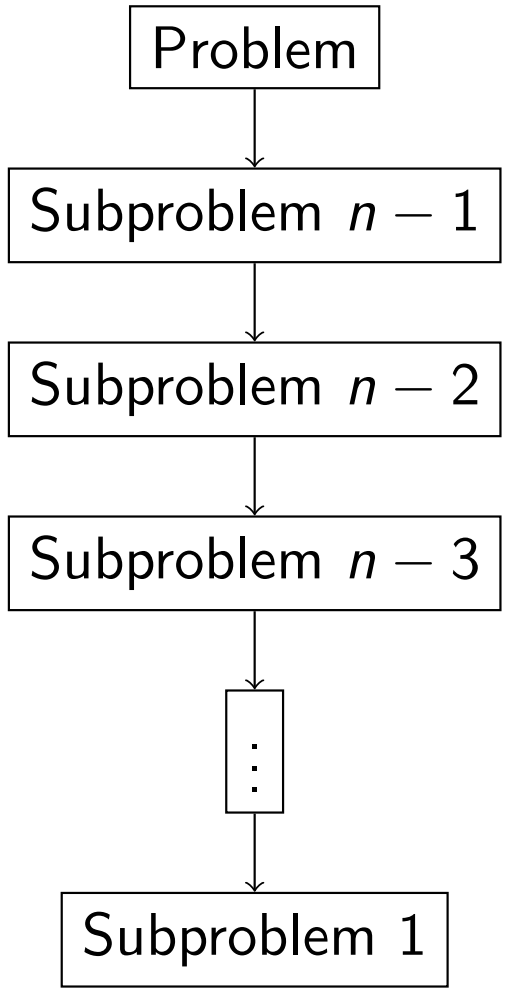
CSE 431/531: Analysis of Algorithms (Summer 2023) Divide and Conquer

Chen Xu

June 22-29, 2023

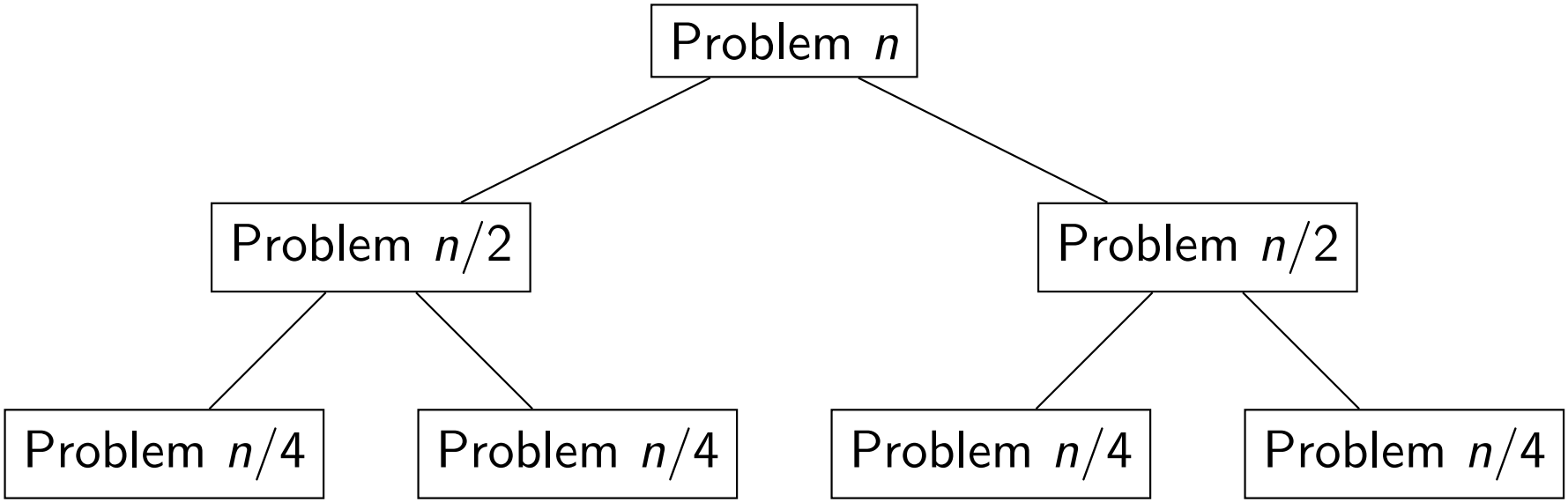
Subproblem structure

- Greedy algorithm:



Subproblem structure

- Divide-and-Conquer(DaC):



Comparison of two strategies

- A working greedy algorithm usually cuts the time from $O(2^n)$ to $O(n^c)$ for some constant c. A DaC usually cuts $O(n^k)$ down to $O(n^{k-1} \log n)$.
- The proof of correctness is the major focus for greedy algorithms. DaC focuses more on running time.
- A greedy algorithm induces on one smaller instance. A DaC induces on multiple smaller instances.

Divide and Conquer

A DaC strategy contains three parts:

- **Divide:** Recursively divide problems into smaller disjoint subproblems.
- **Conquer:** Solve the smaller instances.
- **Combine:** Map the solution to the solutions from the smaller instances.

Recall from the previous lecture

- We have learned the merge sort.
  1. **Divide:** It has two recursive calls to the subproblem.
  2. **Conquer:** Solve the subproblem of half size.
  3. **Combine:** Combine the solution using Merge() procedure.
- Merge sort reduced running time of insertion sort from $O(n^2)$ to $O(n \log n)$.

Inversion Counting

Definition

An *inversion* is a pair (i, j) of indices of array A such that $i < j$ and $A[i] > A[j]$.

Inversion Counting problem

Input: An integer array A

Output: Number of inversions.

Example

Example

$A = [50, 75, 12, 90, 22, 35, 67, 18, 98, 45]$

- Inversion pairs are the indices of
  (50,12),(50,22),(50,35),(50,18),(50,45),(75,12),(75,22),
  (75,35),(75,67),(75,18),(75,45),(90,22),(90,35),(90,67),
  (90,18),(90,45),(22,18),(35,18),(67,18),(67,45),(98,45).
  So there are 21 pairs of inversions.

Trivial algorithm

Inversion counting

```
1: CountInversions(A)
2:     count ← 0                          ▷ Initialize inversion count as 0
3:     for i ← 1 to n − 1 do
4:         for j ← i + 1 to n do
5:             if A[i] > A[j] then
6:                 count ← count + 1
7:     return count
```

Using DaC

- Divide the array into two halves.
- The total number of inversion pairs equals the sum of the inversion pair number of **left half** and **right half** plus the number of inversion pairs **across two halves**.
- We can compute the number of inversion across two halves easily if both left half and right half are sorted.

Computing the number of inversions across

- We can modify the merge procedure from mergesort.
- When it is copying an element from the left array into the new sorted array, we know how many elements have already been processed in the right half.
- The processed ones are the ones that are smaller. So we know that the inversion count at that moment, must increase by the amount of those elements.

MergeCount

MergeCount

```
1: MergeCount(L, R, l, r)
2:     Create array A of size l + r
3:     Initialize variables: i ← 1, j ← 1, Count ← 0
4:     while i ≤ l or j ≤ r do
5:         if j > r or (i ≤ l and L[i] ≤ R[j]) then
6:             A[i + j − 1] ← L[i], i ← i + 1, Count ← Count + (j − 1)
7:         else
8:             A[i + j − 1] ← R[j], j ← j + 1
9:     return A, Count
```

DaC Inversion Counting algorithm

Inversion Counting

1: $\textsc{MergeSortCount}(A,\ n)$
2: $\quad inv \leftarrow 0$
3: $\quad$ **if** $n == 1$ **then return** $(A, 0)$
4: $\quad$ **else**
5: $\quad\quad (L, m_1) \leftarrow \textsc{MergeSortCount}(A[0..mid],\ mid)$
6: $\quad\quad (R, m_2) \leftarrow \textsc{MergeSortCount}(A[mid+1, r],\ r - mid)$
7: $\quad\quad (A, m_3) \leftarrow \textsc{MergeCount}(L,\ R,\ |L|,\ |R|)$
8: $\quad$ **return** $(A, m_1 + m_2 + m_3)$

- Running time is identical to mergesort $O(n \log n)$.

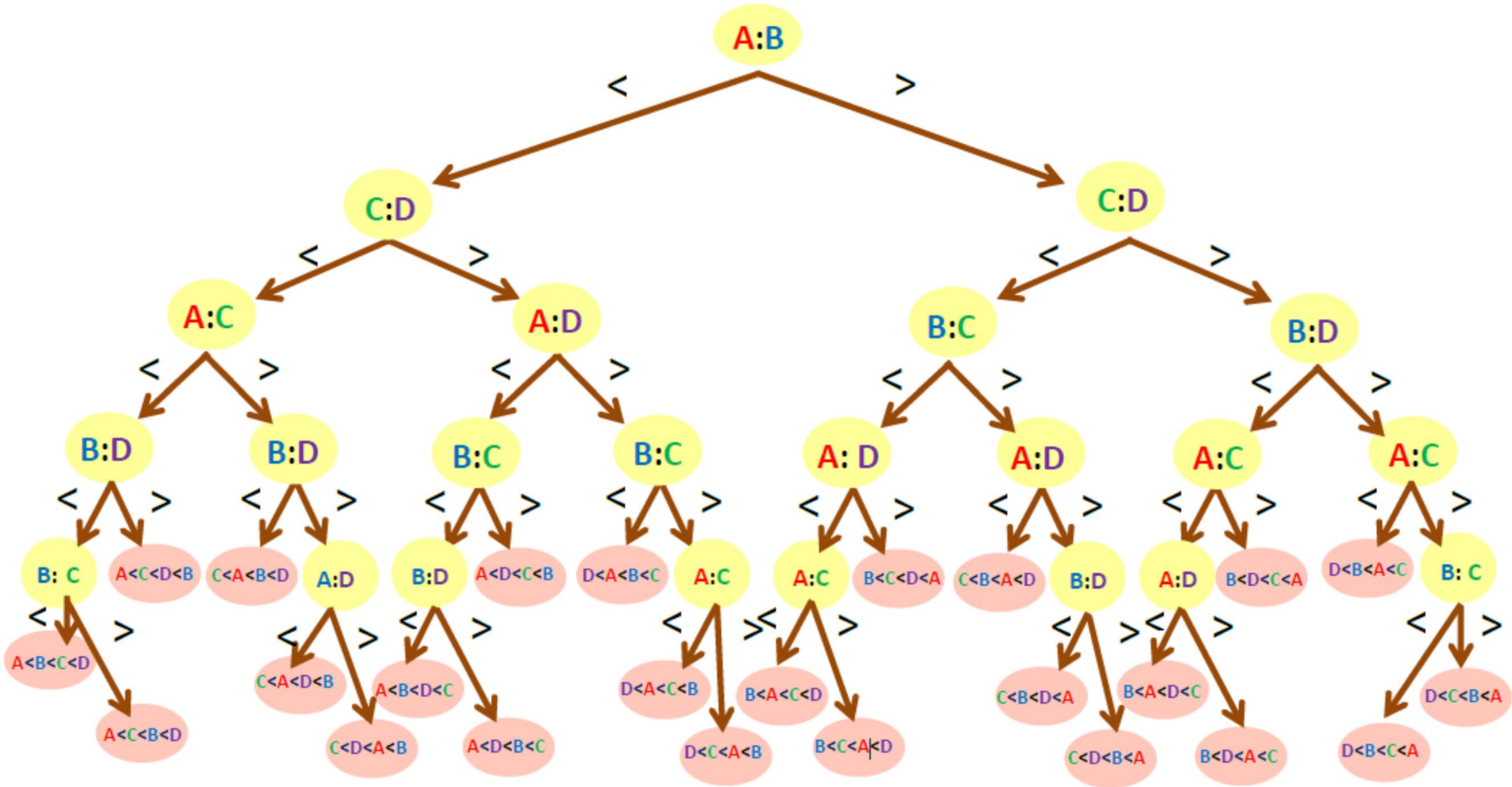
Better Sorting algorithm?

- For MergeSort, at every layer of recursion, the Merge() procedure will create a new array to store the temporary sorted array.
- This is an extra $O(n)$ usage of space.
- We may want to two questions:
  1. Is $O(n \log n)$ the best we can do for comparison based sorting algorithms?
  2. Can we reduce the space usage? While still having $O(n \log n)$ running time?

Lower bound of comparison-based sorting algorithm

- The comparison operation is the only way to resolve the order of two elements. We cannot take advantage of any predefined order on the element domain.
- For n elements, there are $O(n!)$ many different permutations.
- But for one comparison, there can only be 3 outcomes: $<, >, =$.

Decision tree for 4 elements (non-duplicated)



Lower bound of the optimum decision tree for sorting

- Each permutation is placed on one of the leaf nodes.
- We know that to organize m numbers within a binary tree, $\Omega(\log m)$ tree depth is required.
- Imagine each permutation corresponds to a unique number, to organize all of them, $\Omega(\log n!)$ depth is required.
- By Stirling's formula, $\Omega(\log n!)$ is roughly $\Omega(n \log n)$.
- So any comparison based sorting algorithm that runs in worst case $O(n \log n)$ is optimal.

In-place

- But being optimal in running time does not mean the space usage is optimal.

Definition

An algorithm is an *In-place* algorithm if the space usage is $O(\log n)$. This does not count the input space.

- How small is that space usage? This means a constant number of registers indexing the whole input. When input is bounded, for example, size is 2^{32}, then a 32-bit register can index all of the content.
- Usually we consider this situation so we say in-place algorithms use constant space or we say the algorithm does not use extra space, since the usage is too small that it is negligible.

Quick sort is an in-place sorting algorithm

- The quick sort algorithm operates on the input data directly.
- In each recurrence, quick sort chooses a pivot element by some strategy and move all smaller elements before the pivot and larger ones after the pivot.

pivot choosing example

4,3,1,6,8,7,2,5 – We just pick the first one.
3,1,2,4,6,8,7,5 – All smaller than 4 upfront, all larger than 4 behind.
3,1,2,4,6,8,7,5 – We then recur on these two parts.

The best scenario and the worst

- The best scenario is that every pivot we pick is the median element of that array. This will precisely divide the array in a half. In this case, the time complexity is $O(n \log n)$.
- But if we are extremely unlucky, for example, the worst case is when we choose the first element but the whole array is in descending order. This will make every split extremely unbalanced. The worst case therefore is not better than any $O(n^2)$ sorting algorithm.
- Solution: **Randomly pick an element as pivot.** This gives us average case running time $O(n \log n)$.

In-place Quicksort

Quick sort with random pivot

```
1: QuickSort(A, low, high)
2:     if low < high then
3:         pivot ← Partition(A, low, high)
4:         QuickSort(A, low, pivot − 1)
5:         QuickSort(A, pivot + 1, high)
```

Partition Function

Partition Function

```
1:  PARTITION(A, low, high)
2:      p ← random(low, high)
3:      i ← low; SWAP(A[low], A[p])
4:      while True do
5:          while i < j and A[i] < A[j] do j ← j − 1
6:          if i = j then Break;
7:          SWAP(A[i], A[j]), i ← i + 1
8:          while i < j and A[i] < A[j] do i ← i + 1
9:          if i = j then Break;
10:         SWAP(A[i], A[j]), j ← j − 1
11:     return i + 1
```

Selection Problem

Selection Problem

Input: An array A of n numbers
Output: The ith largest number

- Naive approach: Sort the array and return the ith element. Running time is $O(n \log n)$.
- Can we do better?

Modifying QuickSort to solve the problem

- During each recurrence of the quick sort algorithm, we divide the array into two parts with respect to the pivot element.
- After partition, the position of the pivot element is identical to the position in the sorted list. Suppose the position is at p, we know that pivot element is the pth element.
- Therefore we can decide which part the ith element is in.

Selection algorithm

Selection algorithm

```
1:  SELECT(A, low, high, i)
2:      if low = high then
3:          return A[low]
4:      pivot ← PARTITION(A, low, high)
5:      if i = pivot − low + 1 then
6:          return A[pivot]
7:      else if i < pivot − low + 1 then
8:          return SELECT(A, low, pivot − 1, i)
9:      else
10:         return SELECT(A, pivot + 1, high, i − pivot + low − 1)
```

- Recursion is $T(n) = T(n/2) + O(n)$
- Solving it gives us running time $O(n)$.

DaC algorithms can give uncommon running times.

- DaC Polynomial Multiplication – $O(n^{\log_2 3})$
- Strassen's Matrix Multiplication – $O(n^{\log_2 7})$

Polynomial multiplication

Polynomial multiplication

Input: Two polynomial functions of maximum degree n

Output: The product of the two polynomial functions.

Example

Input:

$$p(x) = x^2 + 2x + 3 = [1, 2, 3][x^2, x, 1]^T$$
$$q(x) = 4x^2 + 5x + 6 = [4, 5, 6][x^2, x, 1]^T$$

Output:

$$p(x)q(x)$$
$$= 4x^4 + 5x^3 + 6x^2 + 8x^3 + 10x^2 + 12x + 12x^2 + 15x + 18$$
$$= 4x^4 + 13x^3 + 28x^2 + 27x + 18$$
$$= [4, 13, 28, 27, 18][x^4, x^3, x^2, x, 1]^T$$

We can just consider polynomials as the coefficient arrays.

Trivial algorithm

Polynomial multiplication

```
1: MULTIPLY(A, B)
2:     C ← [0]
3:     for i ← 0 to n do
4:         for j ← 0 to n do
5:             C[i + j] ← C[i + j] + A[i] · B[j]
6:     return C
```

The algorithm has a running time of $O(n^2)$.

Polynomial Multiplication DaC trick

- Will DaC help?
  - If we divide coefficient arrays A, B into four halves: A_H, A_L, B_H, B_L. In order to compute the product C, we will need to call **four** times subproblems because

$$C = Multiply(A_H, B_H)x^n + Multiply(A_H, B_L)x^{n/2} + Multiply(A_L, B_H)x^{n/2} + Multiply(A_L, B_L)$$

  - We shrink the problem size to a half but at the same time we increase the number of calls to the subproblems. The recursion is $T(n) = 4T(n/2) + O(n)$. We still get $O(n^2)$ running time.
  - But we know that the polynomial addition can be done very fast.

Polynomial Multiplication DaC trick

- $$C = Multiply(A_H, B_H)x^n + (Multiply(A_H, B_L) + Multiply(A_L, B_H))x^{n/2} + Multiply(A_L, B_L)$$

  We can use the partial result of $Multiply(A_H, B_H)$ and $Multiply(A_L, B_L)$ to help us solve $Multiply(A_H, B_L) + Multiply(A_L, B_H)$ since

  $$A_H B_L + A_L B_H = (A_H + A_L)(B_H + B_L) - A_H B_H - A_L B_L$$

- So we just need to solve **three** half-sized subproblems instead of four. They are: $Multiply(A_H, B_H)$, $Multiply(A_L, B_L)$, $Multiply(A_H + A_L, B_H + B_L)$

Implementation of DaC Polynomial Multiplication

Polynomial Multiplication with DaC

1: $\textsc{Multiply}(A, B)$
2: **if** $n = 1$ **then**
3: **return** $A \cdot B$
4: $mid \leftarrow \lfloor (n-1)/2 \rfloor$
5: $A_H, A_L \leftarrow \textsc{Split}(A, mid)$
6: $B_H, B_L \leftarrow \textsc{Split}(B, mid)$
7: $C_0 \leftarrow \textsc{Multiply}(A_H, B_H)$
8: $C_2 \leftarrow \textsc{Multiply}(A_L, B_L)$
9: $C_1 \leftarrow \textsc{Multiply}((A_H + A_L), (B_H + B_L)) - C_0 - C_2$
10: **for** $i \leftarrow 0$ to $n-2$ **do**
11: $C[i] \leftarrow C[i] + C_0[i]$
12: $C[i+n] \leftarrow C[i+n] + C_2[i]$
13: $C[i+n/2] \leftarrow C[i+n/2] + C_1[i] - C_0[i] - C_2[i]$
14: **return** C

- Recurrence is $T(n) = 3T(n/2) + O(n)$. By master theorem, running time is $O(n^{\log_2 3})$.
- This is known to be not optimal. A more complicated version of DaC – Fast Fourier Transform can reduce the running time to $O(n \log n)$.
- The key idea is to increase the number of addition, using the partial results to reduce the number of multiplication calls. Similar idea can be applied to 2D case.

Matrix Multiplication

Matrix Multiplication

Input: Two square matrices A, B

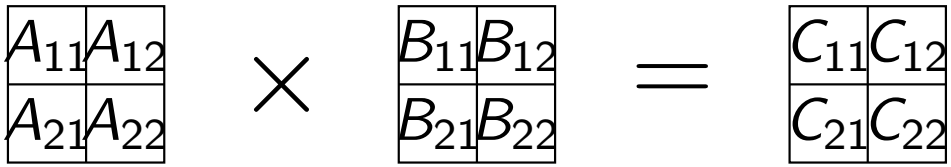
Output: The product matrix $C = AB$

Example

$$A = \begin{bmatrix} 2 & 3 \\ 4 & 1 \end{bmatrix} \quad B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$

$$C = \begin{bmatrix} (2 \cdot 5) + (3 \cdot 7) & (2 \cdot 6) + (3 \cdot 8) \\ (4 \cdot 5) + (1 \cdot 7) & (4 \cdot 6) + (1 \cdot 8) \end{bmatrix} = \begin{bmatrix} 31 & 36 \\ 27 & 32 \end{bmatrix}$$

Trivial DaC Matrix Multiplication



- $C_{11} = A_{11}B_{11} + A_{12}B_{21}$
- $C_{12} = A_{11}B_{12} + A_{12}B_{22}$
- $C_{21} = A_{21}B_{11} + A_{22}B_{21}$
- $C_{22} = A_{21}B_{12} + A_{22}B_{22}$
- 8 calls to the subproblem. So $T(n, n) = 8T(n/2, n/2) + O(n^2)$
- $T(n) = O(n^3)$

Strassen's Matrix Multiplication

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

- $C_{11} = M_1 + M_4 - M_5 + M_7, C_{12} = M_3 + M_5, C_{21} = M_2 + M_4, C_{22} = M_1 - M_2 + M_3 + M_6$
- $M_1 = (A_{11} + A_{22})(B_{11} + B_{22}), M_2 = (A_{21} + A_{22})B_{11}, M_3 = A_{11}(B_{12} - B_{22}), M_4 = A_{22}(B_{21} - B_{11}), M_5 = (A_{11} + A_{12})B_{22}, M_6 = (A_{21} - A_{11})(B_{11} + B_{12}), M_7 = (A_{12} - A_{22})(B_{21} + B_{22})$
- 7 calls to the subproblem. So $T(n, n) = 7T(n/2, n/2) + O(n^2)$
- $T(n) = O(n^{\log_2 7})$

What is the optimal running time for matrix multiplication?

- You might think it is possible to get $O(n^2 \log n)$ running time on Matrix Multiplication just like the polynomial multiplication. However, the best result today is still around $O(n^{2.3})$.
- People spent decades optimizing and it gradually became harder to optimize as if there is a limit for that. We usually refer the optimal running time for matrix multiplication as $O(n^{\omega})$ where ω is a constant between 2 and 3.

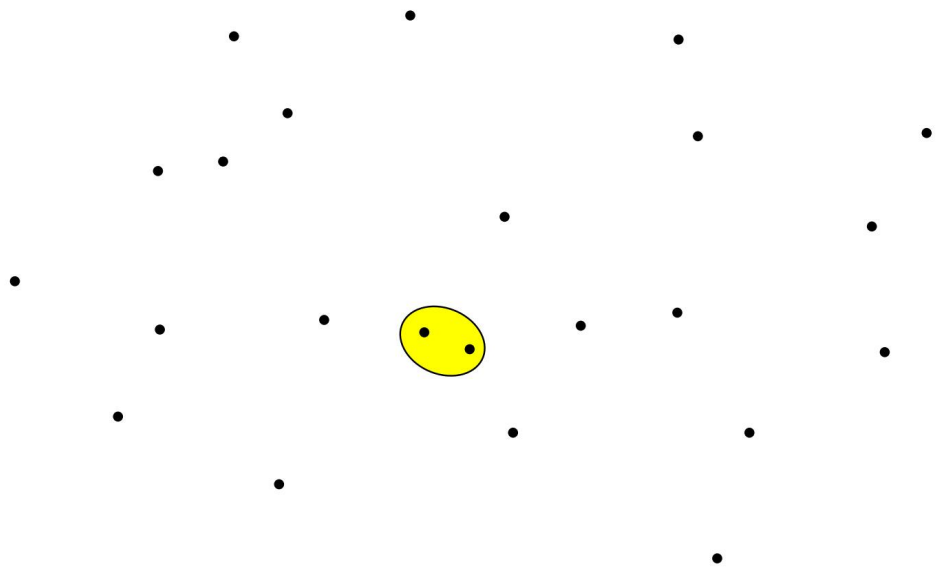
Other applications

- Geometric problems
- Number sequence problems

Find closest pair

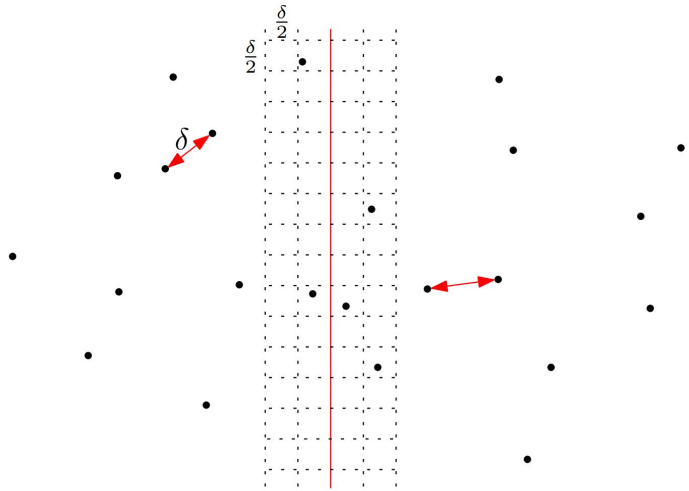
2D Closest pair problem

Input: 2D points with x, y coords: $(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)$
Output: p, q such that (x_p, y_p) and (x_q, y_q) are the closest.



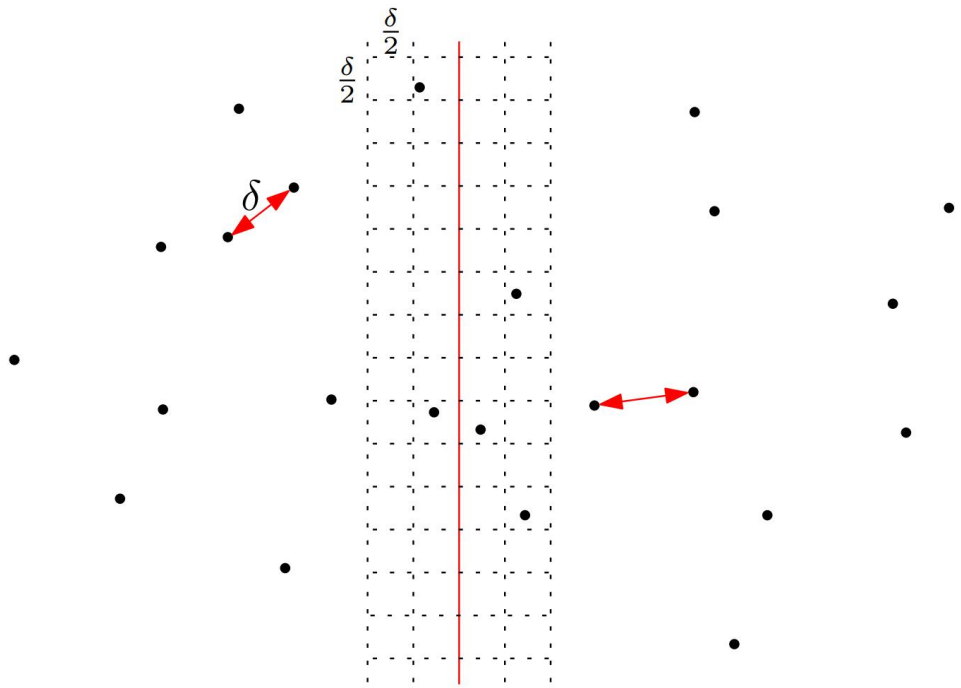
- Trivial algorithm runs in $O(n^2)$ time because it goes through every pair of points.

DaC Closest Pair



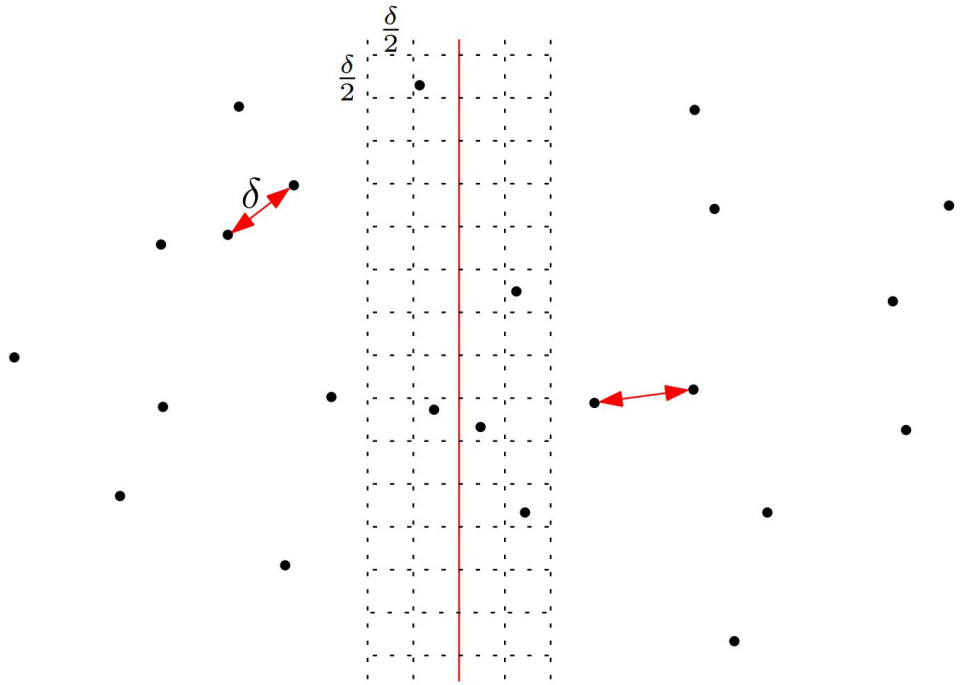
- **Divide:** Draw a vertical line in the middle and divide the points in two halves.
- **Conquer:** Solve the subproblem in the left half and the right half.
- **Combine:** Select the pair that is the closest among the pair from the left half, the pair from the right half and the pair that is across the line.

DaC Closest Pair



- Suppose the closest pair from left and right half has a distance δ, we can further put the points near the line into boxes of width $\frac{\delta}{2}$, this operation can be done in $O(n)$ time.

DaC Closest Pair



- Each box and its neighbour contain at most $O(1)$ number of points. We know that the closest pair cannot be across two boxes. So the number of pairs across the middle is reduced to $O(n)$.
- We have the recurrence $T(n) = 2T(n/2) + O(n)$. The running time is $O(n \log n)$.

Solving nth power of a square matrix

nth power of matrix

Input: A square matrix A
Output: A^n

- Trivial algorithm:
  Solve matrix multiplication n times: $A^n = A^{n-1} \times A$
- DaC algorithm:
  If n is even, $A^n = (A^{n/2})^2$.
  If n is odd, $A^n = (A^{(n-1)/2})^2 \times A$
- The running time is tricky. Note that the entries of A^n will be growing exponentially with respect to n. So addition and multiplication of two entries is not $O(1)$ anymore. The running time depends on the size of the result.

Solving nth number of Fibonacci Sequence

Fibonacci Sequence

$F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}, \forall n \geq 2$

$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...$

nth Fibonacci number

Input: Integer $n > 0$

Output: F_n

Utilizing the nth power of matrix

- Notice that we have:

$$\begin{aligned}
\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} &= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} F_{n-1} \\ F_{n-2} \end{bmatrix} \\
&= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^2 \begin{bmatrix} F_{n-2} \\ F_{n-3} \end{bmatrix} \\
&\cdots \\
&= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \begin{bmatrix} F_1 \\ F_0 \end{bmatrix} \\
&= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \begin{bmatrix} 1 \\ 0 \end{bmatrix}
\end{aligned}$$

- Therefore we can use the DaC matrix power to solve the Fibonacci number faster.