Divide and Conquer: Counting Inversions

Divide and Conquer

- Break the problem into disjoint sub-problems
- Combine these sub-problem solutions efficiently

Examples

- Merge sort
  - Split into left and right half and sort each half separately
  - Merge the sorted halves
- Quicksort
  - Re-arrange into lower and upper partitions, sort each partition separately
  - Place pivot between sorted lower and upper partitions

Recommender systems

- Online services recommend items to you
- Compare your profile with other customers
- Identify people who share your likes and dislikes
- Recommend items that they like
- Comparing profiles: How similar are your rankings to those of others?

Comparing rankings

- You and your friend rank 5 movies $\{A, B, C, D, E\}$
  - Your ranking: D, B, C, A, E
  - Your friend's ranking: B, A, C, D, E
- How to measure how similar these rankings are?
- For each pair of movies, compare preferences
  - You rank B above C, so does your friend
  - You rank D aboe B, your friend rankes B above D

Compare based on inversions

Inversions

- Pair of movies ranked in opposite order
  - You rank D above B, your friend ranks B above D

- No inversion $\implies$ rankings identical
- Every pair inverted $\implies$ maximally dissimilar
- Number of inversions range from 0 to $n(n-1)/2 \rightarrow$ measure of dissimilarity

Permutations

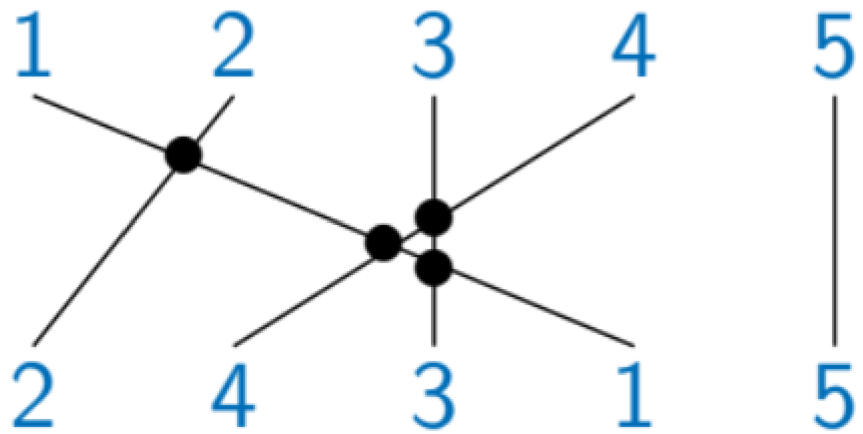
- Fix the order of one ranking as a sorted sequence $1, 2, \ldots, n$
- The other ranking is a permutation of $1, 2, \ldots, n$
- An inversion is a pair $(i, j), i < j$, where j appears before i

Counting inversions

- Number of inversions ranges from 0 to $n(n-1)/2 \rightarrow$ measure of dissimilarity
- Your ranking: D, B, C, A, E
    - D = 1, B = 2, C = 3, A = 4, E = 5
- Your friend's ranking: B, A, C, D, E
    - 2, 4, 3, 1, 5
- Inversions in 2, 4, 3, 1, 5?
- (1, 2), (1, 3), (1, 4), (3, 4)

Graphically

- Write the 2 permutations as 2 rows of nodes
- Connect every pair (j, j) between the two rows



- Every crossing is an inversion
- Brute force - check every $(i, j), O(n^2)$

Divide and Conquer

- Friend's permutation is $i_1, i_2, \ldots, i_n$
- Divide into 2 lists
    - $L = [i_1, i_2, \ldots, i_{n/2}]$

- $R = [i_{n/2+1}, i_{n/2+2}, \ldots, i_n]$
- Recursively count inversions in L and R
- Add inversions across the boundary between L and R
  - $i \in L, j \in R, i > j$
  - How many elements in L are bigger than elements in R?
- How to count inversions across the boundary?
- Adapt merge sort
- Recursively **sort and count** inversions in L and R
- Count inversions while merging - **merge and count**

Merge and Count

- Merge $L = [i_1, i_2, \ldots, i_{n/2}]$ and $R = [i_{n/2+1}, i_{n/2+2}, \ldots, i_n]$, sorted
- Count inversions while merging
  - If we add i_m from R to the output, i_m is smaller than elements currently in L
  - i_m is hence inverted w.r.t. elements currently in L
  - Add current size (total size - current pointer index) of L to the inversion count

```
def merge_and_count(A, B):
  m = len(A)
  n = len(b)
  C = []
  i, j, k, count = 0, 0, 0, 0

  while k < m + n:
    if i == m:
      C.append(B[j])
      j += 1
      k += 1
    elif j == n:
      C.append(A[i])
      i += 1
      k += 1
    elif A[i] < B[j]:
      C.append(A[i])
      i += 1
      k += 1
    else:
      C.append(B[j])
      j += 1
      k += 1
      count = count + (m - i) # m - i is the current length of L

  return (C, count)
```

- `sort_and_count` is merge sort with `merge_and_count`

```
def sort_and_count(A):
  n = len(A)
  if n <= 1:
    return (A, 0)

  (L, countL) = sort_and_count(A[:n//2])
  (R, countR) = sort_and_count(A[n//2:])
  (B, countB) = merge_and_count(L, R) # countB is cross inversions

  return (B, countL + countR + countB)
```

Analysis

- Recurrence is similar to merge sort
  - $T(0) = T(1) = 1$
  - $T(n) = 2T(n/2) + n$
- Solve to get $T(n) = O(n\ log\ n)$
- Note that the number of inversions can still be $O(n^2)$
  - Number ranges from 0 to $n(n-1)/2$
- We are counting them efficiently without enumerating each one

Divide and Conquer: Closest pair of points

Recall: Video game

- Several objects on the screen
- Basic step: Find the closest pair of objects
- n objects - naive algorithm is n^2
  - For each pair of objects, compute their distance
  - Report minimum distance across all pairs
- There is a clever algorithm that takes time $n \ log_2 \ n$
- Uses divide and conquer

The problem statement

- Points p in $2D$ - $p = (x, y)$
- Usual Euclidean distance between $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$
  - $d(p_1, p_2) = \sqrt{(y_2 - y_1)^2 + (x_2 - x_1)^2}$
- Given n points $p_1, p_2, \ldots, p_n$ find the closest pair
  - Assume no two points have same x or y coordinate
  - We can always rotate the points slightly to ensure this
  - or we can modify the algorithm slightly 
- Brute force
  - Compute $d(p_i, p_j)$ for every pair of points
  - $O(n^2)$

Finding the closest pair of points

In 1 dimension

- Given n 1D points $x_1, x_2, \ldots, x_n$, find the closest pair
  - $d(p_i, p_j) = |x_j - x_i|$
- Sort the points - $O(n \ log \ n)$
- In sorted order, nearest points to p are its neighbours
  - $O(n)$ scan to find the minimum separation between adjacent points

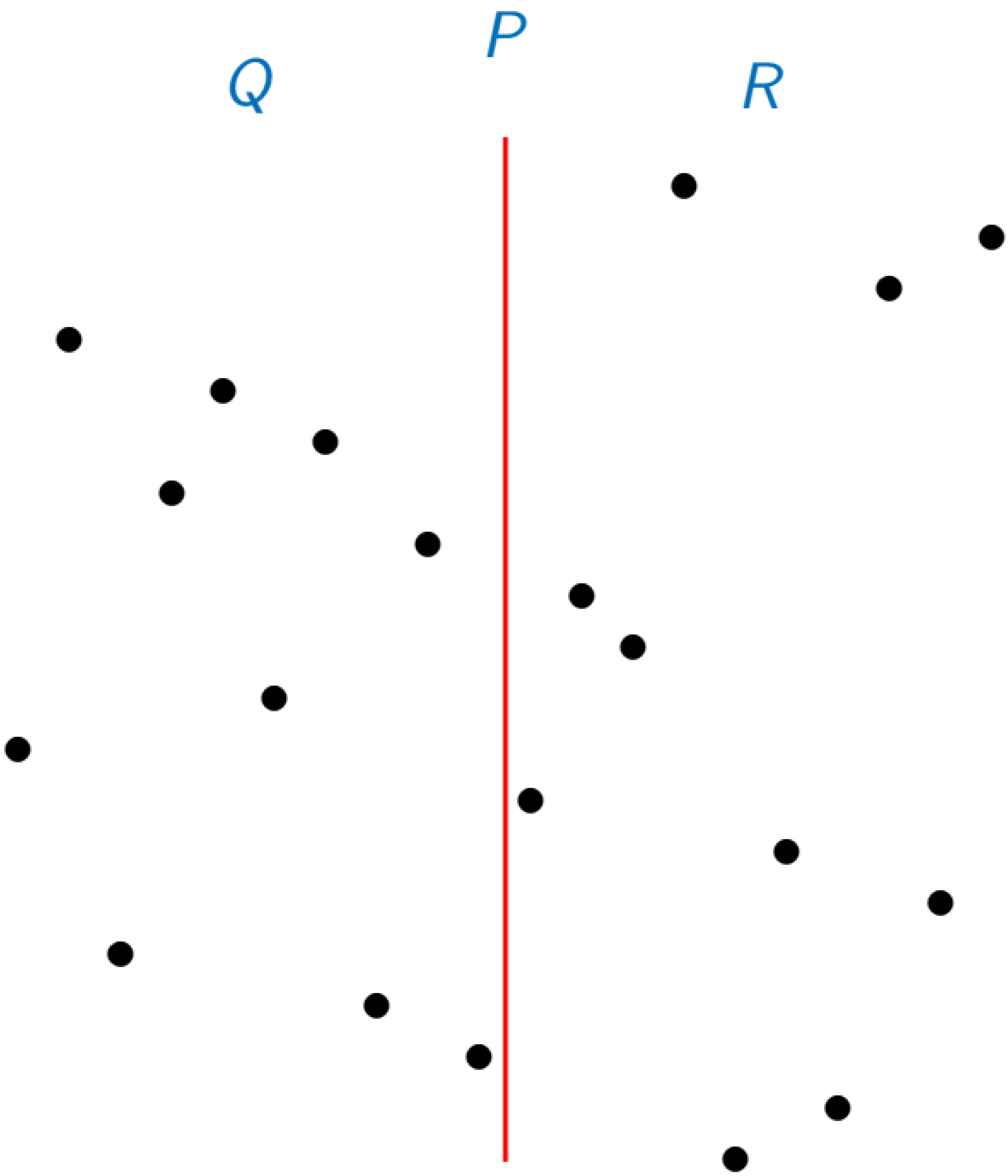
In 2 dimensions

- Divide and Conquer

- Split the points in 2 halves by a vertical line
- Recursively compute closest pairin each half
- Compare shortest distance in each half to shortest distance across the dividing line
- How to do this efficiently?

Dividing 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 a vertical line into equal size Q, R
- How to compute Q_x, Q_y, R_x, R_y efficiently?
- Q_x is the first half of P_x, R_x is the second half of P_x
- Let x_R be the smallest x coordinate in R
- For $p \in P_y$, if x-coordinate of p is less than x_R, move p to Q_y, else R_y
- All of this can be done is $O(n)$



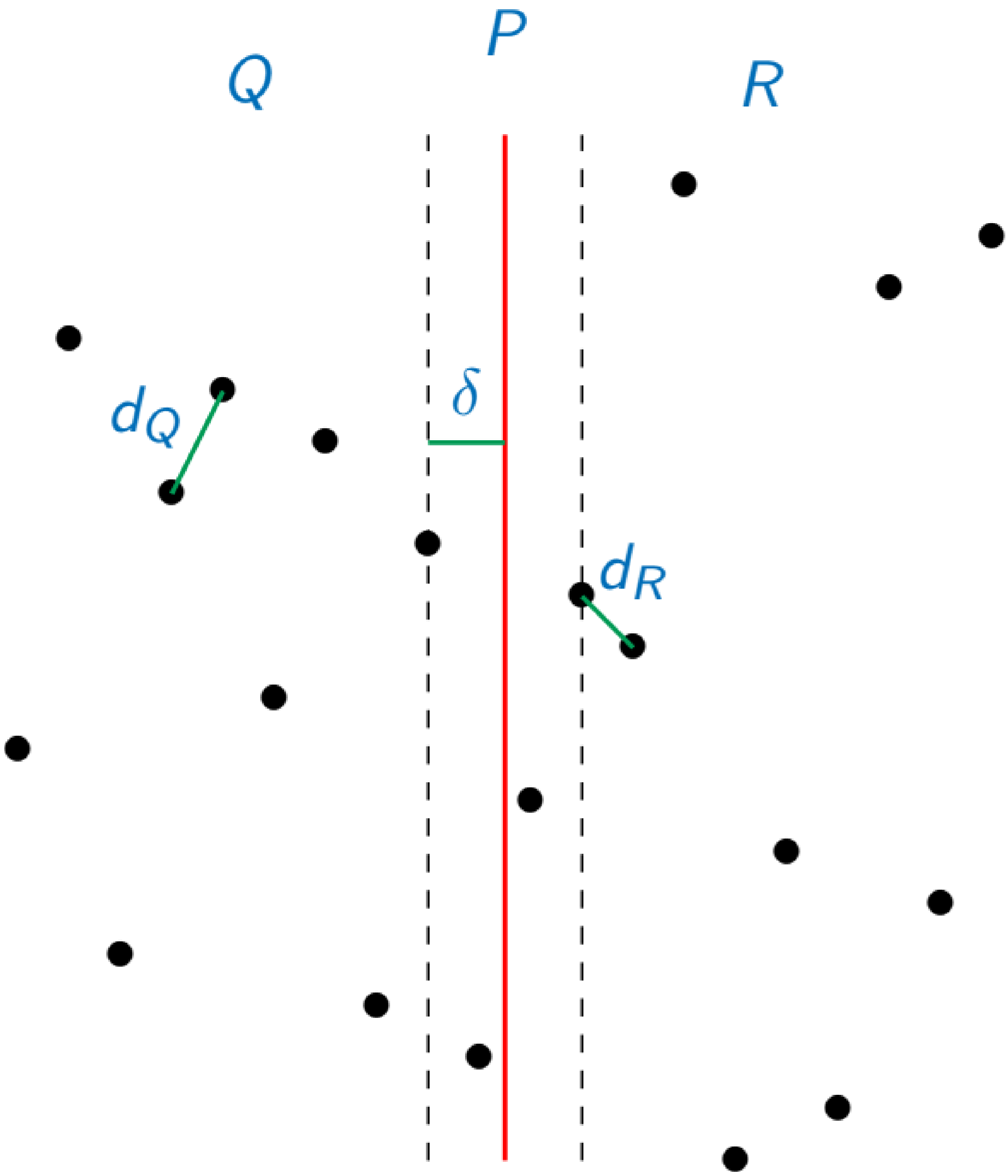
Divide and Conquer

- Want to compute $ClosestPair(P_x, P_y)$
- Split (P_x, P_y) as (Q_x, Q_y), (R_x, R_x)
- Recursively compute $ClosestPair(Q_x, Q_y)$ and $ClosestPair(R_x, R_y)$'
- How to combine these recursive solutions?

Combining Solutions

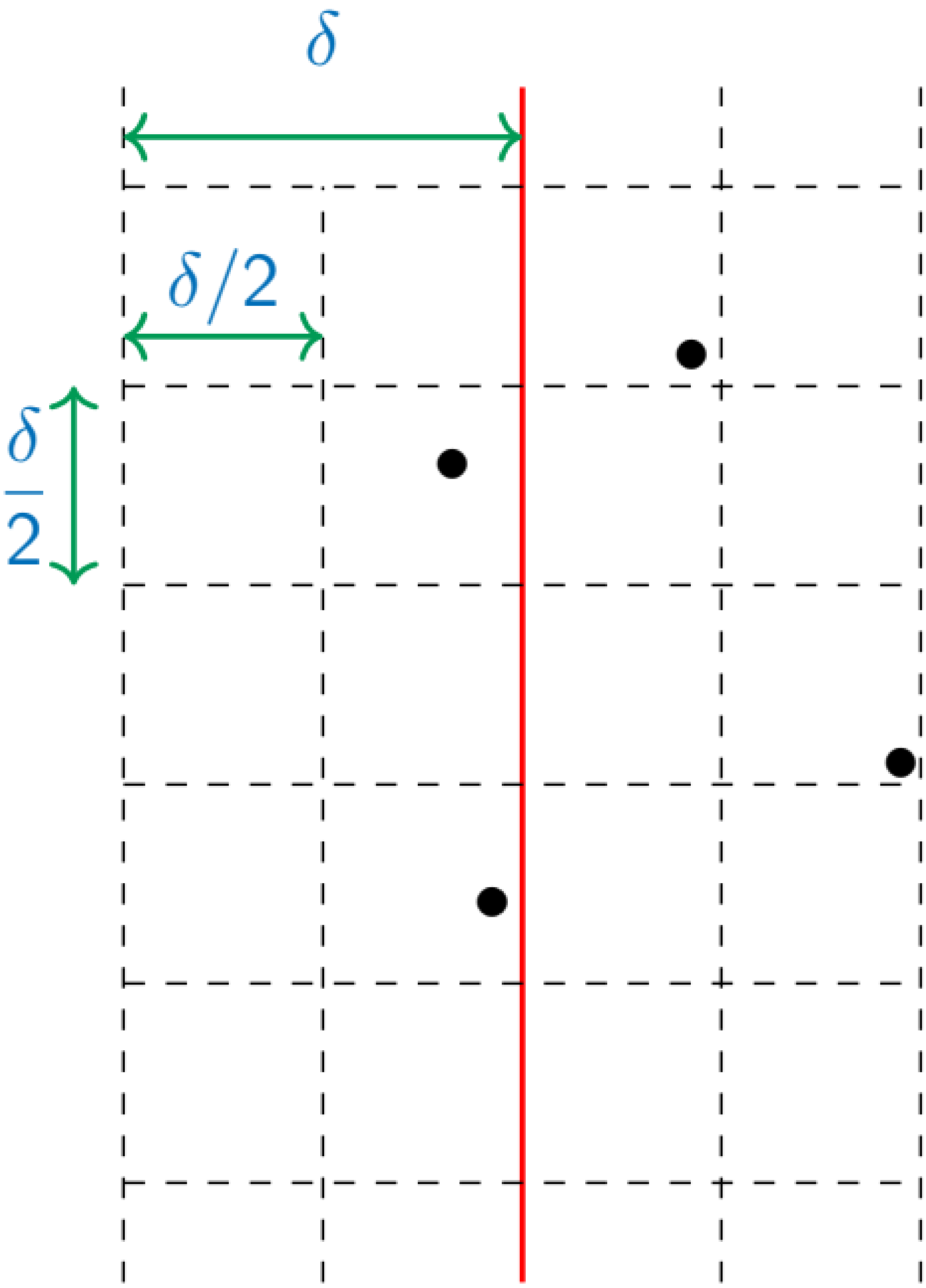
- Let d_Q, d_R be the closest distances in Q, R, respectively
- Set $\delta = min(d_Q, d_R)$

- Only need to consider points within distance δ on either side of the separator
- No pair outside this band can be closer than δ



Combining Solutions

- Divide the distance δ band into boxes of side $\delta/2$
- We cannot have 2 points inside the same box
    - Box diagonal is $\delta/\sqrt{2}$
- Any point within the distance δ must lie in a 4×4 neighbourhood of boxes
    - Check each point against 15 others
- From Q_y, R_y, extract S_y, points in δ band sorted by y
- Scan S_y from bottom to top, comparing each p with next 15 points in S_y
- Linear scan



Algorithm and Analysis

Pseudocode

```
def ClosestPair(Px, Py):

  if len(Px) <= 3:
    compute pairwise distances
    return closest pair and distance

  Construct (Qx, Qy), (Rx, Ry)

  (q1, q2, dQ) = ClosestPair(Qx, Qy)
```

```
(r1, r2, dR) = ClosestPair(Rx, Ry)

Construct Sy from Qy, Ry
Scan Sy, find (s1, s2, dS)

return (q1, q2, dQ), (r1, r2, QR), (s1, s2, dS)
depending on which of dQ, dR, dS is minimum
```

Analysis

- Sort P to get P_x, P_y - $O(n\ log\ n)$
- Recursive algorithm
    - Construct $(Q_x, Q_y), (R_x, R_y)$ - $O(n)$
    - Construct S_y from Q_y, R_y - $O(n)$
    - Scan S_y - $O(n)$
- Recurrence: $T(n) = 2T(n/2) + O(n)$ like merge sort
- Overall, $O(n\ log\ n)$

Divide and Conquer: Integer Multiplication

Integer Multiplication

- How do we multiply two integers x, y?
- Form **partial products** - multiply each digit of y separately by x
- Add up all the partial products
- Works the same in any base - e.g. Binary
- To multiply 2 n-bit numbers
    - n partial products
    - Adding each partial product to cumulative sum is $O(n)$
    - Overall $O(n^2)$
- Can we improve on this?
    - Each partial product seems "necessary"

```
  12            1100
x 13       x    1101
 ---         -------
  36            1100
 12            0000
 ---          1100
 156         1100
             -------
            10011100
```

Divide and Conquer

- Split the n bits into 2 groups of $n/2$

$$
\begin{array}{cccc}
 & x_1 & & x_0 \\
x & b_{n-1} b_{n-2} \cdots b_{\frac{n}{2}} & & b_{\frac{n}{2}-1} b_{\frac{n}{2}-2} \cdots b_0
\end{array}
$$

$$
\begin{array}{cccc}
 & y_1 & & y_0 \\
y & b'_{n-1} b'_{n-2} \cdots b'_{\frac{n}{2}} & & b'_{\frac{n}{2}-1} b'_{\frac{n}{2}-2} \cdots b'_0
\end{array}
$$

- Rewrite xy as $(x_1.2^{n/2} + x_0)(y_1.2^{n/2} + y_0)$
- Regroup as $x_1 y_1.2^n + (x_1 y_0 + x_0 y_1).2^{n/2} + x_0 y_0$
- Four $n/2$-bit multiplications
- $T(1) = 1, T(n) = 4T(n/2) + n$
    - Combining the partial products requires adding $O(n)$ bit numbers

$$
\begin{aligned}
T(n) &= 4T(n/2) + n \\
&= 4(4T(n/4) + n/2) + n \\
&= 4^2 T(n/2^2) + (2+1)n \\
&= 4^2(4T(n/2^3) + n/2^2) \\
&\qquad + (2^1 + 2^0)n \\
&= 4^3 T(n/2^3) + (2^2 + 2^1 + 2^0)n \\
&= \cdots \\
&= 4^{\log n} T(n/2^{\log n}) \\
&\qquad + (2^{\log n - 1} + \cdots + 2^1 + 2^0)n \\
&= O(n^2)
\end{aligned}
$$

Karatsuba's algorithm

- Rewrite xy as $x_1y_1.2^n + (x_1y_0 + x_0y_1).2^{n/2} + x_0y_0$
- $T(n) = 4T(n/2) + n$ is $O(n^2)$
- Divide and Conquer has not helped!
- $(x_1 - x_0)(y_1 - y_0) = x_1y_1 - x_1y_0 - x_0y_1 + x_0y_0$
    - $O(n/2)$ bit multiplication
- Compute x_1y_1, x_0y_0
    - $O(n/2)$ bit multiplications
- $(x_1y_1 + x_0y_0) - (x_1 - x_0)(y_1 - y_0)$ leaves $x_1y_0 + x_0y_1$
    - $3\ O(n/2)$ bit multiplications

The Algorithm

Fast-Multiply(x, y, n)

  if $n = 1$

    return $x \cdot y$

  else

    $m = n/2$

    $(x_1, x_0) = (x/2^m, x \bmod 2^m)$ Bit shifting

    $(y_1, y_0) = (y/2^m, y \bmod 2^m)$ Bit shifting

    $(a, b) = (x_1 - x_0, y_1 - y_0)$

    $p =$ Fast-Multiply(x_1, y_1, m)

    $q =$ Fast-Multiply(x_0, y_0, m)

    $r =$ Fast-Multiply(a, b, m)

    return $p \cdot 2^n + (p + q - r) \cdot 2^{n/2} + q$

Karatsuba's algorithm - Analysis

- $T(1) = 1, \; T(n) = 3T(n/2) + n$
- $$\begin{aligned} T(n) &= 3T(n/2) + n \\ &= 3(3T(n/4) + n/2) + n \\ &= 3^2 T(n/2^2) + (3/2 + 1)n \\ &= 3^2(3T(n/2^3) + n/2^2) \\ &\qquad +((3/2)^1 + 1)n \\ &= 3^3 T(n/2^3) \\ &\qquad +((3/2)^2 + (3/2)^1 + 1)n \\ &= \cdots \\ &= 3^{\log n} T(n/2^{\log_2 n}) \\ &\qquad +((3/2)^{\log n - 1} + \cdots + (3/2)^1 + 1)n \\ &= 3^{\log n} \\ &\qquad +[((3/2)^{\log n - 1} - 1)/((3/2) - 1)]n \end{aligned}$$

- $a^{\log n} = n^{\log a}$
- $3^{\log n} = n^{\log 3}$
- $$\begin{aligned} n \cdot (3/2)^{\log n} &= n \cdot n^{\log(3/2)} \\ &= n \cdot n^{\log 3 - \log 2} \\ &= n^1 \cdot n^{\log 3 - 1} \\ &= n^{1 + \log 3 - 1} \\ &= n^{\log 3} \end{aligned}$$
- $\log 3 \approx 1.59$
- Divide and conquer reduces the complexity of integer multiplication from $O(n^2)$ to $O(n^{1.59})$

Historical note

- In the 1950s, Andrei Kolmogorov, one of the giants of 20th century mathematics, publicly conjectured that multiplication could not be done in subquadratic time
- Kolmogorov mentioned this conjecture at a seminar in Moscow University in 1960
- Anatolii Karatsuba, a 23 year old student, came back 2 weeks later to Kolmogorov with this divide and conquer algorithm
- Karatsuba's original proposal was slightly different
  - Instead of $r = (x_1 - x_0)(y_1 - y_0)$, he used $r = (x_1 + x_0)(y_1 + y_0)$
  - Then, $x_0 y_1 + x_1 y_0 = r - (x_1 y_1 + x_0 y_0)$
  - Difficulty is that $x_1 + x_0, y_1 + y_0$ could have $n + 1$ bits, complicates the analysis
- Using $r = (x_1 - x_0)(y_1 - y_0)$ to simplify the analysis is due to Donald Knuth
- Karatsuba's algorithm can be used in any base, not just for binary multiplication

Divide and Conquer: Recursion Trees

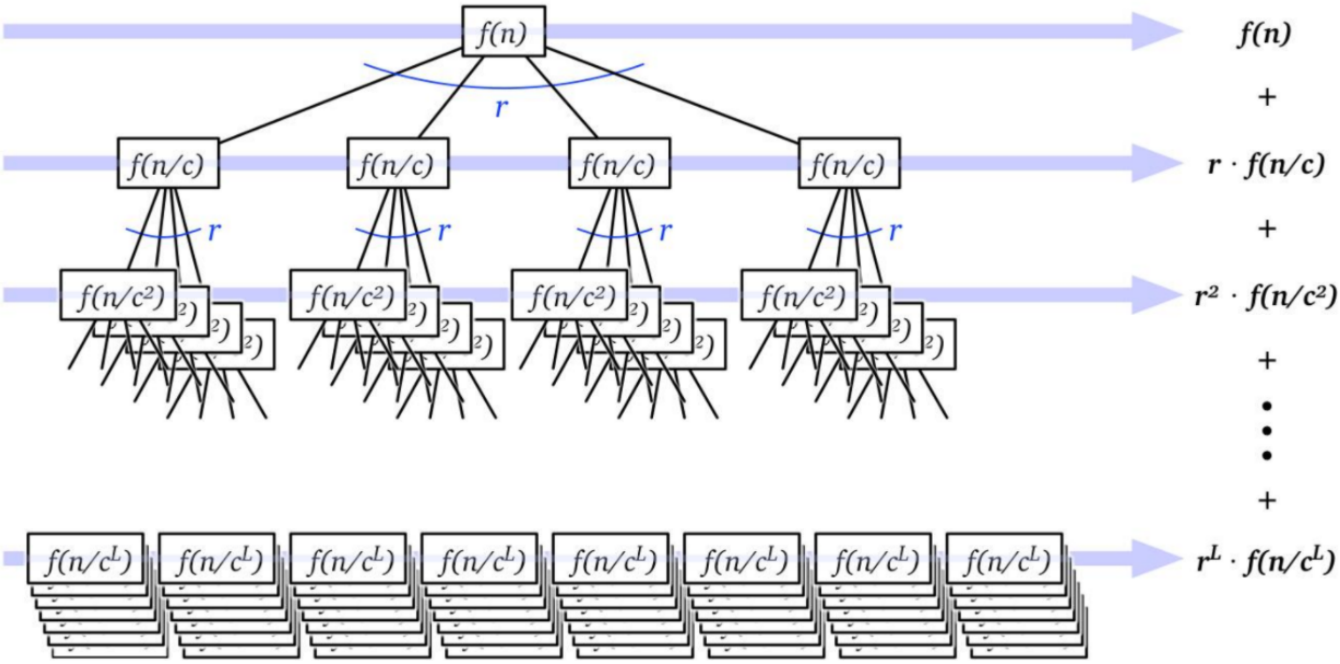
Solving recurrences

- Divide and conquer involves breaking up a problem into disjoint subproblems and combining the solutions efficiently
- Complexity $T(n)$ is expressed as a recurrence
- For searching and sorting, we solved simple recurrence by repeated substitution
  - Binary search: $T(n) = T(n/2) + 1, T(n)$ is $O(log\ n)$
  - Merge sort: $T(n) = 2T(n/2) + n, T(n)$ is $O(n\ log\ n)$
- For integer multiplication, the analysis became more complicated
  - Naive divide and conquer: $T(n) = 4T(n/2) + n, T(n)$ is $O(n^2)$
  - Karatsuba's algorithm: $T(n) = 3T(n/2) + n$ is $O(n^{log_2 3})$
- Is there a uniform way to compute the asymptotic expression for $T(n)$?

Recursion Trees

- **Recursion tree** Rooted tree with one node for each recursive subproblem
- **Value** of each node is time spent on that subproblem **excluding** recursive calls
- Concretely, on an input of size n
  - $f(n)$ is the time spent on non-recursive work
  - r is the number of recursive calls
  - Each recursive call works on a subproblem of size n/c
- Resulting recurrence: $T(n) = rT(n/c) + f(n)$
- Root of recursion tree for $T(n)$ has value of $f(n)$
- Root has r children, each (recursively) the root of a tree for $T(n/c)$
- Each node at level d has value $f(n/c^d)$
  - Assume, for simplicity, that n was a power of c

Recursion tree for $T(n) = rT(n/c) + f(n)$



Recursion Trees

- Leaves correspond to the base case $T(1)$
  - Safe to assume $T(1) = 1$, asymptotic complexity ignores constants
- Level i has r^i nodes, each with value $f(n/c^i)$
- Tree has L levels, $L = log_c\ n$
- Total cost is $T(n) = \sum_{i=0}^{L} r^i.\ f(n/c^i)$
- Number of leaves is r^L
  - Last term in the level by level sum is $r^L.\ f(1) = r^{log_c\ n}.1 = n^{log_c\ r}$
  - Recall that $a^{log_b\ c} = c^{log_b\ a}$

- Tree has $log_c\ n$ levels, last level has the cost $n^{log_c\ r}$
- Total cost is $T(n) = \sum_{i=0}^{L} r^i.\ f(n/c^i)$
- Think of the total cost as a series. Three common cases
- **Decreasing** Each term is a constant factor smaller than the previous term
  - Root dominates the sum, $T(n) = O(f(n))$
- **Equal** All terms in the series are equal
  - $T(n) = O(f(n).\ L) = O(f(n)log\ n) - log_c\ n$ is asymptotically the same as $log\ n$
- **Increasing** Series grows exponentially, each term a constant factor larger than the previous term
  - Leaves dominate the sum, $T(n) = O(n^{log_c\ r})$

Divide and Conquer: Quick Select

Selection

- Find the k^{th} largest value in a sequence of length n
- Sort in descending order and look at position k - $O(n \ log \ n)$
- Can we do better than this?
    - $k = 1$ - maximum, $O(n)$
    - $k = n$ - minimum, $O(n)$
- For any fixed k, k passes, $O(kn)$
- Median - $k = n/2$
    - If we can find median $O(n)$, quicksort becomes $O(n \ log \ n)$

Divide and Conquer

- Recall partitioning for quicksort
    - Pivot partitions sequence as `lower` and `upper`
- Let `m = len(lower)`. 3 cases:
    - `k <= m` - answer lies in `lower`
    - `k == m + 1` - answer lies in `pivot`
    - `k > m + 1` - answer lies in `upper`
- Recursive strategy
    - Case 1: `select(lower, k)`
    - Case 2: `return(pivot)`
    - Case 3: `select(upper, k - (m + 1))`

```
# To find the k-th largest element in L[l:r]
def quick_select(L, l, r, k):
  if (k < 1) or (k > r - 1):
    return None

  pivot, lower, upper = L[l], l + 1, l + 1

  for i in range(l + 1, r):
    if L[i] > pivot:    # Extend the upper segment
      upper += 1
    else:               # Exchange L[i] with the start of upper segment
      L[i], L[lower] = L[lower], L[i]
      lower += 1
```

```
    upper += 1

  # Move the pivot
  L[l], L[lower - 1] = L[lower - 1], L[l]
  lower - 1

  # Recursive calls
  lower_len = lower - l

  if k <= lower_len:
    return quick_select(L, l, lower, k)
  elif k == lower_len + 1:
    return L[lower]
  else:
    return quick_select(L, lower + 1, r, k - (lower_len + 1))
```

Analysis

- Recurrence is similar to quick sort
- $T(1) = 1$
- $T(n) = max(T(m), T(n-(m+1))) + n$, where $m = len(lower)$
- Worst case: m is always 0 or $n-1$
  - $T(n) = T(n-1) + n$
  - $T(n)$ is $O(n^2)$
- Recall: if the pivot is within a fixed fraction, quick sort is $O(n\ log\ n)$
  - E.g. pivot in middle third of values
  - $T(n) = T(n/3) + T(2n/3) + n$
- Can we find a good pivot quickly?

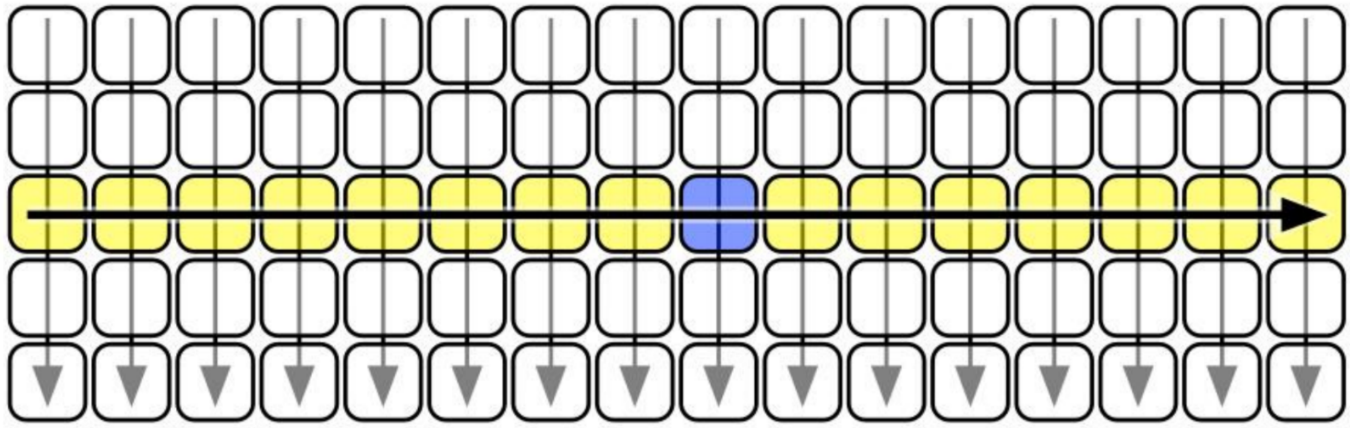
Median of medians

- Divide L into blocks of 5
- Find the median of each block (brute force)
- Let M be the list of block medians
- Recursively apply the process to M
- What can we guarantee about `MoM(L)`?

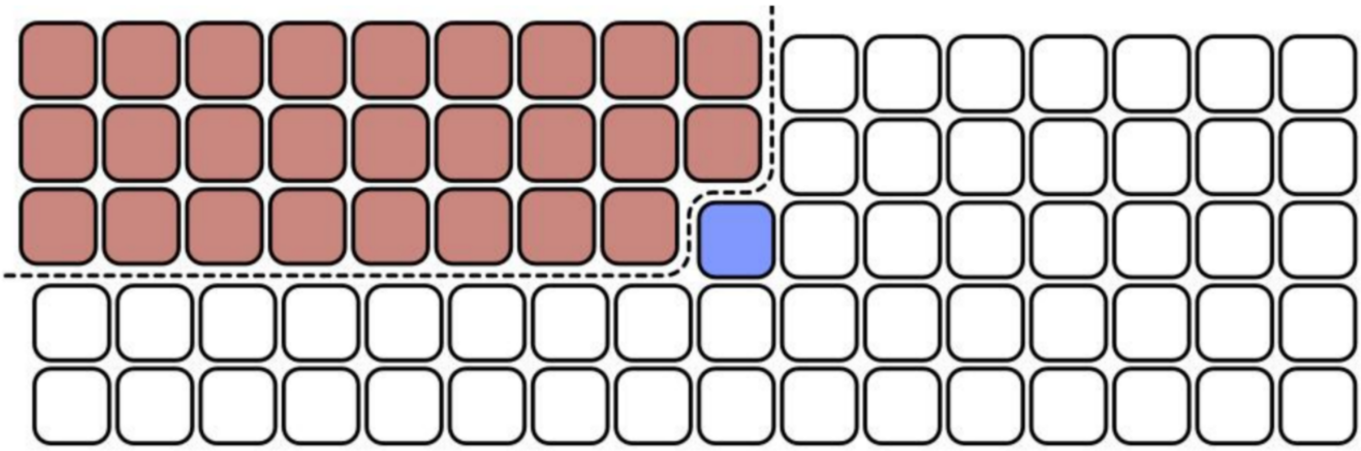
```
def MoM(L):   # Median of medians
  if len(L) <= 5:
    L.sort()
    return L[len(L)//2]

  # Construct list of block medians
```

```
M = []

for i in range(0, len(L), 5):
  X = L[i : i + 5]
  X.sort()
  M.append(X[len(X)//2])

return MoM(M)
```

- We can visualize the blocks as follows
- Each block of 5 is arranged in ascending order, top to bottom
- Block medians are the middle row



- We can visualize the blocks as follows
- Each block of 5 is arranged in ascending order, top to botto
- Block medians are the middle row
- The median of block medians lies between $3len(L)/10$ and $7len(L)/10$



Analysis

- Use median of block medians to locate the pivot for `quick_select`
- `MoM` is $O(n)$
  - $T(1) = 1$

    - $T(n) = T(n/5) + n$
- Recurrence for `fast_select` is now
    - $T(1) = 1$
    - $T(n) = max(T(3m/10), T(7m/10) + n)$, where $m = len(lower)$
- $T(n)$ is $O(n)$
- Can also use `MoM` to make quick sort $O(n \ log \ n)$

```
# Find the k-th largest element in L[l:r]
def fast_select(L, l, r, k):
  if (k < 1) or (k > r - 1):
    return None

  # Find MoM pivot and move to L[l]
  pivot = MoM(L[l:r])
  pivot_pos = min([i for i in range(l, r) if L[i] == pivot])
  L[l], L[pivot_pos] = L[pivot_pos], L[l]

  # Partition as before
  pivot, lower, upper = L[], l + 1, l + 1
  for i in range(l + 1, r):
    ...

  # Recursive calls
  lower_len = lower - l

  if k <= lower_len:
    return fast_select(L, l, lower, k)
  elif k == lower_len + 1:
    return L[lower]
  else:
    return fast_select(L, lower + 1, r, k - (lower_len + 1))
```

Summary

- Median of block medians helps find a good pivot in $O(n)$
- Selection becomes $O(n), quicksortbecomes$O(n \ log \ n)
  $*Noticethat`fast_select`with`k = len(L)/2`findsmedianintime$O(n)$

Historical note

- C.A.R. Hoare described `quick_select` in the same paper that introduced `quick_sort`, 1962
- The median of medians algorithm is due to Manuel Blum, Robert Floyd, Vaughn Pratt, Ron Rivest and Robert Tarjan, 1973

Acknowledgement

Implementation of Quick Select and Fast Select Algorithms

```
def quick_select(L, l, r, k):   # k-th largest in L[l:r]
  if (k < 1) or (k > r - l):
    return None

  pivot, lower, upper = L[l], l + 1, l + 1

  for i in range(l + 1, r):
    if L[i] > pivot:            # Extend the upper segment
      upper += 1
    else:                       # Exchange L[i] with start of upper segment
      L[i], L[lower] = L[lower], L[i]
      lower += 1
      upper += 1

  L[l], L[lower - 1] = L[lower - 1], L[l]
  lower -= 1

  # Recursive calls
  lower_len = lower - l

  if k <= lower_len:
    return quick_select(L, l, lower, k)
  elif k == lower_len + 1:
    return L[lower]
  else:
    return quick_select(L, lower + 1, r, k - (lower_len + 1))
```

```
from random import *
A = [randrange(1000) for i in range(200)]
print(A)
```

```
[459, 622, 418, 938, 112, 602, 681, 763, 566, 859, 949, 917, 181, 168, 147, 530, 661,
```

```
for i in range(0, len(A) + 2):
  print(quick_select(A, 0, len(A), i))
```

```
None
1
5
5
12
17
19
25
```

```
25
29
37
44
53
57
63
64
68
72
73
74
77
79
85
89
94
95
96
97
97
112
117
129
133
140
145
147
147
149
149
151
151
155
168
168
181
183
188
198
199
206
208
212
213
226
231
235
240
244
```

```
def MoM(L):        # Median of medians
  if len(L) <= 5:
    L.sort()
    return L[len(L)//2]

  # Construct list of block medians
  M = []
```

```
  for i in range(0, len(L), 5):
    X = L[i:i + 5]
    X.sort()
    M.append(X[len(X)//2])

  return MoM(M)

from random import *
A = [randrange(1000) for i in range(200)]
print(A)
```

```
[923, 690, 370, 838, 519, 572, 18, 772, 414, 800, 444, 458, 487, 353, 561, 128, 668,
```

```
B = sorted(A)
(B[(3 * len(B))//10], B[len(B)//2], B[(7 * len(B))//10], MoM(A))
```

```
(342, 480, 677, 459)
```

```
import sys
sys.setrecursionlimit(2 ** 31 - 1)
```

```
import time

class TimeError(Exception):
  """A custom exception used to report error in the use of Timer class"""

class Timer:
  def __init__(self):
    self._start = 0
    self._elapsed = 0

  def start(self):
    if self._start is not None:
      raise TimeError('Timer is running. Use .stop()')

    self._start = time.perf_counter()

  def stop(self):
    if self._start is None:
      raise TimeError('Timer is not running. Use .start()')

    self._elapsed = time.perf_counter() - self._start
    self._start = None

  def elapsed(self):
    if self._elapsed is None:
      raise TimeError('Timer has not been run yet. Use .start()')
```

```
    return self._elapsed

  def __str__(self):
    return str(self._elapsed)
```

```
t = Timer()
t.stop()
t.start()

A = [i for i in range(10000)]
print(quick_select(A, 0, len(A), 10000))
t.stop()

print(t)
```

```
9999
6.074782084999697
```

```
# Find the k-th largest element in L[l:r]
def fast_select(L, l, r, k):
  if (k < 1) or (k > r - 1):
    return None

  # Find MoM pivot and move to L[l]
  pivot = MoM(L[l:r])
  pivot_pos = min([i for i in range(l, r) if L[i] == pivot])
  L[l], L[pivot_pos] = L[pivot_pos], L[l]

  # Partition as before
  pivot, lower, upper = L[l], l + 1, l + 1
  for i in range(l + 1, r):
    if L[i] > pivot:  # Extend the upper segment
      upper += 1
    else:             # Exchange L[i] with the start of the upper segment
      L[i], L[lower] = L[lower], L[i]
      lower += 1
      upper += 1

  L[l], L[lower - 1] = L[lower - 1], L[l]
  lower -= 1

  # Recursive calls
  lower_len = lower - l

  if k <= lower_len:
    return fast_select(L, l, lower, k)
  elif k == lower_len + 1:
    return L[lower]
```

```
  else:
    return fast_select(L, lower + 1, r, k - (lower_len + 1))
```

```
t = Timer()
t.stop()
t.start()

A = [i for i in range(10000)]
print(fast_select(A, 0, len(A), 10000))
t.stop()
print(t)
```

```
None
0.0011042640003324777
```