Divide and Conquer

RMIT UNIVERSITY

Learning objectives

1. Understand the divide-and-conquer algorithmic approach.
2. Master Theorem.
3. Understand and apply **Merge Sort** and **Quick Sort**.
4. Understand and apply divide-and-conquer to the **Convex hull** problem.

Agenda

1. Overview of the Divide-and-Conquer approach
2. Master Theorem
3. Sorting techniques: Merge Sort & Quick Sort
4. Quick Hull algorithm

1. Overview

Divide and Conquer

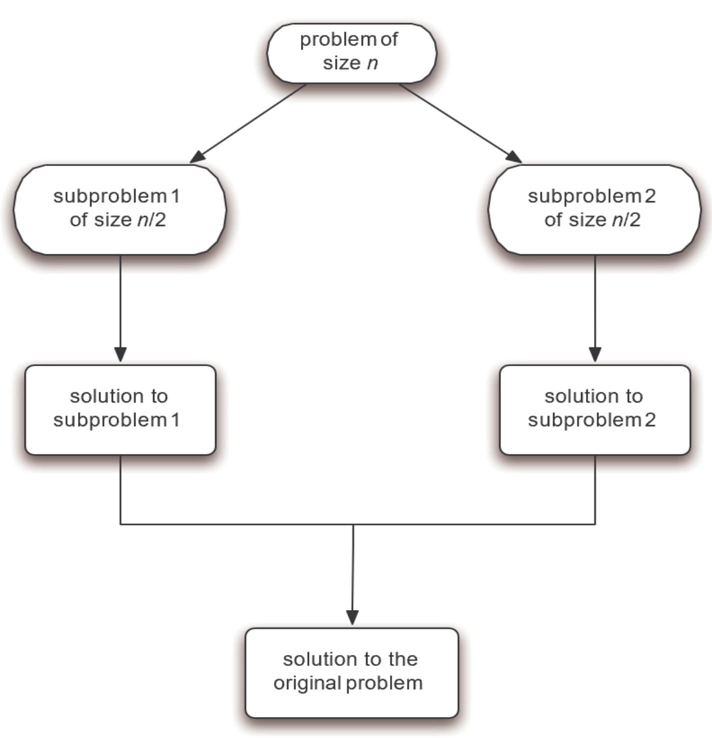
Strategy:

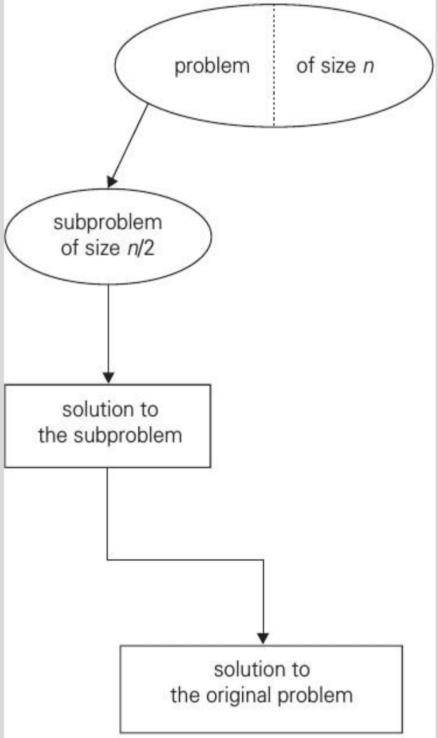
1. Divide the problem instance into smaller subproblems.
2. Solve each subproblem (**recursively**).
3. Combine smaller solutions to solve the original instance.

Pseudocode

```
solve(problem p of size n)
    if n is small enough
        solve p directly
    else
        create a subproblems, each with size n/b
        solve each subproblem recursively
        combine the results of all subproblems
```

Compare with Decrease-by-a-constant-factor





2. Master Theorem

Master Theorem

- A tool to determine an asymptotic complexity for recurrence relations
- Recurrence relation: a sequence in which the n-th term is calculated by the previous terms
  - $T(n) = T(n-1) + 1$
  - $T(n) = 2T(n/2) + n$
- Not all recurrence relations can apply the Master theorem

General Form

- Solve a problem of size n by:
  - Divide it into **a** subproblems of size **n/b**
  - Combine the results of subproblems **f(n)**
- `T(n) = aT(n/b) + f(n)`
- Assumption: T(n) = O(1) when n is small enough (that is, when the problem can be solved directly without recursive calls)

Cases

- $T(n) = aT(n/b) + f(n)$
- First, calculate: $c = \log_b(a)$
- There are three cases
  - $f(n) = O(n^p)$ where $p < c$
    - Then, $T(n) = O(n^c)$
  - $f(n) = O(n^c * \log^k n)$ $k >= 0$
    - Then, $T(n) = O(n^c \log^{k+1} n)$
  - $f(n) = O(n^p)$ where $p > c$ **AND** $a*f(n/b) <= k*f(n)$ for some $k < 1$
    - Then, $T(n) = O(f(n))$

Example 1

- Binary Search
- $T(n) = T(n/2) + 1$
- $a = 1$, $b = 2$, $f(n) = 1$
- $c = \log_2(1) = 0$
- $f(n) = 1 = n^0 = O(n^0 * \log^0 n) = O(n^c * \log^0 n)$ => this is case 2, $k = 0$
- $T(n) = O(n^c * \log^{k+1} n) = O(\log(n))$

Example 2

- Calculate binary tree's height
- $T(n) = 2*T(n/2) + 1$
- $a = 2$, $b = 2$, $f(n) = 1$
- $c = \log_2(2) = 1$
- $f(n) = 1 = n^0 = O(n^0)$ and $0 < 1 = c$, => this is case 1
- $T(n) = O(n^c) = O(n)$

Example 3

- Merge sort
- $T(n) = 2*T(n/2) + n$
- $a = 2$, $b = 2$, $f(n) = n$
- $c = \log_2(2) = 1$
- $f(n) = n = O(n*\log^0 n) = O(n^c * \log^0 n)$ => this is case 2, $k = 0$
- $T(n) = O(n^c*\log^{k+1} n) = O(n * \log(n))$

Example 4

- $T(n) = 3*T(n/2) + n^2$
- $a = 3$, $b = 2$, $f(n) = n^2$
- $c = \log_2(3) = 1.58$
- $f(n) = n^2 = O(n^2) \Rightarrow p = 2$ (here: $p > c$)
- AND we have
- $a*f(n/b) = 3*(n/2)^2 = 3*n^2/4 \le (3/4)*n^2$ (here: $k = 3/4 < 1$)
- This is case 3, so
- $T(n) = O(f(n)) = O(n^2)$

3a. Merge Sort

Merge Sort

Idea:

- We recursively divide an array (we want to sort) **into halves**, until we reach **single element partitions**.
- We then **recursively merge the partitions**, where we have a process that maintains sorting after partitions are merged.
- When we finally merge the last two partitions, we have a sorted array.

Merge Sort Example

15	21	1	25	12	6	8	3	5	19	10	18

Compares 0

Merge Sort Example

15	21	1	25	12	6

8	3	5	19	10	18

Compares

0

Merge Sort Example

15	21	1	25	12	6	8	3	5	19	10	18

Compares $\boxed{0}$

Merge Sort Example

15	21	1	25	12	6	8	3	5	19	10	18

Compares 0

Merge Sort Example

15	21	1	25	12	6	8	3	5	19	10	18

Compares $\boxed{0}$

Merge Sort Example

15	1	21	25	12	6	8	3	5	19	10	18

Compares

1

Merge Sort Example

15	1	21	25	6	12	8	3	5	19	10	18

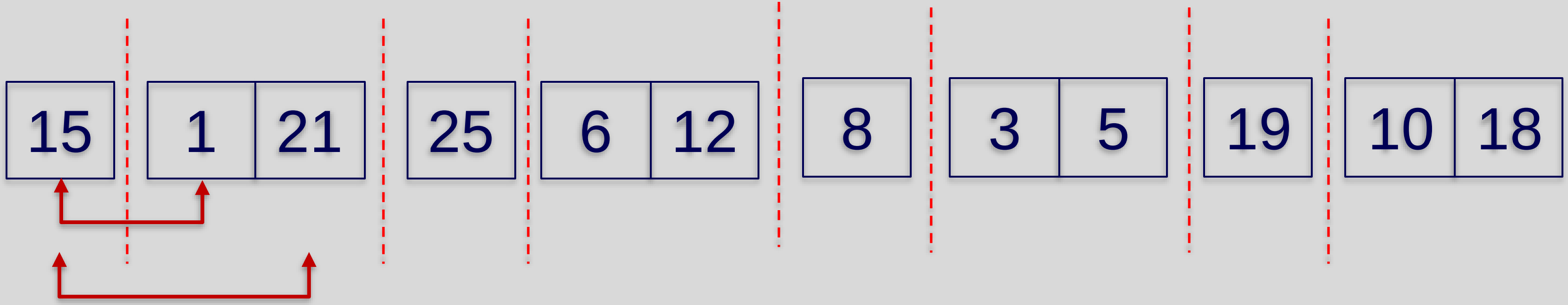
Compares 2

Merge Sort Example

15	1	21	25	6	12	8	3	5	19	10	18

Compares 3

Merge Sort Example



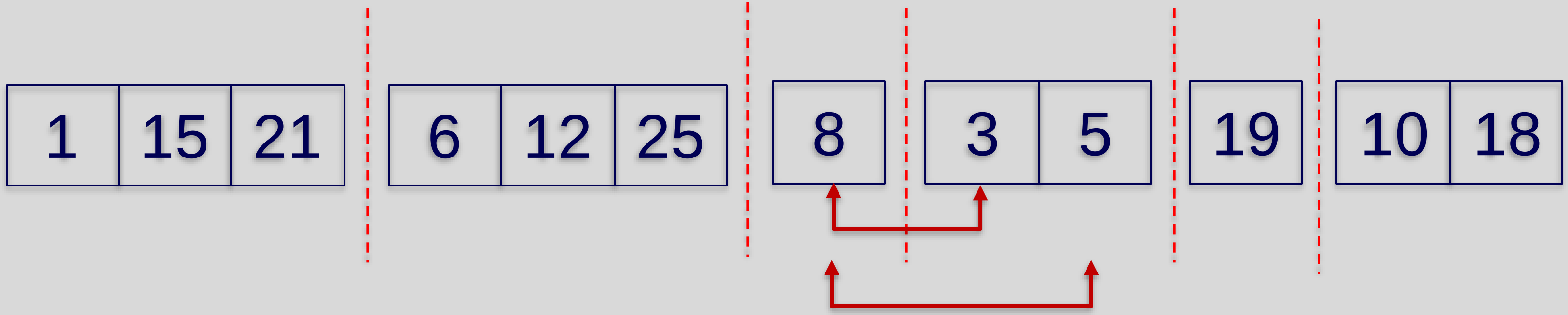
Compares $\boxed{4}$

Merge Sort Example

1	15	21		25		6	12		8		3	5		19		10	18

Compares $\boxed{6}$

Merge Sort Example



Compares 8

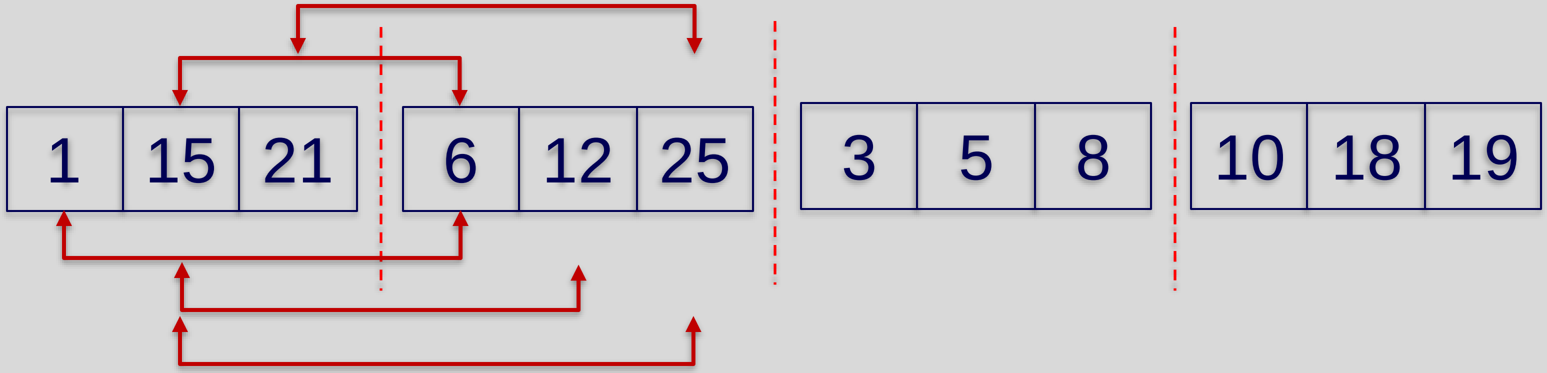
Merge Sort Example

1	15	21	6	12	25	3	5	8	19	10	18

Compares

10

Merge Sort Example



Compares

Merge Sort Example

1	6	12	15	21	25		3	5	8		10	18	19

Compares 17

Merge Sort Example

1	6	12	15	21	25	3	5	8	10	18	19

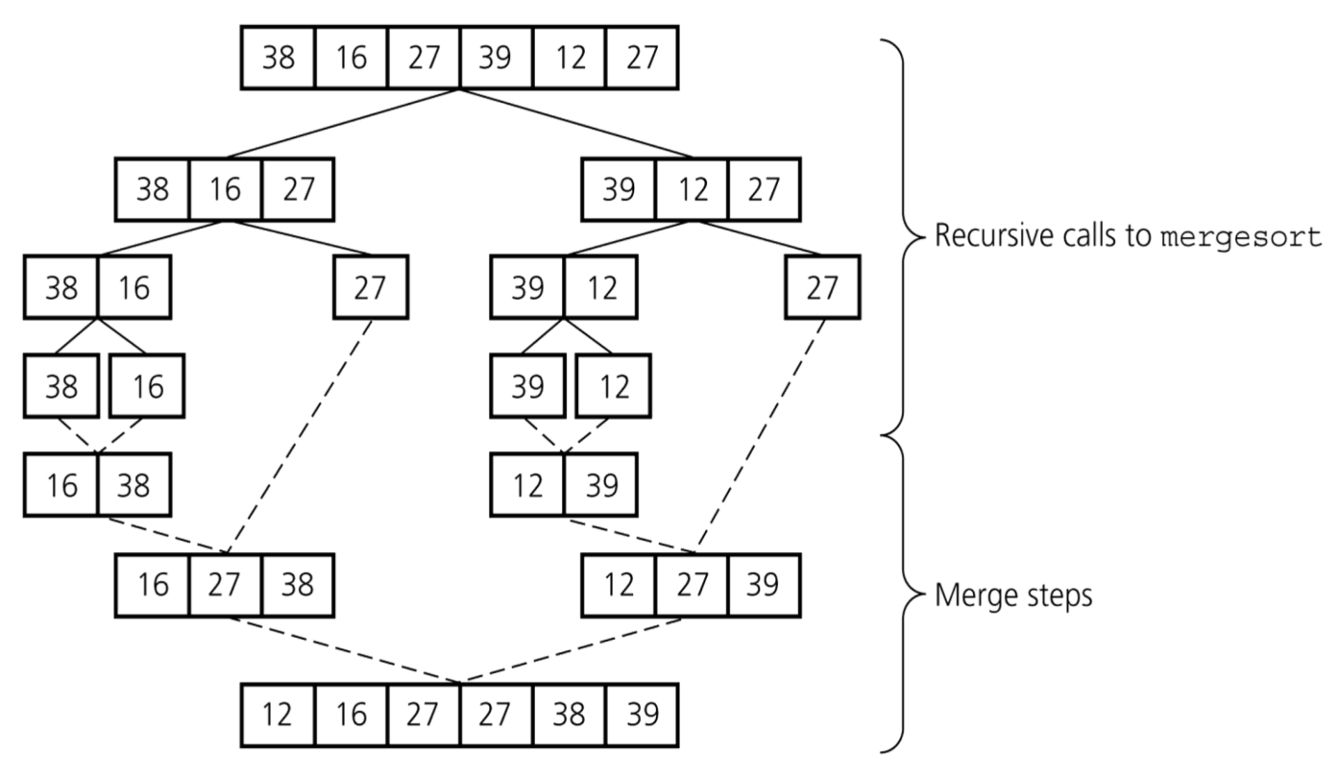
Compares 20

Merge Sort Example

1	3	5	6	8	10	12	15	18	19	21	25

Compares 30

Merge Sort – Example 2



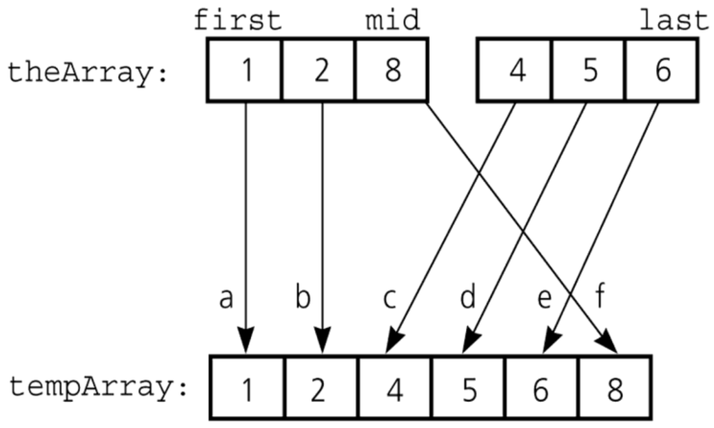
Merge Sort Algorithm

ALGORITHM **MergeSort** $(A[0 \ldots n-1])$
/* Sort an array using a divide-and-conquer merge sort. */
/* INPUT : An array $A[0 \ldots n-1]$ of orderable elements. */
/* OUTPUT : An array $A[0 \ldots n-1]$ sorted in ascending order. */

if $n > 1$ **then**
    $B = A[0 \ldots \lfloor n/2 \rfloor - 1]$ /* *B is first half of A* */
    $C = A[\lfloor n/2 \rfloor \ldots n-1]$ /* *C is second half of A* */
    MergeSort (B)
    MergeSort (C)
    Merge (B, C, A) /* *Merge B and C to help sort A* */
end if

Merge Sort – Analysis of Merge

A worst-case instance of the merge step in `merge sort`

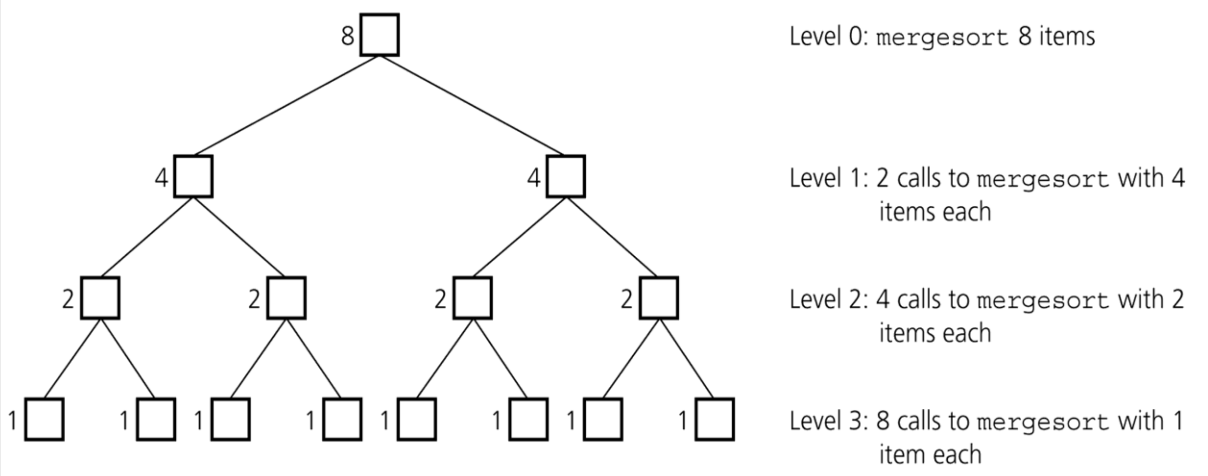


Merge the halves:

a. 1 < 4, so move 1 from `theArray[first..mid]` to `tempArray`
b. 2 < 4, so move 2 from `theArray[first..mid]` to `tempArray`
c. 8 > 4, so move 4 from `theArray[mid+1..last]` to `tempArray`
d. 8 > 5, so move 5 from `theArray[mid+1..last]` to `tempArray`
e. 8 > 6, so move 6 from `theArray[mid+1..last]` to `tempArray`
f. `theArray[mid+1..last]` is finished, so move 8 to `tempArray`

Merge Sort - Analysis

Levels of recursive calls to *merge sort*, given an array of eight items



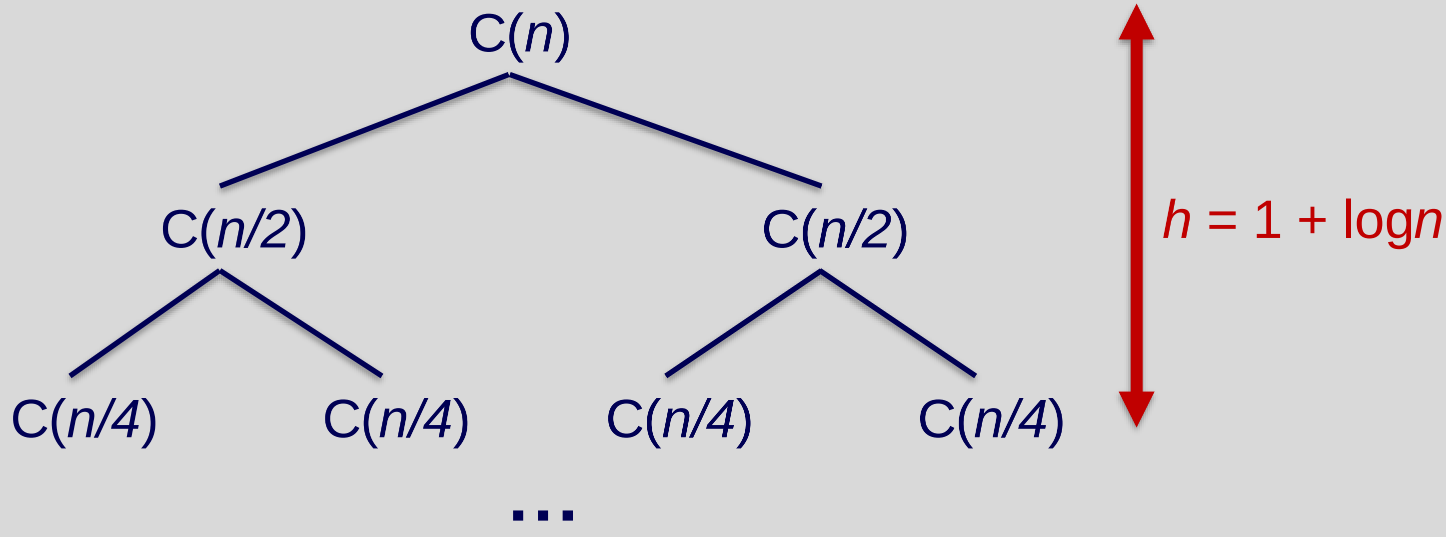
Merge Sort - Analysis

$n = n$

$2(n/2) = n$

$4(n/4) = n$

…



$\Theta(n)$

Merge Sort – Analysis

- Merge sort is extremely efficient algorithm with respect to time
    - Both worst case and average cases are O $(n * \log_2 n)$
- But, merge sort requires an extra array whose size equals to the size of the original array
- If we use a linked list, we do not need an extra array
    - But, we need space for the links
    - And, it will be difficult to divide the list into half (O(n))

Merge() in Merge Sort

Given two sorted subarrays B and C, we want to merge them together to form a sorted array A.

1. Consider first element of each subarray, i.e., B[0] and C[0].
2. Compare them. Copy the smaller one to A[0], and increment current pointer of subarrays that has smaller element and A.
3. Repeat until one of subarrays is empty. Then copy the rest of the other subarray to A.

Comments on Merge Sort

- Guarantees **O(*n**logn*)** time complexity, regardless of the original distribution of data – this sorting method is **insensitive** to the data input.

- The main drawback in this method is the **extra space** required for merging two partitions/sub-arrays, e.g., B and C from pseudo-code.

- Merge sort is a **stable** sorting method.

3b. Quick Sort

Quick Sort

Motivation:

- Merge sort has consistent behaviour for all inputs – what if we seek an algorithm that is fast for the average case?
- Quick sort is such a sorting algorithm, often the best practical choice in terms of efficiency because of its good performance on the average case.
- Quick sort is a divide and conquer algorithm.

Quick Sort – Idea

1. Select an element from the array for which, ***we hope***, about half the elements will come before and half after in a sorted array. Call this element the **pivot**.

2. **Partition the array** so that all elements with a value less than the pivot are in one subarray, and larger elements come in the other subarray.

3. **Swap pivot** into position of array that is between the partitions.

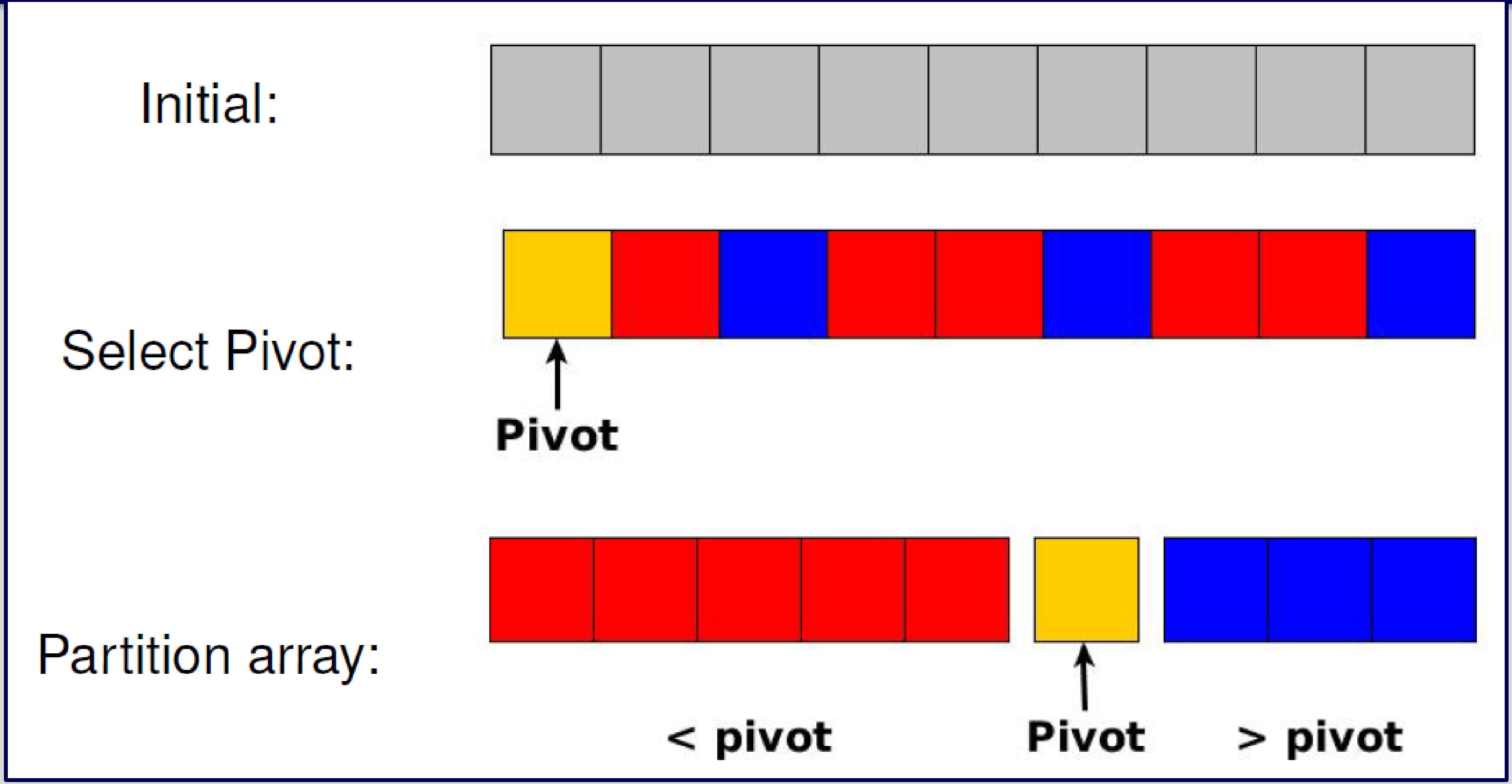
Quick Sort – Idea

4. Recursively apply the same procedure on the two subarrays separately.

5. Terminate when only subarrays are of one element.

6. When terminate, because we do things in-place, the resulting array is sorted.

Quick Sort – Idea



Lomuto Partition Scheme

- The pivot element is the last (right) element
- Initialize two pointers i and j
  - i is used to decide the position of the next element that is <= pivot, j is used to loop through the array
- i = left
- Let j go through the array (i.e., from left to right – 1)
  - If arr[j] <= pivot
    - swap arr[i] with arr[j]
    - i++
- swap arr[i] with arr[right], i stores the index of pivot element

Lomuto Partition Scheme

```
partition(arr[], left, right)
    pivot = arr[right]
    i = left
    for j = left to (right - 1)
        if arr[j] <= pivot then
            swap arr[i] with arr[j]
            i++
    swap arr[i] with arr[right]
    return i
```

Quick Sort/Lomuto Partition

ALGORITHM **QuickSort** $(A[\ell \ldots r])$
/* Sort a subarray using by quicksort. */
/* INPUT : A subarray $A[\ell \ldots r]$ of $A[0 \ldots n-1]$, defined by its left and right indices ℓ and r. */
/* OUTPUT : A subarray $A[\ell \ldots r]$ sorted in ascending order. */

1: **if** $\ell < r$ **then**
2:     /* s is the index to split array. */
3:     $s =$ **QPartition**$(A[\ell \ldots r])$
4:     **QuickSort**$(A[\ell \ldots s-1])$
5:     **QuickSort**$(A[s+1 \ldots r])$
6: **end if**

Hoare Partition Scheme

- The pivot element can be any element
- Initialize two pointers i and j
  - i go from left to right, stop when the element at i is >= pivot
  - j go from right to left, stop when the element at j is <= pivot
  - Swap the two elements pointed to by i and j
  - Continue until i >= j, then return j
- In this partition scheme, all elements from left to j are <= all elements from (j+1) to right. But the element at j is not necessary at its correct position

Hoare Partition Scheme

```
partition(arr[], left, right)
    pivot = arr[left], i = left, j = right
    while (true)
        while arr[i] < pivot
            i++
        while arr[j] > pivot
            j--
        if j <= i then
            return j
        swap arr[i] with arr[j]
        i++ and j--
```

Quick Sort/Hoare Partition

ALGORITHM **QuickSort** $(A[\ell \ldots r])$
/* Sort a subarray using by quicksort. */
/* INPUT : A subarray $A[\ell \ldots r]$ of $A[0 \ldots n-1]$, defined by its left and right indices ℓ and r. */
/* OUTPUT : A subarray $A[\ell \ldots r]$ sorted in ascending order. */

if $\ell < r$ **then**
    /* s is the index to split array. */
    $s =$ **QPartition**$(A[\ell \ldots r])$
    ~~**QuickSort**$(A[\ell \ldots s-1])$~~ **QuickSort**(A[l...s])
    ~~**QuickSort**$(A[s+1 \ldots r])$~~ **QuickSort**(A[s+1...r])
end if

Quick Sort Example

15	21	1	25	12	6	8	3	5	19	10	18

pivot = 15

Quick Sort Example

15	21	1	25	12	6	8	3	5	19	10	18
i											j

pivot = 15

Quick Sort Example

15	21	1	25	12	6	8	3	5	19	10	18

i j

pivot = 15

Quick Sort Example

10	21	1	25	12	6	8	3	5	19	15	18
	i								j		

pivot = 15

swap 15 <-> 10, increase i, decrease j

Quick Sort Example

10	21	1	25	12	6	8	3	5	19	15	18
	i								j		

pivot = 15

Quick Sort Example

10	21	1	25	12	6	8	3	5	19	15	18
	i							j			

pivot = 15

Quick Sort Example

10	5	1	25	12	6	8	3	21	19	15	18

i j

pivot = 15

swap 21 <-> 5, increase i, decrease j

Quick Sort Example

10	5	1	25	12	6	8	3	21	19	15	18
			i				j				

pivot = 15

Quick Sort Example

10	5	1	25	12	6	8	3	21	19	15	18
			i				j				

pivot = 15

Quick Sort Example

10	5	1	3	12	6	8	25	21	19	15	18
				i		j					

pivot = 15

swap 25 <-> 3, increase i, decrease j

Quick Sort Example

10	5	1	3	12	6	8	25	21	19	15	18
						j	i				

pivot = 15

Quick Sort Example

10	5	1	3	12	6	8	25	21	19	15	18
						j	i				

pivot = 15

Quick Sort Example

10	5	1	3	12	6	8	25	21	19	15	18
						j	i				

pivot = 15, j <= i, return j

Note: [10, 5, 1, 3, 12, 6, 8] <= [25, 21, 19, 15, 18], but 15 (pivot) is not positioned at its final location

Quick Sort Complexity

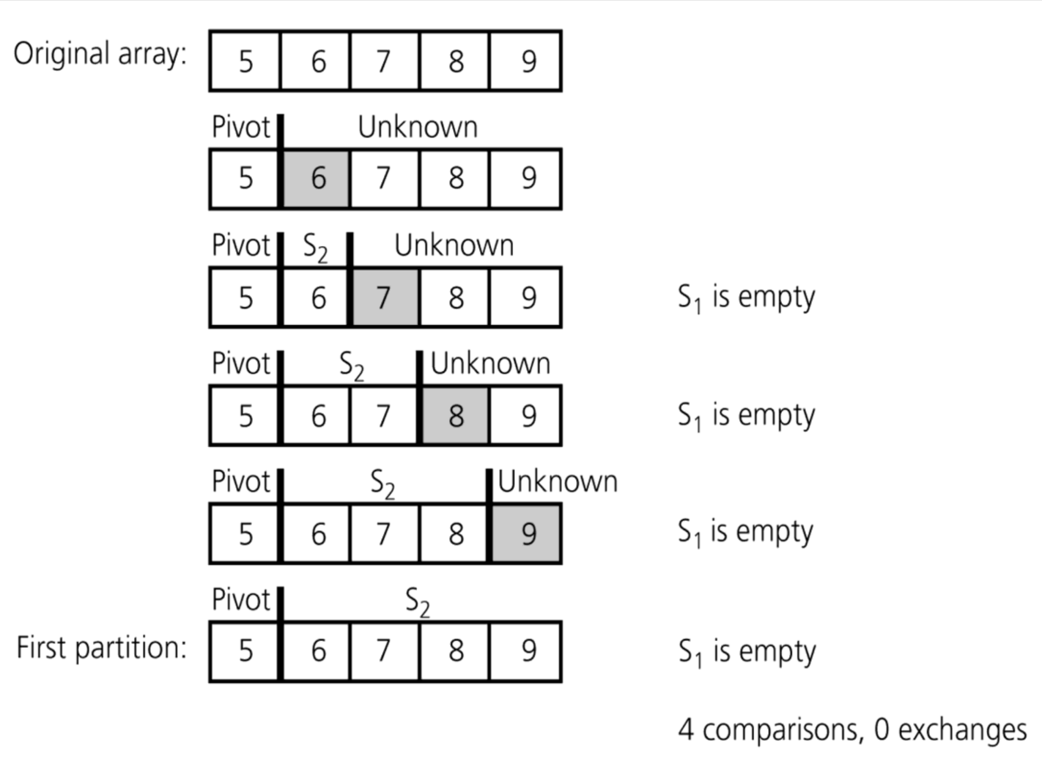
- **Best Case:**
  - Occurs when the pivot repeatedly splits the dataset into **two equal sized** subsets
  - The complexity is $O(n\,log_2 n)$

Quick Sort Complexity

- **Worst Case:**
  - If the **pivot is chosen poorly**, one of the partitions may be empty, and the other reduced by only one element.
  - Then the quick sort is **slower than brute-force sorting** (due to partitioning overheads).
  - The complexity is $n + (n - 1) + (n - 2) + ... + 1 \approx n^2/2 \in O(n^2)$.
  - Occurs when array is **already sorted** or **reverse order sorted**

Quicksort – Analysis

A worst-case partitioning with `quick sort`



Quick Sort Complexity

- **Average case:**
  - Number of comparisons $C(n) \approx 1.39*n\log n$
  - That means 39% more comparisons than merge sort
  - But **faster than merge sort in practice** because of lower cost of other high-frequency operations,
  - And uses considerably less space (no need to copy to temporary arrays).

Partition – Choosing the pivot

Which array item should be selected as pivot?

- Somehow we have to select a pivot, and we hope that we will get a good partitioning
- If the items in the array arranged randomly, we choose a pivot randomly
- We can choose the first or last element as a pivot (it may not give a good partitioning)
- We can use different techniques to select the pivot – for example the median.
  - Does this change the order of complexity?
  - What would be better, the median or the average?
  - What is the complexity of calculating the mean/median

Quick Sort – Pivots

Choosing a pivot:

- **First or last element:** worst case appears for already sorted or reverse sorted arrays (as we saw last slide).
- **Median of three:** requires extra compares but generally avoids worst case.
- **Random element:** Poor cases are very unlikely, but efficient implementations can be non-trivial.

- As long as selected pivot is not always the worst case, Quick sort **on average performs well**.

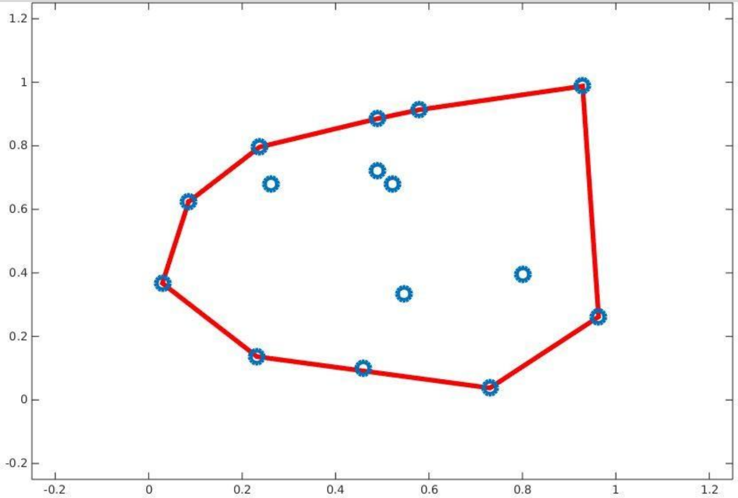
Quicksort – Analysis

- Quick Sort is $O(n*\log_2 n)$ in the best case and average case
- Quick Sort is $O(n^2)$ in the worst case, for example when the array is sorted and we choose the first element as the pivot
- Although the worst case behavior is not so good, its average case behavior is much better than its worst case
    - So, Quick sort is one of best sorting algorithms using key comparisons
- Quick Sort is not a stable sorting method.

4. Convex Hull (again)

Convex Hull problem

- The convex hull of a set of points is the smallest convex polygon that contains all the points, i.e., all the points are “within” the polygon.



Quick Hull algorithm

- **Recall:** Brute force convex hull algorithm = compute lines between all pair of points then do comparison to see if points all fall on one side.
- Can we use divide and conquer principles to design a faster algorithm? Yes of course!

Quick Hull – Idea

- Reduce the number of points that we have to consider for the boundary of the convex hull.
- Use divide and conquer to (quickly) partition the set of points into possible and not possible boundary points.

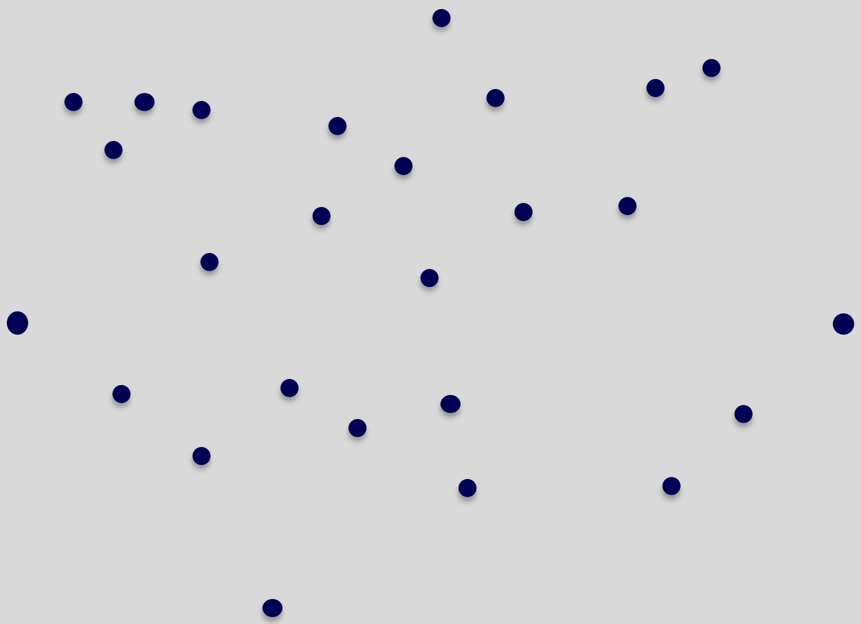
Quick Hull – Idea

1. Sort all points in increasing order of x and then y.
2. Choose the leftmost and rightmost point. Call these points a and b.
3. Separate the remaining points into two sets S and T . All points above line *ab* are in S and all below are in T.
4. Find the point c in S which is farthest from line *ab*.
5. Discard all points inside the triangle *abc*.

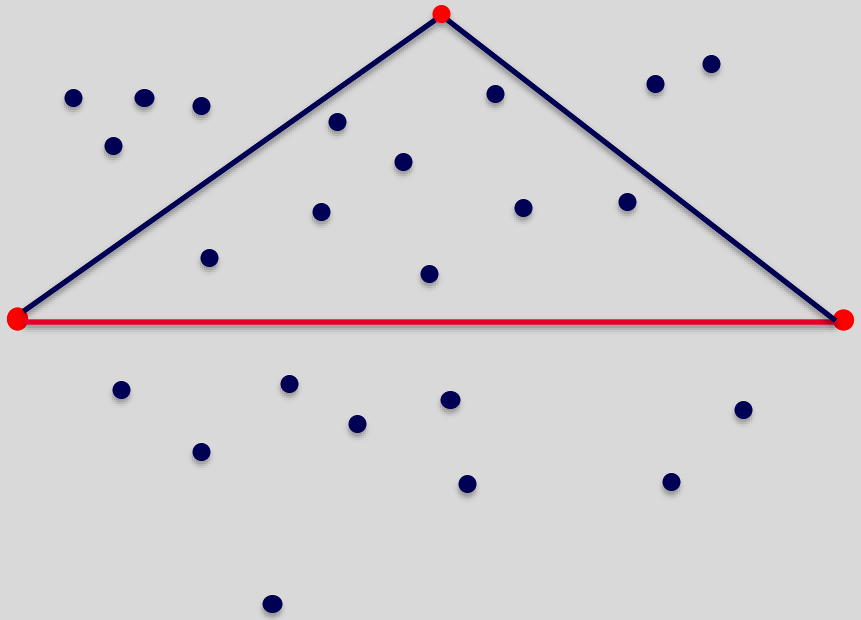
Quick Hull – Idea

6. Put all points outside of *ac* in set Sj.
7. Put all points outside of *bc* in set Tj.
8. Run recursively on *ac* and *bc*.
9. Abort when the subset contains only the two endpoints of the current line.

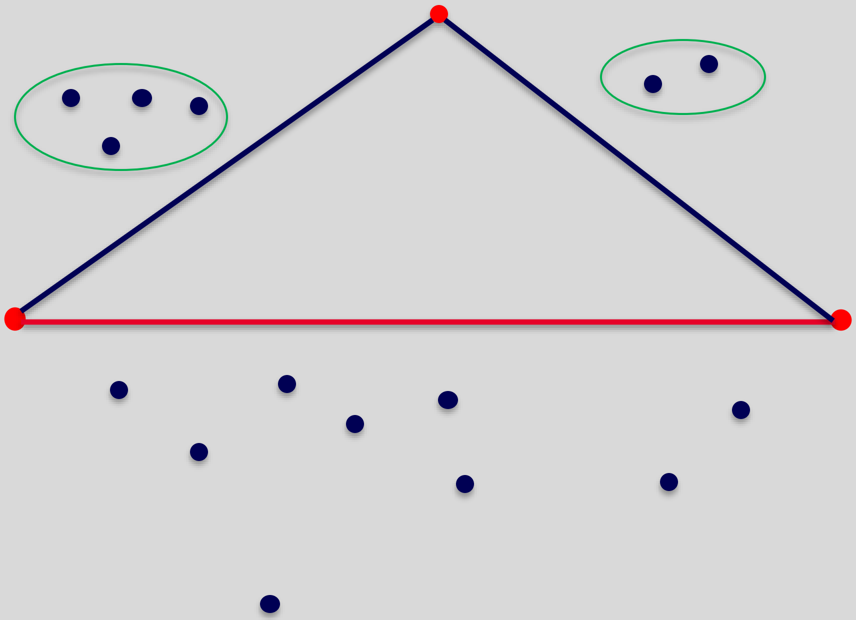
Quick Hull – Example



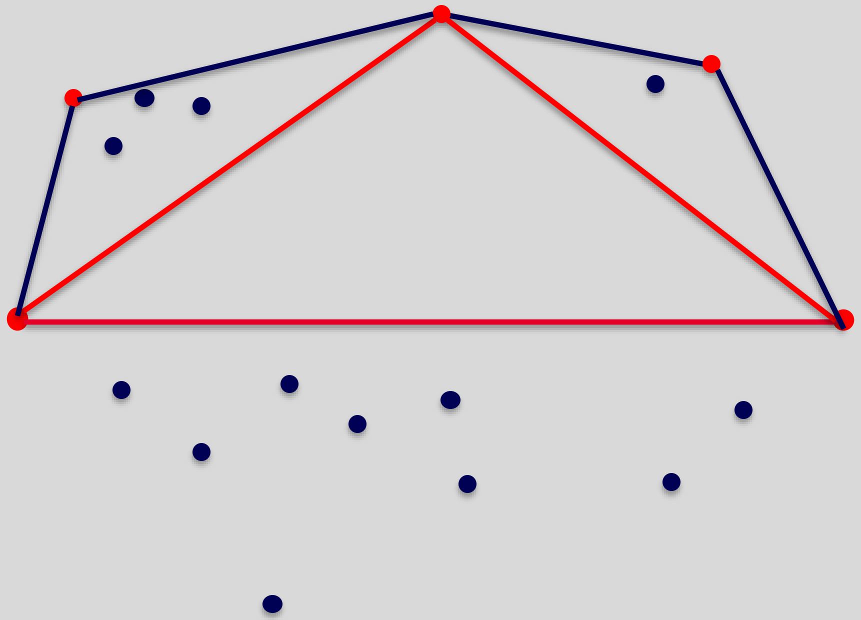
Quick Hull – Example



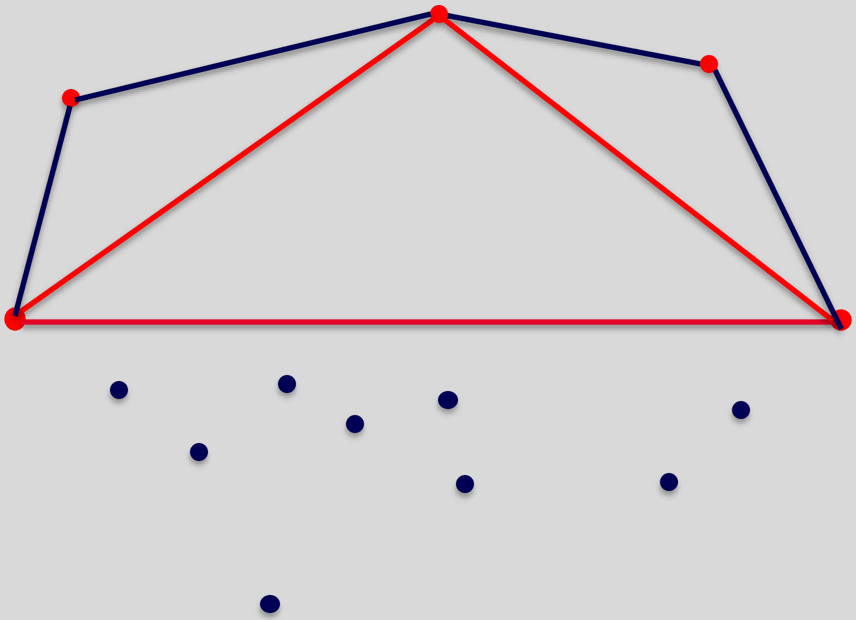
Quick Hull – Example



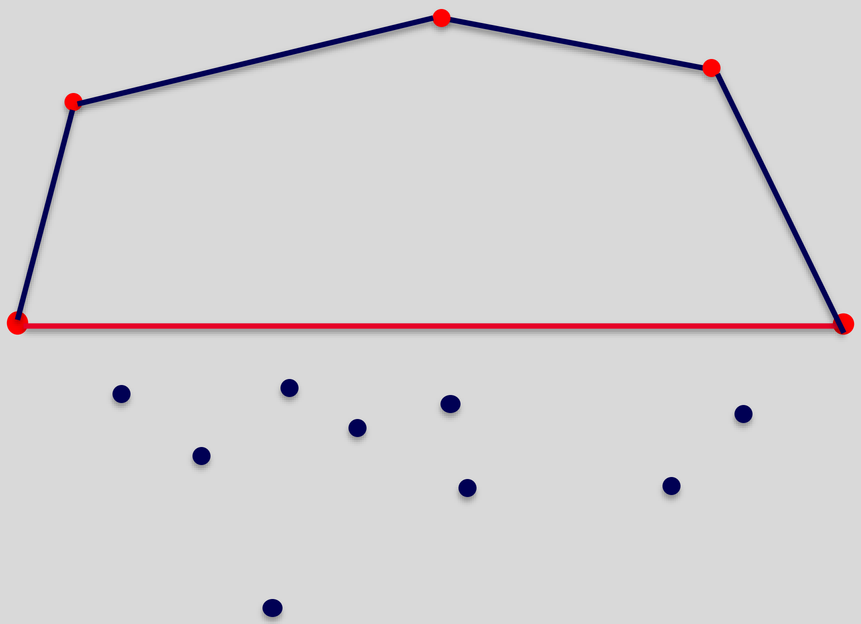
Quick Hull – Example



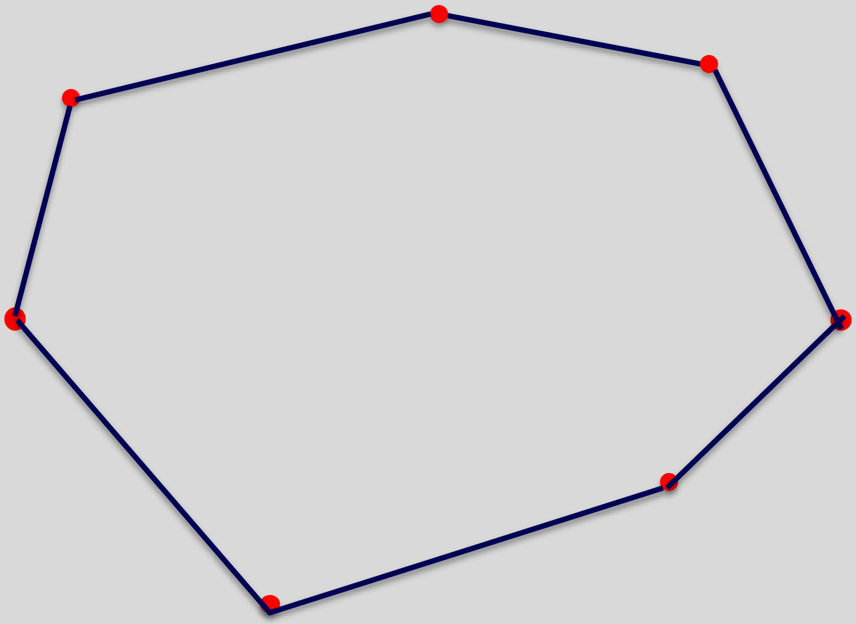
Quick Hull – Example



Quick Hull – Example



Quick Hull – Example



Quick Hull – Time Efficiency

- **Worst Case:** O(n^2) just as in quicksort.
- **Average Case:** O(*n*logn) under reasonable assumptions about the distribution of points given (assuming points are sorted).

Wrapping things up

Summary

- Introduced the divide-and-conquer algorithmic approach
- Master theorem to calculate asymptotic complexity of recurrence relations
- Sorting: merge sort and quick sort
- Convex hull by divide-and-conquer

RMIT UNIVERSITY