Algorithms – I (CS29003/203)

Autumn 2022, IIT Kharagpur

Divide and Conquer, Recursion

Divide and Conquer

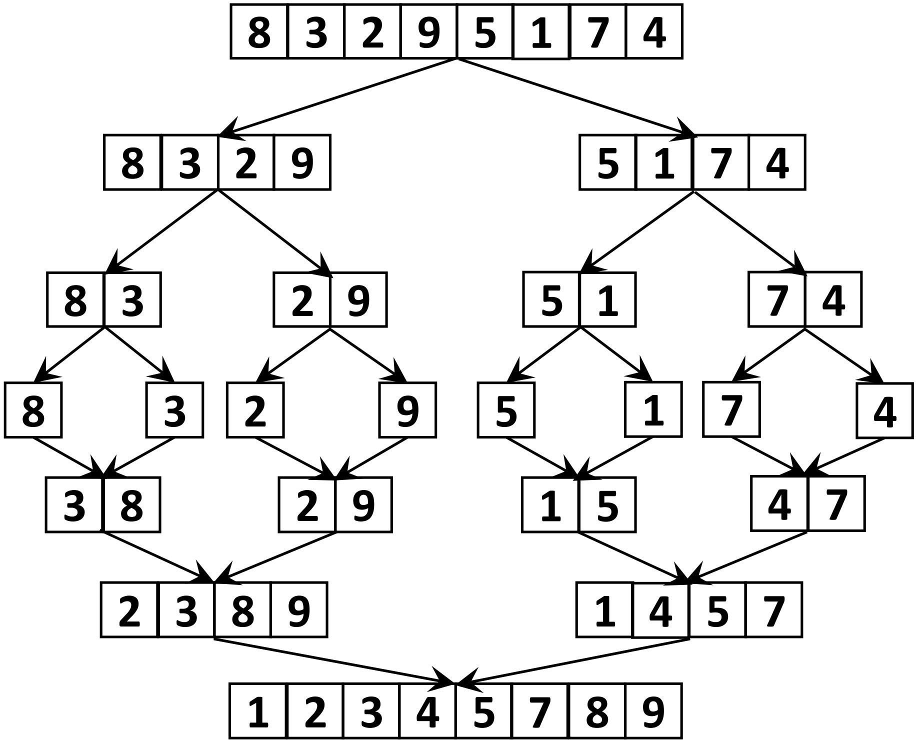
- General techniques
- Merge sort
- Recursion Tree Method
- Master Theorem

Divide and Conquer

- Divide and conquer involves three steps

- **Divide** a problem into pieces (smaller instances of same problem)
    - E.g., divide into halves
- **Conquer** the subproblems by solving them recursively
    - Can just call the same algorithm on the subproblems (recursively solving)
    - Base case: when n=1 (or is small)
- **Combine** solutions to pieces to get answer to original problem
    - Combining results from recursive calls.
    - Usually the hardest part in the algorithm design

Merge Sort


8 3 2 9 5 1 7 4
8 3 2 9
5 1 7 4
8 3
2 9
5 1
7 4
8
3
2
9
5
1
7
4
3 8
2 9
1 5
4 7
2 3 8 9
1 4 5 7
1 2 3 4 5 7 8 9

Merge Sort

- Dividing is easy. The key operation, though, is merge.
- How to do merging of two sorted arrays to produce a merged sorted array?

2	**5**	**9**

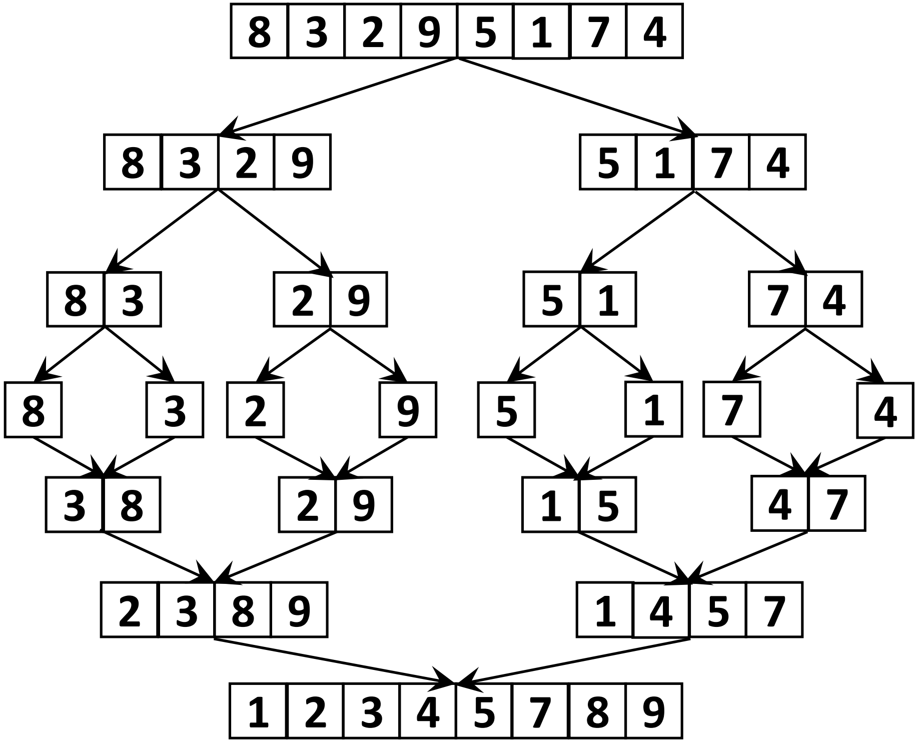
3	**4**	**8**

Merge Sort Pseudocode

```
MERGE-SORT($A, p, r$)
    if $p == r$
        Return
    $q = \lfloor \frac{p+r}{2} \rfloor$
    MERGE-SORT($A, p, q$)
    MERGE-SORT($A, q+1, r$)
    MERGE($A, p, q, r$)
```

```
MERGE($A, p, q, r$)
    // Create two temporary arrays to store
    // left and right halves of the input array
    $n_1 = q - p + 1$        // Size of left array
    $n_2 = r - q$            // Size of right array
    for $i$ = 1 to $n_1$
        $L[i] = A[p + i - 1]$
    for $j$ = 1 to $n_2$
        $R[i] = A[q + i]$
    // We need 3 indices - One for scanning
    // through $L$, one for $R$ and one for $A$
    // which will receive sorted numbers too

    $i = 1, \quad j = 1$
    for $k = p$ to $r$
        if $L[i] \leq R[j]$
            A[k] = L[i]
            $i++$
        else
            $A[k] = R[j]$
            $j++$
```

Merge Sort – In What Order?



Merge Sort – Runtime Analysis

```
MERGE($A, p, q, r$)
    // Create two temporary arrays to store
    // left and right halves of the input array
    $n_1 = q - p + 1$        // Size of left array
    $n_2 = r - q$            // Size of right array
    for $i$ = 1 to $n_1$
        $L[i]$ = $A[p + i - 1]$
    for $j$ = 1 to $n_2$
        $R[i]$ = $A[q + i]$
    // We need 3 indices - One for scanning
    // through $L$, one for $R$ and one for $A$
    // which will receive sorted numbers too
    $L[n_1 + 1] = \infty$,   $R[n_2 + 1] = \infty$
    $i = 1$,   $j = 1$
    for $k = p$ to $r$
        if $L[i] \le R[j]$
            A[k] = L[i]
            $i++$
        else
            $A[k] = R[j]$
            $j++$
```

Total number of elements in two arrays is $n_1 + n_2$

$\xrightarrow{}$ $\Theta(1)$ [Constant time]

$\xrightarrow{}$ $\Theta(n_1)$

$\xrightarrow{}$ $\Theta(n_2)$

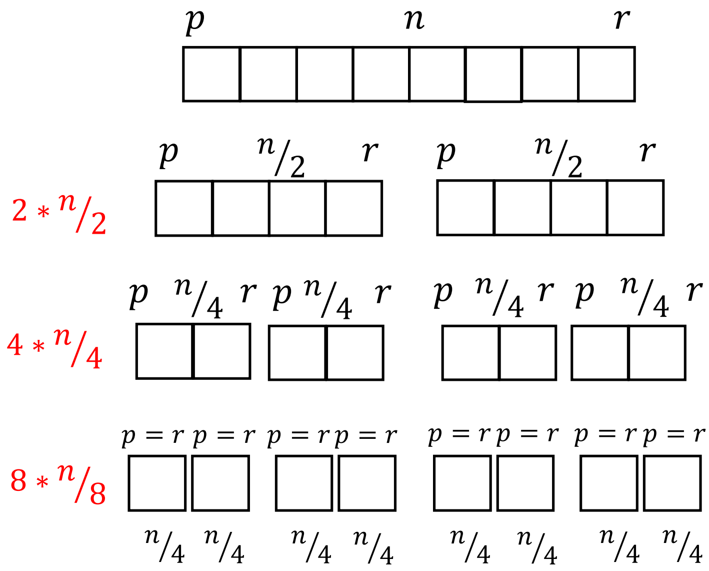
$\xrightarrow{}$ $\Theta(1)$

$\xrightarrow{}$ $\Theta(n_1 + n_2)$

Overall runtime of "MERGE" subroutine is Θ(total number of elements after merging)

Merge Sort – Runtime Analysis

```
MERGE-SORT(A, p, r)
    if $p == r$                  } → $\Theta(1)$
        Return
    $q = \lfloor \frac{p+r}{2} \rfloor$        } → $\Theta(1)$
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)            } → $\Theta(n)$
```


p n r
$2 * n/2$
p $n/2$ r p $n/2$ r
$4 * n/4$
p $n/4$ r p $n/4$ r p $n/4$ r p $n/4$ r
$8 * n/8$
$p = r$ $p = r$ $p = r$ $p = r$ $p = r$ $p = r$ $p = r$ $p = r$
$n/4$ $n/4$ $n/4$ $n/4$ $n/4$ $n/4$ $n/4$ $n/4$

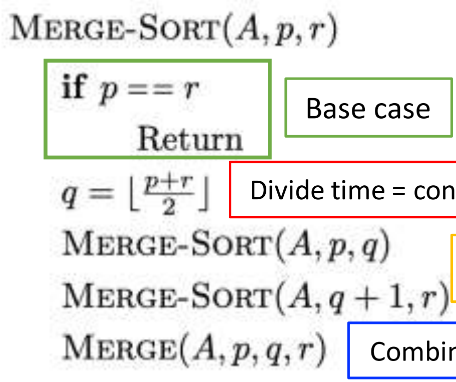
- How do we count the recursive calls?
- Recursive calls belong to the next level

Merge Sort – Runtime Analysis

- What are the number of levels in mergesort?
- $\log n$
- There are $\log n$ levels and at each level the number of operations is $\Theta(n)$
- The runtime of mergesort is $\Theta(n \log n)$

- Next we will learn how to write an equation that represents the running time of a divide and conquer algorithm.

Recurrence Relation



```
MERGE-SORT$(A, p, r)$
    if $p == r$
        Return
    $q = \lfloor \frac{p+r}{2} \rfloor$
    MERGE-SORT$(A, p, q)$
    MERGE-SORT$(A, q+1, r)$
    MERGE$(A, p, q, r)$
```

Base case

Divide time = const

Conquer time = $T(\frac{n}{2})$ each

Combine time = $\Theta(n)$

- Let $T(n)$ be the running time of the algorithm when the input size is n
- Base case takes constant time
- $T(n) = \begin{cases} c & n = 1 \\ & n > 1 \end{cases}$
- $T(n) = \begin{cases} c & n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & n > 1 \end{cases}$
- Recurrence relation: Represents the running time of a recursive algorithm

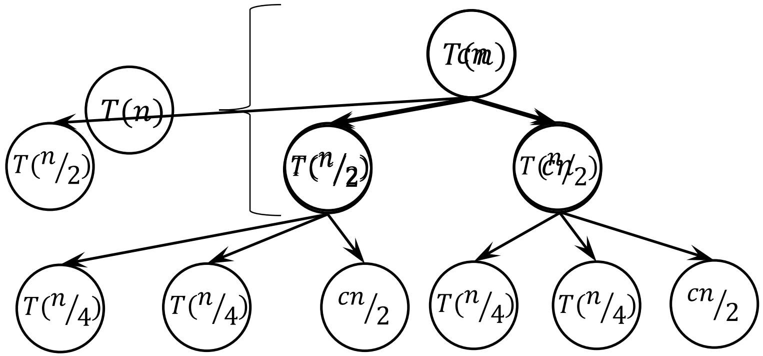
Solving Recurrences – Recursion Tree

- $T(n) = \begin{cases} c & n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & n > 1 \end{cases}$
- We can write $\Theta(n)$ as some const.*n
- Does the base case need to be always for $n = 1$?
- General form:
- $T(n) = \begin{cases} d & n = n_0 \\ 2T\left(\frac{n}{2}\right) + cn & n > n_0 \end{cases}$

Solving Recurrences – Recursion Tree

$$T(n) = \begin{cases} d & n = 1 \\ 2T\left(\frac{n}{2}\right) + cn & n > 1 \end{cases}$$

- Left hand side is $T(n)$. This is a node
- It expands to 2, $T(^n/_2)$ nodes and one cn node
- The convention is to replace $T(n)$ with these nodes
- And continue



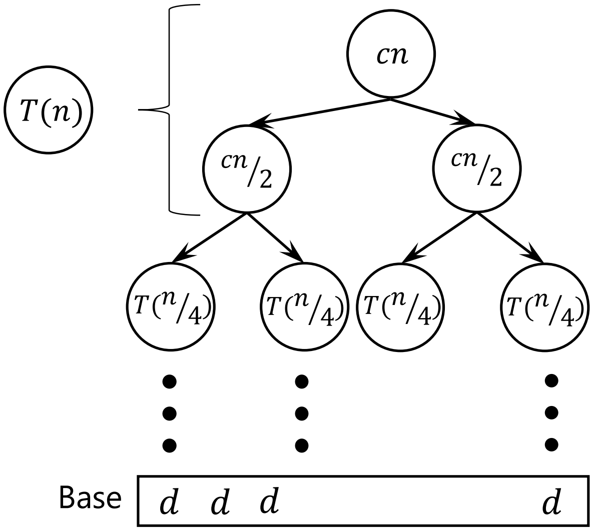
Solving Recurrences – Recursion Tree

$$T(n) = \begin{cases} d & n = 1 \\ 2T\left(\frac{n}{2}\right) + cn & n > 1 \end{cases}$$

- Left hand side is $T(n)$. This is a node
- It expands to 2, $T(^n/_2)$ nodes and one cn node
- The convention is to replace $T(n)$ with these nodes
- And continue

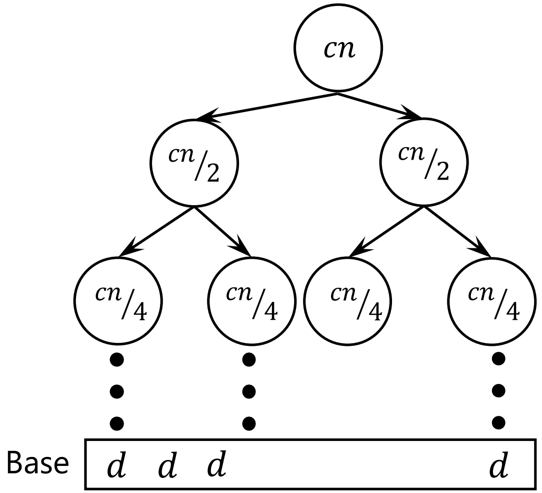
The tree ends when the base case is reached

How many d ?



Solving Recurrences – Recursion Tree

$$T(n) = \begin{cases} d & n = 1 \\ 2T\left(\frac{n}{2}\right) + cn & n > 1 \end{cases}$$

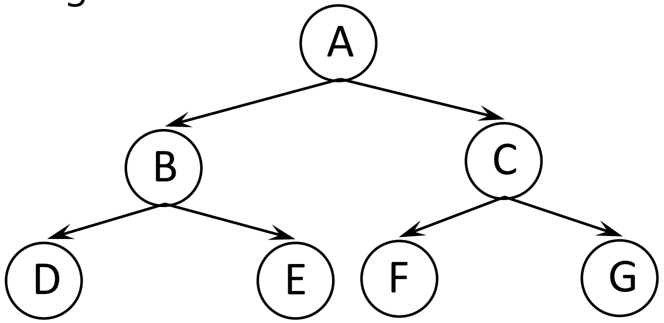


- Total runtime comes from counting number of operations from the internal nodes and counting them from base nodes
- Lets get introduced to trees and some related terminologies as we need
- More will come later

Solving Recurrences – Recursion Tree

- Graphs are sets of nodes and edges
- Trees are a form of a graph with some specialties
- Trees have direction (parent/child relationships) and don't contain cycles

- Some terminologies



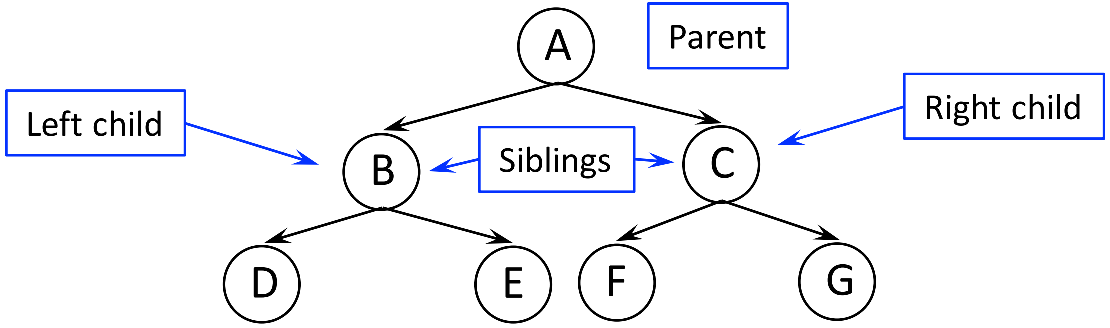
Root

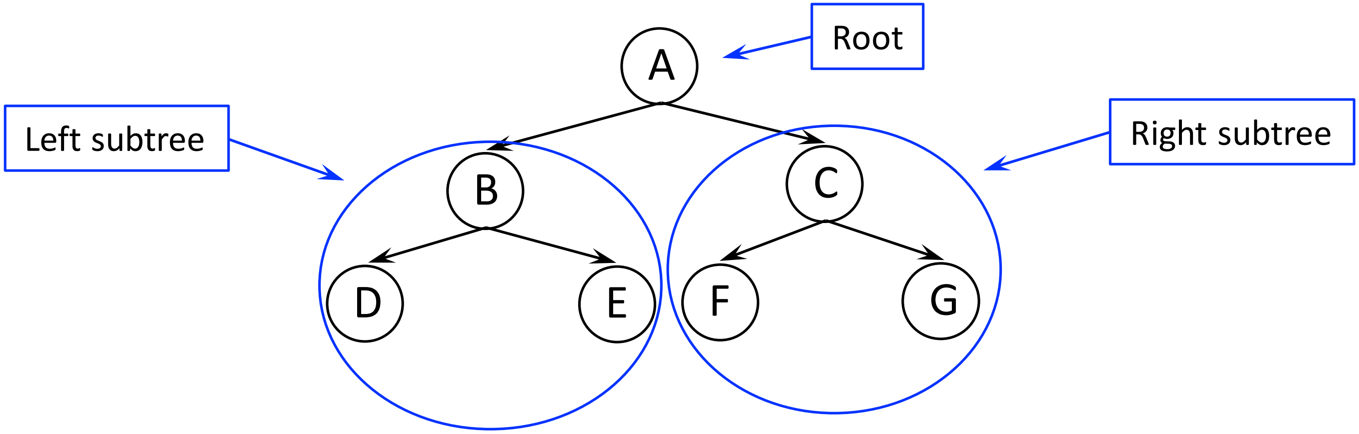
Internal

Leaf

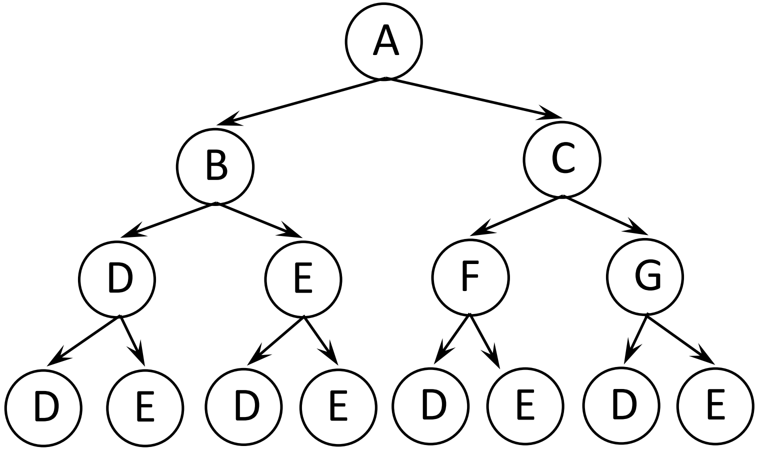
- In a Binary Tree, a node can have at most two children

Some Terminologies





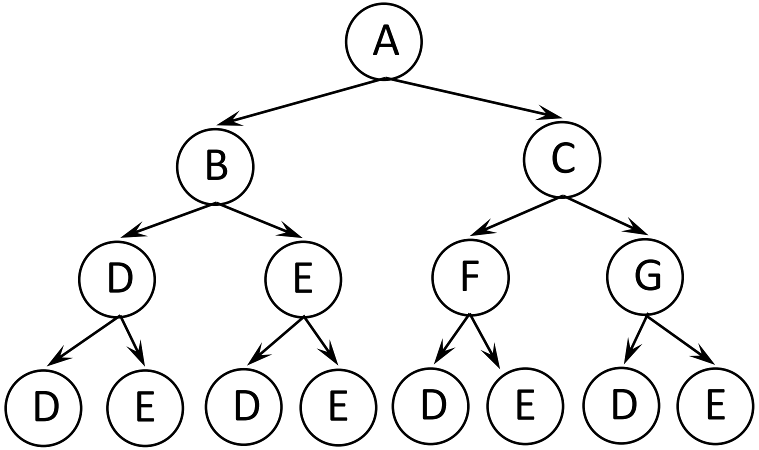
Some Terminologies



	Max # of Nodes
Level 0	$1 = 2^0$
Level 1	$2 = 2^1$
Level 2	$4 = 2^2$
Level 3	$8 = 2^3$

- Number of nodes at level $i = 2^i$
- Number of leaves $L =$ Number of nodes at level $H\ is\ 2^H$ [H is height]
- Naturally height $H = \log_2 L$
- Note: Number of Levels = Height (H) + 1
- Total number of nodes, $N = \sum_{i=0}^{H} 2^i = 2^{H+1} - 1$

Some Terminologies

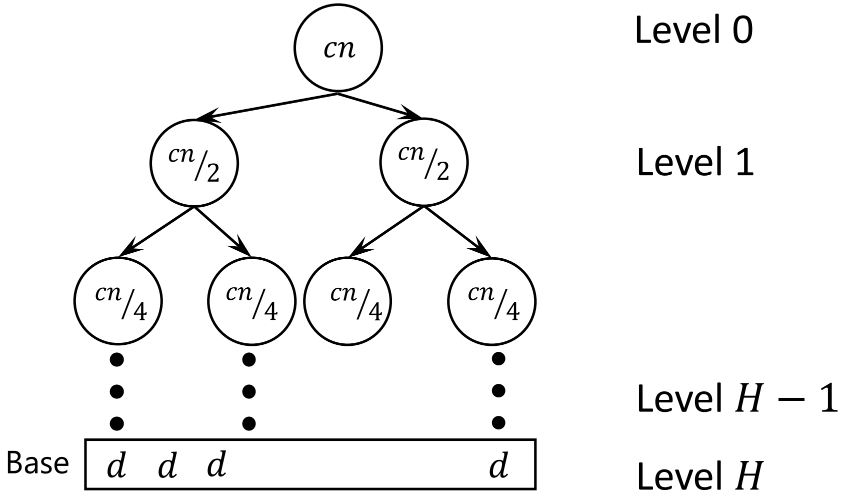


For general `k'-array Tree

- Number of nodes at level $i = k^i$
- Number of leaves $L =$ Number of nodes at level $H = k^H$ [H is height]
- Naturally height $H = \log_k L$
- Total number of nodes, $N = \sum_{i=0}^{H} k^i = \frac{k^{H+1}-1}{k-1}$

Solving Recurrences – Recursion Tree

$$T(n) = \begin{cases} d & n = 1 \\ 2T\left(\frac{n}{2}\right) + cn & n > 1 \end{cases}$$



- To get the total runtime, first we need to get the height of the tree
$$H = \log_2 n$$
- Number of leaves $L = 2^H = 2^{\log_2 n} = n$
- Base cost = $d \times L = dn = \Theta(n)$

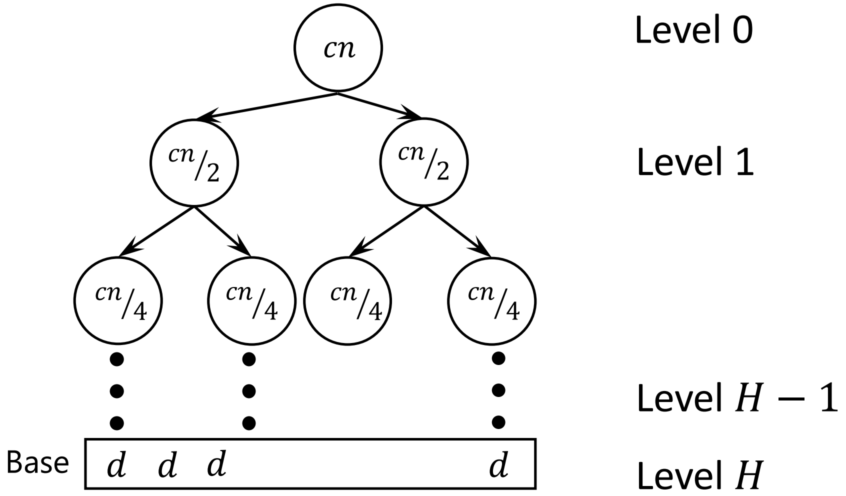
Solving Recurrences – Recursion Tree

cn

$2 * {}^{cn}/_{2} = cn$

$4 * {}^{cn}/_{4} = cn$

⋮



- What about the internal costs?
- We need to sum the costs at each internal node
- Since the costs at each level is same we can multiply by height
- Internal cost = $cn * H = cn \log_2 n = \Theta\,(n \log_2 n)$
- Total cost =$\Theta(n) + \theta(n \log_2 n) = \theta(n \log_2 n)$

Solving Recurrences – Recursion Tree

- Lets take another example

$$T(n) = \begin{cases} d & n = 1 \\ 4T\left(\frac{n}{2}\right) + cn & n > 1 \end{cases}$$

- Does it make sense?
- The general form of the recurrence can be

$$\mathrm{T(n)} = aT\left(\frac{n}{b}\right) + f(n)$$

  - Where, a is number of subproblems (branching factor)
  - b is the size of each subproblem (division ratio)
  - $f(n)$ total time for divide and combine operations

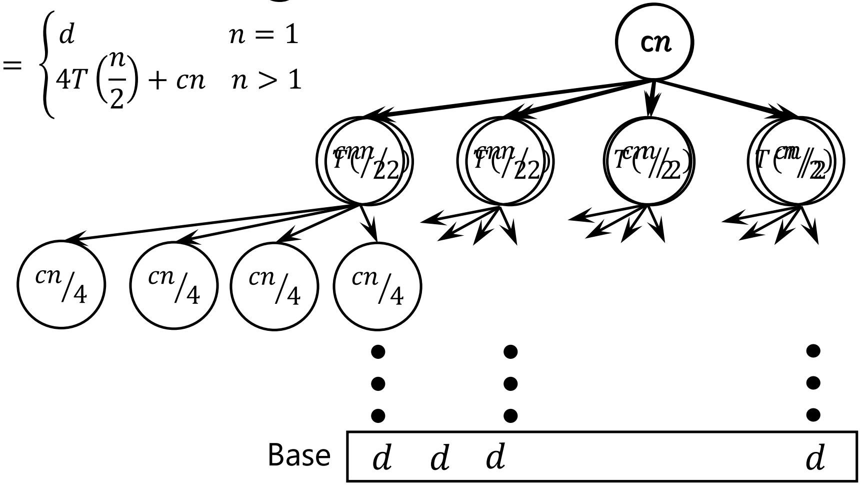
- What are these values for binary search?

$$\mathrm{T(n)} = T\left(\frac{n}{2}\right) + c$$

- with base case time - constant

Solving Recurrences – Recursion Tree

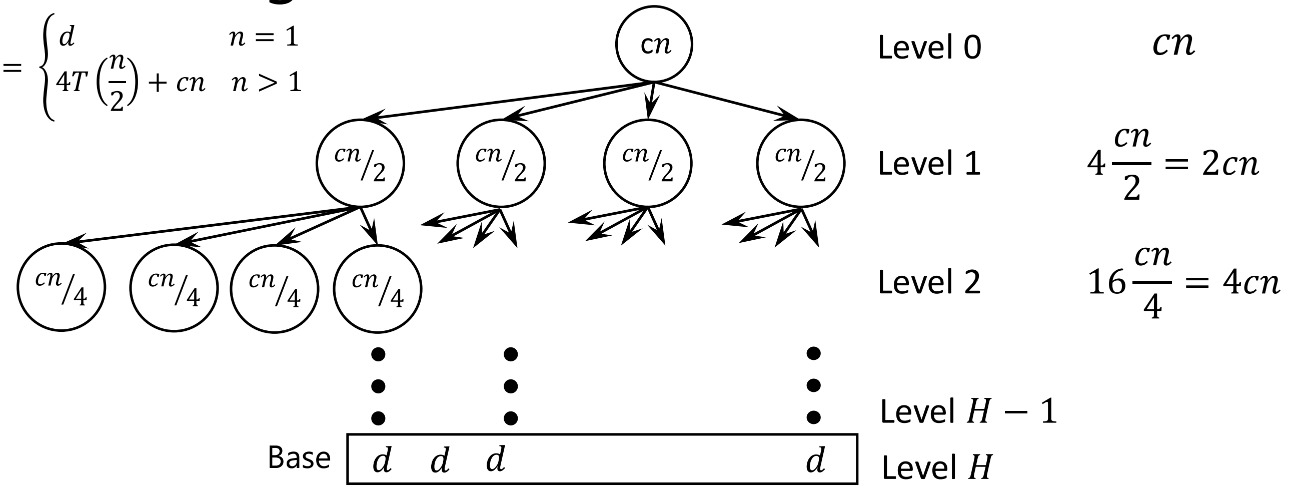
$$T(n) = \begin{cases} d & n = 1 \\ 4T\left(\frac{n}{2}\right) + cn & n > 1 \end{cases}$$



- What will be the number of leaves? More than or less than previous?
- What about the height? More than or less than previous?
- Height $H = \log_b n = \log_2 n$
- Number of leaves $L = a^H = 4^{\log_2 n} = n^2$

Solving Recurrences – Recursion Tree

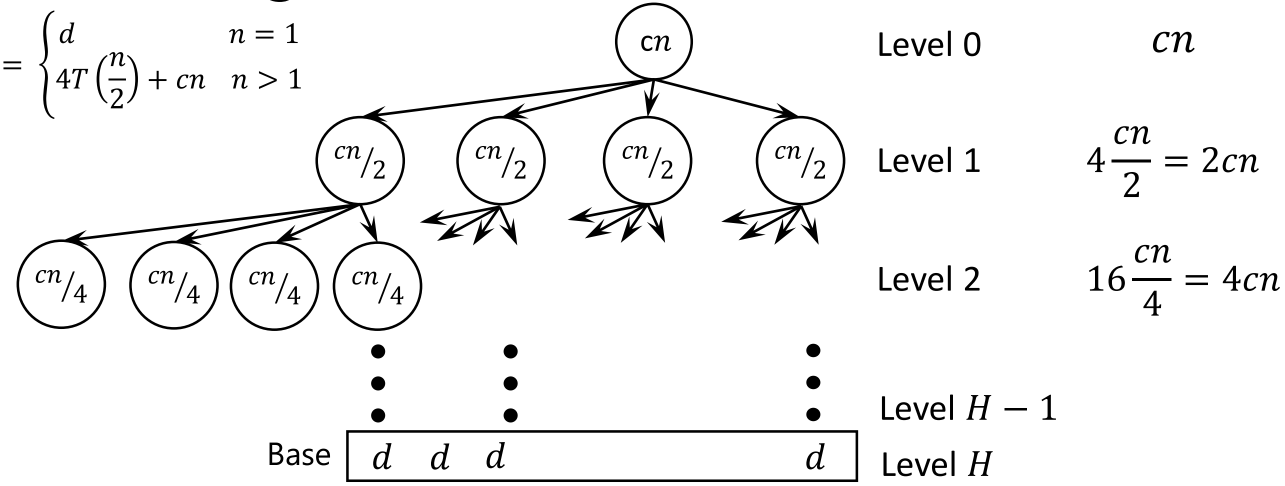
$$T(n) = \begin{cases} d & n = 1 \\ 4T\left(\frac{n}{2}\right) + cn & n > 1 \end{cases}$$



- So base cost, $= d \times L = dn^2 = \theta(n^2)$
- Now the internal cost
- So, number of operations at level i is $2^i cn$
- Internal cost = $\sum_{i=0}^{H-1} 2^i cn = cn(2^H - 1) = cn\left(2^{\log_2 n} - 1\right) = cn(n-1)$ $= cn^2 - cn$

Solving Recurrences – Recursion Tree

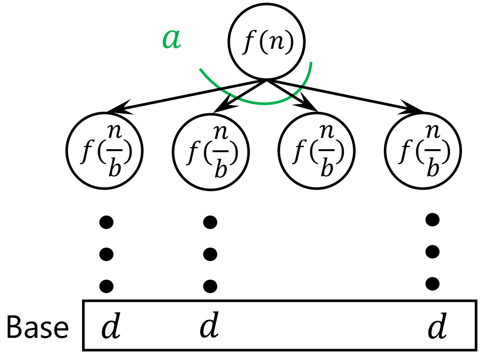
$$T(n) = \begin{cases} d & n = 1 \\ 4T\left(\frac{n}{2}\right) + cn & n > 1 \end{cases}$$



- So base cost, $= d \times L = dn^2 = \theta(n^2)$
- Internal cost = $cn^2 - cn$
- Total cost = $dn^2 + cn^2 - cn = \theta(n^2)$

Solving Recurrences – Master Theorem

- Review of the general procedure
- General form:
- $T(n) = \begin{cases} d & n \le n_0 \\ aT\left(\frac{n}{b}\right) + f(n) & n > n_0 \end{cases}$
  - where d is base case runtime
  - a is number of subproblems (branching factor)
  - b is the size of each subproblem (division ratio)
  - $f(n)$ total time for divide and combine operations
- Compute Height $H = \log_b n$
- Compute number of leaves $L = a^H = a^{\log_b n} = n^{\log_b a}$
- Compute base cost (Generally - $L \times d$)
- Compute internal cost $f(n) + af\left(\frac{n}{b}\right) + a^2 f\left(\frac{n}{b^2}\right) + \cdots$
- Sum base and internal cost



Solving Recurrences – Master Theorem

- Master theorem provides a consolidated set of rules
- It depends on the relation between the number of leaves ($L = n^{\log_b a}$) and the function $f(n)$
- <u>Case I</u>: If L is **polynomially** larger i.e., $f(n) = O\left(n^{\log_b a - \Delta}\right)$, where Δ is some constant > 0, then $T(n) = \Theta(n^{\log_b a})$
- Δ is the difference in the polynomial degree between L and $f(n)$
- If $L = n^2$ and $f(n) = n$, then Δ=? → 1
- Similarly, if $L = n^2$ and $f(n) = n^{\frac{3}{2}}$, then Δ=? → 0.5
- Another way of thinking this is – the ratio between L and $f(n)$ must be at least a polynomial
- Take $L = n^2$ and $f(n) = \frac{n^2}{\log n}$, Then $\frac{L}{f(n)} = \log n$ and $\log n$ is less than polynomial

Solving Recurrences – Master Theorem

- Case I: If L is **polynomially** larger i.e., $f(n) = O\left(n^{\log_b a - \Delta}\right)$, where Δ is some constant > 0, then $T(n) = \Theta(n^{\log_b a})$
  - Δ is the difference in the polynomial degree between L and $f(n)$
- Case II: If $f(n)$ and L are asymptotically same, i.e., $f(n) = \Theta\left(n^{\log_b a}\right)$ then $T(n) = \Theta\left(n^{\log_b a} \log n\right) = \Theta(f(n) \log n)$
- Case III: If $f(n)$ is **polynomially** larger i.e., $f(n) = \Omega\left(n^{\log_b a + \Delta}\right)$, then $T(n) = \Theta(f(n))$ if $\boxed{af(\frac{n}{b}) \leq kf(n) \text{ for } k < 1}$ → Regularity Condition

$$T(n) = 2T\left(\frac{n}{2}\right) + cn$$
$$a = 2, b = 2$$
$$L = n^{\log_b a} = n$$
$$f(n) = cn$$
Case II: $T(n) = \Theta(n \log n)$

$$T(n) = 4T\left(\frac{n}{2}\right) + cn$$
$$a = 4, b = 2$$
$$L = n^{\log_b a} = n^2$$
$$f(n) = cn$$
Case I: $T(n) = \Theta\left(n^{\log_b a}\right) = \Theta(n^2)$

Quicksort

- Quicksort is a sorting algorithm which puts elements in the right places one by one
- What is a “right place”?
- Is there any element which is at the right place?

3	7	9	5	15	11	4

4	7	9	5	15	11	17

4	7	5	9	15	11	12

Quicksort

- Lets walkthrough an example to understand how quicksort works

7	6	10	5	9	2	1	15	7

- The crucial step in quicksort is partioning (divide into two parts such that left of **pivot** is less and right of pivot id more)

Quicksort

```
PARTITION$(A, p, r)$
    // Initialize the pivot, start and end pointer
    $pivot = A[p]; start = p; end = r$
    while $start < end$
        while $pivot \geq A[start]$
            $start++$
        while $pivot < A[end]$
            $end--$
        if $start < end$
            SWAP$(A[start], A[end])$
    // When start and end cross over
    // swap pivot and end
    SWAP$(pivot, A[end])$
    // Return final position of pivot
    Return $end$
```

```
QUICKSORT$(A, p, r)$
    if $p < r$
        $q =$ PARTITION$(A, p, r)$
        QUICKSORT(A, p, q-1)
        QUICKSORT(A, q+1, r)
```

Quicksort – Runtime Analysis

```
PARTITION($A, p, r$)
    // Initialize the pivot, start and end pointer
    $pivot = A[p]; start = p; end = r$
    while $start < end$
        while $pivot \geq A[start]$
            $start++$
        while $pivot < A[end]$
            $end--$
        if $start < end$
            SWAP($A[start], A[end]$)
    // When start and end cross over
    // swap pivot and end
    SWAP($pivot, A[end]$)
    // Return final position of pivot
    Return $end$
```

$\Theta(1)$

$\Theta(n)$

Overall runtime of "PARTITION" subroutine is $\Theta(n)$

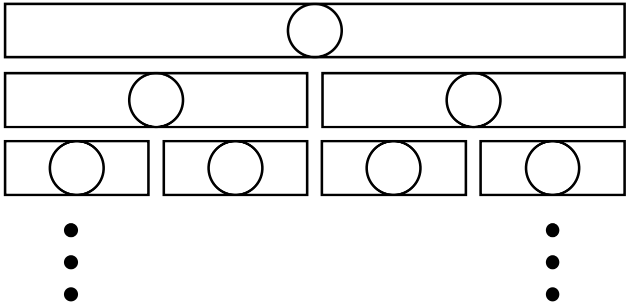
$\Theta(1)$

Quicksort – Runtime Analysis

```
QUICKSORT(A, p, r)
    if $p < r$
        $q$ = PARTITION(A, p, r)   ⟶ $\Theta(n)$
        QUICKSORT(A, p, q-1)
        QUICKSORT(A, q+1, r)       } Hard to tell about these.
```

- Lets take two extreme cases

Partitions are perfectly balanced everytime



- Number of levels = $\log_2 n$
- Recurrence relation

$$T(n) = 2T\left(\frac{n}{2}\right) + cn$$

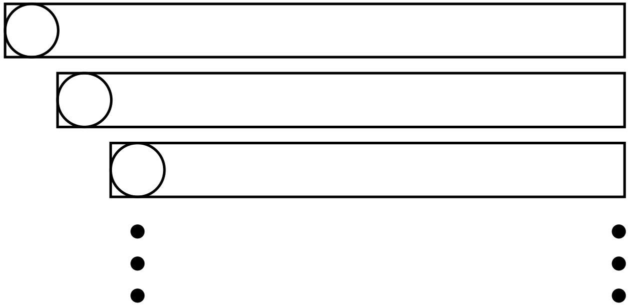
- Same as mergesort – $\Theta(n \log n)$
- However, this is the best case scenario

Quicksort – Runtime Analysis

```
QUICKSORT(A, p, r)
    if $p < r$
        $q = $ PARTITION(A, p, r)    → $\Theta(n)$
        QUICKSORT(A, p, q-1)
        QUICKSORT(A, q+1, r)         } Hard to tell about these.
```

- Lets take two extreme cases

Partitions are most unbalanced everytime

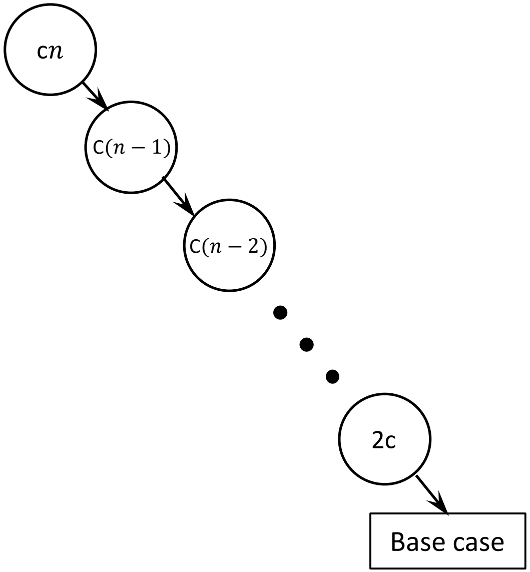


- Number of levels = $\Theta(n)$
- Recurrence relation

$$T(n) = T(n-1) + \underbrace{T(0)}_{Const.} + cn$$

- Lets try to solve this recusrsive eqn.

Quicksort – Runtime Analysis



- Base case = $const.$
- Internal cost at level i is $c(n-i)$
- Total internal cost

$$c(2+3+\cdots n)$$

- Number of levels = $\Theta(n)$
- Recurrence relation

$$T(n) = T(n-1) + \cancel{T(0)}^{\color{red}{Const.}} + cn$$

- Same as insertion sort – $\Theta(n^2)$
- This is the worst case scenario

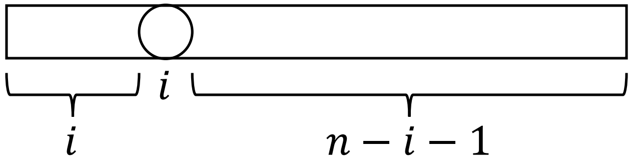
Quicksort – Runtime Analysis

- What should the input look like for the best case?
    - The medians are chosen as pivots always
- What should the input look like for the worst case?
    - The input array is already sorted
- Probability of getting an already sorted array is $\frac{1}{n!}$

- However, the problem can be tackled if instead of always taking the first element as pivot, we take any element as pivot in random
- The average-case time complexity of the randomized version of quicksort is $\Theta(n \log n)$
- The proof of this involves a more complicated recurrence than we have seen so far.

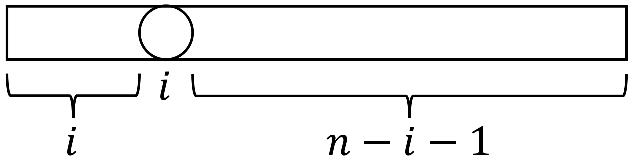
Quicksort – Runtime Analysis

- We will follow "Introduction to Algorithms and Data Structures" by M. J. Dinneen, G. Gimelfarb, M. C. Wilson for the proof
- Let $T(n)$ denote the average-case running time for size n list
- In the first step, the time taken to compare all the elements with the pivot is cn
- Let $0 \leq i \leq n-1$ is the final position of the pivot. So, the two sublists are of size i and $n-i-1$ respectively



- The recurrence relation for this i is $T(n) = T(i) + T(n-1-i) + cn$
    - Small detail: Last term should be $\Theta(n)$, as it depends on the exact implementaiton (e.g., whether comparison starts from pivot or one element after, etc.)

Quicksort – Runtime Analysis

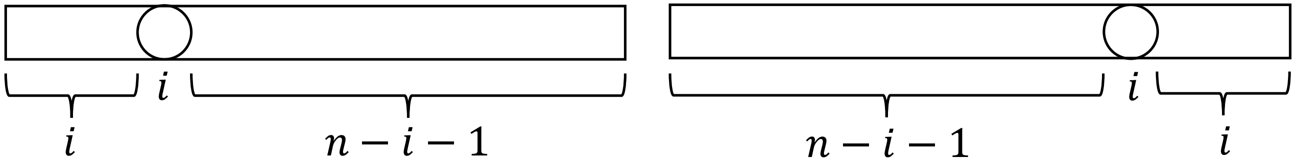


- The recurrence relation for this i is $T(n) = T(i) + T(n-1-i) + cn$
- The final pivot position i can be any of the n positions with equal probability
- So, the value of $T(n)$ to be $T(0) + T(n-1) + cn$ (i.e., $i = 0$) has probability $\frac{1}{n}$
- Similarly, the value of $T(n)$ to be $T(1) + T(n-2) + cn$ (i.e., $i = 1$) has probability $\frac{1}{n}$ and so on
- So the expected value of $T(n)$ is given by,

$$T(n) = \frac{1}{n}\sum_{i=0}^{n-1}(T(i) + T(n-1-i) + cn) = \frac{1}{n}\sum_{i=0}^{n-1}\big(T(i) + T(n-1-i)\big) + cn$$

Quicksort – Runtime Analysis

- $T(n) = \frac{1}{n}\sum_{i=0}^{n-1}(T(i) + T(n-1-i)) + cn$
- $T(0) + T(n-1) = T(n-1) + T(0)\ for\ i = 0\ and\ i = n-1$
- $T(1) + T(n-2) = T(n-2) + T(1)\ for\ i = 1\ and\ i = n-2$ and so on



- $T(n) = \frac{1}{n}\sum_{i=0}^{n-1}(T(i) + T(n-1-i)) + cn = \frac{2}{n}\sum_{i=0}^{n-1}T(i) + cn$

- Let us rewrite this as $nT(n) = 2\sum_{i=0}^{n-1}T(i) + cn^2$

Quicksort – Runtime Analysis

- $nT(n) = 2\sum_{i=0}^{n-1} T(i) + cn^2$
- Replace n by $n-1$ and subtract from the above
- $$nT(n) = 2\big(T(0) + T(1) + \cdots T(n-2) + T(n-1)\big) + cn^2$$
  $$(n-1)T(n-1) = 2(T(0) + T(1) + \cdots + T(n-3) + T(n-2)) + c(n-1)^2$$
- $nT(n) - (n-1)T(n-1) = 2T(n-1) + c(n^2 - (n-1)^2)$
- $nT(n) = (n+1)T(n-1) + c(2n-1)$
- Divide everything by $n(n+1)$
- $\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + c\frac{2n-1}{n(n+1)} = \frac{T(n-1)}{n} + c\left(\frac{3}{n+1} - \frac{1}{n}\right)$
- Last term uses partial fraction decomposition

Quicksort – Runtime Analysis

- $\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + c\,\frac{2n-1}{n(n+1)} = \frac{T(n-1)}{n} + c\left(\frac{3}{n+1} - \frac{1}{n}\right)$
- Lets put $n = 1, 2,3, \ldots n$

- $\cancel{\frac{T(1)}{2}} = \frac{T(0)}{1} + \frac{3c}{2} - \frac{c}{1}$
- $\cancel{\frac{T(2)}{3}} = \cancel{\frac{T(1)}{2}} + \frac{3c}{3} - \frac{c}{2}$
- $\cancel{\frac{T(3)}{4}} = \cancel{\frac{T(2)}{3}} + \frac{3c}{4} - \frac{c}{3}$
- $\frac{T(n)}{n+1} = \cancel{\frac{T(n-1)}{n}} + \frac{3c}{n+1} - \frac{c}{n}$
- $\frac{T(n)}{n+1} = \frac{T(0)}{1} + 3c\left(\frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n+1}\right) - c\left(\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}\right)$

Quicksort – Runtime Analysis

- $\frac{T(n)}{n+1} = \frac{T(0)}{1} + 3c\left(\frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n+1}\right) - c\left(\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}\right)$
- $\frac{T(n)}{n+1} = \frac{T(0)}{1} - c + \frac{3c}{n+1} + 2c\left(\frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n}\right) = d - c + \frac{3c}{n+1} + 2c(H_n - 1)$
- $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$; $d = T(0)$
- $T(n) = (d - c)(n + 1) + 3c + 2c(n + 1)(H_n - 1)$
- $\quad = d + (d - 3c)n + 2cH_n + 2cnH_n$
- Using a result $H_n = \Theta(\log n)$, we can write
- $T(n) = d + (d - 3c)n + 2c\Theta(\log n) + 2c\Theta(n \log n) = \Theta(n \log n)$
- $H_n = \Theta(\log n)$ -> This will shown to be true for a special case, next

Quicksort – Runtime Analysis

- In case, the number of terms in the Harmonic series is a perfect power of 2,i.e., $n = 2^k$ or $k = \log_2 n$,

- $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{2^k} \leq 1 + \left(\frac{1}{2} + \frac{1}{2}\right) + \left(\frac{1}{4} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4}\right) + (\frac{1}{8} + \cdots$

  $k = 1$
  $k = 2$
  $k = 3$

- All the bracketed sums are 1

- So, $H_n \leq k + \frac{1}{n} = \log_2 n + \frac{1}{n} \leq \log_2 n + 1$ as $1 > \frac{1}{n}$

Quicksort – Runtime Analysis

- So, $H_n \leq \log_2 n + 1$ … (1)

- Again, for $n = 2^k$ or $k = \log_2 n$,
- $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} \ldots + \frac{1}{2^k}$
- $\geq 1 + \frac{1}{2} + \left(\frac{1}{4} + \frac{1}{4}\right) + \left(\frac{1}{8} + \frac{1}{8} + \frac{1}{8} + \frac{1}{8}\right) + \left(\frac{1}{2^k} + \cdots + \frac{1}{2^k}\right)$
- $= 1 + \left(\frac{1}{2} + \frac{1}{2} + k\ times\right) = 1 + \frac{1}{2}k \geq \frac{1}{2}k = \frac{1}{2}\log_2 n$

- $H_n \geq \frac{1}{2}\log_2 n$ … (2)
- Combining (1) and (2), $\frac{1}{2}\log_2 n \leq H_n \leq \log_2 n$ +1
- $H_n = \Theta(\log_2 n)$

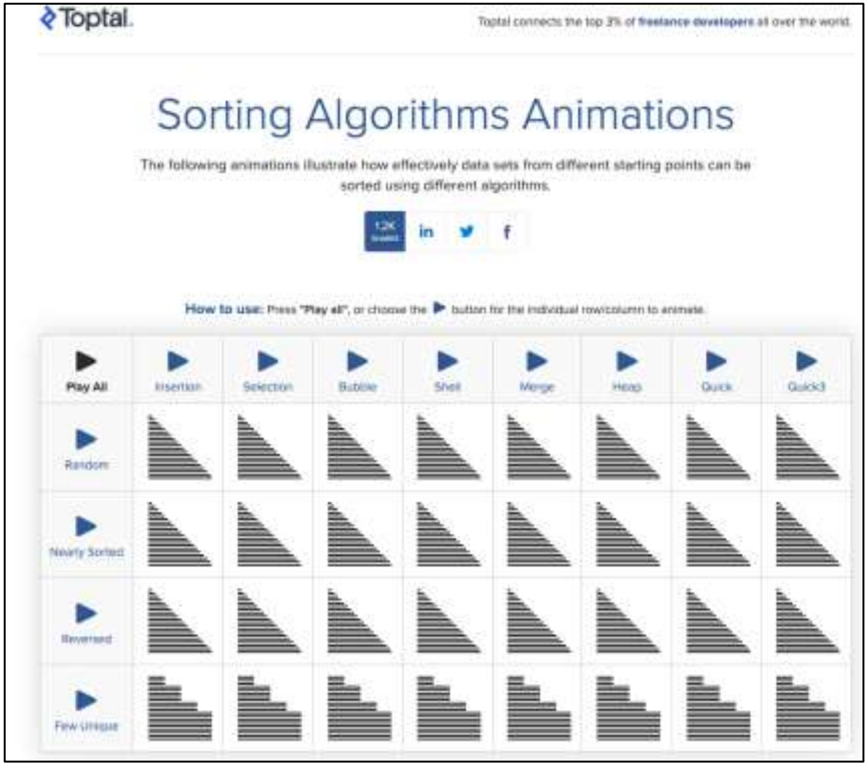
Quicksort

- Quicksort was proposed by Sir Charles Antony Richard (Tony) Hoare in 1959/60 at age 26

source: http://curation.cs.manchester.ac.uk/Turing100/www.turing100.manchester.ac.uk/speakers/invited-list/11-speakers/39-hoare.html

- He was trying to sort words for a machine translation project from Russian to English
- In his own words – "I have been very lucky. What a wonderful way to start a career in Computing, by discovering a new sorting algorithm!"
- Emeritus Professor, Oxford University
- Turing award winner, 1980.

A Quick Animation



Toptal. Toptal connects the top 3% of freelance developers all over the world.

Sorting Algorithms Animations

The following animations illustrate how effectively data sets from different starting points can be sorted using different algorithms.

How to use: Press "Play all", or choose the ▶ button for the individual row/column to animate.

Click to play https://tinyurl.com/3k8pt3a3

Median Finding

- Given a set of numbers, a median is the "halfway point" of the set
- More formally, given a set of n numbers, medians occur at $i = \lfloor \frac{n+1}{2} \rfloor$ (**lower median**) and at $i = \lceil \frac{n+1}{2} \rceil$ (**upper median**)
- Following the book, by "median", we will mean the lower median

- We would like to find the median.
- If a list is sorted, the median can be easily found out. But that means at least $\Theta(n \log n)$ operation
- We would like to do it faster than that

- We would like to find i^{th} rank element in a sorted list

Median Finding

- The problem is called the **selection problem**
- **Input**: Set A of n (distinct) numbers and an integer i, with $1 \leq i \leq n$
- **Output**: The element $x \in A$ that is larger than exactly $i - 1$ other elements of A

- The idea we will be using is that after the partition operation we know the “right” position of the pivot element. We will use it to discard some “obvious” elements

6	2	9	1	12	5	10	3

Median Finding - PseudoCode

Median Finding – Running Time Analysis

- Best case is that the first pivot is i
- What is the running time?
- $\Theta(n)$ [At least once you have to scan the array to get the partition]
- What is the worst case?
- (Similar to quicksort) To get the pivot always at one of the ends
- The running time is $\Theta(n^2)$ – Same as quicksort

- Another best case scenario is – perfectly balanced partitioning at all levels
- Recurrence relation for such case is
- $$T(n) = T\left(\frac{n}{2}\right) + cn$$

Median Finding – Running Time Analysis

$$T(n) = T\left(\frac{n}{2}\right) + cn$$

$$a = 1, b = 2$$

$$L = n^{\log_b a} = n^0 = 1$$

$$f(n) = cn$$

Case III: We need to check the regularity condition

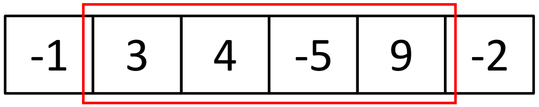
$af\left(\frac{n}{b}\right) \leq kf(n) \Rightarrow c\frac{n}{2} \leq kcn$ – This is true for $\frac{1}{2} \leq k < 1$

$$T(n) = \Theta(n)$$

- Just like quicksort, it can be proven that the average case complexity is also of the same order as the best case complexity i.e., $\Theta(n)$

Maximum Sum Subarray

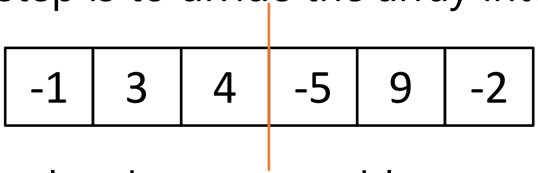
- Given an array, we need to find a subarray whose sum of elements among all possible subarray sums is the maximum



- The problem will be trivial if the numbers are all positive
- By inspection the solution is the subarray from 3 to 9
- What can be a brute-force strategy to find this?
- As we are looking for start and end positions of the subarray, we can try for all possible start indices and all possible end indices
- This means n-choose-2 i.e., $\Theta(n^2)$ algorithm
- Can we use divide-and-conquer paradigm to improve the runtime?

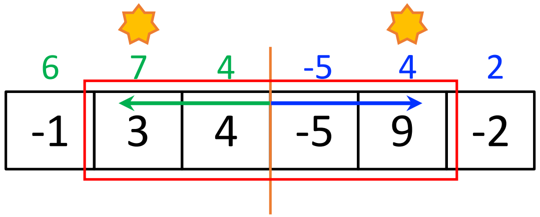
Maximum Sum Subarray

- The natural first step is to divide the array into half



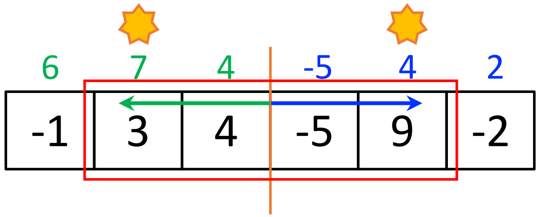
- Then recursively solve the same problem on the divided subarrays
- Next we need to combine. While combining, the maximum sum can come from the two subarrays as well from an array that spans across the two subarrays
- We need to find the maximum sum corresponding to the subarray crossing the centerline. And we need to find it efficiently.
- If we can find the maximum crossing subarray in $\Theta(n)$ time, then we have a chance to get $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$ recursion leading to $\Theta(n \log n)$ overall runtime

Maximum Sum Subarray



- To do so, we will cleverly use the fact that we are trying to find a subarray whose one end lies on the right of the centerline and the other end lies on the left
- The maximum subarray crossing the centerline will be a concatenation of 1) a subarray whose one end is the centerline and goes to the left 2) a subarray whose one end is the centerline and goes to the right
- So, start at the centerline, keep going to the left, keep doing the sum and keep track of the maximum of the sum. Similarly, for the subarray going to the right from the centerline

Maximum Sum Subarray



- Now we need to compare three sums – 1) one from entirely the left subarray 2) one from entirely the right subarray 3) one from the maximum crossing subarray
- Maximum of these three we will return
- Runtime of finding the maximum crossing subarray?
- $\Theta(n)$

Maximum Sum Subarray - Pseudocode

```
MAXCROSSING($A, p, q, r$)
    // Initialize a very low value and sum
    $left\text{-}sum = -\infty, sum = 0$
    for $i = q$ downto $p$
        $sum = sum + A[i]$
        if $sum > left\text{-}sum$
            $left\text{-}sum = sum$
            $max\text{-}left = i$
    // Process the right subarray
    $right\text{-}sum = -\infty, sum = 0$
    for $i = q + 1$ to $r$
        $sum = sum + A[i]$
        if $sum > right\text{-}sum$
            $right\text{-}sum = sum$
            $max\text{-}right = i$
    return $max\text{-}left, max\text{-}right, left\text{-}sum + right\text{-}sum$
```

```
MAXSUBARRAY($A, p, r$)
    // Base case
    if $p == r$
        return p, r, A[p]
    $q = \lfloor \frac{p+r}{2} \rfloor$
    ($left\text{-}p, left\text{-}r, left\text{-}sum$) = MAXSUBARRAY(A,p,q)
    ($right\text{-}p, right\text{-}r, right\text{-}sum$) = MAXSUBARRAY(A,q+1,r)
    ($cross\text{-}p, cross\text{-}r, cross\text{-}sum$) = MAXCROSSING(A,p,q,r)
    $max\text{-}sum$ = MAX(left-sum,right-sum,cross-sum)
    if $max\text{-}sum == left\text{-}sum$
        return $left\text{-}p, left\text{-}r, left\text{-}sum$
    if $max\text{-}sum == right\text{-}sum$
        return $right\text{-}p, right\text{-}r, right\text{-}sum$
    else
        return $cross\text{-}p, cross\text{-}r, cross\text{-}sum$
```

Maximum Sum Subarray Runtime Analysis

```
MAXCROSSING($A, p, q, r$)
    // Initialize a very low value and sum                  } → $\Theta(1)$
    $left\text{-}sum = -\infty$, $sum = 0$                  }
    for $i = q$ downto $p$                                  }
        $sum = sum + A[i]$                                  }
        if $sum > left\text{-}sum$                          }
            $left\text{-}sum = sum$                         }
            $max\text{-}left = i$                           }
    // Process the right subarray                           }
    $right\text{-}sum = -\infty$, $sum = 0$                 } → $\Theta(n)$
    for $i = q + 1$ to $r$                                  }
        $sum = sum + A[i]$                                  }
        if $sum > right\text{-}sum$                         }
            $right\text{-}sum = sum$                        }
            $max\text{-}right = i$                          }
    return $max\text{-}left, max\text{-}right, left\text{-}sum + right\text{-}sum$   } → $\Theta(1)$
```

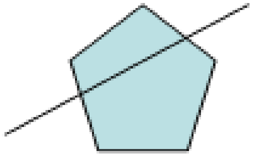
Maximum Sum Subarray Runtime Analysis

$$\boxed{T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) \quad \Rightarrow \Theta(n \log n)}$$

```
MAXSUBARRAY($A, p, r$)
    // Base case
    if $p == r$                                                    $\rightarrow \Theta(1)$
        return p, r, A[p]
    $q = \lfloor \frac{p+r}{2} \rfloor$                                        $\rightarrow \Theta(1)$
    ($left$-$p$, $left$-$r$, $left$-$sum$) = MAXSUBARRAY(A,p,q)          $\rightarrow T(\frac{n}{2})$
    ($right$-$p$, $right$-$r$, $right$-$sum$) = MAXSUBARRAY(A,q+1,r)     $\rightarrow T(\frac{n}{2})$
    ($cross$-$p$, $cross$-$r$, $cross$-$sum$) = MAXCROSSING(A,p,q,r)     $\rightarrow \Theta(n)$
    $max$-$sum$ = MAX(left-sum,right-sum,cross-sum)                 $\rightarrow \Theta(1)$
    if $max$-$sum$ == $left$-$sum$
        return $left$-$p$, $left$-$r$, $left$-$sum$
    if $max$-$sum$ == $right$-$sum$
        return $right$-$p$, $right$-$r$, $right$-$sum$
    else
        return $cross$-$p$, $cross$-$r$, $cross$-$sum$
```

Finding Convex Hull

- We will now see two examples in computational geometry where divide and conquer is used effectively
- The first problem is finding the **convex hull** of a set of points
- Formally - The convex hull of a set S of points is the smallest **convex polygon** containing all points in S
- So, what is a convex polygon?
- A convex polygon is a polygon with all its interior angles $< 180°$
    - A line drawn through a convex polygon will intersect the polygon exactly twice
    - All the diagonals of a convex polygon lie entirely inside the polygon



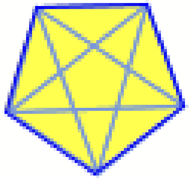
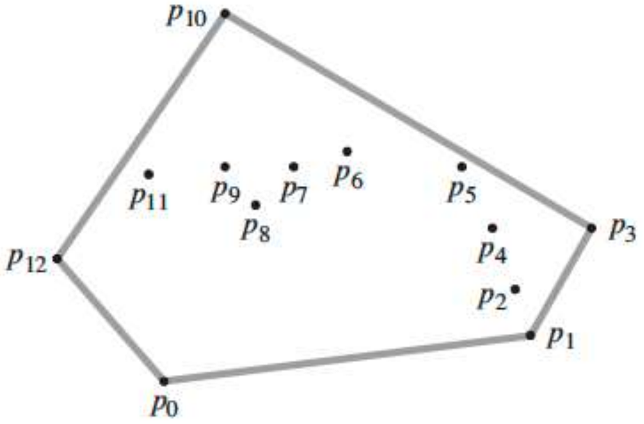
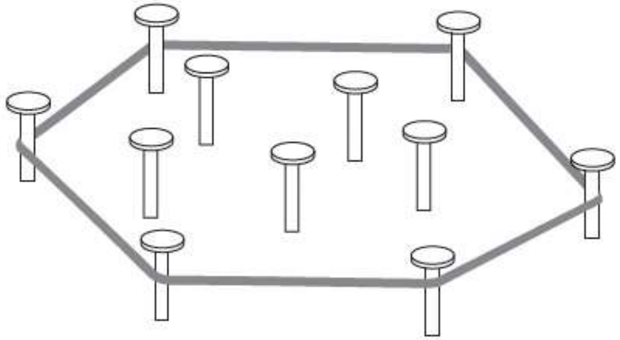


Image source: Math Open Reference

Finding Convex Hull

- Convex hull of a set of points in a plane is the smallest convex polygon that contains all of them either inside or on its boundary





- Imagine that the points are represented by nails sticking out from a board. Convex hull is the shape formed by a tight rubber band that surrounds all the nails

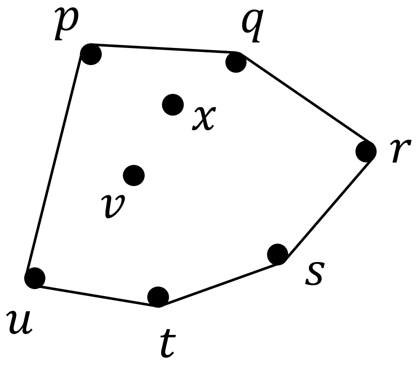
Finding Convex Hull

- We would like to have an algorithm that finds the segments/sides of the convex hull given the set of points (their (x, y) co-ordinates)
- The book has two efficient algorithms - Graham's scan and Jarvis's march

- However, we would follow a divide-and-conquer algorithm, famously called as "two finger" algorithm by Prof. Srini Devadas, MIT in his "Design and Analysis of Algorithms" course
- Much of the material is taken from his lecture [Link].

Finding Convex Hull – Problem Definition

- Given n points in plane i.e., set $\mathbb{S} = \{(x_i, y_i) | i = 1,2, \ldots, n\}$ and assuming no two points having same x coordinates, no two points having same y coordinates and no three points in a line, we need to find the convex hull (CH($\mathbb{S}$)) containing all points in $\mathbb{S}$
- The assumptions are not limiting, they are just convenient

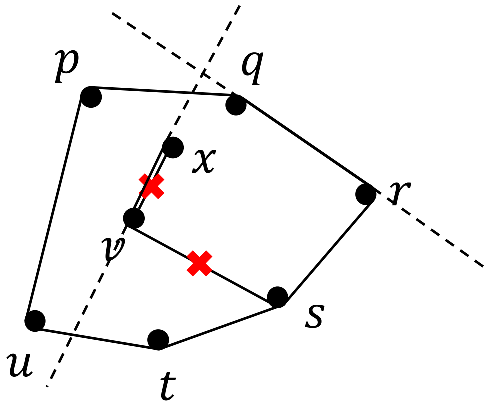


- We will use the following convention to represent the hull
- Sequence of points on the boundary in clockwise manner

$$p - q - r - s - t - u - p$$

Finding Convex Hull – Brute Force

- Can we think of a simple (can be brute force) algorithm to generate the segments comprising the convex hull?
- Draw a line segment connecting a pair of points and check if all the other points are on one side of the segment



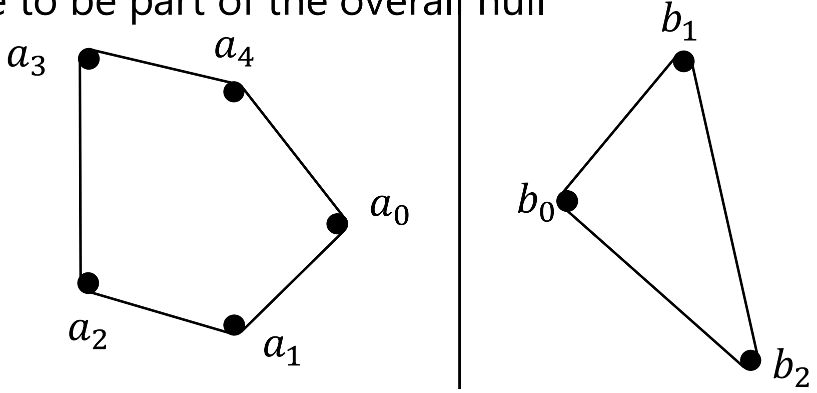
- Assuming n points, what is the complexity?
- $\Theta(n^2)$ number of segments
- We need to test roughly n points against each segment
- So, total runtime complexity of the brute-force approach is $\Theta(n^3)$

Finding Convex Hull – Divide and Conquer

- Lets try to apply divide and conquer
- ‘Divide’ is pretty straight-forward - dividing them roughly into half
- The points are sorted in terms of x-coordinates. Everything to the left of the median is one subproblem and everything to the right is another subproblem
- Note that sorting is a single time operation. And it is $\Theta(n \log n)$
- Solve for convex hulls for the subproblems
- Base case is when the number of points are small – say 2 or 3
- Now merging the solutions is where things start to happen

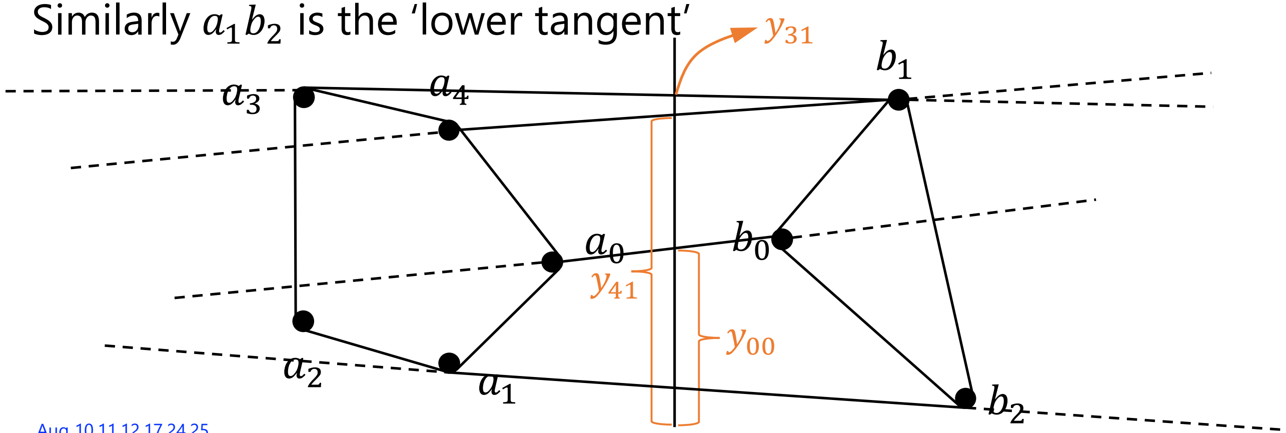
Finding Convex Hull – How to Merge?

- Lets try to apply divide and conquer
- The naming convention is – for the left subproblem, the first point is the closest to the dividing line (highest x). Similarly, for the right subproblem, the first point is the closest to the dividing line (least x)
- We have got the sub-hulls. To generate the 'combined' hull, we need to generate additional segments that are not part of the sub-hulls but are part of the overall hull. Similarly we need to remove the ones that cease to be part of the overall hull



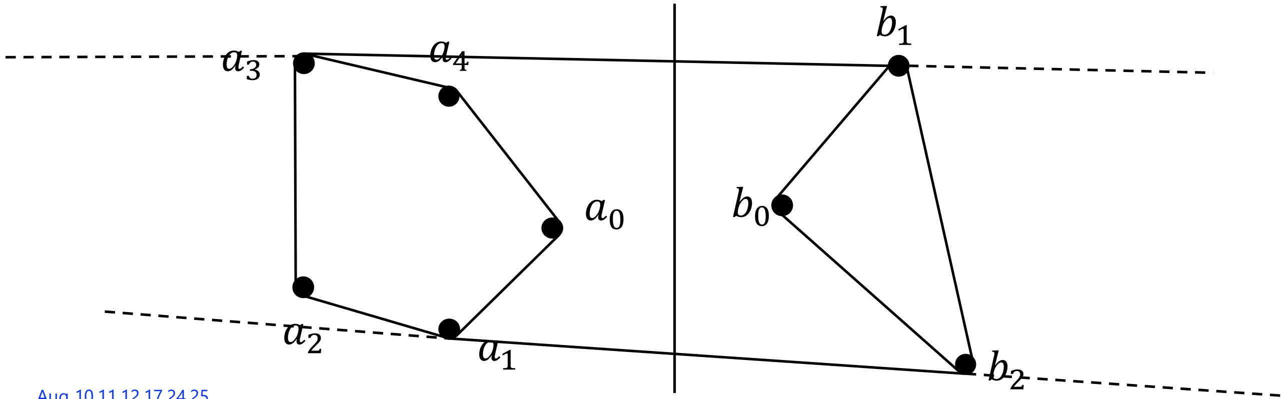
Finding Convex Hull – How to Merge?

- One way can be picking one point from either side and check if all other points are on one side of the segment joining the two
- $a_3 b_1$ looks good. It has the highest y intercept
- The maximum y intercept would make sure there is no point higher than the segment
- Let us call it an 'upper tangent'
- Similarly $a_1 b_2$ is the 'lower tangent'

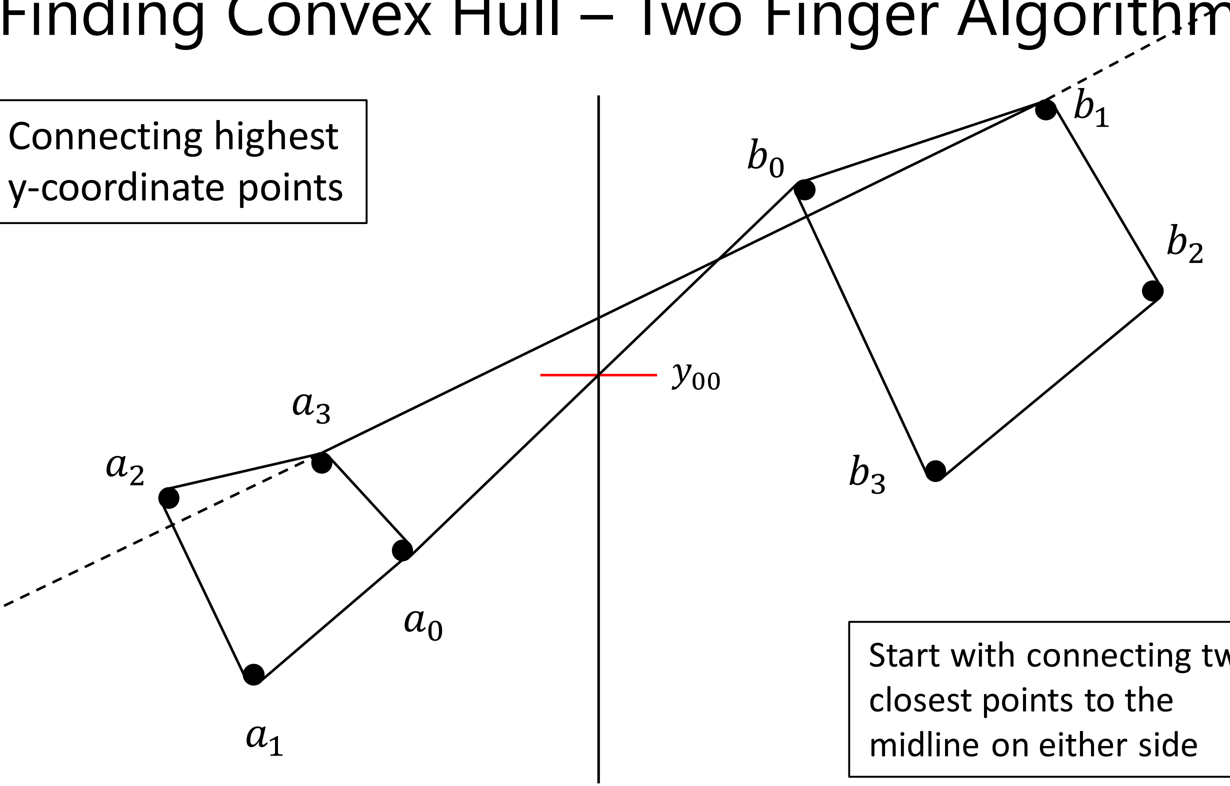


Finding Convex Hull – How to Merge?

- This trivial strategy of choosing one point each from either side is $\Theta(n^2)$
- Can we do better?
- May be – for upper tangent, can I go for connecting highest y coordinate points from either side?
- This will be constant time but not correct always



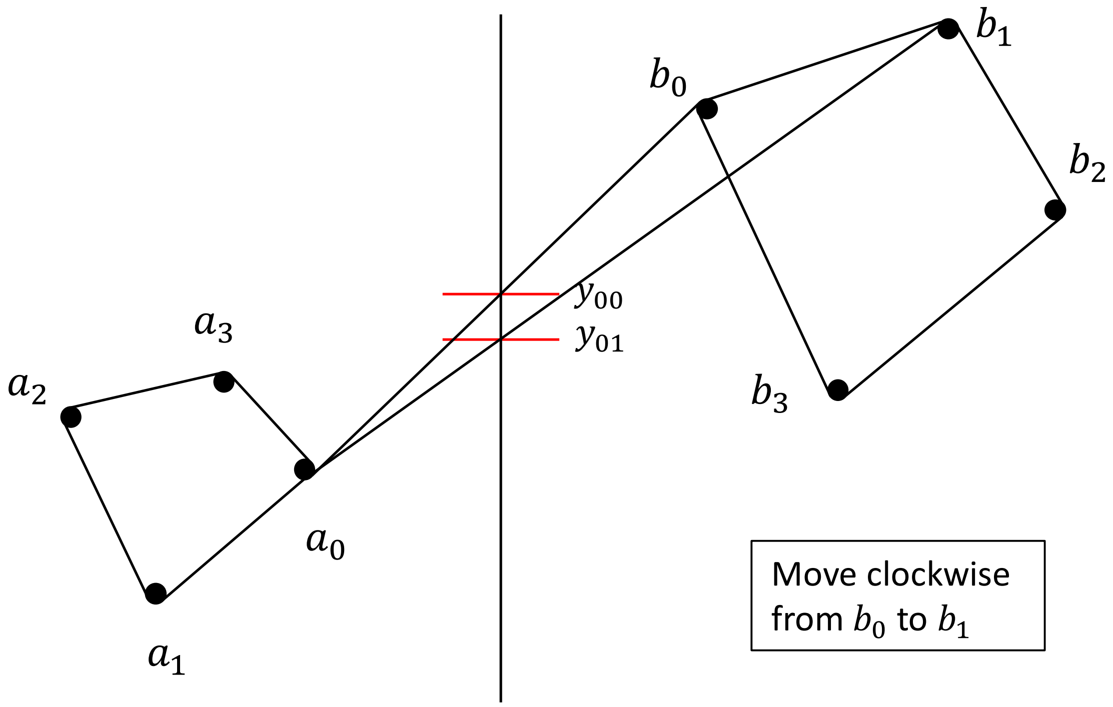
Finding Convex Hull – Two Finger Algorithm



Connecting highest y-coordinate points

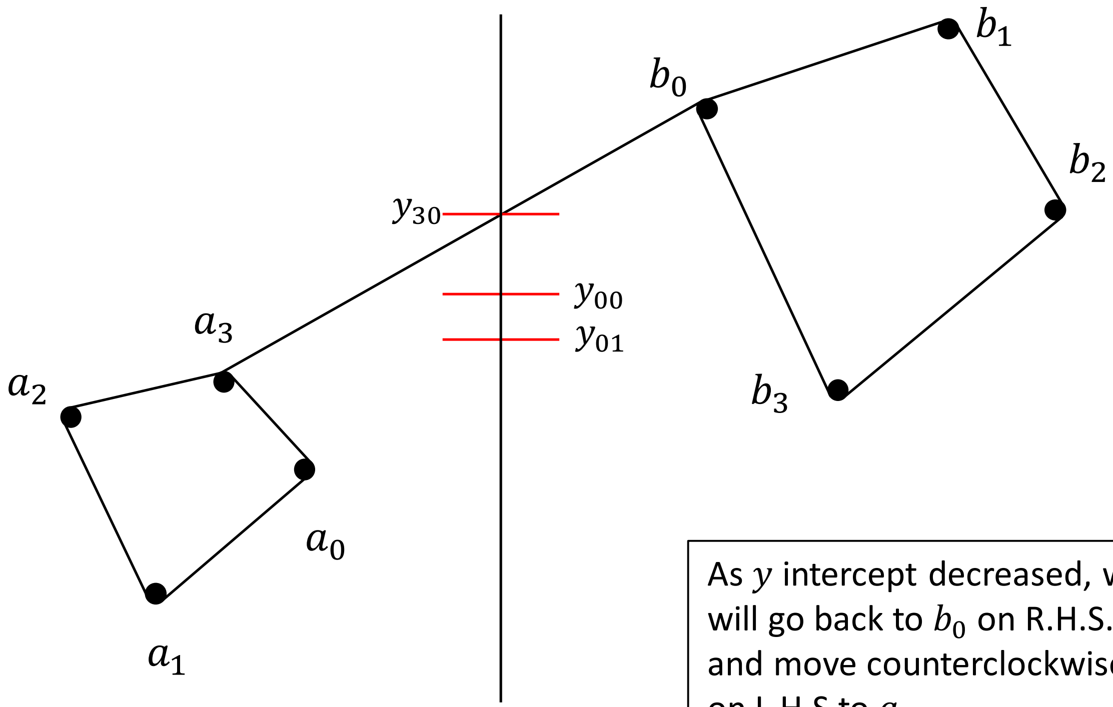
Start with connecting two closest points to the midline on either side

Finding Convex Hull – Two Finger Algorithm



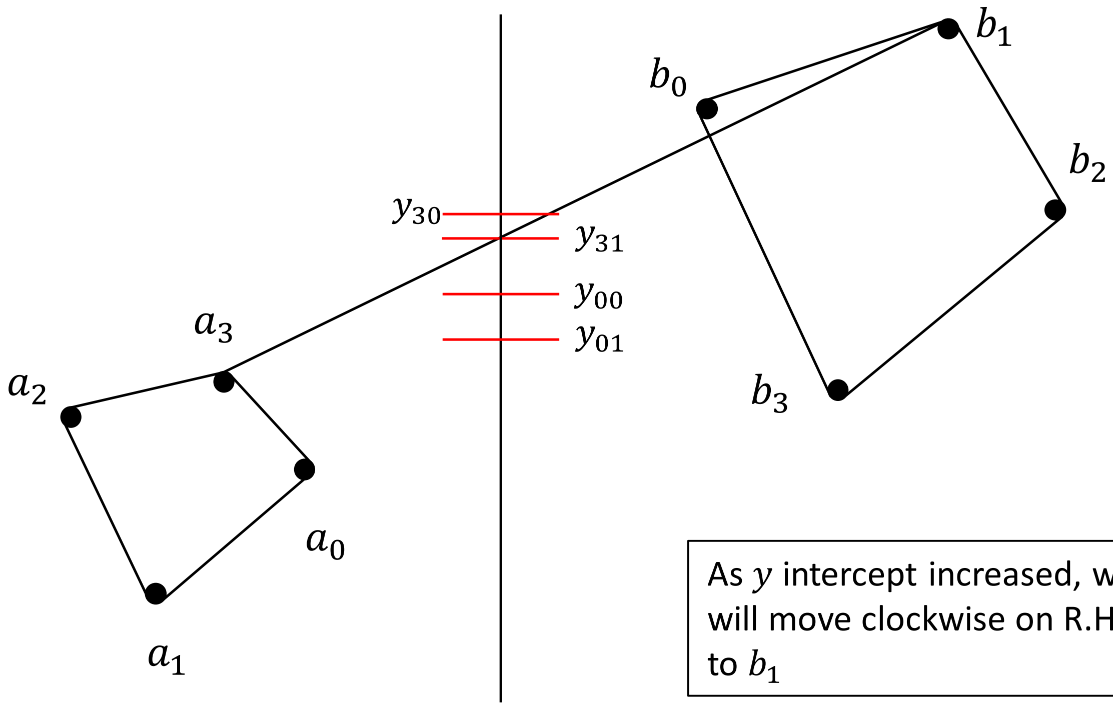
Move clockwise from b_0 to b_1

Finding Convex Hull – Two Finger Algorithm



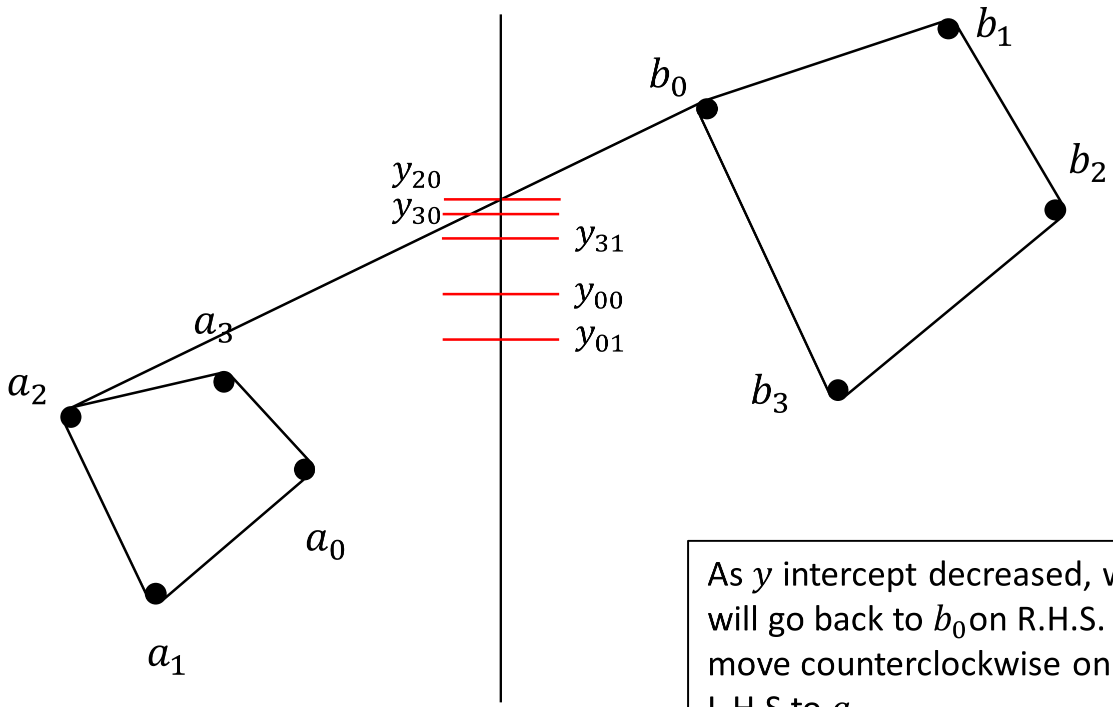
As y intercept decreased, we will go back to b_0 on R.H.S. and move counterclockwise on L.H.S to a_3

Finding Convex Hull – Two Finger Algorithm



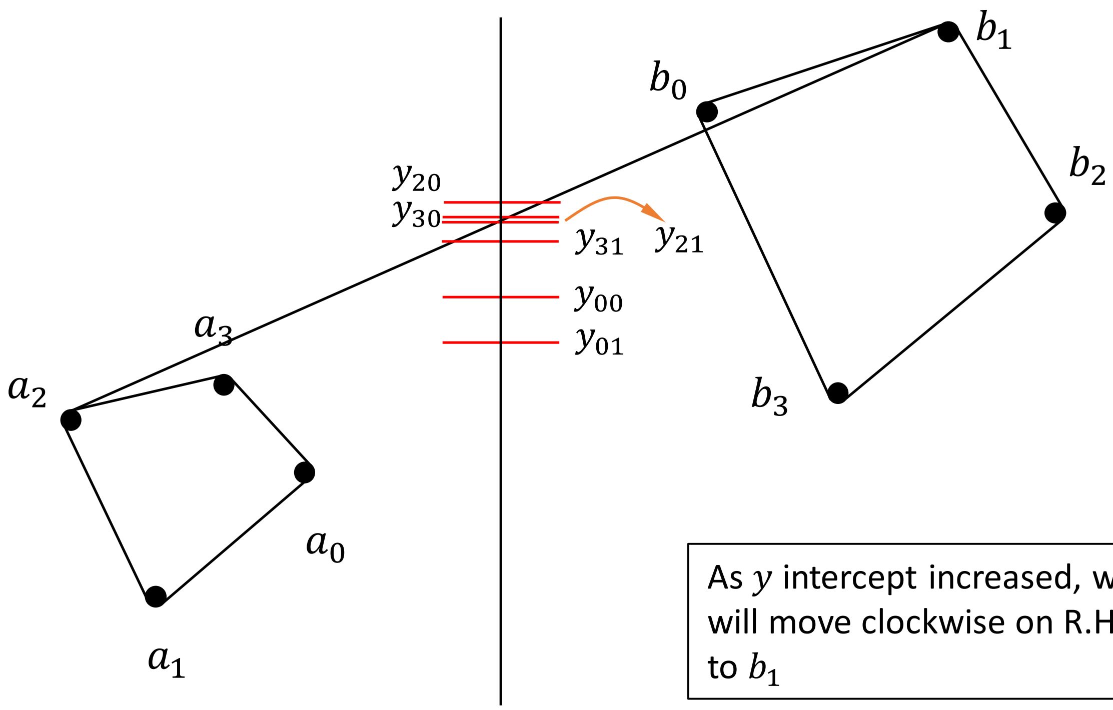
As y intercept increased, we will move clockwise on R.H.S to b_1

Finding Convex Hull – Two Finger Algorithm



As y intercept decreased, we will go back to b_0 on R.H.S. and move counterclockwise on L.H.S to a_2

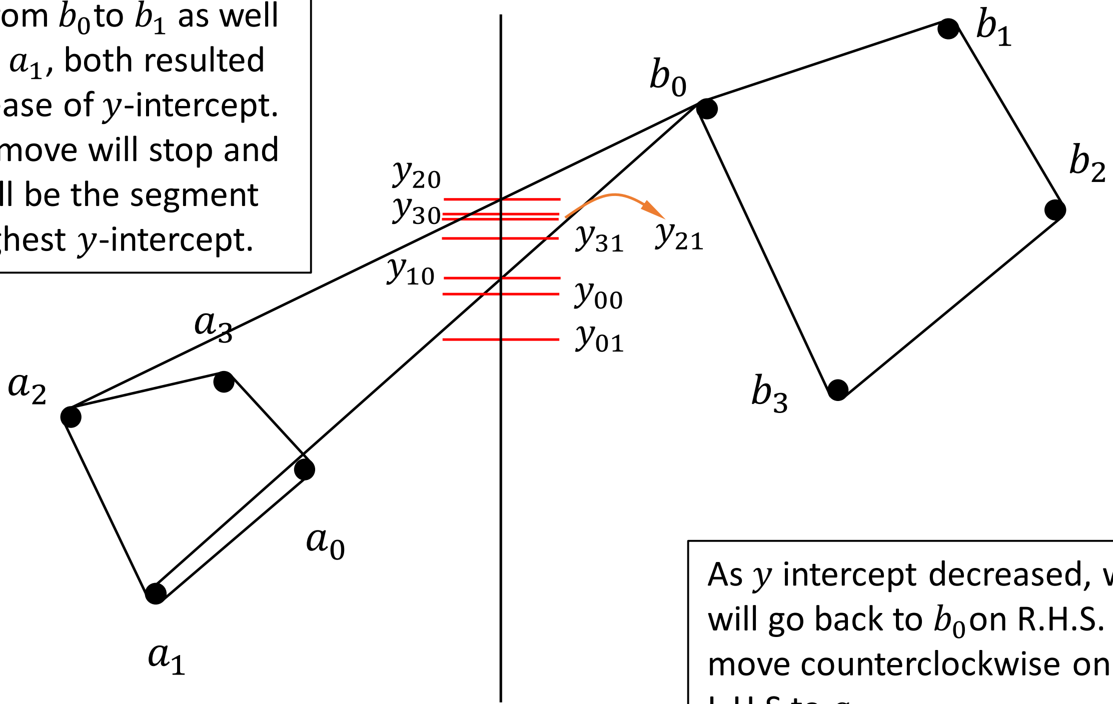
Finding Convex Hull – Two Finger Algorithm



As y intercept increased, we will move clockwise on R.H.S to b_1

Finding Convex Hull – Two Finger Algorithm

Move from $b_0$to b_1 as well as a_2 to a_1, both resulted in decrease of y-intercept. So, the move will stop and a_2b_0 will be the segment with highest y-intercept.



As y intercept decreased, we will go back to $b_0$on R.H.S. and move counterclockwise on L.H.S to a_1

PseudoCode

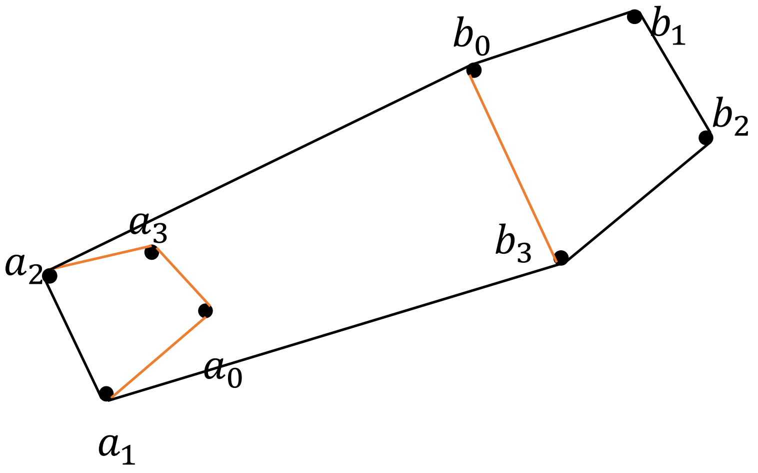
```
// Initialize counters
i=j=0;
$while\ (y_{(i,j+1)} > y_{(i,j)}\ OR\ y_{(i-1,j)} > y_{(i,j)})$
    $if\ (y_{(i,j+1)}) > y_{(i,j)}$ ) // j+1 is actually (j+1)%p
        $j = (j+1)\%p$
    if ($y_{(i-1,j)} > y_{(i,j)}$) // i-1 is actually (i-1)%q
        $i = (i-1)\%q$

Return $(i,j)$
```

p and q are points on the left and right halves respectively

What about the runtime complexity of the two finger algorithm?

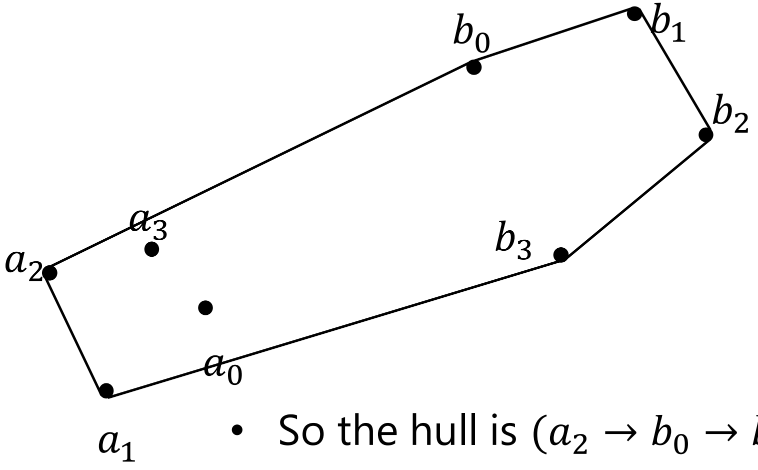
One Final Bit

- How to remove the segments that are part of the sub-hulls but not of the overall hull?
- Lets say we have found the upper and the lower tangents as (a_i, b_j) and (a_k, b_m) which for our example were (a_2, b_0) and (a_1, b_3)
- Our two sub-hulls are a_0, a_1, a_2, a_3 and b_0, b_1, b_2, b_3



One Final Bit

- How to remove the segments that are part of the sub-hulls but not of the overall hull?
- Lets say we have found the upper and the lower tangents as (a_i, b_j) and (a_k, b_m) which for our example were (a_2, b_0) and (a_1, b_3)
- Our two sub-hulls are a_0, a_1, a_2, a_3 and b_0, b_1, b_2, b_3

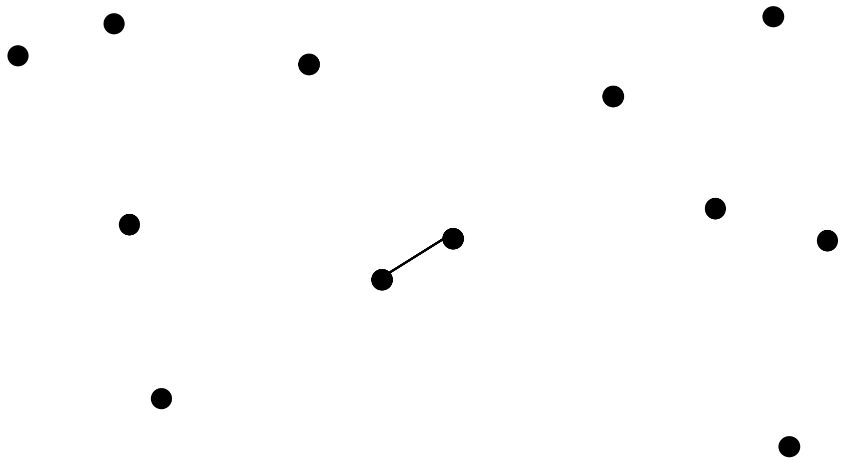


- Link $a_i \rightarrow b_j$ (here $a_2 \rightarrow b_0$)
- Traverse the b list till b_m is met
- Link $b_m$to a_k (here $b_3$to a_1)
- Traverse the a list till a_i is met

- So the hull is $(a_2 \rightarrow b_0 \rightarrow b_1 \rightarrow b_2 \rightarrow b_3 \rightarrow a_0 \rightarrow a_1 \rightarrow a_2)$

Closest Pair of Points

- Problem definition: Given n points $(x_1, y_1), \ldots, (x_n, y_n)$ in the plane, find the pair (x_i, y_i) and (x_j, y_j)whose Euclidean distance is as small as possible
- Naïve approach: Try every pair of points. $\Theta(n^2)$ runtime

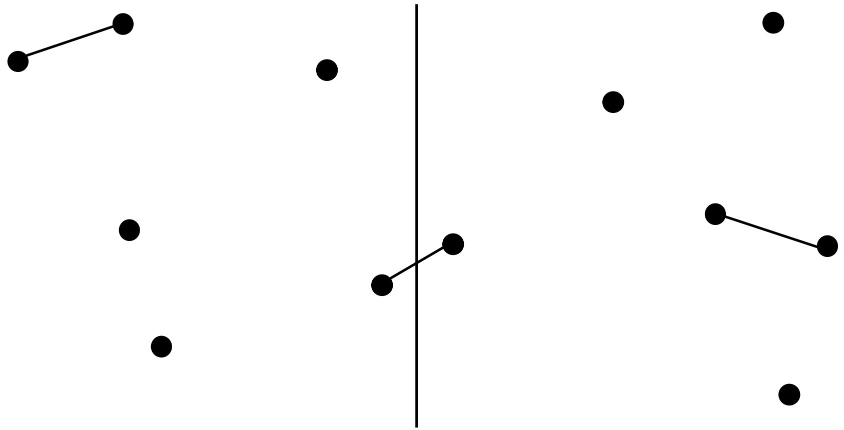


Closest Pair of Points

- Can we avoid comparing every pair of points?
- How about single dimension case?
- In one dimensional case, the structure of the problem setting helps. we only need the distances between adjacent points. So $\Theta(n)$.
- But we need to sort them first according to the x-coordinate values. So its $\Theta(n \log n)$

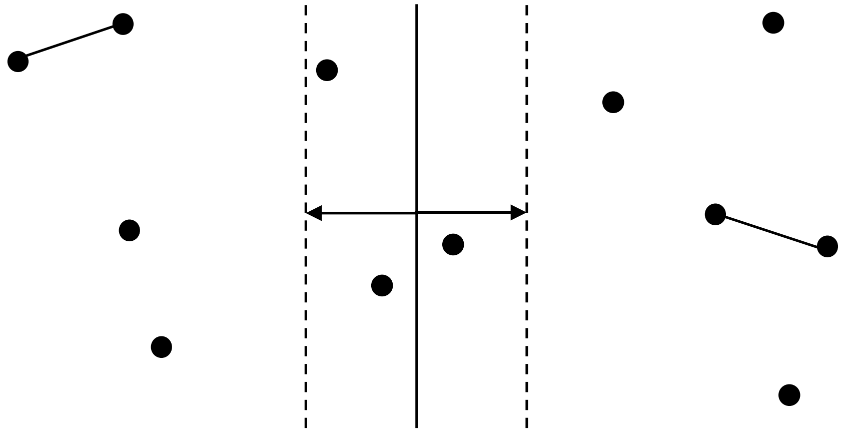
2 Dimensions – Divide and Conquer

- Split the set of points into two halves
  - For 2-D points, a good option to split is using a vertical line that divides the set into two of roughly the equal size
  - For that we would need to sort the points according to x-coordinates
- Recursively compute closest pairs in these halves until base case
- While combining compute closest pair across two halves



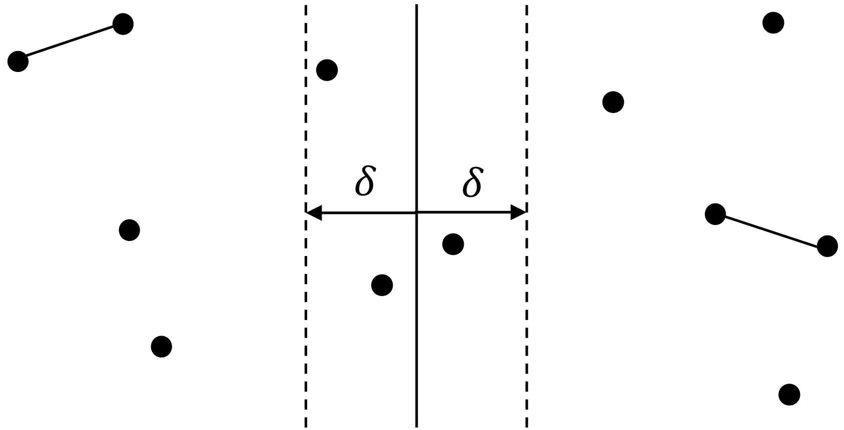
2 Dimensions – Divide and Conquer

- One way to compute closest pair across the midline, is to choose one point from either side and compute the distance
- This will be $\Theta(n^2)$
- But do we really need to consider every pair across the midline?
- We only need to consider checking a narrow band across the midline where the band width is twice the minimum distance found on either halves



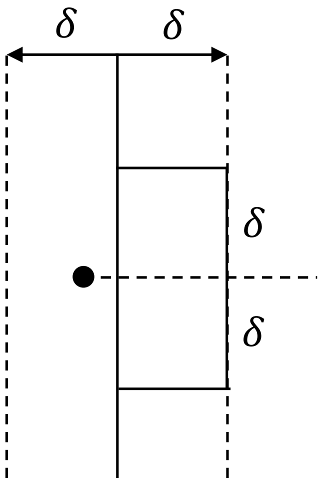
Closest Pair of Points

- Let the closest pair distance on the left and right halves be δ_L and δ_R respectively and $\delta = \min(\delta_L, \delta_R)$
- However, the same issue of all points to be residing inside the band can occur
- Savior: Points on each side separated by at least δ. Not enough room for many of them to be packed nearby
- This means we need to check only a few pairs crossing the line

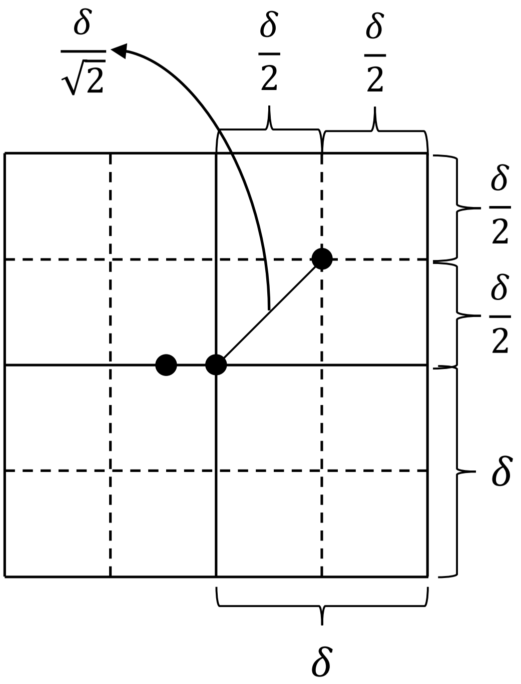


Closest Pair of Points

- Lets consider the point shown in the band
- The candidate points can only lie in the two neighboring boxes of size $\delta \times \delta$ shown to the right
- Now, how many points can be in these two neighboring boxes?



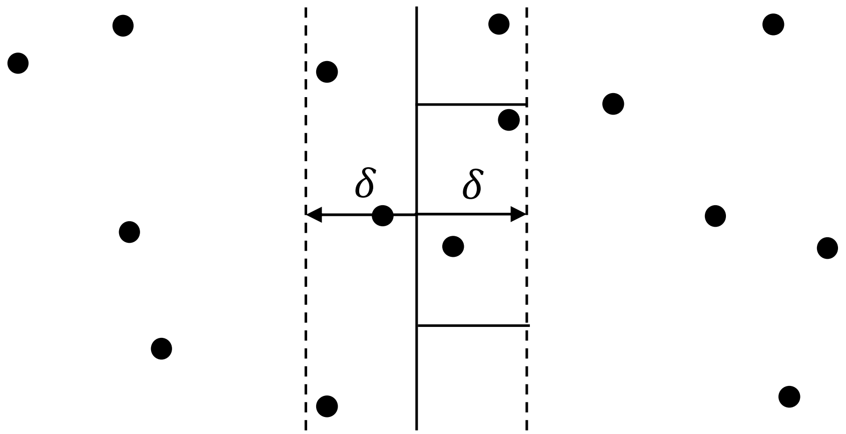
Closest Pair of Points



- Now, how many points can be in these two neighboring boxes?
- Lets subdivide the boxes by half
- Maximum one point can be there in these $\frac{\delta}{2} \times \frac{\delta}{2}$ boxes
- So at most 8 points to check on the right hand side

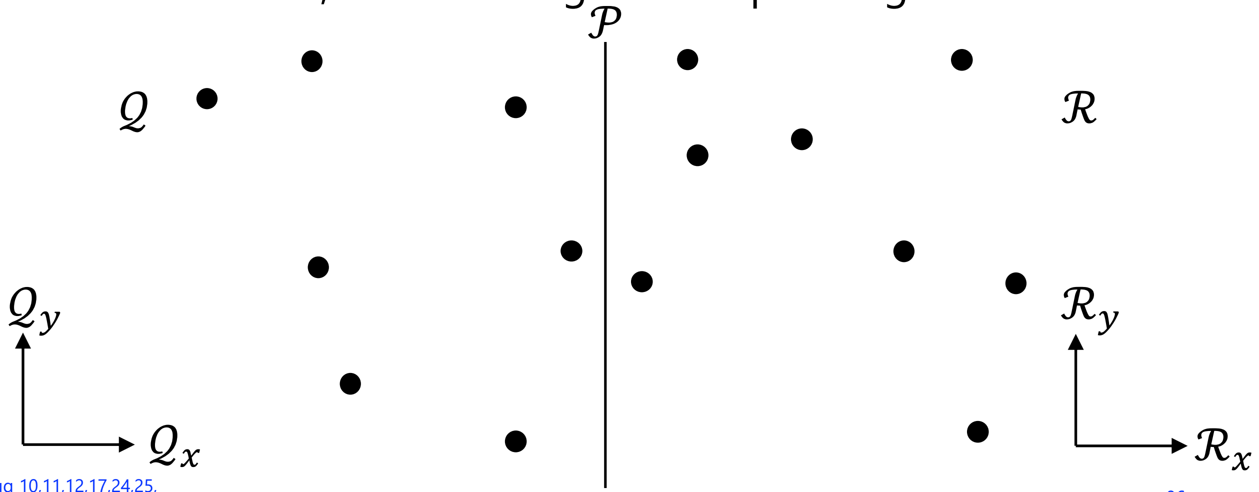
Closest Pair of Points

- So, for every point in the δ band on the left we have to scan points in the neighboring boxes
- This requires the points to be sorted according to y-coordinates also
- Time for comparison in this window is linear as it's a fixed number (8 points).



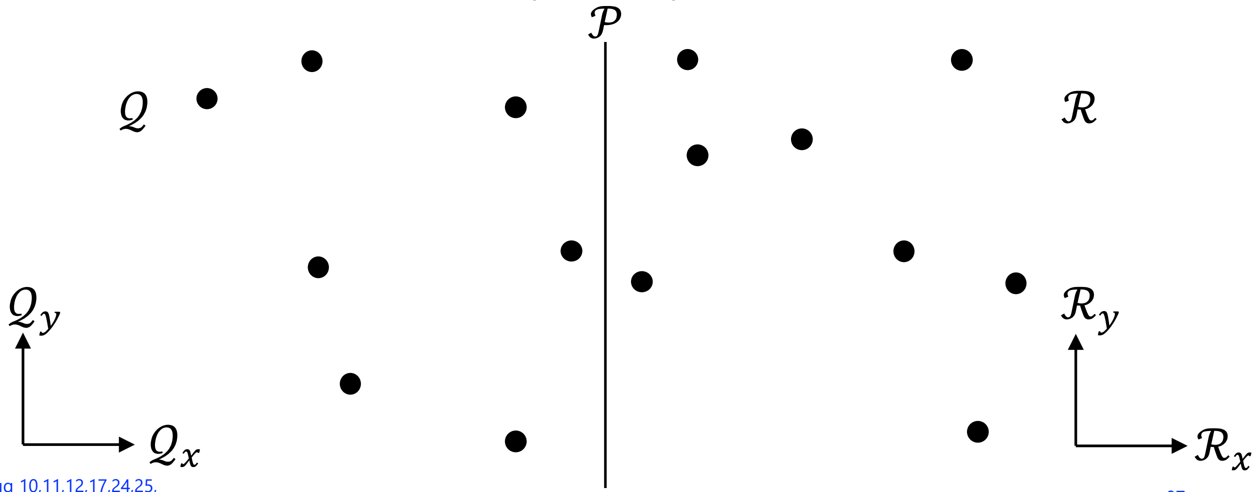
Closest Pair of Points - Algorithm

- Input: Set of 2-D points (denoted by set $\mathcal{P}$)
- Assumption: No two points are same in its x or y coordinates. Not a limiting assumption, just convenient
- Step 0: Sort all points on both x and y coordinates to get two different lists, say $\mathcal{P}_x$ and $\mathcal{P}_y$
- When we divide $\mathcal{P}$, we need to get corresponding lists in each half



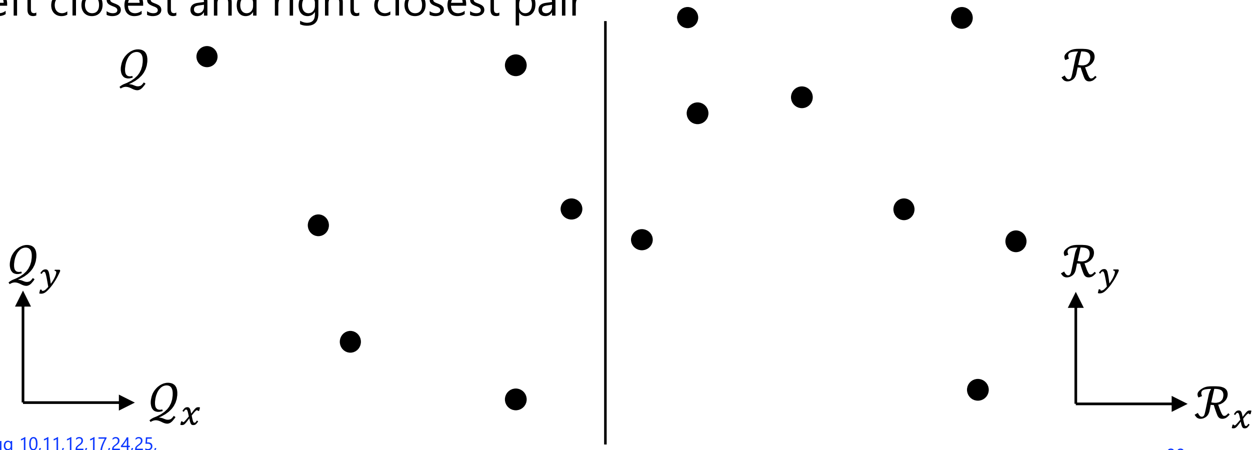
Closest Pair of Points - Algorithm

- Getting $\mathcal{Q}_x$ and $\mathcal{R}_x$ are easy
- To get $\mathcal{Q}_y$ and $\mathcal{R}_y$, we have to scan P_y and points for which x coordinate is less than the midline will be pushed to the list $\mathcal{Q}_y$ and points for which x coordinate is more than the midline will be pushed to the list $\mathcal{R}_y$
- So, in linear time we get $\mathcal{Q}_x, \mathcal{Q}_y, \mathcal{R}_x, \mathcal{R}_y$



Closest Pair of Points - Algorithm

- Step 0: Sort all points on both x and y coordinates to get two different lists, say $\mathcal{P}_x$ and $\mathcal{P}_y$
- Step 1: Call 'ClosestPair($\mathcal{P}_x$, $\mathcal{P}_y$)'
- Step 2: If hits base case (2 Points) return the distance between two
- Step 3: Recursively call 'ClosestPair($\mathcal{Q}_x$, $\mathcal{Q}_y$)' and 'ClosestPair($\mathcal{R}_x$, $\mathcal{R}_y$)'
- Step 4: Get closest 'across-midline' pair and return minimum of this, left closest and right closest pair



Closest Pair of Points - Analysis

- Step 0: Sort all points on both x and y coordinates to get two different lists, say $\mathcal{P}_x$ and $\mathcal{P}_y$ -> $\Theta(n \log n)$
- Step 1: Call 'ClosestPair($\mathcal{P}_x, \mathcal{P}_y$)' -> $T(n)$
- Step 2: If hits base case (2 Points) return the distance between two -> $\Theta(1)$
- Step 3: Recursively call 'ClosestPair($\mathcal{Q}_x, \mathcal{Q}_y$)' and 'ClosestPair($\mathcal{R}_x, \mathcal{R}_y$)' -> $T(\frac{n}{2})$
- Step 4: Get closest 'across-midline' pair and return minimum of this, left closest and right closest pair -> $\Theta(n)$

- So the recurrence relation is

$$T(n) = 2T\left(\frac{n}{2}\right) + cn$$

- So, runtime

$$T(n) = \Theta(n \log n)$$

Thank You!!