Algorithms 03

CS201

Kaustuv Nag

Divide-and-Conquer

- Involves three steps.
  - *Divide* the problem into a number of subproblems that are smaller instances of the same problem.
  - *Conquer* the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.
  - *Combine* the solutions to the subproblems into the solution for the original problem.
- When the subproblems are large enough to solve recursively, we call that the *recursive case*.
- Once the subproblems become small enough that we no longer recurse, we say that the recursion "bottoms out" and that we have gotten down to the *base case*.
- Sometimes, in addition to subproblems that are smaller instances of the same problem, we have to solve subproblems that are not quite the same as the original problem.

Recurrences

- A recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs.
- In practice, we neglect certain technical details when we state and solve recurrences.
  - we often omit floors and ceilings.
    - Technically, the recurrence describing the worst-case running time of merge sort is really

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n) & \text{if } n > 1 \end{cases}$$

      However, we consider it to be

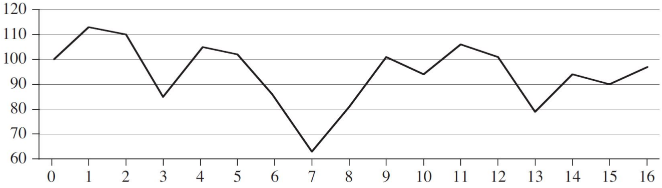
$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

  - Boundary conditions represent another class of details that we typically ignore.
    - For instnace, without explicitly giving values for small n we write

$$T(n) = 2T(n/2) + \Theta(n)$$

The maximum-subarray Problem

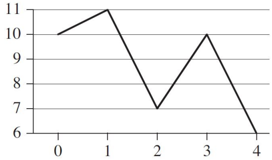
- You are allowed to buy one unit of stock only one time and then sell it at a later date, buying and selling after the close of trading for the day.
- To compensate for this restriction, you are allowed to learn what the price of the stock will be in the future.
- Your goal is to maximize your profit.



Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97

The maximum-subarray Problem

- Maybe you can always maximize profit by either buying at the lowest price or selling at the highest price.
- Find the highest and lowest prices. Then work left from the highest price to find the lowest prior price, work right from the lowest price to find the highest later price, and take the pair with the greater difference.
- A simple counterexample



Day	0	1	2	3	4
Price	10	11	7	10	6
Change		1	−4	3	−4

The maximum-subarray Problem

- We can easily devise a brute-force solution to this problem: just try every possible pair of buy and sell dates in which the buy date precedes the sell date.
- A period of n days has $\binom{n}{2}$ such pairs of dates. Since $\binom{n}{2}$ is $\Theta(n^2)$, and the best we can hope for is to evaluate each pair of dates in constant time, this approach would take $\Omega(n^2)$ time.
- Can we do better?

The maximum-subarray Problem

A Transformation

- Instead of looking at the daily prices, let us instead consider the daily change in price, where the change on day i is the difference between the prices after day $i-1$ and after day i .
- If we store these values in an array A, we now want to find the nonempty, contiguous subarray of A whose values have the largest sum.
- We call this contiguous subarray the maximum subarray.

Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
Change		13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

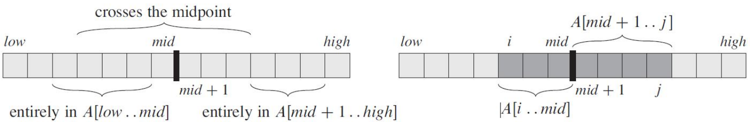
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

maximum subarray

The maximum-subarray Problem

A Solution using Divide and Conquer

- We find the midpoint, say *mid*, of the subarray, and consider the subarrays $A[low \cdots mid]$ and $A[mid+1 \cdots high]$.
- any contiguous subarray $A[i \cdots j]$ of $A[low \cdots mid]$ must lie in exactly one of the following places:
  - entirely in the subarray $A[low \cdots mid]$, so that $low \leq i \leq j \leq mid$,
  - entirely in the subarray $A[mid+1 \cdots high]$, so that $mid < i \leq j \leq high$, or
  - crossing the midpoint, so that $low \leq i \leq mid < j \leq high$.



The maximum-subarray Problem

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
left-sum = -∞
sum = 0
for i = mid downto low
    sum = sum + A[i]
    if sum > left-sum
        left-sum = sum
        max-left = i
right-sum = -∞
sum = 0
for j = mid + 1 to high
    sum = sum + A[j]
    if sum > right-sum
        right-sum = sum
        max-right = j
return (max-left, max-right, left-sum + right-sum)
```

The maximum-subarray Problem

```
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)
```

The maximum-subarray Problem

Analyzing the Algorithm

Runtime of the algorithm:

$$
\begin{aligned}
T(n) &= \Theta(1) + 2T(n/2) + \Theta(n) + \Theta(1) \\
&= 2T(n/2) + \Theta(n)
\end{aligned}
$$

Matrix Multiplication

Let $A = (a_{ij})$ and $B = (b_{ij})$ are $n \times n$ square matrices. Matrix multiplication is:

$$C = A \cdot B$$

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

```
SQUARE-MATRIX-MULTIPLY(A, B)
n = A.rows
let C be a new n × n matrix
for i = 1 to n
    for j = 1 to n
        c_ij = 0
        for k = 1 to n
            c_ij = c_ij + a_ik · b_kj
return C
```

Matrix Multiplication

Suppose that we partition each of A, B, and C into four $n/2 \times n/2$ matrices

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

We rewrite the equation $C = A \cdot B$ as

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

$$\begin{aligned} C_{11} &= A_{11} \cdot B_{11} + A_{12} \cdot B_{21} \\ C_{12} &= A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\ C_{21} &= A_{21} \cdot B_{11} + A_{22} \cdot B_{21} \\ C_{22} &= A_{21} \cdot B_{12} + A_{22} \cdot B_{22} \end{aligned}$$

Matrix Multiplication

```
SQUARE-MATRIX-MULTIPLY-RECURSIVE(A, B)
n = A.rows
let C be a new n × n matrix
if n == 1
    c11 = a11 · b11
else partition A, B, and C
    C11 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B11)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B21)
    C12 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B12)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B22)
    C21 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B11)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B21)
    C22 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B12)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B22)
return C
```

We should not copy the entries, we should use index calculations.

Matrix Multiplication

Analyzing the Algorithm

- Partitioning each $n \times n$ matrix by index calculation takes $\Theta(1)$ time.
- Each of four matrix additions in lines 6–9 contains $n^2/4$ entries, and so each of the four matrix additions takes $\Theta(n^2)$ time.
- Runtime of the algorithm:

$$
\begin{aligned}
T(n) &= \Theta(1) + 8T(n/2) + \Theta(n^2) \\
&= 8T(n/2) + \Theta(n^2) \\
&= \Theta(n^3)
\end{aligned}
$$

Matrix Multiplication: Strassen's Method

- Recursively create 10 matrices:

$$
\begin{aligned}
S_1 &= B_{12} - B_{22} \\
S_2 &= A_{11} + A_{12} \\
S_3 &= A_{21} + A_{22} \\
S_4 &= B_{21} - B_{11} \\
S_5 &= A_{11} + A_{22} \\
S_6 &= B_{11} + B_{22} \\
S_7 &= A_{12} - A_{22} \\
S_8 &= B_{21} + B_{22} \\
S_9 &= A_{11} - A_{21} \\
S_{10} &= B_{11} + B_{12}
\end{aligned}
$$

- Since we must add or subtract $n/2 \times n/2$ matrices 10 times, this step takes $\Theta(n^2)$ time.

Matrix Multiplication: Strassen's Method

- Recursively compute 7 matrices:

$$
\begin{aligned}
P_1 &= A_{11} \cdot S_1 &&= A_{11} \cdot B_{12} - A_{11} \cdot B_{22} \,, \\
P_2 &= S_2 \cdot B_{22} &&= A_{11} \cdot B_{22} + A_{12} \cdot B_{22} \,, \\
P_3 &= S_3 \cdot B_{11} &&= A_{21} \cdot B_{11} + A_{22} \cdot B_{11} \,, \\
P_4 &= A_{22} \cdot S_4 &&= A_{22} \cdot B_{21} - A_{22} \cdot B_{11} \,, \\
P_5 &= S_5 \cdot S_6 &&= A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22} \,, \\
P_6 &= S_7 \cdot S_8 &&= A_{12} \cdot B_{21} + A_{12} \cdot B_{22} - A_{22} \cdot B_{21} - A_{22} \cdot B_{22} \,, \\
P_7 &= S_9 \cdot S_{10} &&= A_{11} \cdot B_{11} + A_{11} \cdot B_{12} - A_{21} \cdot B_{11} - A_{21} \cdot B_{12} \,.
\end{aligned}
$$

- The only multiplications we need to perform are those in the middle column of the above equations. The right-hand column just shows what these products equal in terms of the original submatrices created in the previous step.

Matrix Multiplication: Strassen's Method

- Recursively construct the four $n/2 \times n/2$ submatrices of the product C:

$$\begin{aligned} C_{11} &= P_5 + P_4 - P_2 + P_6 \\ C_{12} &= P_1 + P_2 \\ C_{21} &= P_3 + P_4 \\ C_{22} &= P_5 + P_1 - P_3 - P_7 \end{aligned}$$

- we add or subtract $n/2 \times n/2$ matrices eight times in this step, and so this step indeed takes $\Theta(n^2)$ time.

Matrix Multiplication: Strassen's Method

- C_{11}:

$$
\begin{array}{rrrrrrrr}
A_{11}\cdot B_{11} & + A_{11}\cdot B_{22} & + A_{22}\cdot B_{11} & + A_{22}\cdot B_{22} & & & & \\
 & & - A_{22}\cdot B_{11} & & + A_{22}\cdot B_{21} & & & \\
 & - A_{11}\cdot B_{22} & & & & - A_{12}\cdot B_{22} & & \\
 & & & - A_{22}\cdot B_{22} & - A_{22}\cdot B_{21} & + A_{12}\cdot B_{22} & + A_{12}\cdot B_{21} & \\
\hline
A_{11}\cdot B_{11} & & & & & & + A_{12}\cdot B_{21} & ,
\end{array}
$$

Matrix Multiplication: Strassen's Method

- C_{12}:

$$
\begin{array}{r}
A_{11} \cdot B_{12} - A_{11} \cdot B_{22} \phantom{+ A_{12} \cdot B_{22}} \\
+ A_{11} \cdot B_{22} + A_{12} \cdot B_{22} \\
\hline
A_{11} \cdot B_{12} \phantom{- A_{11} \cdot B_{22} + A_{11} \cdot B_{22}} + A_{12} \cdot B_{22} \ ,
\end{array}
$$

Matrix Multiplication: Strassen's Method

- C_{21}:

$$\begin{array}{l} A_{21} \cdot B_{11} + A_{22} \cdot B_{11} \\ \qquad\qquad - A_{22} \cdot B_{11} + A_{22} \cdot B_{21} \\ \hline A_{21} \cdot B_{11} \qquad\qquad\qquad + A_{22} \cdot B_{21} \ , \end{array}$$

Matrix Multiplication: Strassen's Method

- C_{22}:

$$
\begin{array}{rrrrrrrr}
A_{11} \cdot B_{11} & + A_{11} \cdot B_{22} & + A_{22} \cdot B_{11} & + A_{22} \cdot B_{22} & & & & \\
 & - A_{11} \cdot B_{22} & & & + A_{11} \cdot B_{12} & & & \\
 & & - A_{22} \cdot B_{11} & & & - A_{21} \cdot B_{11} & & \\
- A_{11} \cdot B_{11} & & & & - A_{11} \cdot B_{12} & + A_{21} \cdot B_{11} & + A_{21} \cdot B_{12} & \\
\hline
 & & & A_{22} \cdot B_{22} & & & + A_{21} \cdot B_{12} & ,
\end{array}
$$

Matrix Multiplication: Strassen's Method

Analyzing the Algorithm

Runtime of the algorithm:

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

Solving this yields:

$$T(n) = \Theta(n^{\lg 7}) \approx \Theta(n^{2.81})$$

Comparing Rankings

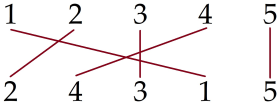
- There are five movies: A, B, C, D, E. They are ranked.
  - Rank by Person 1: D, B, C, A, E.
  - Rank by Person 2: B, A, C, D, E.
- How to measure the similarities between these rankings?
  - Compare preferences for each pair of movies.
- Inversion: Pair of movies ranked in opposite order.
  - No inversion: rankings are identical.
  - $n(n-1)/2$ inversions: every pair is inverted.
    - Maximum dissimilarity of rankings.

Counting Inversions

- Equivalent formulation
  - Fix the order of one ranking as a sorted sequence, $1, 2, \ldots, n$.
  - The other ranking is a permutation of $1, 2, \ldots, n$.
  - An inversion is a pair $(i,j), i < j$, where j appears before i in the permutation.
- Ranking 1: D, B, C, A, E.
  - $D = 1, B = 2, C = 3, A = 4, E = 5$.
- Ranking 2: B, A, C, D, E.
  - Corresonding Permutation: $2, 4, 3, 1, 5$.
- Inversions: $(1,2), (1,3), (1,4), (3,4)$

Counting Inversions

- Graphically
  - Ranking 1: D, B, C, A, E $(1, 2, 3, 4, 5)$.
  - Ranking 2: B, A, C, D, E $(2, 4, 3, 1, 5)$.



- Brute force: Check every pairs (i, j)
  - Requires $O(n^2)$ time.

Counting Inversions: Divide and Conquer

- Consider "Ranking 2": $[i_1, i_2, \ldots, i_n]$
- Divide into to lists $L = [i_1, i_2, \ldots, i_{n/2}]$ and $R = [i_{n/2+1}, i_{n/2+2}, \ldots, i_n]$.
- Recursively count inversions in L and R.
- Add inversions accross L and R.
  - How many elements in R are greater than elements in L?

Counting Inversions: Divide and Conquer

Adapting Merge Sort

- Divide $[i_1, i_2, \dots, i_n]$ to lists $L = [i_1, i_2, \dots, i_{n/2}]$ and $R = [i_{n/2+1}, i_{n/2+2}, \dots, i_n]$.
- Recursively *sort and count* inversions in L and R.
- Count inversions across L and R while merging
  - Merge and count.

Counting Inversions: Divide and Conquer

Merge and Count

- $L = [i_1, i_2, \ldots, i_{n/2}]$ and $R = [i_{n/2+1}, i_{n/2+2}, \ldots, i_n]$ are sorted.
- Count inversions across L and R while merging
  - Any element from R added to output is inverted with respect to all elements in L.
  - Add current size of L to the number of inversions.

Counting Inversions: Divide and Conquer

```
function MergeCount(A, m, B, n)
    // Merge A[0..m-1], B[0..n-1] into C[0..m+n-1]

    i = 0; j = 0; k = 0; count = 0;
    // Count positions in A, B, C and inversion count

    while (k < m + n)
        // Case 1: Move head of A into C, no inversions
        if (j == n or A[i] <= B[j])
            C[k] = A[i]; i++; k++;
        // Case 2: Move head of B into C, update Count
        if (i == m or A[i] > B[j])
            C[k] = B[j]; j++; k++;
            count = count + (m-i);

    return (count, C)
```

Counting Inversions: Divide and Conquer

```
function MergeSortCount(A, left, right)
    // Sort the segment A[left..right-1] into B

    if (right - left == 1) // Base case, no inversions
        B[0] = A[left]; count = 0;
        return (0, B)

    if (right - left > 1) // Recursive call
        mid = (left + right) / 2
        (countL, L) = MergeSortCount(A, left, mid)
        (CountR, R) = MergeSortCount(A, mid, right)
        (countM, B) = MergeCount(L, mid - left, R, right - mid)
        return (clountL + countR + countM, B)
```

Counting Inversions: Divide and Conquer

Analyzing the Algorithm

- Similar to merge sort.
- Runtime of the algorithm:

$$\begin{aligned} T(n) &= 2T(n/2) + O(n) \\ &= O(n \log(n)) \end{aligned}$$

- Total number of inversions could be $n(n-1)/2 = O(n^2)$.
- We are counting them efficiently without enumerating each of them.

Closest Pair of Points

- Several Objects on a two-dimensional plane.
- Objective: find the closest pair of objects.
- Given n objects, naive algorithm is $O(n^2)$.
  - For each pair of objects, compute their distance.
  - Report minimum distance over all such pairs.

Closest Pair of Points: Formal Way of Expression

- A point p is given by xy coordinates.
- Distance between $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ is

$$d(p_1, p_2) = \sqrt{(x_2 - x_1)^2 + (y2 - y1)^2}$$

- Objective: Given n points $(p_1, p_2, \ldots, p_n)$ find the closest pair.
  - Assume no two points are identical.
- Brute force
  - Try every pair (p_i, p_j) and find the pair with the minimum distance.
  - $O(n^2)$

Closest Pair of Points: In 1 Dimension

- A point p is given by x coordinate.
- Distance between p_1 and p_2 is

$$d(p_1, p_2) = |p1 - p2|$$

- Given n points $(p_1, p_2, \ldots, p_n)$
  - Sort the points - $O(n \log(n))$
  - Compute minimum separation between adjacent points after sorting - $O(n)$

Closest Pair of Points: In 2 Dimensions

- Split set of points into two halves by vertical line.
- Recursively compute closest pair in left and right half.
- Compute closest pairs across separating line.
  - How to do it in an efficient manner?

Closest Pair of Points: Sorting Points

- Given n points $P = \{p_1, p_2, \ldots, p_n\}$, compute
  - P_x, P sorted by x coordinate.
  - P_y, P sorted by y coordinate.
- Divide P by vertical line into equal size sets Q and R
- Need to efficiently compute Q_x, Q_y, R_x, R_y.

Closest Pair of Points: Sorting Points

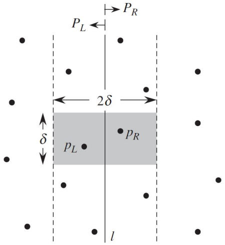
Need to efficiently compute Q_x, Q_y, R_x, R_y

- Q_x is the first half of P_x, R_x is the second half of P_x.
- When splitting Q_x, note the largest coordinate in Q, x_Q.
- Separate P_y as Q_y, R_y by checking x coordinate with x_Q.
- The above steps require $O(n)$.

Closest Pair of Points

Divide and Conquer: Combining Solutions

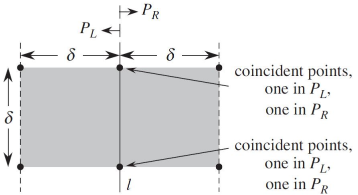
- Let d_Q be the closest distance in Q and d_R be the closest distance in R. Let, $d = \min(d_Q, d_R)$.
- Only need to consider points across the separate at most distance d from separator
  - Any pair outside this band cannot be closest pair overall.
- We use the *sparsity property* of the point set.
  - The closest pair of points is no further apart than d.
  - For each point p to the left of the dividing line we have to compare the distances to the points that lie in the rectangle of dimensions $(d, 2d)$ bordering the dividing line on the right side.
  - This rectangle can contain at most six points with pairwise distances at least d.
  - It is sufficient to compute at most 6 left-right distances.



Closest Pair of Points

Divide and Conquer: Combining Solutions

- If we allow coincident points, at most 8 points of P can reside within this rectangle.
  - Since points on line l may be in either the lest side or the right side, there may be up to 4 points on each side.
  - This limit is achieved if there are two pairs of coincident points such that each pair consists of one point from left side and one point from right side, one pair is at the intersection of l and the top of the rectangle, and the other pair is where l intersects the bottom of the rectangle.



Closest Pair of Points: Divide and Conquer

```
function ClosestPair(Px, Py)

if (|Px| <= 3)
    Compute pariwise distances and return closest pair distance

Construct (Qx, Qy, Rx, Ry)

(dQ, q1, q2) = ClosestPair(Qx, Qy)
(dR, r1, r2) = ClosestPair(Rx, Ry)

Construct Sy and scan to find (dS, s1, s2)

Return (dQ, q1, q2), (dR, r1, r2), (dS, s1, s2)
depending on which among (dQ, dR, dS) is minimum
```

Closest Pair of Points

Analyzing the Algorithm

- Computing (P_x, P_y) from P takes $O(n\log(n))$.
- Recursive algorithm
  - Setting up (Q_x, Q_y, R_x, R_y) from P_x, P_y is $O(n)$.
  - Setting up S_y from Q_y, R_y is $O(n)$.
  - Scanning S_y is $O(n)$.
- Recurrence is the same as merge sort: $T(n) = O(n\log(n))$

White Board

White Board