtained by combining all solutions of small problems.

Template

```
Solve(I)
begin
        n ← size(I)
        if ( n ≤ smallsize ) then solution ← DirectSolve(I)
                else    divide I into I1 I2 … Ik
                        for j ← 1 step 1 until k do
                                solution(Ij) ← Solve (Ij)
                        enddo
                        solution ← combine(all solution(I))
        endif
        return ← solution
end
```

Complexity

Let k be the number of smaller instances into which the input divided, where n_j is the size of the instance j,

D(n) be run time used by divide,

C(n) be run time used by combine.

The general form of the recurrence equation that describes the amount of work done by the algorithm is

$$T(n) = D(n) + \sum_{0 \le j \le k} T(n_j) + C(n), \text{ for } n > \text{smallsize}.$$

It is clear that for any $n \le$ smallsize, time used T(n) is the time used by DirectSolve.

Binary search

An algorithm to search a sorted array. It begins with an interval covering the whole array. If the search value is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

Binary search

Algorithm BinarySearch
Input: Array A[1..n] with n elements
k is the search key
Output: The index j such that A[j] = k

```
BinarySearch(A[1..n],k)
begin
    if ( n < 1 ) then return −1
    else
        split ← (n+1)/2
        if ( k = A[split] ) then return split
        else     if ( k < A[split] ) then return BinarySearch(A[1..split−
                 else return BinarySearch(A[split+1..n],k) endif
    endif
end
```

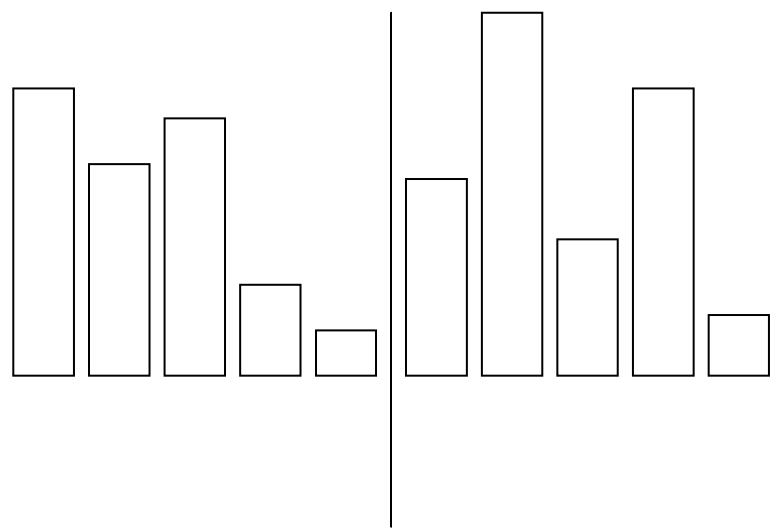
Binary search

Worst-Case Analysis of Binary Search

Let us define the problem size of BinarySearch as n, the number of entries in the range of Array A to be searched. How many times can we divide n by two without getting a result less than one? In other words, what is the largest d for which $n/2^d \geq 1$? We solve for d: $2^d \leq n$ and $d \leq \lg(n)$. Therefore we can do $\lfloor \lg(n) \rfloor$ comparisons following recursive calls, and one comparison before any recursive calls, for at most $\lfloor \lg(n) \rfloor + 1$ comparisons in all. Thus the running time is $\Theta(\log n)$.

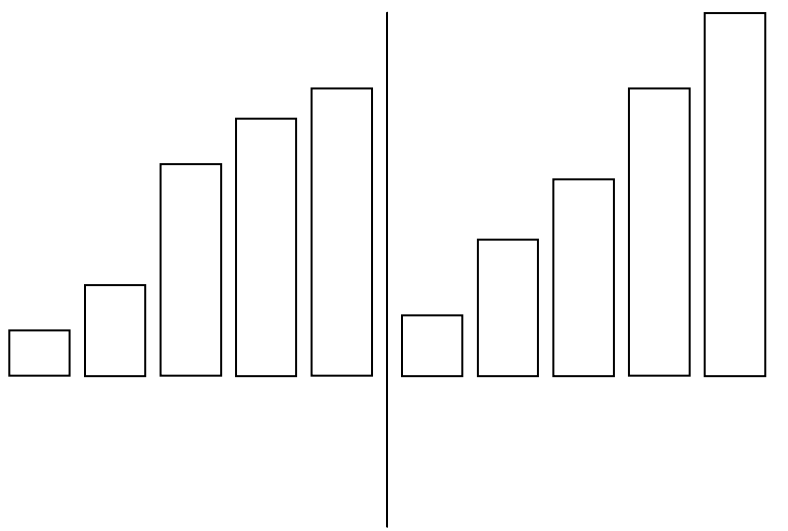
MergeSort

Given two sequences A and B sorted in nondecreasing order, merge them to create one sorted sequence C. Merging sorted subsequences is essential to the strategy of Mergesort.



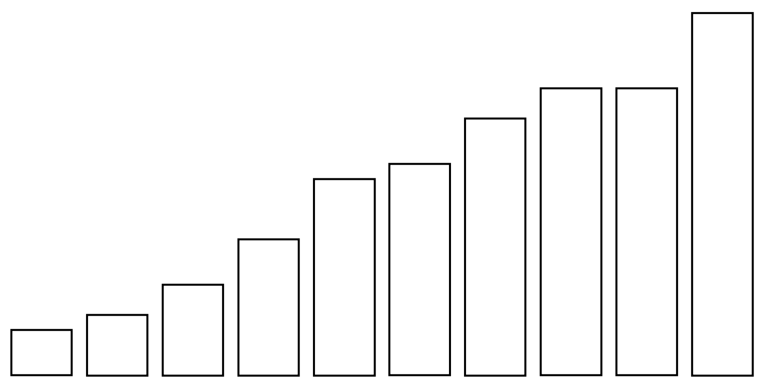
MergeSort

Given two sequences A and B sorted in nondecreasing order, merge them to create one sorted sequence C. Merging sorted subsequences is essential to the strategy of Mergesort.



MergeSort

Given two sequences A and B sorted in nondecreasing order, merge them to create one sorted sequence C. Merging sorted subsequences is essential to the strategy of Mergesort.



MergeSort

Merging sorted sequences (times)

Whenever a comparison of keys from A and B is done, at least one element is moved to C and never examined again. After the last comparison, at least two elements have not yet been moved to C. The greater one is moved immediately, but now C has at most n-1 elements, and no more comparisons will be done. Those that remain in the other array are moved to C without any further comparisons. So at most n-1 comparisons are done. The worst case, using all n-1 comparisons, occurs when A[1] and B[1] belong in the first two positions in C.

MergeSort

Merging sorted sequences (space)

It might appear from the way in which Merge algorithm is written that merging sequences with a total of n entries requires enough memory locations for 2n entries, since all entries are copied to C. In some cases, however, the amount of extra space needed can be decreased. One case is that the sequences are linked lists, and A and B are not needed (as lists) after the merge is completed. Then the list nodes of A and B can be recycled as C in created.

MergeSort

Algorithm MergeSort

Input: Array A[1..n] with n elements

Output: Array A[1..n] with n elements and for 2≤j ≤ n, A[j-1] ≤ A[j].

```
Mergesort(A[1..n])
begin
    if ( 1 < n ) then
        miditem ← ⌊ (1+n)/2⌋
        Mergesort(A[1..miditem])
        Mergesort(A[miditem+1..n])
        Merge(A[1..miditem],A[miditem+1..n],A[1..n])
    endif
end
```

MergeSort

Mergesort analysis

First, we find the asymptotic order of the worst-case number of key comparisons for Mergesort. As usual, we define the problem size as n, the number of elements in the range to be sorted. The recurrence equation for the worst-case behavior of Mergesort is

$$t(n) = t(\lfloor n/2 \rfloor) + t(\lceil n/2 \rceil) + n - 1,$$

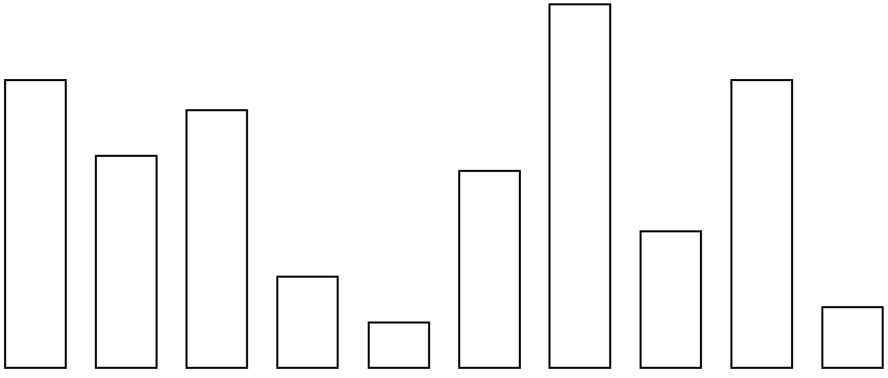
and $t(1) = 0$.

The master method tells us immediately that $t(n) = \Theta(n \log n)$. So we finally have a sorting algorithm whose worst-case behavior is in $\Theta(n \log n)$.

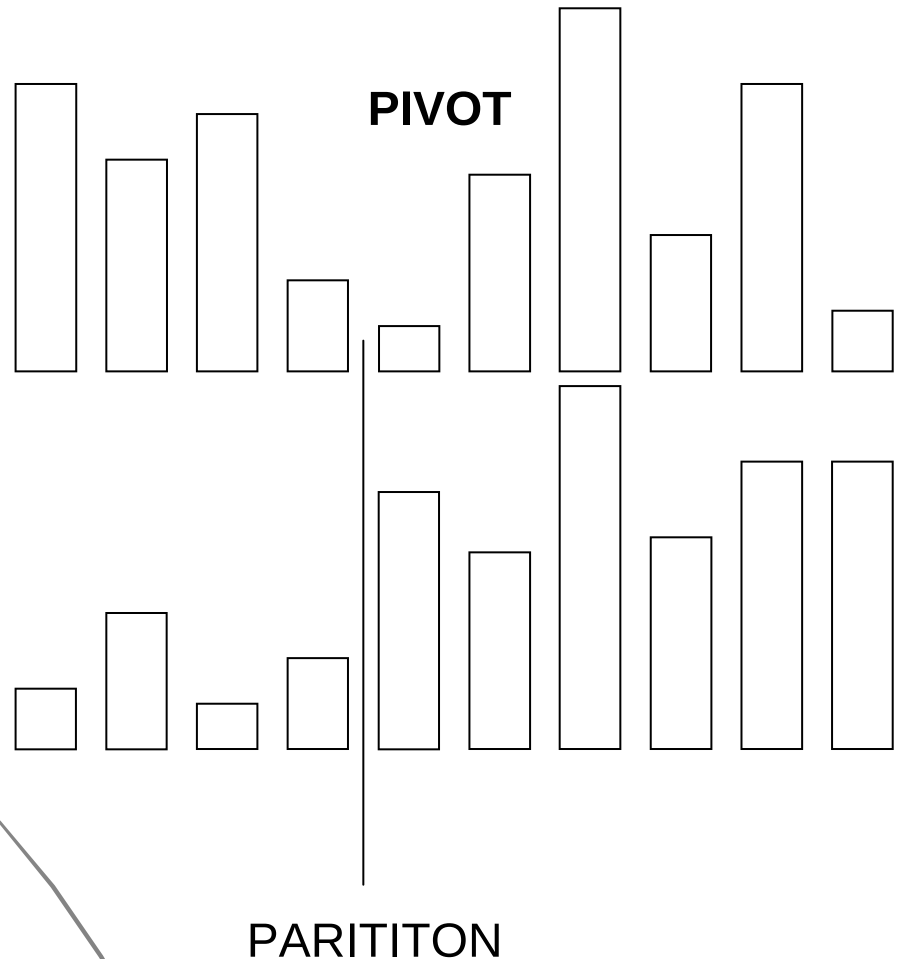
Quicksort

An in-place sort algorithm that uses the divide-and-conquer paradigm. It picks an element from the array (the pivot), partitions the remaining elements into those greater than and less than this pivot, and recursively sorts the partitions. There are many variants of the basic scheme above: to select the pivot, to partition the array, to stop the recursion on small partitions, etc.

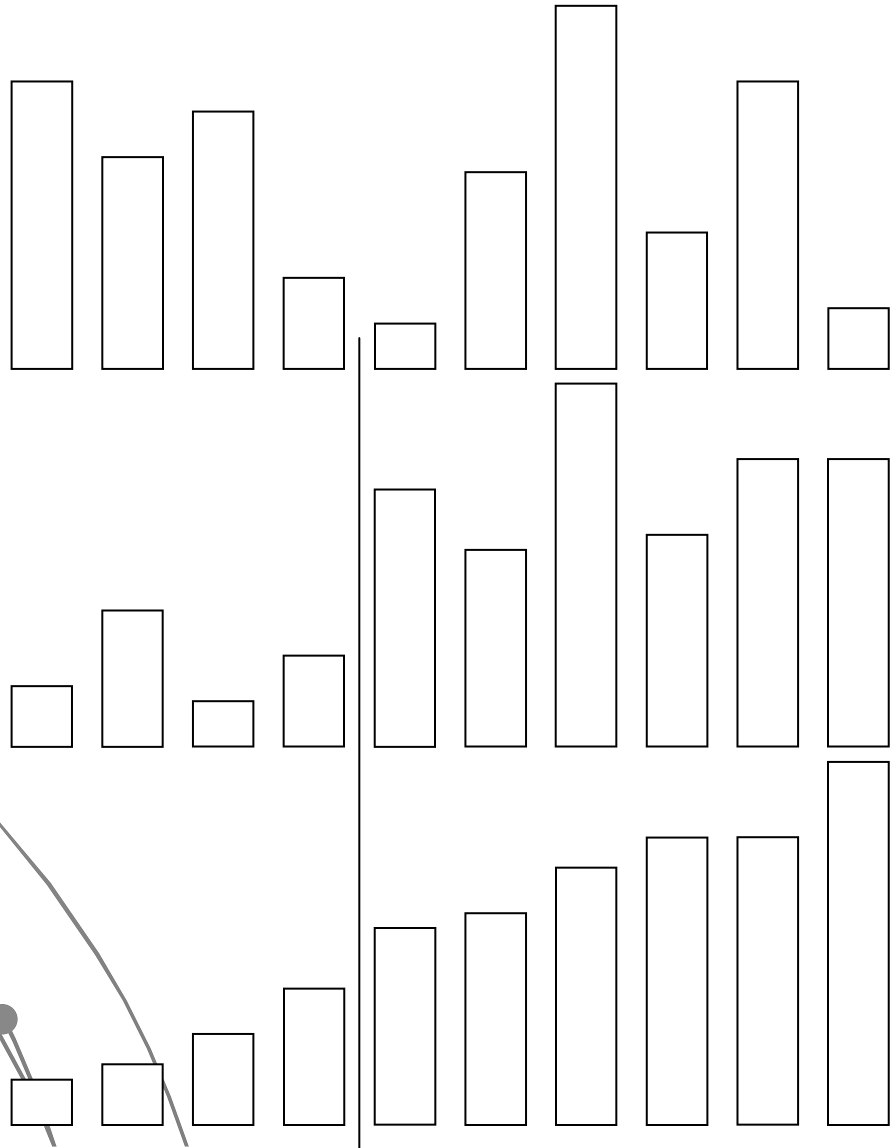
Quicksort



Quicksort



Quicksort



QUICKSORT

QUICKSORT

Quicksort

Algorithm Quicksort
Input: Array A[1..n] with n elements
Output: Array A[1..n] with n elements and for 2≤j≤n, A[j-1] ≤ A[j].

```
Quicksort(A[1..n])
begin
    splitpoint ← Partition(A[1..n])
    if ( splitpoint ≠ first ) then
        Quicksort(A[first..splitpoint])
    endif
    if ( splitpoint ≠ last ) then
        Quicksort(A[splitpoint+1..last])
    endif
end
```

Quicksort

PARTITION: Pivot = 5.

5	4	7	1	8	9	3	11	2	12

2	4	7	1	8	9	3	11	5	12

2	4	7	1	8	9	3	11	5	12

2	4	3	1	8	9	7	11	5	12

2	4	3	1	8	9	7	11	5	12

Quicksort

Worst case: Partition compares each key to pivot, so if there are n positions in the range of the array it is working on, it does n-1 key comparisons. If pivot is the smallest key, and all that has been accomplished is splitting the range into an one element subrange and a subrange with n-1 elements. Thus, if pivot is the smallest key each time Partition is called, then the total number of key comparisons done is

$$\sum_{1 \le j \le n} (j-1) = n(n-1)/2$$

and time used = $\Theta(n^2)$.

Quicksort

Average Behavior: We assume that the keys are distinct and that all permutations of the keys are equally likely. Let k be the number of elements in the left subrange (then n-k be the number of elements in the another subrange). It means that pivot is the (k+1)th element of the array (after sorted). Each possible position for the split point k is equally likely (has probability 1/k) so, letting k=n and t(n) be the number of comparisons done for range of this size, we have the recurrence equation

$$
\begin{aligned}
t(n) &= (1/n)\left(\sum_{1\le k\le n-1} (t(k) + t(n-k)) + t(1) + t(n-1) \right) + \Theta(n). \\
&= (1/n)\left(\sum_{1\le k\le n-1} t(k) + \sum_{1\le k\le n-1} t(n-k) \right) + \Theta(n). \\
&= (2/n)\left(\sum_{1\le k\le n-1} t(k) \right) + \Theta(n).
\end{aligned}
$$

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

Selection problem

Suppose that A is an array containing n elements with keys from some linearly ordered set, and let k be an integer such that $1 \le k \le n$. The selection problem is the problem of finding an element with the k^{th} smallest key in A. Such an element is said to have rank k.

Selection problem

A Divide-and-Conquer Approach

Suppose we can partition the keys into two sets, S_1 and S_2, such that all keys in S_1 are smaller than all keys in S_2. Then we know that the k^{th} element is in S_1 or S_2, and we can ignore the other set and restrict our search to the larger set.

Selection problem

Example: Partitioning in search of the median

Suppose n=255 be size of the problem. We are seeking the median element (whose rank k=128). Suppose after partitioning, that S_1 has 96 elements and S_2 has 159 elements. Then the median of the whole set is in S_2, and it is the 32^{nd}-smallest element in S_2. Thus the problem reduces to finding the element of rank 32 in S_2, which has 159 elements.

Quick selection

Algorithm QuickSelect

Input: Array A[1..n] with n elements

k is an integer such that $1 \le k \le n$

Output: The kth smallest element of A

```
QuickSelect(A[1..n],k)
begin
    if ( n = 1 ) then return A[1]
    else
        split ← RandomizedPartition(A[1..n])
        if ( k ≤ split ) then return QuickSelect(A[1..split],k)
        else return QuickSelect(A[split+1..n],k-split)
    endif
end
```

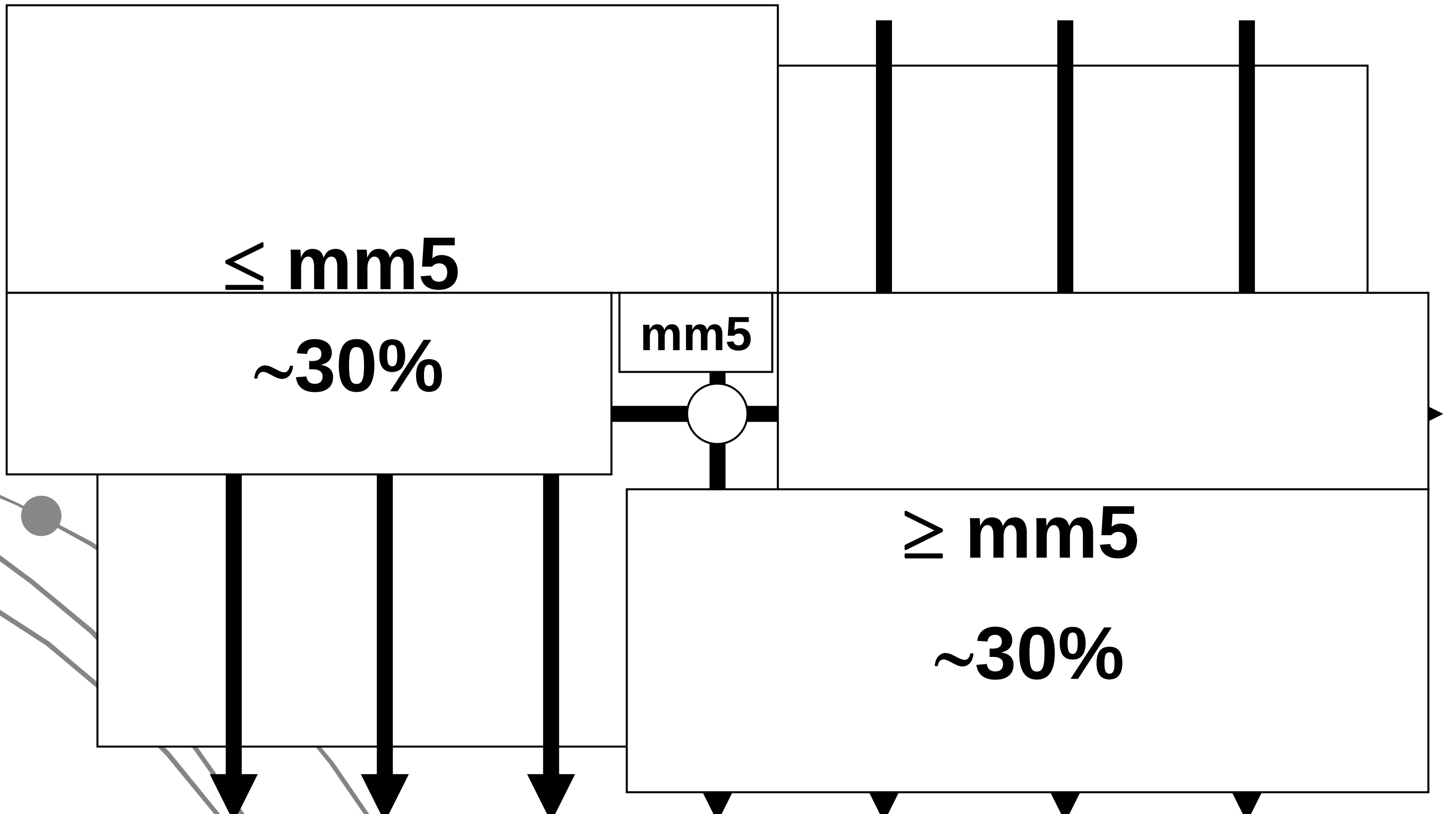
Quick selection

Let t(n) be time used for QuickSelect of n elements, then

$$
\begin{aligned}
t(n) &\le t(\max(k,n-k)) + \Theta(n) \\
&\le (1/n) \left(\textstyle\sum_{1\le k\le n} t(\max(k,n-k)) \right) + \Theta(n) \\
&= (1/n) \left(t(n-1) + \textstyle\sum_{1\le k\le n-1} t(\max(k,n-k)) \right) + \Theta(n) \\
&= (2/n) \left(\textstyle\sum_{\lceil n/2 \rceil\le k\le n-1} t(k) \right) + \Theta(n) \\
&= O(n).
\end{aligned}
$$

Note: In the worst case, $t(n) = \Theta(n^2)$.

MM5



MM5

Smaller problem **partition** **selection**

$$
\begin{aligned}
T(n) &\le T(0.7n) + \Theta(n) + \Theta(n) + T(n/5) + \Theta(n) \\
&= T(0.7n) + T(0.2n) + \Theta(n) \\
&\le \Theta(n)
\end{aligned}
$$

medians **removed**

MM5

$$
\begin{aligned}
T(n) &\le T(0.7n) + \Theta(n) + \Theta(n) + T(n/5) + \Theta(n) \\
&= T(0.7n) + T(0.2n) + \Theta(n) \\
&\le \Theta(n)
\end{aligned}
$$

Modular

Find the value of

$$\mathbf{a^k \bmod n}$$

using divide & conquer technique.

Fact:

$a^k \bmod n = (a^{k/2} \bmod n)^2 \bmod n$: k=even

$a^k \bmod n = a(a^{\lfloor k/2 \rfloor} \bmod n)^2 \bmod n$: k=odd

Modular

170 $2^{370} \bmod 371 = (2^{185} \bmod 371)^2 \bmod 371$

151

$2^{185} \bmod 371 = 2(2^{92} \bmod 371)^2 \bmod 371$

46

$2^{92} \bmod 371 = (2^{46} \bmod 371)^2 \bmod 371$

135 $2^{46} \bmod 371 = (2^{23} \bmod 371)^2 \bmod 371$

298 $2^{23} \bmod 371 = 2(2^{11} \bmod 371)^2 \bmod 371$

193 $2^{11} \bmod 371 = 2(2^{5} \bmod 371)^2 \bmod 371$

32

Matrix Multiplication

$A_{1,1}$ $A_{1,2}$ $B_{1,1}$ $B_{1,2}$

A B

$A_{2,1}$ $\times$

ma ma

$$\boxed{T(n) = 8T(n/2) + \Theta(n^2)}$$

$B_{1,1} + A_{1,2}\,B_{2,1}$

$A_{1,1}\,B_{1,2} + A_{1,2}\,B_{2,2}$

$A_{2,1}\,B_{1,1} + A_{2,2}\,B_{2,1}$

$A_{2,1}\,B_{1,2} + A_{2,2}\,B_{2,2}$

Matrix Multiplication

STRASSEN 1968

C_{11}	C_{12}
C_{21}	C_{22}

$M_1 = (A_{12}-A_{22})(B_{21}+B_{22})$

$M_2 = (A_{11}+A_{22})(B_{11}+B_{22})$

$M_3 = (A_{11}-A_{21})(B_{11}+B_{12})$

$M_4 = (A_{11}-A_{12})B_{22}$

$M_5 = A_{11}(B_{12}-B_{22})$

M_6

$M_7 = (A_{21}+A_{22})B_{11}$

$C_{11} = M_1+M$

$C_{21} = M_8+M_7$

$C_{22} = M_2-M_3+M_5-M_7$

$$\boxed{T(n) = 7T(n/2) + \Theta(n^2) = O(n^{2.81})}$$

SuperStar

Find a superstar in the party

Superstar

1. All people know the superstar.
2. The superstar knows nobody.

Exercises

$\sqrt{40}$

Exercises

Draw an algorithm for finding the k^{th} element in the list using MM5.

$\sqrt{40}$

Draw an algorithm for computing the multiplication of two n-square matrices.

Draw an algorithm for finding the superstar.

$\sqrt{40}$

Exercises

Using a divide and conquer technique to find the value of

$$\sqrt{40}$$

$\sqrt{40}$

a	b	x=(a+b)/2	x^2-n
0	40	20	360
0	20	10	60
0	10	5	-15
5	10	7.5	16.25
5	7.5	6.25	-0.9375
6.25	7.5	6.875	7.265625
6.25	6.875	6.5625	3.066406
6.25	6.5625	6.40625	1.040039

Exercises

Given a set of real numbers { a_1, a_2, a_3,…,a_n}.
Is there a pair of a_i and a_j satisfying

$\sqrt{40}$

$$a_i + a_j = k$$

for a fixed number k ?

Find an algorithm with O(n log n) to answer this question.

Exercises

Given a set of numbers { a_1, a_2, a_3,…,a_n}.

$\sqrt{40}$

Find an algorithm with O(n log n) to find a mode element in this set.

(In the case that there are more than one mode, only one mode is chosen to be the answer.)

Exercises

Given a sorted list a_1 a_2 a_3 … a_n.

$\sqrt{40}$

Suppose that the list is rotated by an unknown k,

i.e., a_{n-k+1} a_{n-k+2} … a_n a_1 a_2 … a_{n-k}

Find an algorithm with O(log n) for finding the maximum number in the list.