Chapter 1: Divide and Conquer

Advanced Algorithms

Divide · Break Down · Conquer · Combine | Master theorem | Merge sort and quicksort | Fast exponentiation | Quickhull | Closest pair of points | Karatsuba | Strassen

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides:
https://sercankulcu.github.io/files/advanced_algorithms/slides/Chapter_01_Divide_and_Conquer.html

Learning Outcomes

Paradigm

Explain the steps of divide and conquer and decide whether a problem suits the approach.

Analysis

Set up a recurrence and use the master theorem to find the running time of a divide-and-conquer algorithm.

Geometric algorithms

Trace Quickhull for the convex hull and the strip method for the closest pair of points.

Fast multiplication

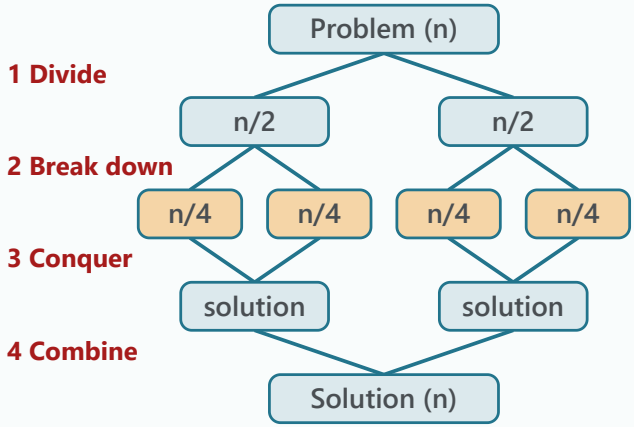
Show how Karatsuba (integers) and Strassen (matrices) gain speed by saving one recursive call.

PARADIGM

The Divide–Break Down–Conquer Paradigm

Divide and conquer solves complex problems effectively in four steps:

1. **Divide:** split the problem into smaller, manageable subproblems.
2. **Break down:** keep splitting until the subproblems are small enough to solve directly (recursion).
3. **Conquer:** solve each subproblem separately.
4. **Combine:** merge the subproblem solutions into a solution of the original problem.



Key: the subproblems are **of the same kind** as the original, only smaller, and **independent** of each other, so the same function calls itself recursively.

PARADIGM

Why Divide and Conquer?

- It breaks a complex problem into **small, manageable** pieces that are easier to understand and solve.
- The same method (the same function) is **reused** at every level: short, reusable code.
- Uses: sorting, searching, computational geometry, big-number and matrix arithmetic, and game-playing AI.

Advantages

+ Often an **asymptotic gain**: $O(n^2) \rightarrow O(n \log n)$, $O(n) \rightarrow O(\log n)$.
+ Independent subproblems can be solved in **parallel**.
+ Small subproblems fit in cache: good memory locality.
+ Correctness proofs by induction are easy.

Disadvantages

– Recursion overhead: call stack, extra memory (e.g., O(n) for merge sort).
– If subproblems **overlap**, the same work is repeated → exponential time.
– For small n the constant factors dominate; in practice, switch to a simple method below a threshold.

General Template and Recurrence

```
Result divideAndConquer(Problem p) {
  if (p.size() <= THRESHOLD)
    return solveDirectly(p);        // base case
  Problem[] subs = divide(p);       // a subproblems of size n/b
  Result[] sols = new Result[subs.length];
  for (int i = 0; i < subs.length; i++)
    sols[i] = divideAndConquer(subs[i]); // conquer: recurse
  return combine(sols);             // combine
}
```

Running time:

$T(n) = \mathbf{a} \cdot T(n/\mathbf{b}) + \mathbf{f(n)}$

a: number of subproblems · n/b: subproblem size · f(n): cost of dividing + combining

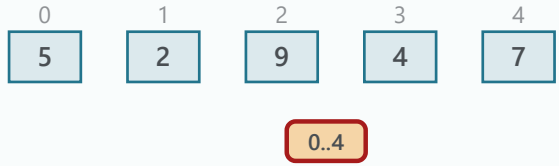
- Without a base case, the recursion never ends.
- The split should be **balanced**: n/2 + n/2 is good, 1 + (n − 1) is bad.
- Speedups usually come from reducing **a** (Karatsuba, Strassen) or shrinking **f(n)**.

The Template in Action: Maximum Element

```
1 int findMax(int[] a, int lo, int hi) {
2   if (lo == hi) return a[lo];       // base case
▶   int mid = (lo + hi) / 2;
▶   int left = findMax(a, lo, mid);
5   int right = findMax(a, mid + 1, hi);
6   return Math.max(left, right);     // combine
7 }
```

CALL STACK

- `findMax(0, 4)`
- `main()`



Array [5, 2, 9, 4, 7]; `findMax(a, 0, 4)` is called. The range has more than one element → mid = 2; the **left half** [0..2] is solved first.

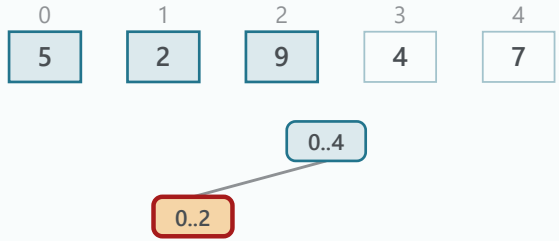
Step 1 / 13

VARIABLES

`lo = 0` `hi = 4` `mid = 2` `left = ?` `right = ?`

The Template in Action: Maximum Element

```
1 int findMax(int[] a, int lo, int hi) {
2   if (lo == hi) return a[lo];       // base case
▶   int mid = (lo + hi) / 2;
▶   int left = findMax(a, lo, mid);
5   int right = findMax(a, mid + 1, hi);
6   return Math.max(left, right);     // combine
7 }
```



CALL STACK

- `findMax(0, 2)`
- `findMax(0, 4)`
- `main()`

`findMax(0, 2)`: mid = 1 → left half [0..1] first, then right half [2..2].

Step 2 / 13

VARIABLES

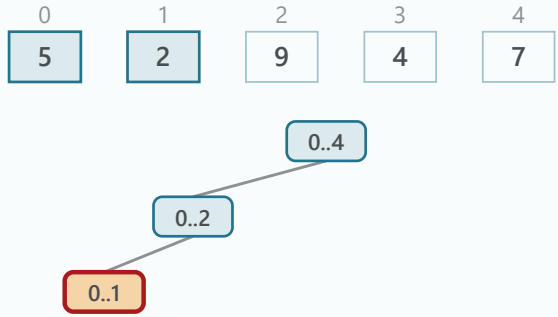
`lo = 0` `hi = 2` `mid = 1` `left = ?` `right = ?`

The Template in Action: Maximum Element

```
1 int findMax(int[] a, int lo, int hi) {
2   if (lo == hi) return a[lo];       // base case
▶   int mid = (lo + hi) / 2;
▶   int left = findMax(a, lo, mid);
5   int right = findMax(a, mid + 1, hi);
6   return Math.max(left, right);     // combine
7 }
```

CALL STACK

- `findMax(0, 1)`
- `findMax(0, 2)`
- `findMax(0, 4)`
- `main()`



`findMax(0, 1)` : mid = 0 → left half [0..0] first, then right half [1..1].

Step 3 / 13

VARIABLES

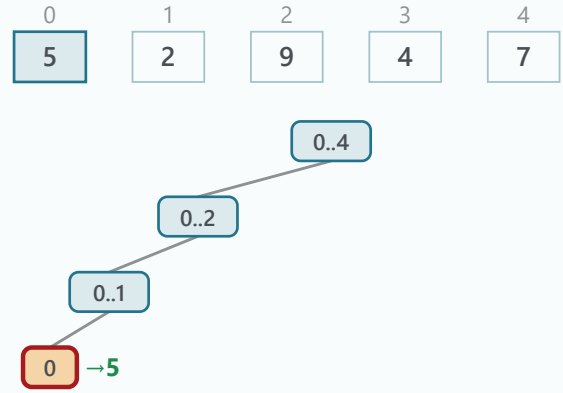
`lo = 0` `hi = 1` `mid = 0` `left = ?` `right = ?`

The Template in Action: Maximum Element

```
1 int findMax(int[] a, int lo, int hi) {
▶   if (lo == hi) return a[lo];        // base case
3   int mid = (lo + hi) / 2;
4   int left = findMax(a, lo, mid);
5   int right = findMax(a, mid + 1, hi);
6   return Math.max(left, right);     // combine
7 }
```

CALL STACK

- **findMax(0, 0)**
- findMax(0, 1)
- findMax(0, 2)
- findMax(0, 4)
- main()



`findMax(0, 0)` : one element → **base case**, returns a[0] = **5**.

Step 4 / 13

VARIABLES

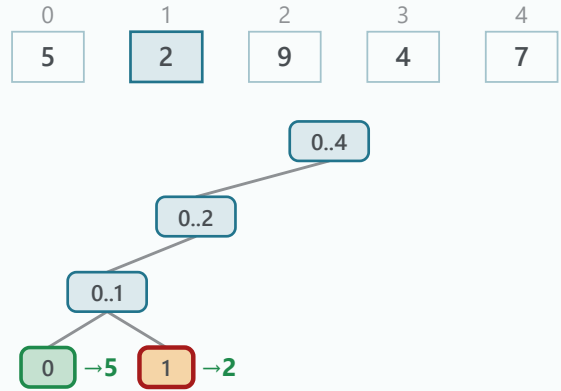
`lo = 0` `hi = 0` `mid = –` `left = –` `right = –`

The Template in Action: Maximum Element

```
1 int findMax(int[] a, int lo, int hi) {
▶   if (lo == hi) return a[lo];       // base case
3   int mid = (lo + hi) / 2;
4   int left = findMax(a, lo, mid);
5   int right = findMax(a, mid + 1, hi);
6   return Math.max(left, right);     // combine
7 }
```

CALL STACK

- **findMax(1, 1)**
- findMax(0, 1)
- findMax(0, 2)
- findMax(0, 4)
- main()



`findMax(1, 1)` : one element → **base case**, returns a[1] = **2**.

Step 5 / 13

VARIABLES

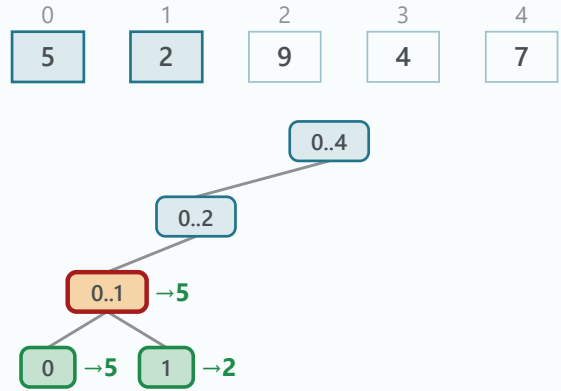
`lo = 1` `hi = 1` `mid = –` `left = –` `right = –`

The Template in Action: Maximum Element

```
1 int findMax(int[] a, int lo, int hi) {
2   if (lo == hi) return a[lo];       // base case
3   int mid = (lo + hi) / 2;
4   int left = findMax(a, lo, mid);
5   int right = findMax(a, mid + 1, hi);
▶   return Math.max(left, right);     // combine
7 }
```

CALL STACK

- **findMax(0, 1)**
- findMax(0, 2)
- findMax(0, 4)
- main()



`findMax(0, 1)` : left = 5, right = 2 → **combine**: returns max = **5**.

Step 6 / 13

VARIABLES

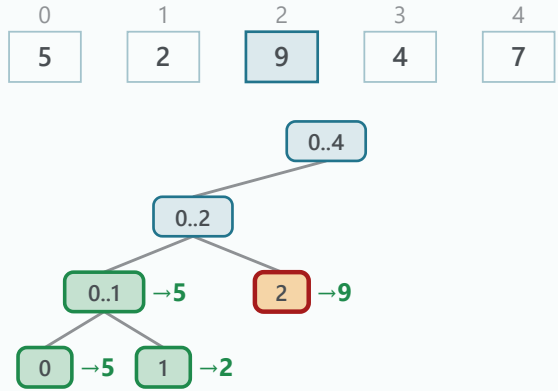
`lo = 0` `hi = 1` `mid = 0` `left = 5` `right = 2`

The Template in Action: Maximum Element

```
1 int findMax(int[] a, int lo, int hi) {
▶   if (lo == hi) return a[lo];        // base case
3   int mid = (lo + hi) / 2;
4   int left = findMax(a, lo, mid);
5   int right = findMax(a, mid + 1, hi);
6   return Math.max(left, right);     // combine
7 }
```

CALL STACK

- **`findMax(2, 2)`**
- `findMax(0, 2)`
- `findMax(0, 4)`
- `main()`



`findMax(2, 2)` : one element → **base case**, returns a[2] = **9**.

Step 7 / 13

VARIABLES

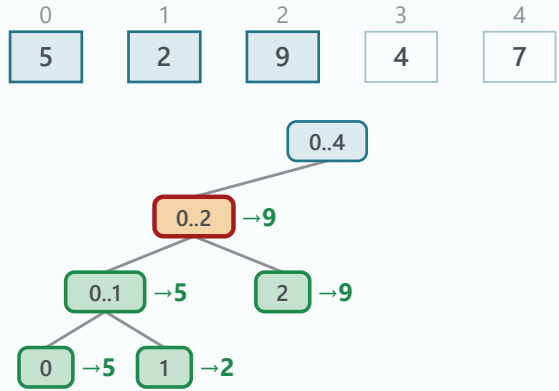
`lo = 2` `hi = 2` `mid = –` `left = –` `right = –`

The Template in Action: Maximum Element

```
1 int findMax(int[] a, int lo, int hi) {
2   if (lo == hi) return a[lo];        // base case
3   int mid = (lo + hi) / 2;
4   int left = findMax(a, lo, mid);
5   int right = findMax(a, mid + 1, hi);
▶   return Math.max(left, right);      // combine
7 }
```

CALL STACK

- `findMax(0, 2)`
- `findMax(0, 4)`
- `main()`



`findMax(0, 2)` : left = 5, right = 9 → **combine**: returns max = **9**.

Step 8 / 13

VARIABLES

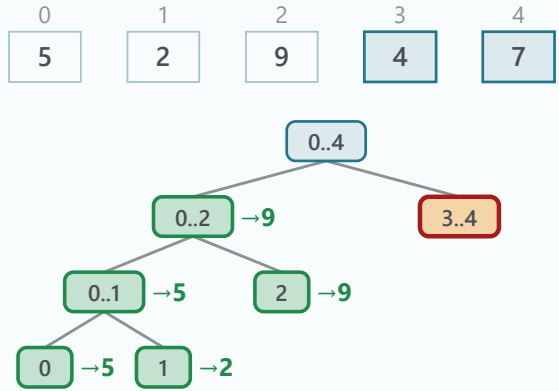
`lo = 0` `hi = 2` `mid = 1` `left = 5` `right = 9`

The Template in Action: Maximum Element

```
1 int findMax(int[] a, int lo, int hi) {
2   if (lo == hi) return a[lo];       // base case
▶   int mid = (lo + hi) / 2;
▶   int left = findMax(a, lo, mid);
5   int right = findMax(a, mid + 1, hi);
6   return Math.max(left, right);     // combine
7 }
```

CALL STACK

- `findMax(3, 4)`
- `findMax(0, 4)`
- `main()`



`findMax(3, 4)` : mid = 3 → left half [3..3] first, then right half [4..4].

Step 9 / 13

VARIABLES

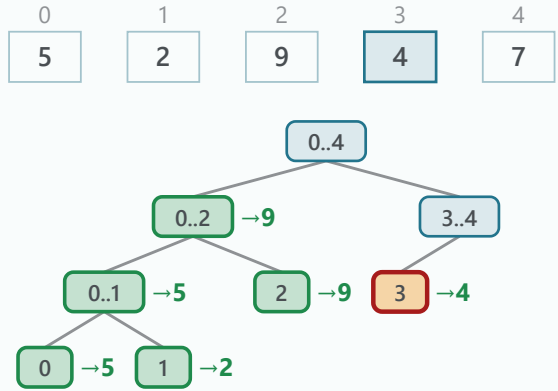
`lo = 3` `hi = 4` `mid = 3` `left = ?` `right = ?`

The Template in Action: Maximum Element

```
1 int findMax(int[] a, int lo, int hi) {
▶   if (lo == hi) return a[lo];        // base case
3   int mid = (lo + hi) / 2;
4   int left = findMax(a, lo, mid);
5   int right = findMax(a, mid + 1, hi);
6   return Math.max(left, right);     // combine
7 }
```

CALL STACK

- **findMax(3, 3)**
- findMax(3, 4)
- findMax(0, 4)
- main()



`findMax(3, 3)` : one element → **base case**, returns a[3] = **4**.

Step 10 / 13

VARIABLES

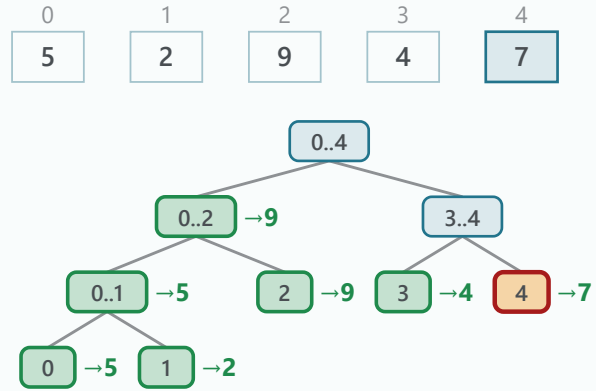
`lo = 3` `hi = 3` `mid = –` `left = –` `right = –`

The Template in Action: Maximum Element

```
1 int findMax(int[] a, int lo, int hi) {
▶   if (lo == hi) return a[lo];        // base case
3   int mid = (lo + hi) / 2;
4   int left = findMax(a, lo, mid);
5   int right = findMax(a, mid + 1, hi);
6   return Math.max(left, right);      // combine
7 }
```

CALL STACK

- **findMax(4, 4)**
- findMax(3, 4)
- findMax(0, 4)
- main()



`findMax(4, 4)` : one element → **base case**, returns a[4] = **7**.

Step 11 / 13

VARIABLES

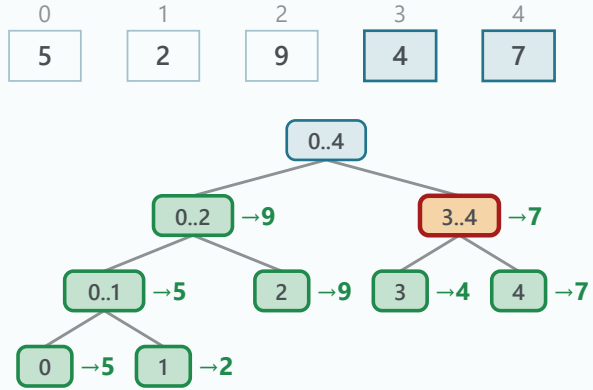
`lo = 4` `hi = 4` `mid = –` `left = –` `right = –`

The Template in Action: Maximum Element

```
1 int findMax(int[] a, int lo, int hi) {
2   if (lo == hi) return a[lo];        // base case
3   int mid = (lo + hi) / 2;
4   int left = findMax(a, lo, mid);
5   int right = findMax(a, mid + 1, hi);
▶   return Math.max(left, right);      // combine
7 }
```

CALL STACK

- `findMax(3, 4)`
- `findMax(0, 4)`
- `main()`



`findMax(3, 4)` : left = 4, right = 7 → **combine**: returns max = **7**.

Step 12 / 13

VARIABLES

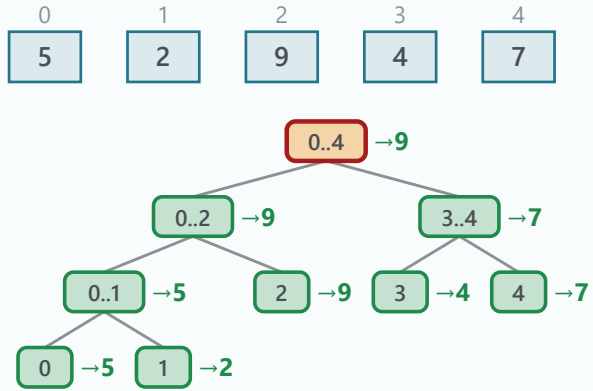
`lo = 3` `hi = 4` `mid = 3` `left = 4` `right = 7`

The Template in Action: Maximum Element

```
1 int findMax(int[] a, int lo, int hi) {
2   if (lo == hi) return a[lo];       // base case
3   int mid = (lo + hi) / 2;
4   int left = findMax(a, lo, mid);
5   int right = findMax(a, mid + 1, hi);
▶   return Math.max(left, right);     // combine
7 }
```

CALL STACK

- `findMax(0, 4)`
- `main()`



`findMax(0, 4)`: left = 9, right = 7 → **combine**: returns max = **9**. Result **9**: n leaves, n − 1 comparisons; T(n) = 2T(n/2) + O(1) = **O(n)**.

Step 13 / 13

VARIABLES

`lo = 0` `hi = 4` `mid = 2` `left = 9` `right = 7`

PARADIGM

Classic Divide-and-Conquer Algorithms

Algorithm	Divide	Combine	Recurrence	Time
Binary search	check the middle, discard one half	none	$T(n/2) + O(1)$	$O(\log n)$
Merge sort	split the array in half	merge the sorted halves	$2T(n/2) + O(n)$	$O(n \log n)$
Quicksort	partition around a pivot	none (in place)	$T(k) + T(n-k-1) + O(n)$	avg. $O(n \log n)$
Fast exponentiation	$x^n = (x^{\lfloor n/2 \rfloor})^2 \cdot (x)$	square	$T(n/2) + O(1)$	$O(\log n)$
Quickhull	triangle with the farthest point	join the hull pieces	$T(n_1) + T(n_2) + O(n)$	avg. $O(n \log n)$
Closest pair of points	split at the median x	scan the middle strip	$2T(n/2) + O(n)$	$O(n \log n)$
Karatsuba	split the number in half	add up 3 products	$3T(n/2) + O(n)$	$O(n^{1.585})$
Strassen	split the matrix into 4 blocks	add up 7 products	$7T(n/2) + O(n^2)$	$O(n^{2.807})$

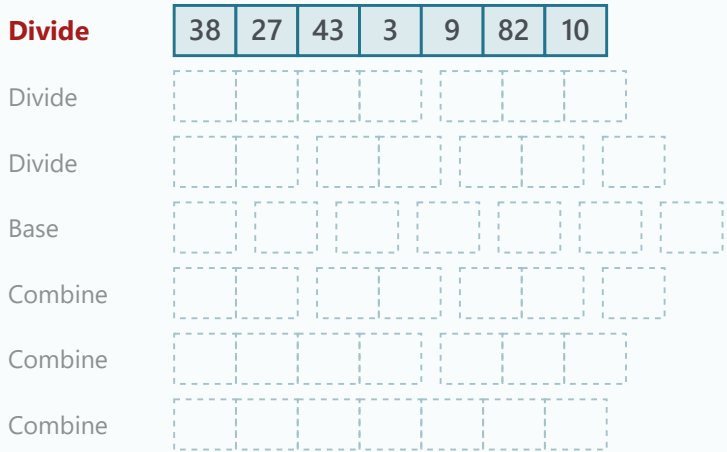
The Master Theorem and This Chapter's Algorithms

$T(n) = a \cdot T(n/b) + f(n)$, $a \geq 1$, $b > 1$. Compare $f(n)$ with the total leaf work $n^{\log_b a}$: if $f(n)$ is polynomially smaller, **$\Theta(n^{\log_b a})$** (case 1, leaves dominate); if it is of the same order, **$\Theta(n^{\log_b a} \cdot \log n)$** (case 2); if it is polynomially larger and the regularity condition holds, **$\Theta(f(n))$** (case 3, root dominates).

Algorithm	a	b	f(n)	$n^{\log_b a}$	Case	T(n)
Binary search (decrease and conquer), fast exponentiation	1	2	1	$n^0 = 1$	2	$\Theta(\log n)$
Merge sort, closest pair, balanced quicksort	2	2	n	n	2	$\Theta(n \log n)$
Karatsuba	3	2	n	$n^{1.585}$	1	$\Theta(n^{1.585})$
Naive block matrix multiplication	8	2	n^2	n^3	1	$\Theta(n^3)$
Strassen	7	2	n^2	$n^{2.807}$	1	$\Theta(n^{2.807})$

All of the gain in Karatsuba and Strassen comes from reducing **a** by one: 4 → 3 and 8 → 7. Quickhull ($T(n_1) + T(n_2) + n$) and unbalanced quicksort ($T(n - 1) + n$) have unequal splits, so the master theorem does not apply; solve them with a recursion tree.

Merge Sort: A Divide-and-Conquer View

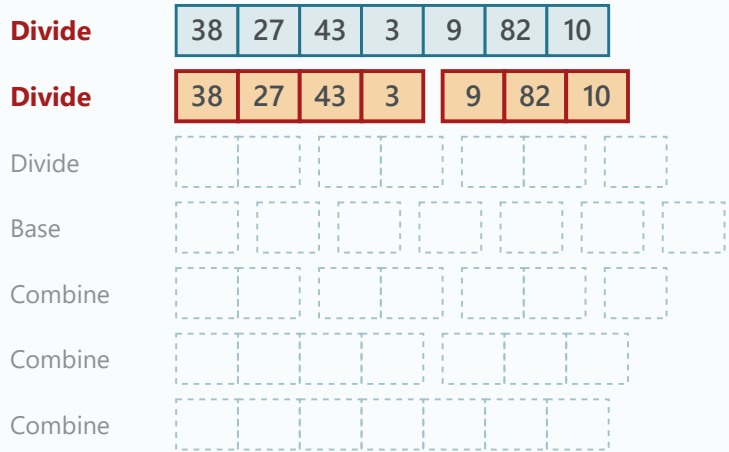


Array: [38, 27, 43, 3, 9, 82, 10], n = 7.

Step 1 / 7

- **Divide:** split the array in the middle, O(1). **Conquer:** sort both halves recursively. **Combine:** merge the sorted halves in one pass, Θ(n).
- T(n) = 2T(n/2) + Θ(n) → case 2 → **Θ(n log n)** in the best, average, and worst case alike. The real work happens in the merge.

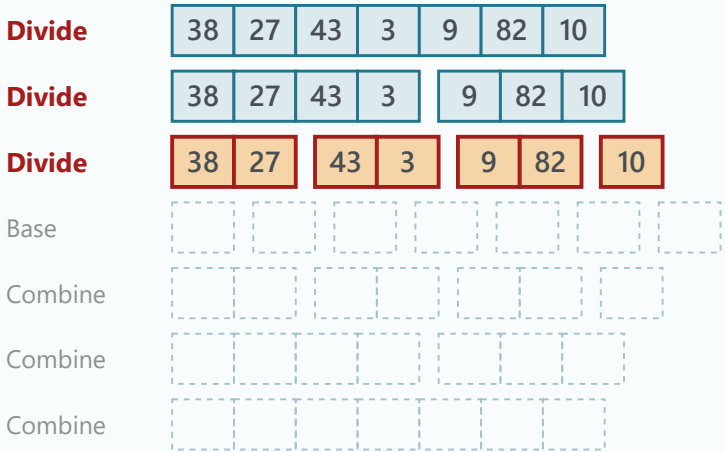
Merge Sort: A Divide-and-Conquer View



Divide: mid = (0 + 6)/2 = 3 → [38 27 43 3] and [9 82 10].

- **Divide:** split the array in the middle, O(1). **Conquer:** sort both halves recursively. **Combine:** merge the sorted halves in one pass, Θ(n).
- T(n) = 2T(n/2) + Θ(n) → case 2 → **Θ(n log n)** in the best, average, and worst case alike. The real work happens in the merge.

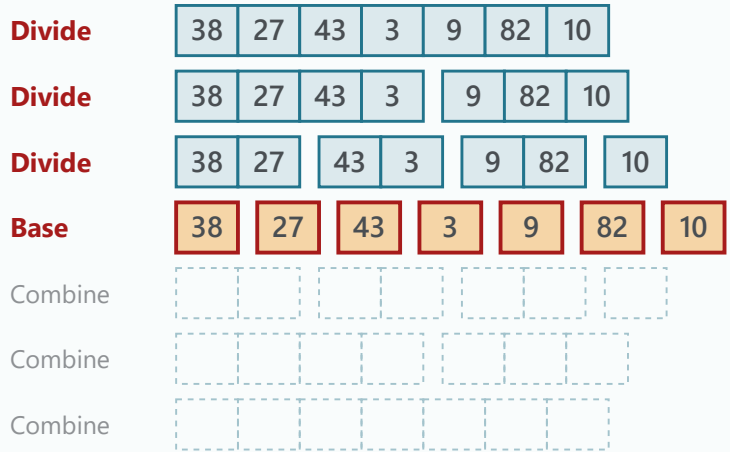
Merge Sort: A Divide-and-Conquer View



The halves are split again: [38 27] [43 3] [9 82] [10].

- **Divide:** split the array in the middle, O(1). **Conquer:** sort both halves recursively. **Combine:** merge the sorted halves in one pass, Θ(n).
- T(n) = 2T(n/2) + Θ(n) → case 2 → **Θ(n log n)** in the best, average, and worst case alike. The real work happens in the merge.

Merge Sort: A Divide-and-Conquer View



One-element arrays are already sorted: the **base case**.

- **Divide:** split the array in the middle, O(1). **Conquer:** sort both halves recursively. **Combine:** merge the sorted halves in one pass, Θ(n).
- T(n) = 2T(n/2) + Θ(n) → case 2 → **Θ(n log n)** in the best, average, and worst case alike. The real work happens in the merge.

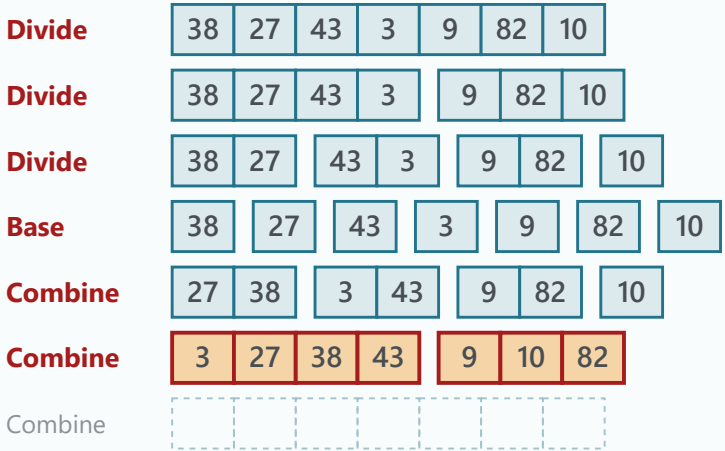
Merge Sort: A Divide-and-Conquer View



Combine: pairs are merged in order: [27 38] [3 43] [9 82] [10].

- **Divide:** split the array in the middle, O(1). **Conquer:** sort both halves recursively. **Combine:** merge the sorted halves in one pass, Θ(n).
- T(n) = 2T(n/2) + Θ(n) → case 2 → **Θ(n log n)** in the best, average, and worst case alike. The real work happens in the merge.

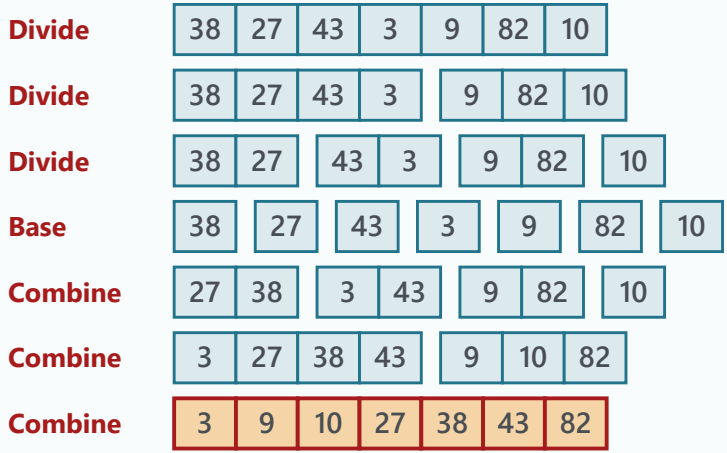
Merge Sort: A Divide-and-Conquer View



[3 27 38 43] and [9 10 82]: each merge takes as many steps as the two halves have elements.

- **Divide:** split the array in the middle, O(1). **Conquer:** sort both halves recursively. **Combine:** merge the sorted halves in one pass, Θ(n).
- T(n) = 2T(n/2) + Θ(n) → case 2 → **Θ(n log n)** in the best, average, and worst case alike. The real work happens in the merge.

Merge Sort: A Divide-and-Conquer View



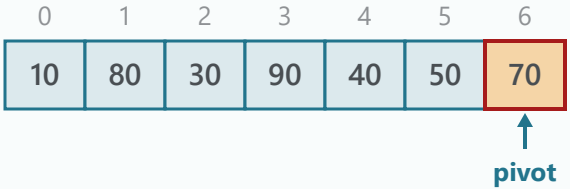
Final merge: **[3 9 10 27 38 43 82]**. Each level costs O(n); $\lceil \log_2 7 \rceil = 3$ levels.

- **Divide:** split the array in the middle, O(1). **Conquer:** sort both halves recursively. **Combine:** merge the sorted halves in one pass, Θ(n).
- T(n) = 2T(n/2) + Θ(n) → case 2 → **Θ(n log n)** in the best, average, and worst case alike. The real work happens in the merge.

BASIC EXAMPLES

Quicksort: A Divide-and-Conquer View

Divide: partition around a pivot, Θ(n). **Conquer:** sort both parts recursively. **Combine:** nothing; the parts are sorted in place. The real work happens in the partition.



Array [10 80 30 90 40 50 70]; Lomuto partition: the **last element is the pivot** (70).

Step 1 / 7

Balanced split

If the pivot is near the median, T(n) = 2T(n/2) + O(n) → **O(n log n)** (average case).

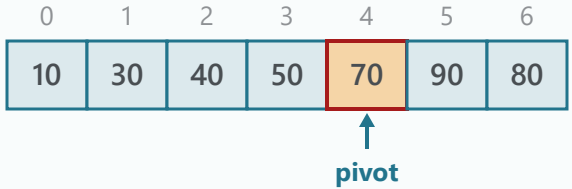
Unbalanced split

Sorted input with the last element as pivot: T(n) = T(n − 1) + O(n) → **$O(n^2)$**. Remedy: random pivot, median of three.

BASIC EXAMPLES

Quicksort: A Divide-and-Conquer View

Divide: partition around a pivot, Θ(n). **Conquer:** sort both parts recursively. **Combine:** nothing; the parts are sorted in place. The real work happens in the partition.



partition(arr, 0, 6), pivot **70** → placed at index 4. Left: [10 30 40 50], right: [90 80].

Step 2 / 7

Balanced split

If the pivot is near the median, T(n) = 2T(n/2) + O(n) → **O(n log n)** (average case).

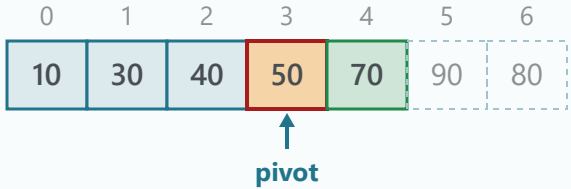
Unbalanced split

Sorted input with the last element as pivot: T(n) = T(n − 1) + O(n) → **$O(n^2)$**. Remedy: random pivot, median of three.

BASIC EXAMPLES

Quicksort: A Divide-and-Conquer View

Divide: partition around a pivot, Θ(n). **Conquer:** sort both parts recursively. **Combine:** nothing; the parts are sorted in place. The real work happens in the partition.



partition(arr, 0, 3), pivot **50** → placed at index 3. Left: [10 30 40], right: [—]. **Unbalanced split!**

Step 3 / 7

Balanced split

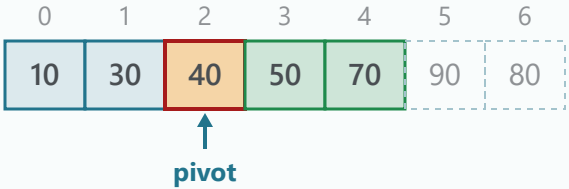
If the pivot is near the median, T(n) = 2T(n/2) + O(n) → **O(n log n)** (average case).

Unbalanced split

Sorted input with the last element as pivot: T(n) = T(n − 1) + O(n) → **$O(n^2)$**. Remedy: random pivot, median of three.

Quicksort: A Divide-and-Conquer View

Divide: partition around a pivot, Θ(n). **Conquer:** sort both parts recursively. **Combine:** nothing; the parts are sorted in place. The real work happens in the partition.



partition(arr, 0, 2), pivot **40** → placed at index 2. Left: [10 30], right: [—]. **Unbalanced split!**

Step 4 / 7

Balanced split

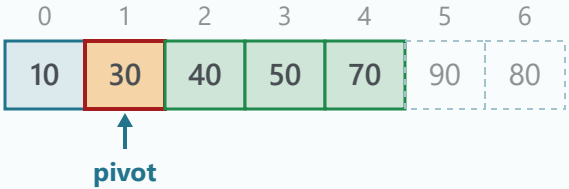
If the pivot is near the median, T(n) = 2T(n/2) + O(n) → **O(n log n)** (average case).

Unbalanced split

Sorted input with the last element as pivot: T(n) = T(n − 1) + O(n) → **$O(n^2)$**. Remedy: random pivot, median of three.

Quicksort: A Divide-and-Conquer View

Divide: partition around a pivot, Θ(n). **Conquer:** sort both parts recursively. **Combine:** nothing; the parts are sorted in place. The real work happens in the partition.



partition(arr, 0, 1), pivot **30** → placed at index 1. Left: [10], right: [—]. **Unbalanced split!**

Step 5 / 7

Balanced split

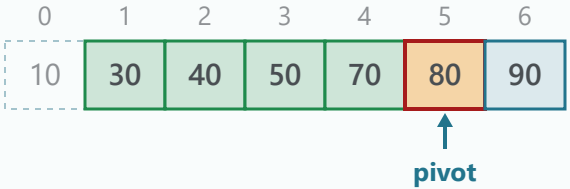
If the pivot is near the median, T(n) = 2T(n/2) + O(n) → **O(n log n)** (average case).

Unbalanced split

Sorted input with the last element as pivot: T(n) = T(n − 1) + O(n) → **$O(n^2)$**. Remedy: random pivot, median of three.

Quicksort: A Divide-and-Conquer View

Divide: partition around a pivot, Θ(n). **Conquer:** sort both parts recursively. **Combine:** nothing; the parts are sorted in place. The real work happens in the partition.



partition(arr, 5, 6), pivot **80** → placed at index 5. Left: [—], right: [90]. **Unbalanced split!**

Balanced split

If the pivot is near the median, T(n) = 2T(n/2) + O(n) → **O(n log n)** (average case).

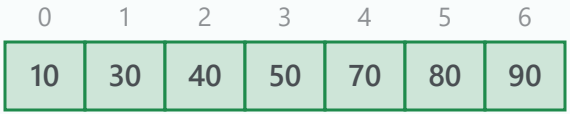
Unbalanced split

Sorted input with the last element as pivot: T(n) = T(n − 1) + O(n) → **$O(n^2)$**. Remedy: random pivot, median of three.

BASIC EXAMPLES

Quicksort: A Divide-and-Conquer View

Divide: partition around a pivot, Θ(n). **Conquer:** sort both parts recursively. **Combine:** nothing; the parts are sorted in place. The real work happens in the partition.



Sorted: **[10 30 40 50 70 80 90]**. No combine step; the work is done in the partition.

Step 7 / 7

Balanced split

If the pivot is near the median, T(n) = 2T(n/2) + O(n) → **O(n log n)** (average case).

Unbalanced split

Sorted input with the last element as pivot: T(n) = T(n − 1) + O(n) → **$O(n^2)$**. Remedy: random pivot, median of three.

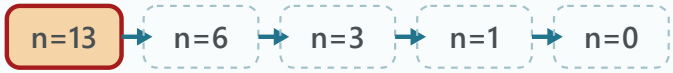
Fast Exponentiation: Tracing 3^{13}

```
1 long power(long x, int n) {
2   if (n == 0) return 1;          // base
  case
▶   long y = power(x, n / 2);
4   if (n % 2 == 0) return y * y; // even
5   return x * y * y;              // odd
6 }
```

CALL STACK

- `power(3, 13)`
- `main()`

The naive method uses 12 multiplications, this one 7; for n = 10^6, 999,999 versus 27 → **O(log n)**: T(n) = T(n/2) + O(1).



$3^{13} = 3\cdot(3^6)^2$; $3^6 = (3^3)^2$; $3^3 = 3\cdot(3^1)^2$

power(3, 13) is called → first **power(3, 6)** must be computed (n/2 = 6).

Step 1 / 9

VARIABLES

x = **3** n = **13** y = **?**

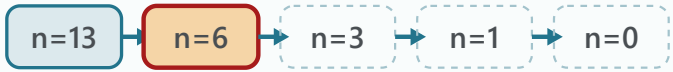
Fast Exponentiation: Tracing 3^{13}

```
1 long power(long x, int n) {
2   if (n == 0) return 1;          // base
  case
▶   long y = power(x, n / 2);
4   if (n % 2 == 0) return y * y; // even
5   return x * y * y;              // odd
6 }
```

CALL STACK

- **power(3, 6)**
- power(3, 13)
- main()

The naive method uses 12 multiplications, this one 7; for n = 10^6, 999,999 versus 27 → **O(log n)**: T(n) = T(n/2) + O(1).



$3^{13} = 3\cdot(3^6)^2$; $3^6 = (3^3)^2$; $3^3 = 3\cdot(3^1)^2$

power(3, 6) is called → first **power(3, 3)** must be computed (n/2 = 3).

Step 2 / 9

VARIABLES

x = **3** | n = **6** | y = **?**

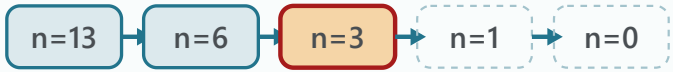
Fast Exponentiation: Tracing 3^{13}

```
1 long power(long x, int n) {
2   if (n == 0) return 1;          // base
  case
▶   long y = power(x, n / 2);
4   if (n % 2 == 0) return y * y; // even
5   return x * y * y;              // odd
6 }
```

CALL STACK

- `power(3, 3)`
- `power(3, 6)`
- `power(3, 13)`
- `main()`

The naive method uses 12 multiplications, this one 7; for n = 10^6, 999,999 versus 27 → **O(log n)**: T(n) = T(n/2) + O(1).



$3^{13} = 3\cdot(3^6)^2$; $3^6 = (3^3)^2$; $3^3 = 3\cdot(3^1)^2$

power(3, 3) is called → first **power(3, 1)** must be computed (n/2 = 1).

Step 3 / 9

VARIABLES

x = **3** | n = **3** | y = **?**

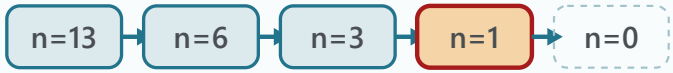
Fast Exponentiation: Tracing 3^{13}

```
1 long power(long x, int n) {
2   if (n == 0) return 1;          // base
  case
▶   long y = power(x, n / 2);
4   if (n % 2 == 0) return y * y; // even
5   return x * y * y;              // odd
6 }
```

CALL STACK

- **power(3, 1)**
- power(3, 3)
- power(3, 6)
- power(3, 13)
- main()

The naive method uses 12 multiplications, this one 7; for n = 10^6, 999,999 versus 27 → **O(log n)**: T(n) = T(n/2) + O(1).



$3^{13} = 3\cdot(3^6)^2$; $3^6 = (3^3)^2$; $3^3 = 3\cdot(3^1)^2$

power(3, 1) is called → first **power(3, 0)** must be computed (n/2 = 0).

Step 4 / 9

VARIABLES

x = **3** n = **1** y = **?**

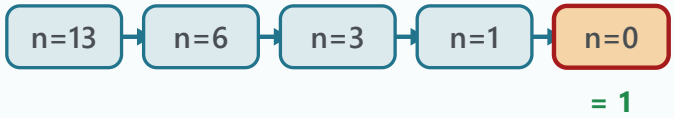
Fast Exponentiation: Tracing 3^{13}

```
1 long power(long x, int n) {
▶   if (n == 0) return 1;          // base
  case
3   long y = power(x, n / 2);
4   if (n % 2 == 0) return y * y; // even
5   return x * y * y;             // odd
6 }
```

CALL STACK

- `power(3, 0)`
- `power(3, 1)`
- `power(3, 3)`
- `power(3, 6)`
- `power(3, 13)`
- `main()`

The naive method uses 12 multiplications, this one 7; for n = 10^6, 999,999 versus 27 → **O(log n)**: T(n) = T(n/2) + O(1).



$3^{13} = 3\cdot(3^6)^2$; $3^6 = (3^3)^2$; $3^3 = 3\cdot(3^1)^2$

n = 0 → **base case**: returns $3^0 = 1$.

Step 5 / 9

VARIABLES

`x = 3` `n = 0` `y = –`

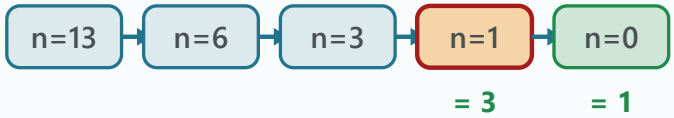
Fast Exponentiation: Tracing 3^{13}

```
1 long power(long x, int n) {
2   if (n == 0) return 1;          // base case
▶   long y = power(x, n / 2);
4   if (n % 2 == 0) return y * y; // even
▶   return x * y * y;              // odd
6 }
```

CALL STACK

- **power(3, 1)**
- power(3, 3)
- power(3, 6)
- power(3, 13)
- main()

The naive method uses 12 multiplications, this one 7; for n = 10^6, 999,999 versus 27 → **O(log n)**: T(n) = T(n/2) + O(1).



$3^{13} = 3\cdot(3^6)^2$; $3^6 = (3^3)^2$; $3^3 = 3\cdot(3^1)^2$

power(3, 1): y = 1; 1 is odd → x·y·y = 3·1·1 = **3**.

Step 6 / 9

VARIABLES

x = **3** n = **1** y = **1**

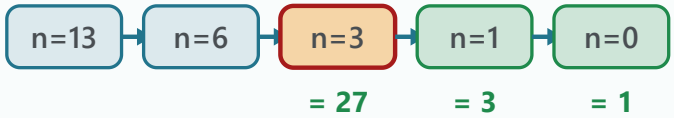
Fast Exponentiation: Tracing 3^{13}

```
1 long power(long x, int n) {
2   if (n == 0) return 1;          // base case
▶   long y = power(x, n / 2);
4   if (n % 2 == 0) return y * y; // even
▶   return x * y * y;              // odd
6 }
```

CALL STACK

- `power(3, 3)`
- `power(3, 6)`
- `power(3, 13)`
- `main()`

The naive method uses 12 multiplications, this one 7; for n = 10^6, 999,999 versus 27 → **O(log n)**: T(n) = T(n/2) + O(1).



$3^{13} = 3\cdot(3^6)^2$; $3^6 = (3^3)^2$; $3^3 = 3\cdot(3^1)^2$

power(3, 3): y = 3; 3 is odd → x·y·y = 3·3·3 = **27**.

Step 7 / 9

VARIABLES

x = **3** n = **3** y = **3**

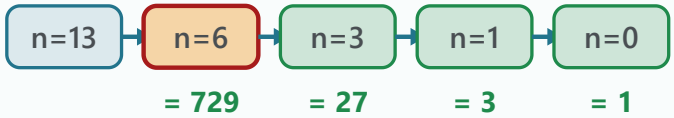
Fast Exponentiation: Tracing 3^{13}

```
1 long power(long x, int n) {
2   if (n == 0) return 1;          // base
  case
▶   long y = power(x, n / 2);
▶   if (n % 2 == 0) return y * y; // even
5   return x * y * y;              // odd
6 }
```

CALL STACK

- **power(3, 6)**
- power(3, 13)
- main()

The naive method uses 12 multiplications, this one 7; for n = 10^6, 999,999 versus 27 → **O(log n)**: T(n) = T(n/2) + O(1).



$3^{13} = 3\cdot(3^6)^2$; $3^6 = (3^3)^2$; $3^3 = 3\cdot(3^1)^2$

power(3, 6): y = 27; 6 is even → y·y = 27·27 = **729**.

Step 8 / 9

VARIABLES

x = **3** n = **6** y = **27**

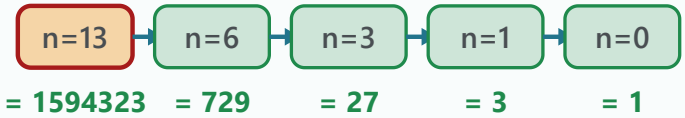
Fast Exponentiation: Tracing 3^{13}

```
1 long power(long x, int n) {
2   if (n == 0) return 1;          // base
  case
▶   long y = power(x, n / 2);
4   if (n % 2 == 0) return y * y; // even
▶   return x * y * y;              // odd
6 }
```

CALL STACK

- `power(3, 13)`
- `main()`

The naive method uses 12 multiplications, this one 7; for n = 10^6, 999,999 versus 27 → **O(log n)**: T(n) = T(n/2) + O(1).



$3^{13} = 3\cdot(3^6)^2$; $3^6 = (3^3)^2$; $3^3 = 3\cdot(3^1)^2$

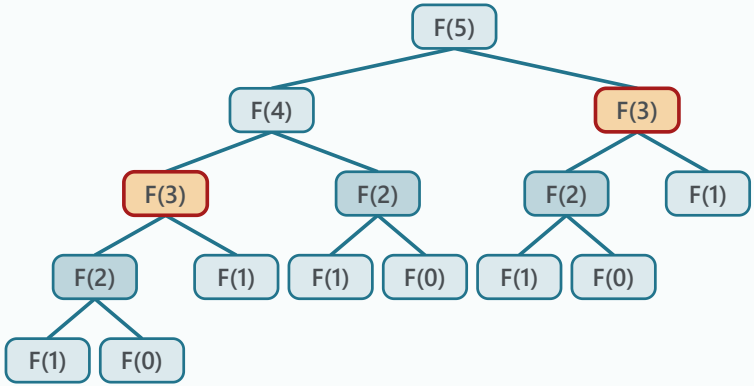
power(3, 13): y = 729; 13 is odd → x·y·y = 3·729·729 = **1594323**.

Step 9 / 9

VARIABLES

x = **3** n = **13** y = **729**

When Does It Fail? Overlapping Subproblems



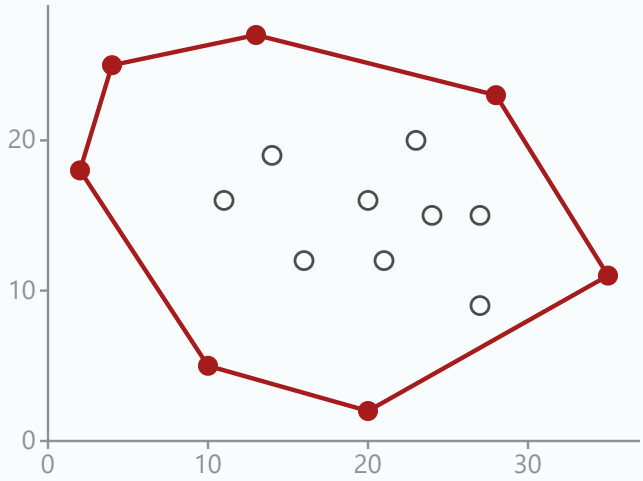

```
long fib(int n) {
  if (n < 2) return n;
  return fib(n - 1) + fib(n - 2);
}
```

- **15 calls** for F(5): F(3) is computed 2 times, F(2) 3 times.
- The subproblems are **not independent**; they overlap → $T(n) = T(n-1) + T(n-2) + O(1) = \mathbf{O(\varphi^n)} \approx O(1.618^n)$.
- Fix: store the results → **dynamic programming**, O(n).

The Fibonacci recursion can be written in divide-and-conquer form and gives the right answer, but it is a **bad** example: divide and conquer is efficient only when the subproblems are independent.

Convex Hull

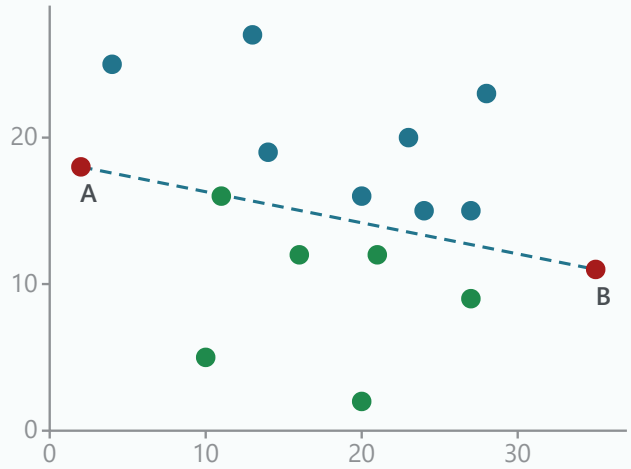
- Let S be a set of n points in the plane. The **convex hull** of S is the **smallest convex polygon** containing S.
- Hammer a nail into each point and stretch a **rubber band** around them: the shape it takes is the hull.
- The hull's vertices are points of S; the output is the list of these vertices in counterclockwise order.
- Convex: the segment joining any two points inside lies entirely within the polygon.



Uses: collision detection (games, robotics), geographic information systems, shape analysis in image processing, farthest pair of points, clustering.

The Quickhull Algorithm

1. The points with the smallest (A) and largest (B) x are on the hull.
2. Line AB splits the points into **two subsets**: above and below it.
3. In a subset, the point C **farthest** from AB is on the hull.
4. **Points inside** triangle ACB **are discarded**: they cannot be on the hull.
5. **Recurse** on the points outside AC and outside CB.
6. Stop when no points remain; the points found form the hull.



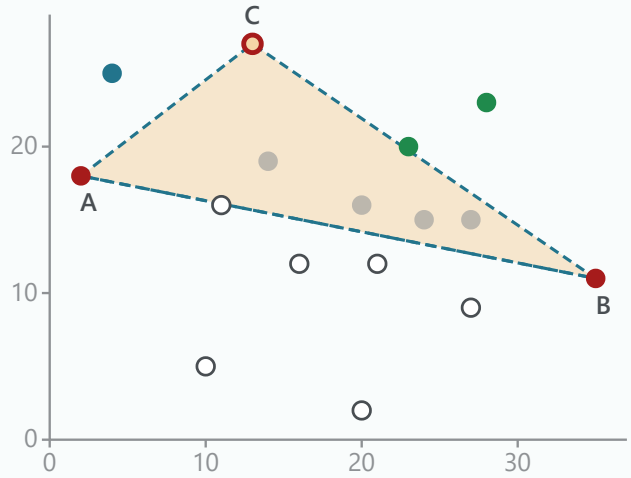
Steps 1–2: the leftmost A and the rightmost B are on the hull; line AB splits the points into **two sets** (upper / lower).

Step 1 / 3

Like quicksort (hence the name): the farthest point is the pivot, and the triangle's sides partition the rest.

The Quickhull Algorithm

1. The points with the smallest (A) and largest (B) x are on the hull.
2. Line AB splits the points into **two subsets**: above and below it.
3. In a subset, the point C **farthest** from AB is on the hull.
4. **Points inside** triangle ACB **are discarded**: they cannot be on the hull.
5. **Recurse** on the points outside AC and outside CB.
6. Stop when no points remain; the points found form the hull.

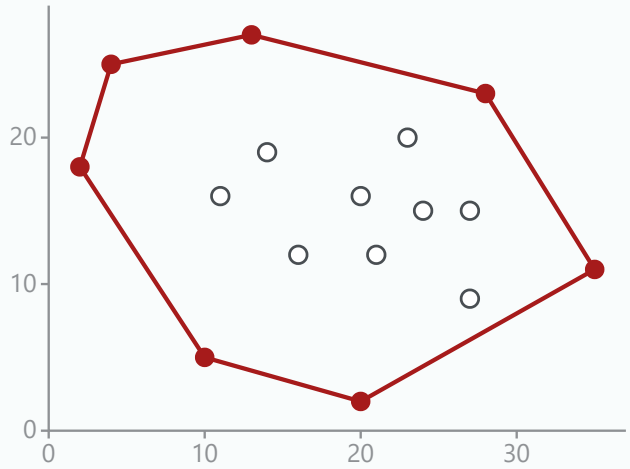


Steps 3–5: C (13,27), **farthest** from AB in the upper set, is on the hull. **The 4 points inside ACB are discarded**; recurse outside AC and CB.

Like quicksort (hence the name): the farthest point is the pivot, and the triangle's sides partition the rest.

The Quickhull Algorithm

1. The points with the smallest (A) and largest (B) x are on the hull.
2. Line AB splits the points into **two subsets**: above and below it.
3. In a subset, the point C **farthest** from AB is on the hull.
4. **Points inside** triangle ACB **are discarded**: they cannot be on the hull.
5. **Recurse** on the points outside AC and outside CB.
6. Stop when no points remain; the points found form the hull.



Step 6: recurse until no point is left outside → **a 7-vertex hull**.

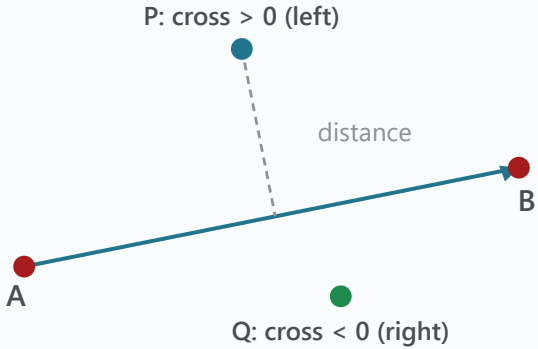
Like quicksort (hence the name): the farthest point is the pivot, and the triangle's sides partition the rest.

Geometry: Orientation Test and Distance

Quickhull needs only **integer** multiplication; no angles, square roots, or division:

```
long cross(Point A, Point B, Point P) {
  return (long)(B.x - A.x) * (P.y - A.y)
       - (long)(B.y - A.y) * (P.x - A.x);
}
```

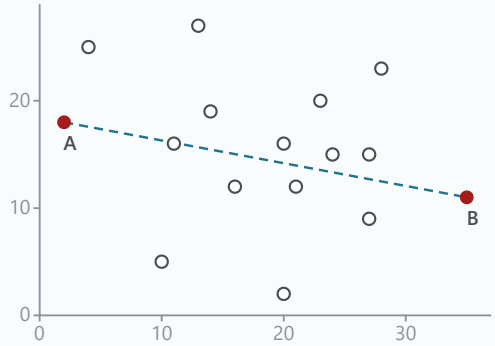
- **cross > 0**: P is to the left of A→B; **< 0**: to the right; **= 0**: on the line.
- |cross| = 2 × (area of triangle ABP) = |AB| × distance. Since |AB| is fixed, **farthest point = largest cross**.



Example: A = (2,18), B = (35,11). For C = (13,27), cross = 33·9 − (−7)·11 = **374**; for (28,23), 33·5 + 7·26 = **347**. C has the largest value → C is the farthest point.

Quickhull: Code Trace

```
1  void quickHull(List<Point> P) {
▶    Point A = leftmost(P), B = rightmost(P);
▶    hull.add(A); hull.add(B);
4    findHull(leftOf(P, A, B), A, B); // upper set
5    findHull(leftOf(P, B, A), B, A); // lower set
6  }
7  void findHull(List<Point> S, Point A, Point B)
   {
8    if (S.isEmpty()) return; // AB: hull edge
9    Point C = farthest(S, A, B); // max cross
10   hull.add(C);
11   findHull(leftOf(S, A, C), A, C); // outside
   AC
12   findHull(leftOf(S, C, B), C, B); // outside
   CB
13 }  // inside ACB: dropped
```



The leftmost A = (2,18) and the rightmost B = (35,11) are added to the hull.

Step 1 / 15

CALL STACK

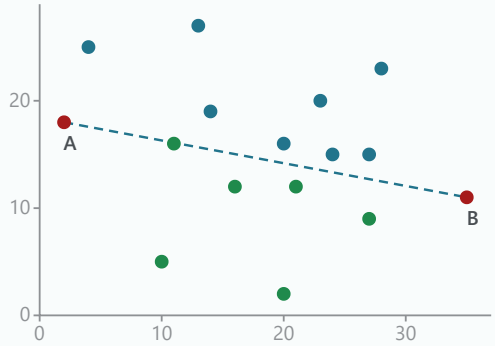
`quickHull(P)`

VARIABLES

`A = (2,18)` `B = (35,11)` `|hull| = 2`

Quickhull: Code Trace

```
1  void quickHull(List<Point> P) {
2    Point A = leftmost(P), B = rightmost(P);
3    hull.add(A); hull.add(B);
▶    findHull(leftOf(P, A, B), A, B); // upper set
5    findHull(leftOf(P, B, A), B, A); // lower set
6  }
7  void findHull(List<Point> S, Point A, Point B)
   {
8    if (S.isEmpty()) return; // AB: hull edge
9    Point C = farthest(S, A, B); // max cross
10   hull.add(C);
11   findHull(leftOf(S, A, C), A, C); // outside
   AC
12   findHull(leftOf(S, C, B), C, B); // outside
   CB
13 }  // inside ACB: dropped
```



8 points lie left of AB (upper), **6** right of it (lower). The upper set comes first.

Step 2 / 15

CALL STACK

`findHull(8 pts, (2,18)→(35,11))`

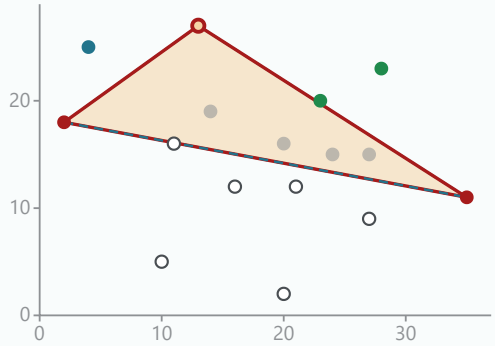
`quickHull(P)`

VARIABLES

`A = (2,18)` `B = (35,11)` `|hull| = 2`

Quickhull: Code Trace

```
1  void quickHull(List<Point> P) {
2    Point A = leftmost(P), B = rightmost(P);
3    hull.add(A); hull.add(B);
4    findHull(leftOf(P, A, B), A, B); // upper set
5    findHull(leftOf(P, B, A), B, A); // lower set
6  }
7  void findHull(List<Point> S, Point A, Point B)
   {
8    if (S.isEmpty()) return; // AB: hull edge
▶    Point C = farthest(S, A, B); // max cross
▶    hull.add(C);
11   findHull(leftOf(S, A, C), A, C); // outside
   AC
12   findHull(leftOf(S, C, B), C, B); // outside
   CB
13 }  // inside ACB: dropped
```



|S| = 8: the farthest point **C = (13,27)** joins the hull. 4 points inside the triangle are discarded; 1 point lies outside AC, 2 outside CB.

Step 3 / 15

CALL STACK

`findHull(8 pts, (2,18)→(35,11))`

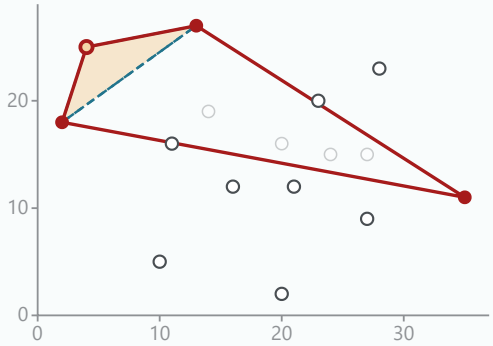
`quickHull(P)`

VARIABLES

`A = (2,18)` `B = (35,11)` `C = (13,27)` `|S| = 8` `|hull| = 3`

Quickhull: Code Trace

```
1  void quickHull(List<Point> P) {
2    Point A = leftmost(P), B = rightmost(P);
3    hull.add(A); hull.add(B);
4    findHull(leftOf(P, A, B), A, B); // upper set
5    findHull(leftOf(P, B, A), B, A); // lower set
6  }
7  void findHull(List<Point> S, Point A, Point B)
   {
8    if (S.isEmpty()) return; // AB: hull edge
▶    Point C = farthest(S, A, B); // max cross
▶    hull.add(C);
11   findHull(leftOf(S, A, C), A, C); // outside
   AC
12   findHull(leftOf(S, C, B), C, B); // outside
   CB
13 }  // inside ACB: dropped
```



|S| = 1: the farthest point **C = (4,25)** joins the hull. 0 points inside the triangle are discarded; 0 outside AC, 0 outside CB.

Step 4 / 15

CALL STACK

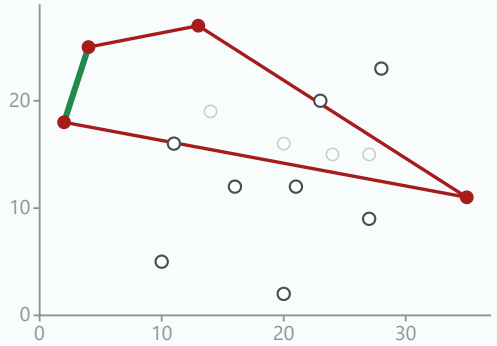
- `findHull(1 pts, (2,18)→(13,27))`
- `findHull(8 pts, (2,18)→(35,11))`
- `quickHull(P)`

VARIABLES

`A = (2,18)` `B = (13,27)` `C = (4,25)` `|S| = 1` `|hull| = 4`

Quickhull: Code Trace

```
1  void quickHull(List<Point> P) {
2    Point A = leftmost(P), B = rightmost(P);
3    hull.add(A); hull.add(B);
4    findHull(leftOf(P, A, B), A, B); // upper set
5    findHull(leftOf(P, B, A), B, A); // lower set
6  }
7  void findHull(List<Point> S, Point A, Point B)
   {
▶    if (S.isEmpty()) return; // AB: hull edge
9    Point C = farthest(S, A, B); // max cross
10   hull.add(C);
11   findHull(leftOf(S, A, C), A, C); // outside
   AC
12   findHull(leftOf(S, C, B), C, B); // outside
   CB
13 }  // inside ACB: dropped
```



S is empty → (2,18)–(4,25) is a **hull edge**; return.

Step 5 / 15

CALL STACK

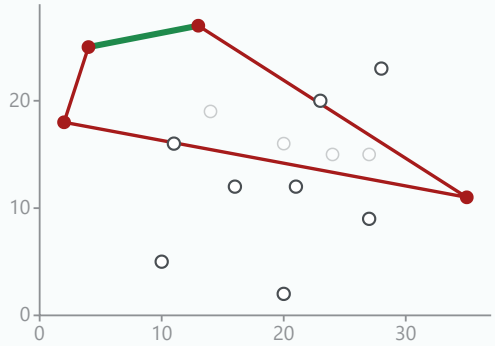
- `findHull(0 pts, (2,18)→(4,25))`
- `findHull(1 pts, (2,18)→(13,27))`
- `findHull(8 pts, (2,18)→(35,11))`
- `quickHull(P)`

VARIABLES

`A = (2,18)` `B = (4,25)` `C = –` `|S| = 0` `|hull| = 4`

Quickhull: Code Trace

```
1  void quickHull(List<Point> P) {
2    Point A = leftmost(P), B = rightmost(P);
3    hull.add(A); hull.add(B);
4    findHull(leftOf(P, A, B), A, B); // upper set
5    findHull(leftOf(P, B, A), B, A); // lower set
6  }
7  void findHull(List<Point> S, Point A, Point B)
   {
▶    if (S.isEmpty()) return; // AB: hull edge
9    Point C = farthest(S, A, B); // max cross
10   hull.add(C);
11   findHull(leftOf(S, A, C), A, C); // outside
   AC
12   findHull(leftOf(S, C, B), C, B); // outside
   CB
13 }  // inside ACB: dropped
```



S is empty → (4,25)–(13,27) is a **hull edge**; return.

Step 6 / 15

CALL STACK

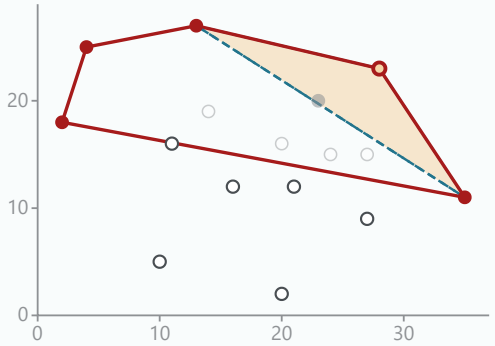
- **findHull(0 pts, (4,25)→(13,27))**
- findHull(1 pts, (2,18)→(13,27))
- findHull(8 pts, (2,18)→(35,11))
- quickHull(P)

VARIABLES

A = **(4,25)** B = **(13,27)** C = **–** |S| = **0** |hull| = **4**

Quickhull: Code Trace

```
1  void quickHull(List<Point> P) {
2    Point A = leftmost(P), B = rightmost(P);
3    hull.add(A); hull.add(B);
4    findHull(leftOf(P, A, B), A, B); // upper set
5    findHull(leftOf(P, B, A), B, A); // lower set
6  }
7  void findHull(List<Point> S, Point A, Point B)
   {
8    if (S.isEmpty()) return; // AB: hull edge
▶    Point C = farthest(S, A, B); // max cross
▶    hull.add(C);
11   findHull(leftOf(S, A, C), A, C); // outside
   AC
12   findHull(leftOf(S, C, B), C, B); // outside
   CB
13 }  // inside ACB: dropped
```



|S| = 2: the farthest point **C = (28,23)** joins the hull. 1 point inside the triangle is discarded; 0 outside AC, 0 outside CB.

Step 7 / 15

CALL STACK

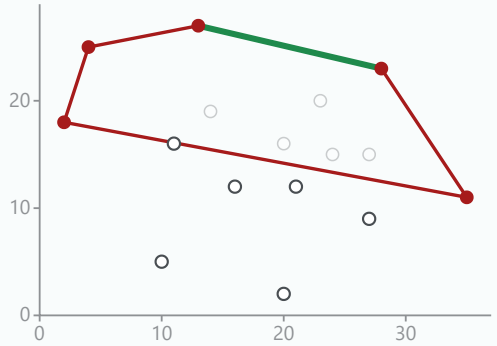
- `findHull(2 pts, (13,27)→(35,11))`
- `findHull(8 pts, (2,18)→(35,11))`
- `quickHull(P)`

VARIABLES

`A = (13,27)` `B = (35,11)` `C = (28,23)` `|S| = 2` `|hull| = 5`

Quickhull: Code Trace

```
1  void quickHull(List<Point> P) {
2    Point A = leftmost(P), B = rightmost(P);
3    hull.add(A); hull.add(B);
4    findHull(leftOf(P, A, B), A, B); // upper set
5    findHull(leftOf(P, B, A), B, A); // lower set
6  }
7  void findHull(List<Point> S, Point A, Point B)
   {
▶    if (S.isEmpty()) return; // AB: hull edge
9    Point C = farthest(S, A, B); // max cross
10   hull.add(C);
11   findHull(leftOf(S, A, C), A, C); // outside
   AC
12   findHull(leftOf(S, C, B), C, B); // outside
   CB
13 }  // inside ACB: dropped
```



S is empty → (13,27)–(28,23) is a **hull edge**; return.

Step 8 / 15

CALL STACK

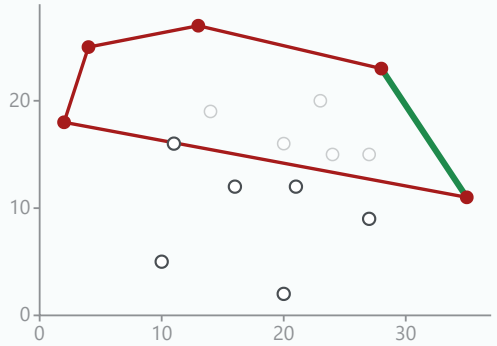
- `findHull(0 pts, (13,27)→(28,23))`
- `findHull(2 pts, (13,27)→(35,11))`
- `findHull(8 pts, (2,18)→(35,11))`
- `quickHull(P)`

VARIABLES

`A = (13,27)` `B = (28,23)` `C = –` `|S| = 0` `|hull| = 5`

Quickhull: Code Trace

```
 1 void quickHull(List<Point> P) {
 2   Point A = leftmost(P), B = rightmost(P);
 3   hull.add(A); hull.add(B);
 4   findHull(leftOf(P, A, B), A, B); // upper set
 5   findHull(leftOf(P, B, A), B, A); // lower set
 6 }
 7 void findHull(List<Point> S, Point A, Point B)
   {
▶  if (S.isEmpty()) return; // AB: hull edge
 9   Point C = farthest(S, A, B); // max cross
10   hull.add(C);
11   findHull(leftOf(S, A, C), A, C); // outside
   AC
12   findHull(leftOf(S, C, B), C, B); // outside
   CB
13 }  // inside ACB: dropped
```



S is empty → (28,23)–(35,11) is a **hull edge**; return.

Step 9 / 15

CALL STACK

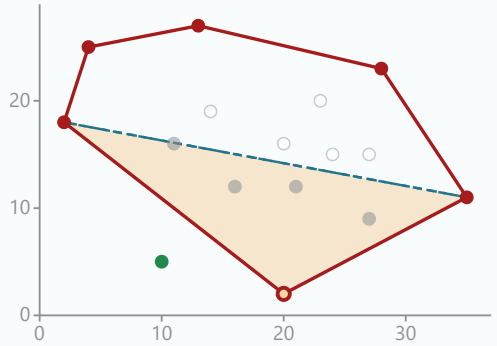
- `findHull(0 pts, (28,23)→(35,11))`
- `findHull(2 pts, (13,27)→(35,11))`
- `findHull(8 pts, (2,18)→(35,11))`
- `quickHull(P)`

VARIABLES

`A = (28,23)` `B = (35,11)` `C = –` `|S| = 0` `|hull| = 5`

Quickhull: Code Trace

```
1  void quickHull(List<Point> P) {
2    Point A = leftmost(P), B = rightmost(P);
3    hull.add(A); hull.add(B);
4    findHull(leftOf(P, A, B), A, B); // upper set
5    findHull(leftOf(P, B, A), B, A); // lower set
6  }
7  void findHull(List<Point> S, Point A, Point B)
   {
8    if (S.isEmpty()) return; // AB: hull edge
▶    Point C = farthest(S, A, B); // max cross
▶    hull.add(C);
11   findHull(leftOf(S, A, C), A, C); // outside
   AC
12   findHull(leftOf(S, C, B), C, B); // outside
   CB
13 }  // inside ACB: dropped
```



|S| = 6: the farthest point **C = (20,2)** joins the hull. 4 points inside the triangle are discarded; 0 outside AC, 1 outside CB.

Step 10 / 15

CALL STACK

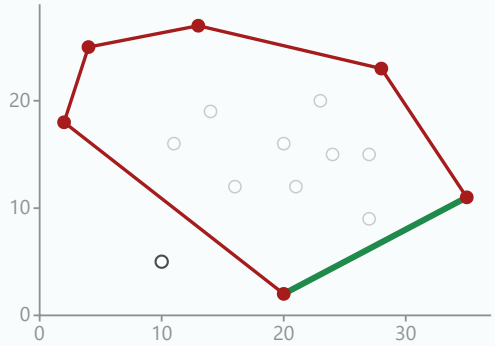
- `findHull(6 pts, (35,11)→(2,18))`
- `quickHull(P)`

VARIABLES

`A = (35,11)` `B = (2,18)` `C = (20,2)` `|S| = 6` `|hull| = 6`

Quickhull: Code Trace

```
 1 void quickHull(List<Point> P) {
 2   Point A = leftmost(P), B = rightmost(P);
 3   hull.add(A); hull.add(B);
 4   findHull(leftOf(P, A, B), A, B); // upper set
 5   findHull(leftOf(P, B, A), B, A); // lower set
 6 }
 7 void findHull(List<Point> S, Point A, Point B)
   {
▶  if (S.isEmpty()) return; // AB: hull edge
 9   Point C = farthest(S, A, B); // max cross
10   hull.add(C);
11   findHull(leftOf(S, A, C), A, C); // outside
   AC
12   findHull(leftOf(S, C, B), C, B); // outside
   CB
13 }  // inside ACB: dropped
```



S is empty → (35,11)–(20,2) is a **hull edge**; return.

Step 11 / 15

CALL STACK

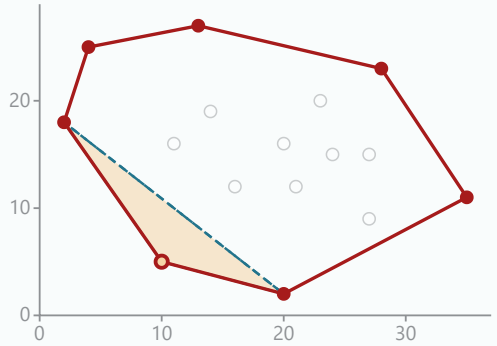
- `findHull(0 pts, (35,11)→(20,2))`
- `findHull(6 pts, (35,11)→(2,18))`
- `quickHull(P)`

VARIABLES

`A = (35,11)` `B = (20,2)` `C = –` `|S| = 0` `|hull| = 6`

Quickhull: Code Trace

```
1  void quickHull(List<Point> P) {
2    Point A = leftmost(P), B = rightmost(P);
3    hull.add(A); hull.add(B);
4    findHull(leftOf(P, A, B), A, B); // upper set
5    findHull(leftOf(P, B, A), B, A); // lower set
6  }
7  void findHull(List<Point> S, Point A, Point B)
   {
8    if (S.isEmpty()) return; // AB: hull edge
▶    Point C = farthest(S, A, B); // max cross
▶    hull.add(C);
11   findHull(leftOf(S, A, C), A, C); // outside
   AC
12   findHull(leftOf(S, C, B), C, B); // outside
   CB
13 }  // inside ACB: dropped
```



|S| = 1: the farthest point **C = (10,5)** joins the hull. 0 points inside the triangle are discarded; 0 outside AC, 0 outside CB.

Step 12 / 15

CALL STACK

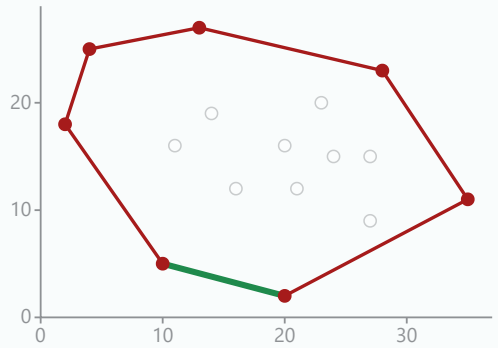
- `findHull(1 pts, (20,2)→(2,18))`
- `findHull(6 pts, (35,11)→(2,18))`
- `quickHull(P)`

VARIABLES

`A = (20,2)` `B = (2,18)` `C = (10,5)` `|S| = 1` `|hull| = 7`

Quickhull: Code Trace

```
 1 void quickHull(List<Point> P) {
 2   Point A = leftmost(P), B = rightmost(P);
 3   hull.add(A); hull.add(B);
 4   findHull(leftOf(P, A, B), A, B); // upper set
 5   findHull(leftOf(P, B, A), B, A); // lower set
 6 }
 7 void findHull(List<Point> S, Point A, Point B)
   {
▶    if (S.isEmpty()) return; // AB: hull edge
 9   Point C = farthest(S, A, B); // max cross
10   hull.add(C);
11   findHull(leftOf(S, A, C), A, C); // outside
   AC
12   findHull(leftOf(S, C, B), C, B); // outside
   CB
13 }  // inside ACB: dropped
```



S is empty → (20,2)–(10,5) is a **hull edge**; return.

Step 13 / 15

CALL STACK

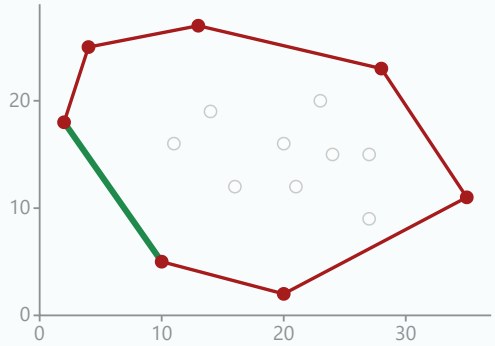
- `findHull(0 pts, (20,2)→(10,5))`
- `findHull(1 pts, (20,2)→(2,18))`
- `findHull(6 pts, (35,11)→(2,18))`
- `quickHull(P)`

VARIABLES

`A = (20,2)` `B = (10,5)` `C = –` `|S| = 0` `|hull| = 7`

Quickhull: Code Trace

```
 1 void quickHull(List<Point> P) {
 2   Point A = leftmost(P), B = rightmost(P);
 3   hull.add(A); hull.add(B);
 4   findHull(leftOf(P, A, B), A, B); // upper set
 5   findHull(leftOf(P, B, A), B, A); // lower set
 6 }
 7 void findHull(List<Point> S, Point A, Point B)
   {
▶  if (S.isEmpty()) return; // AB: hull edge
 9   Point C = farthest(S, A, B); // max cross
10   hull.add(C);
11   findHull(leftOf(S, A, C), A, C); // outside
   AC
12   findHull(leftOf(S, C, B), C, B); // outside
   CB
13 }  // inside ACB: dropped
```



S is empty → (10,5)–(2,18) is a **hull edge**; return.

Step 14 / 15

CALL STACK

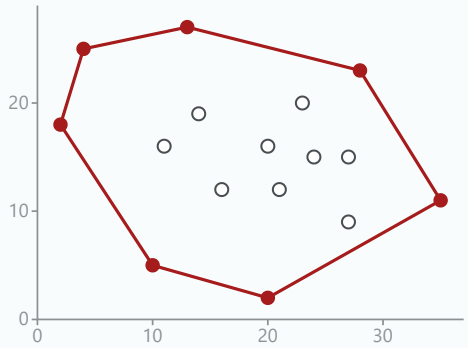
- `findHull(0 pts, (10,5)→(2,18))`
- `findHull(1 pts, (20,2)→(2,18))`
- `findHull(6 pts, (35,11)→(2,18))`
- `quickHull(P)`

VARIABLES

`A = (10,5)` `B = (2,18)` `C = –` `|S| = 0` `|hull| = 7`

Quickhull: Code Trace

```
1  void quickHull(List<Point> P) {
2    Point A = leftmost(P), B = rightmost(P);
3    hull.add(A); hull.add(B);
4    findHull(leftOf(P, A, B), A, B); // upper set
▶    findHull(leftOf(P, B, A), B, A); // lower set
6  }
7  void findHull(List<Point> S, Point A, Point B)
   {
8    if (S.isEmpty()) return; // AB: hull edge
9    Point C = farthest(S, A, B); // max cross
10   hull.add(C);
11   findHull(leftOf(S, A, C), A, C); // outside
   AC
12   findHull(leftOf(S, C, B), C, B); // outside
   CB
13 }  // inside ACB: dropped
```



Done: **7 vertices** found, 9 points discarded. 12 `findHull` calls in total.

Step 15 / 15

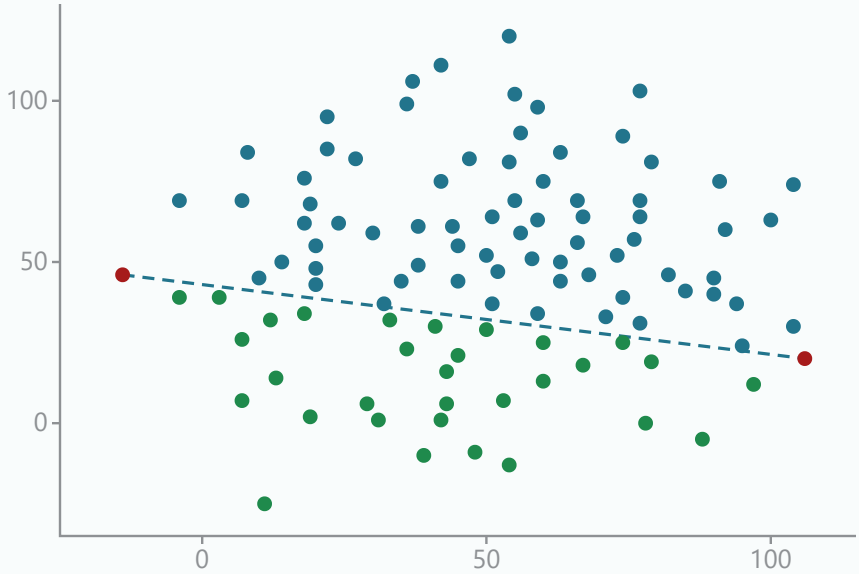
CALL STACK

`quickHull(P)`

VARIABLES

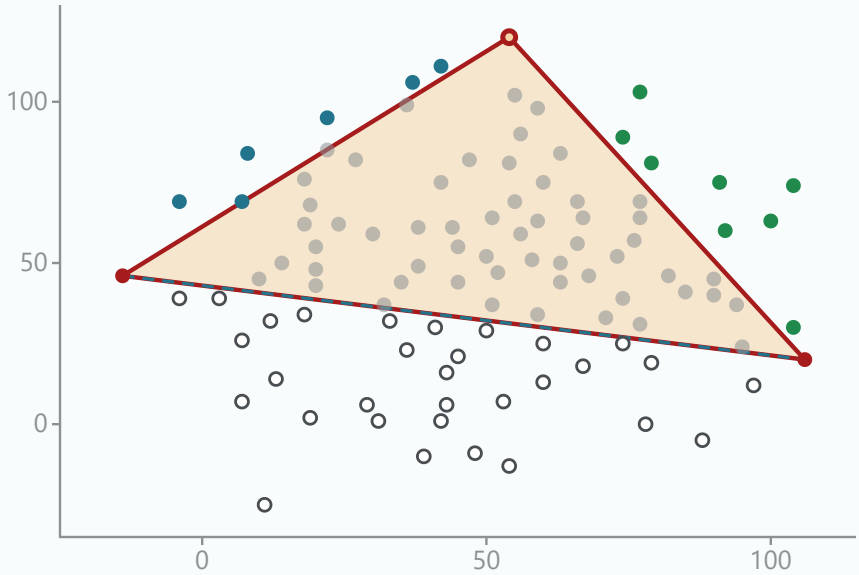
`A = (2,18)` `B = (35,11)` `C = –` `|S| = 0` `|hull| = 7`

Quickhull: A 104-Point Example



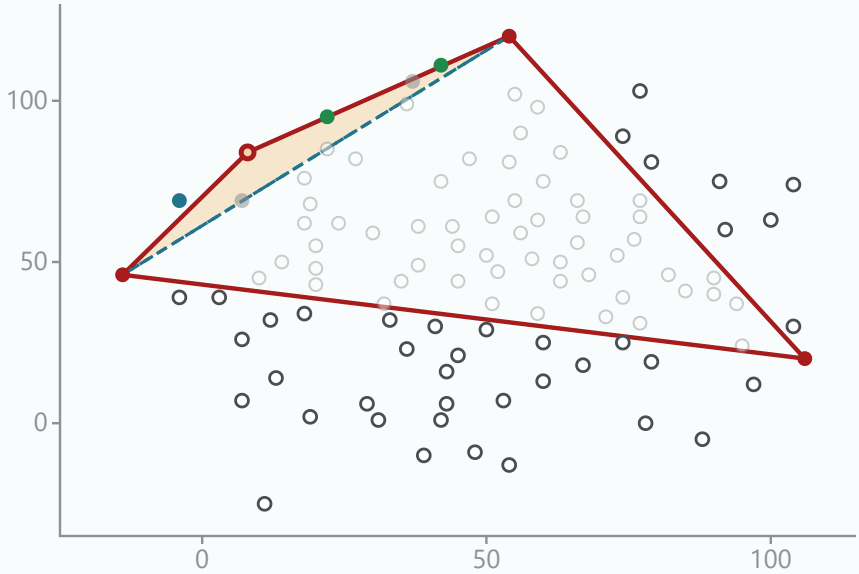
104 points. The leftmost (-14,46) and rightmost (106,20) are joined; 71 points lie above, 31 below.

Quickhull: A 104-Point Example



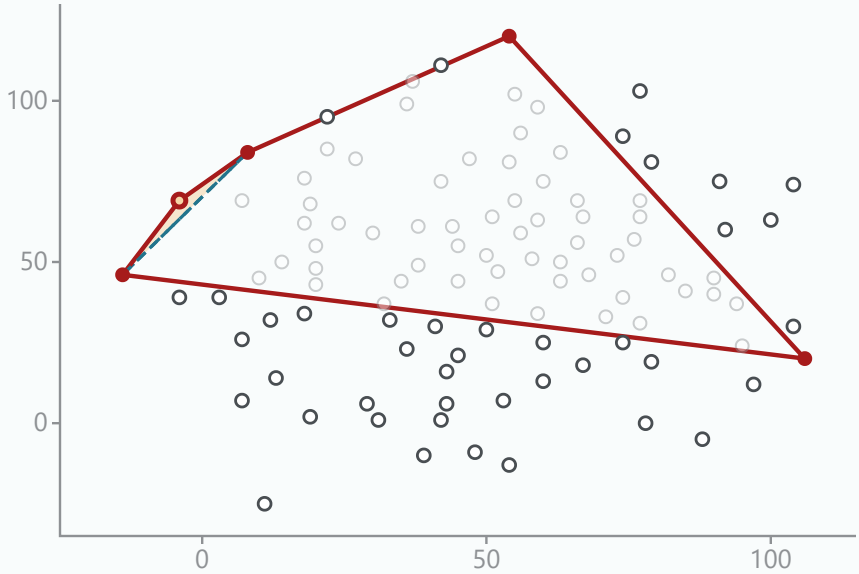
Farthest from (-14,46)→(106,20): **(54,120)**; **56** points inside the triangle discarded. Candidates left: 45.

Quickhull: A 104-Point Example



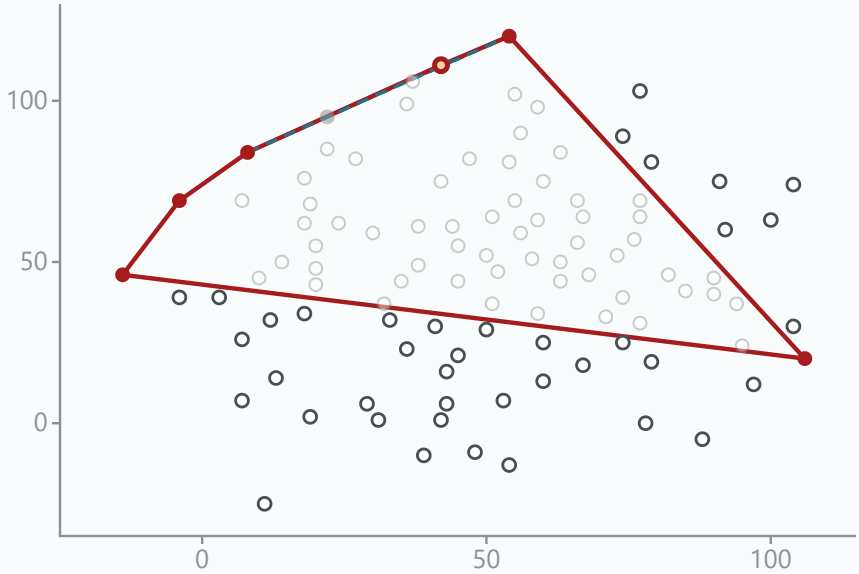
Farthest from (-14,46)→(54,120): **(8,84)**; **2** points inside the triangle discarded. Candidates left: 42.

Quickhull: A 104-Point Example



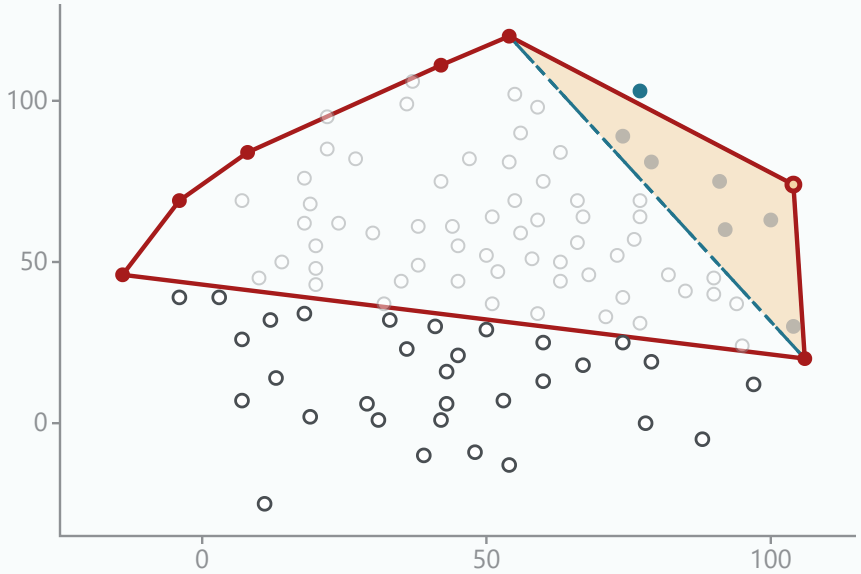
Farthest from (-14,46)→(8,84): **(-4,69)**; **0** points inside the triangle discarded. Candidates left: 41.

Quickhull: A 104-Point Example



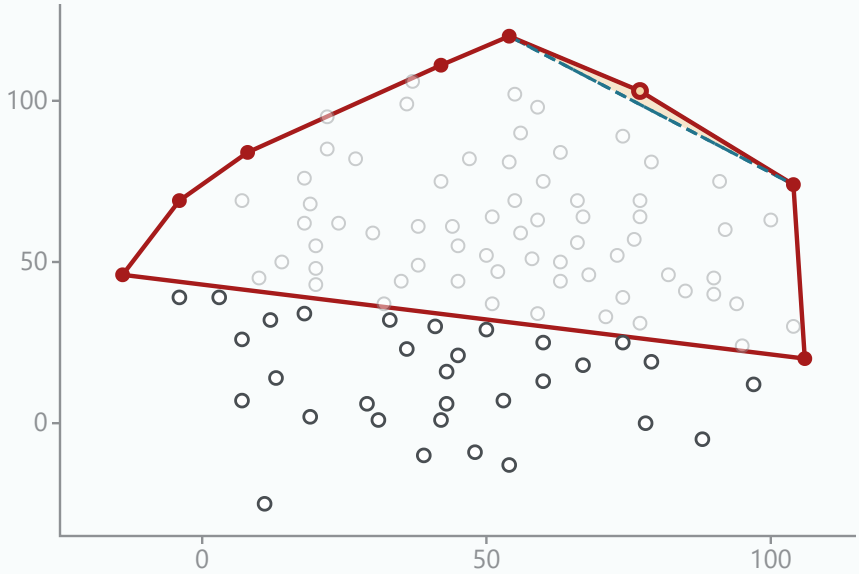
Farthest from (8,84)→(54,120): **(42,111)**; **1** point inside the triangle discarded. Candidates left: 39.

Quickhull: A 104-Point Example



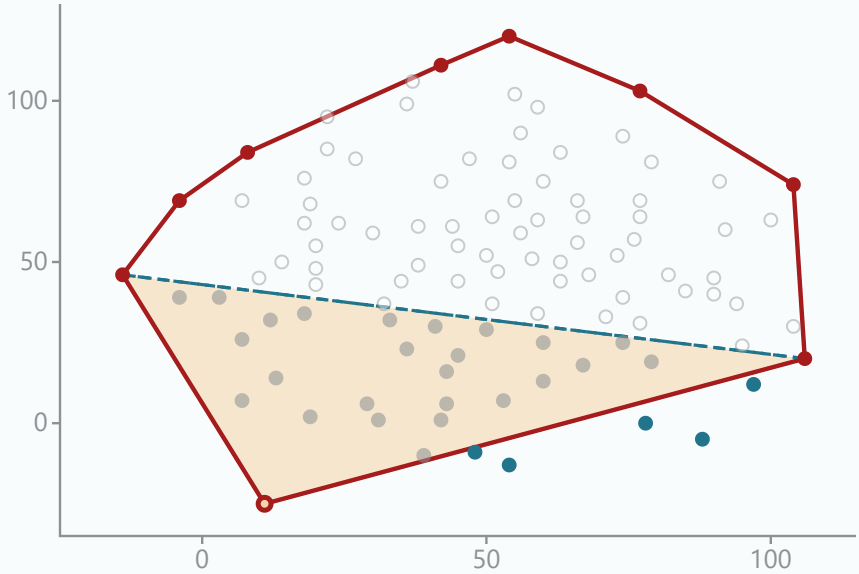
Farthest from (54,120)→(106,20): **(104,74)**; **6** points inside the triangle discarded. Candidates left: 32.

Quickhull: A 104-Point Example



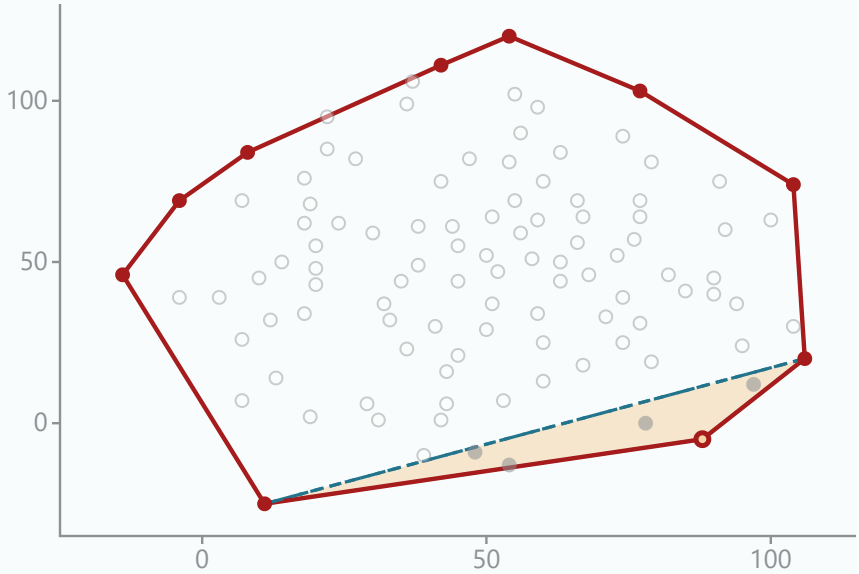
Farthest from (54,120)→(104,74): **(77,103)**; **0** points inside the triangle discarded. Candidates left: 31.

Quickhull: A 104-Point Example



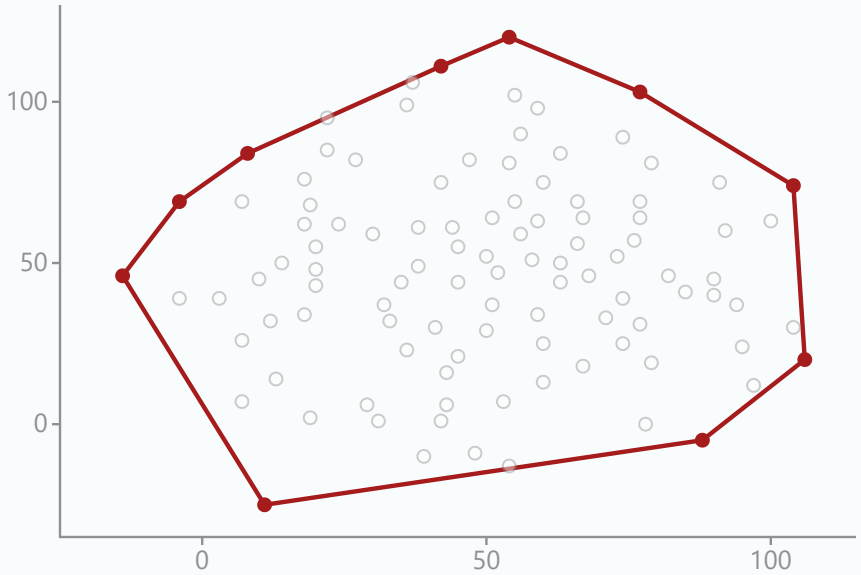
Farthest from (106,20)→(-14,46): **(11,-25)**; **25** points inside the triangle discarded. Candidates left: 5.

Quickhull: A 104-Point Example



Farthest from (106,20)→(11,-25): **(88,-5)**; **4** points inside the triangle discarded. Candidates left: 0.

Quickhull: A 104-Point Example



Done: the hull of 104 points has **10 vertices**. Most points are discarded at every step, so the work shrinks fast.

Quickhull Analysis and Other Hull Algorithms

- Each call scans S once (farthest point + two partitions): **O(|S|)**.
- $T(n) = T(n_1) + T(n_2) + O(n)$, $n_1 + n_2 \le n - 1$.
- **Average** (balanced splits, many points discarded): **O(n log n)**; often even faster for uniformly random points.
- **Worst** case: only one point is split off at each step → $T(n) = T(n - 1) + O(n) =$ **$O(n^2)$**.
- Output-sensitive bound: O(n·h) for a hull with h vertices.

Algorithm	Time
Brute force (every candidate edge)	$O(n^3)$
Jarvis march (gift wrapping)	O(n·h)
Graham scan	O(n log n)
Andrew's monotone chain	O(n log n)
Quickhull	avg. O(n log n), worst $O(n^2)$
Chan	O(n log h)

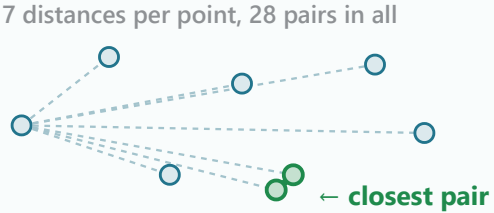
Lower bound: the hull can also sort (place the points on the parabola $y = x^2$), so **Ω(n log n)** time is needed in the comparison model.

Closest Pair of Points

Problem: among n points in the plane, find the pair with the smallest **Euclidean distance** $d(p, q) = \sqrt{((p_x - q_x)^2 + (p_y - q_y)^2)}$.

```
double bruteForce(List<Point> p) {
  double best = Double.MAX_VALUE;
  for (int i = 0; i < p.size(); i++)
    for (int j = i + 1; j < p.size(); j++)
      best = Math.min(best, dist(p.get(i), p.get(j)));
  return best;
}
```

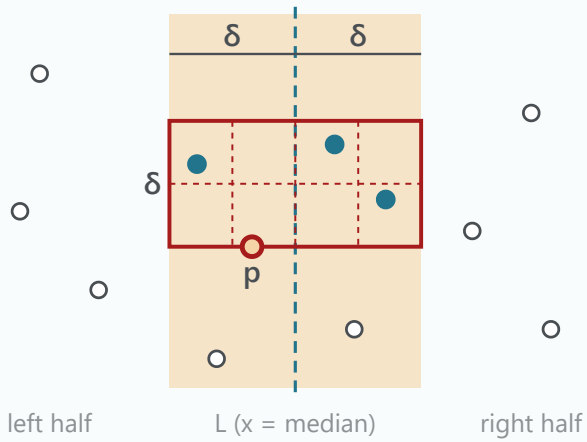
- **Brute force:** $n(n - 1)/2$ pairs → **$O(n^2)$**. For $n = 10^6$, $\approx 5 \cdot 10^{11}$ distances.
- Easy **in one dimension**: sort and compare only neighbors → O(n log n).
- In two dimensions, neighbors in x order are not enough; **divide and conquer** achieves O(n log n).



Applications: the two closest aircraft in air traffic control, clustering, collision detection, nearby facilities in geographic data.

Divide and Conquer: The Middle Strip

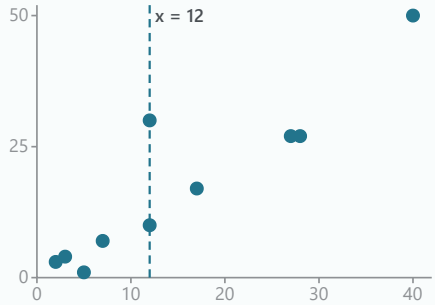
1. Sort the points once by **x** and once by **y**.
2. **Divide:** split into two halves with the vertical line L through the middle point.
3. **Conquer:** recursively find δ_L and δ_R in the halves; $\delta = \min(\delta_L, \delta_R)$.
4. **Combine:** a closer pair must cross L → check only the **strip** points with $|x - L| < \delta$.
5. Scan the strip in y order; compare each point only with the following points whose y differs by less than δ.



Why at most 7? Split the δ × 2δ rectangle above p into 8 squares of side δ/2. Two points in one square would lie in the same half at distance < δ, which is impossible → ≤ 1 point per square → p is compared with at most **7** points.

Closest Pair: Code Trace

```
1  double closest(List<Point> X, List<Point> Y) {
▶    int n = X.size(), mid = n / 2;
3    if (n <= 3) return bruteForce(X);
▶    Point m = X.get(mid);  // YL/YR: the halves of Y
5    double dL = closest(X.subList(0, mid), YL);
6    double dR = closest(X.subList(mid, n), YR);
7    double d = Math.min(dL, dR);
8    List<Point> s = new ArrayList<>();  // strip
9    for (Point p : Y)                   // in y order
10     if (Math.abs(p.x - m.x) < d) s.add(p);
11   for (int i = 0; i < s.size(); i++) {
12     Point p = s.get(i);
13     for (int j = i + 1; j < s.size()
14          && s.get(j).y - p.y < d; j++)
15       d = Math.min(d, dist(p, s.get(j)));
16   }
17   return d;
18 }
```



n = 10 > 3 → mid = 5, m = (12,10). Left: 5 points, right: 5.

Step 1 / 13

CALL STACK

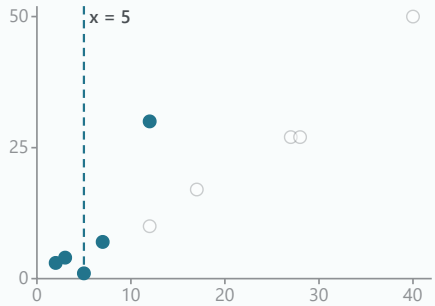
- `closest(X[0..9])`
- `main()`

VARIABLES

`n = 10` `m = (12,10)` `dL = ?` `dR = ?` `d = ?`

Closest Pair: Code Trace

```
1  double closest(List<Point> X, List<Point> Y) {
▶    int n = X.size(), mid = n / 2;
3    if (n <= 3) return bruteForce(X);
▶    Point m = X.get(mid);  // YL/YR: the halves of Y
5    double dL = closest(X.subList(0, mid), YL);
6    double dR = closest(X.subList(mid, n), YR);
7    double d = Math.min(dL, dR);
8    List<Point> s = new ArrayList<>();  // strip
9    for (Point p : Y)                   // in y order
10     if (Math.abs(p.x - m.x) < d) s.add(p);
11   for (int i = 0; i < s.size(); i++) {
12     Point p = s.get(i);
13     for (int j = i + 1; j < s.size()
14          && s.get(j).y - p.y < d; j++)
15       d = Math.min(d, dist(p, s.get(j)));
16   }
17   return d;
18 }
```



n = 5 > 3 → mid = 2, m = (5,1). Left: 2 points, right: 3.

Step 2 / 13

CALL STACK

- **closest(X[0..4])**
- closest(X[0..9])
- main()

VARIABLES

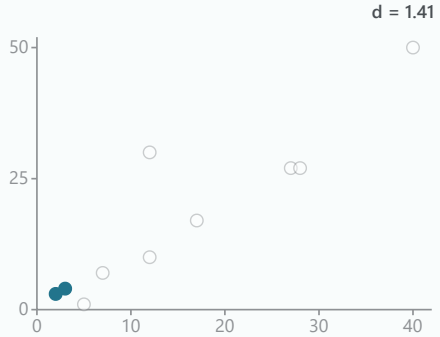
n = **5** | m = **(5,1)** | dL = **?** | dR = **?** | d = **?**

Closest Pair: Code Trace

```
1  double closest(List<Point> X, List<Point> Y) {
2    int n = X.size(), mid = n / 2;
▶    if (n <= 3) return bruteForce(X);
4    Point m = X.get(mid);  // YL/YR: the halves of Y
5    double dL = closest(X.subList(0, mid), YL);
6    double dR = closest(X.subList(mid, n), YR);
7    double d = Math.min(dL, dR);
8    List<Point> s = new ArrayList<>();  // strip
9    for (Point p : Y)                   // in y order
10     if (Math.abs(p.x - m.x) < d) s.add(p);
11   for (int i = 0; i < s.size(); i++) {
12     Point p = s.get(i);
13     for (int j = i + 1; j < s.size()
14          && s.get(j).y - p.y < d; j++)
15       d = Math.min(d, dist(p, s.get(j)));
16   }
17   return d;
18 }
```

CALL STACK

- **closest(X[0..1])**
- closest(X[0..4])
- closest(X[0..9])
- main()



n = 2 ≤ 3 → brute force: (2,3), (3,4). Closest (2,3)–(3,4), d = **1.41**.

Step 3 / 13

VARIABLES

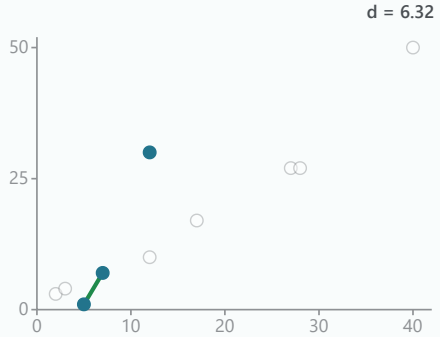
`n = 2` `m = –` `dL = –` `dR = –` `d = 1.41`

Closest Pair: Code Trace

```
1  double closest(List<Point> X, List<Point> Y) {
2    int n = X.size(), mid = n / 2;
▶    if (n <= 3) return bruteForce(X);
4    Point m = X.get(mid);  // YL/YR: the halves of Y
5    double dL = closest(X.subList(0, mid), YL);
6    double dR = closest(X.subList(mid, n), YR);
7    double d = Math.min(dL, dR);
8    List<Point> s = new ArrayList<>();  // strip
9    for (Point p : Y)                   // in y order
10     if (Math.abs(p.x - m.x) < d) s.add(p);
11   for (int i = 0; i < s.size(); i++) {
12     Point p = s.get(i);
13     for (int j = i + 1; j < s.size()
14          && s.get(j).y - p.y < d; j++)
15       d = Math.min(d, dist(p, s.get(j)));
16   }
17   return d;
18 }
```

CALL STACK

- **closest(X[2..4])**
- closest(X[0..4])
- closest(X[0..9])
- main()



n = 3 ≤ 3 → brute force: (5,1), (7,7), (12,30). Closest (5,1)–(7,7), d = **6.32**.

Step 4 / 13

VARIABLES

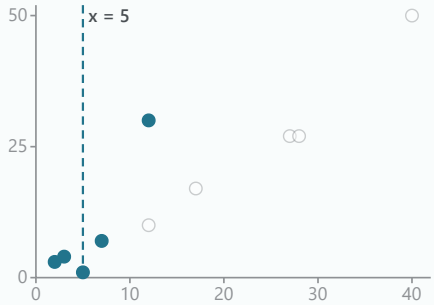
`n = 3` `m = –` `dL = –` `dR = –` `d = 6.32`

Closest Pair: Code Trace

```
1  double closest(List<Point> X, List<Point> Y) {
2    int n = X.size(), mid = n / 2;
3    if (n <= 3) return bruteForce(X);
4    Point m = X.get(mid);  // YL/YR: the halves of Y
5    double dL = closest(X.subList(0, mid), YL);
6    double dR = closest(X.subList(mid, n), YR);
▶    double d = Math.min(dL, dR);
8    List<Point> s = new ArrayList<>();  // strip
9    for (Point p : Y)                   // in y order
10     if (Math.abs(p.x - m.x) < d) s.add(p);
11   for (int i = 0; i < s.size(); i++) {
12     Point p = s.get(i);
13     for (int j = i + 1; j < s.size()
14          && s.get(j).y - p.y < d; j++)
15       d = Math.min(d, dist(p, s.get(j)));
16   }
17   return d;
18 }
```

CALL STACK

- `closest(X[0..4])`
- `closest(X[0..9])`
- `main()`



Both halves returned: dL = 1.41, dR = 6.32 → d = **1.41**.

Step 5 / 13

VARIABLES

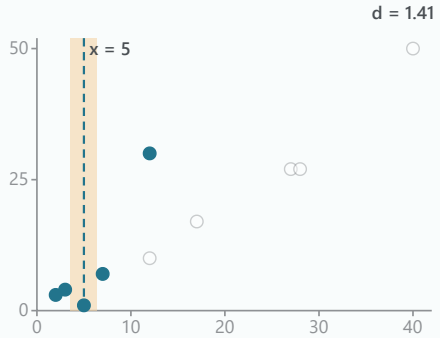
`n = 5` `m = (5,1)` `dL = 1.41` `dR = 6.32` `d = 1.41`

Closest Pair: Code Trace

```
 1 double closest(List<Point> X, List<Point> Y) {
 2   int n = X.size(), mid = n / 2;
 3   if (n <= 3) return bruteForce(X);
 4   Point m = X.get(mid);  // YL/YR: the halves of Y
 5   double dL = closest(X.subList(0, mid), YL);
 6   double dR = closest(X.subList(mid, n), YR);
 7   double d = Math.min(dL, dR);
 8   List<Point> s = new ArrayList<>();  // strip
 9   for (Point p : Y)                   // in y order
▶      if (Math.abs(p.x - m.x) < d) s.add(p);
11   for (int i = 0; i < s.size(); i++) {
12     Point p = s.get(i);
13     for (int j = i + 1; j < s.size()
14          && s.get(j).y - p.y < d; j++)
▶        d = Math.min(d, dist(p, s.get(j)));
16   }
▶    return d;
18 }
```

CALL STACK

- **closest(X[0..4])**
- closest(X[0..9])
- main()



Strip |x − 5| < 1.41: [(5,1)]; a single point → no comparisons. Returns **1.41**.

Step 6 / 13

VARIABLES

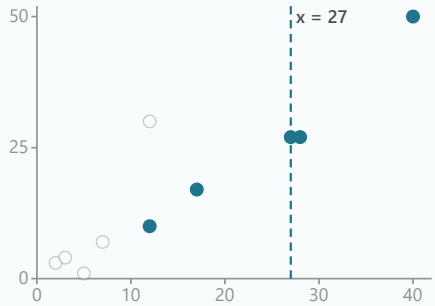
`n = 5` `m = (5,1)` `dL = 1.41` `dR = 6.32` `d = 1.41` `|strip| = 1`

Closest Pair: Code Trace

```
1  double closest(List<Point> X, List<Point> Y) {
▶    int n = X.size(), mid = n / 2;
3    if (n <= 3) return bruteForce(X);
▶    Point m = X.get(mid);  // YL/YR: the halves of Y
5    double dL = closest(X.subList(0, mid), YL);
6    double dR = closest(X.subList(mid, n), YR);
7    double d = Math.min(dL, dR);
8    List<Point> s = new ArrayList<>();  // strip
9    for (Point p : Y)                   // in y order
10     if (Math.abs(p.x - m.x) < d) s.add(p);
11   for (int i = 0; i < s.size(); i++) {
12     Point p = s.get(i);
13     for (int j = i + 1; j < s.size()
14          && s.get(j).y - p.y < d; j++)
15       d = Math.min(d, dist(p, s.get(j)));
16   }
17   return d;
18 }
```

CALL STACK

- **closest(X[5..9])**
- closest(X[0..9])
- main()



n = 5 > 3 → mid = 2, m = (27,27). Left: 2 points, right: 3.

Step 7 / 13

VARIABLES

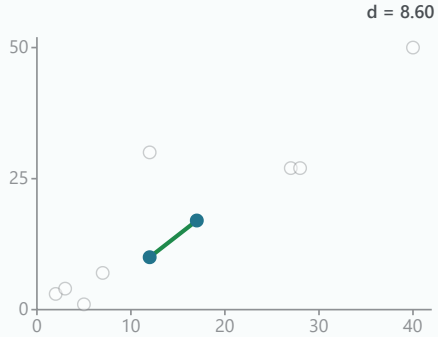
`n = 5` `m = (27,27)` `dL = ?` `dR = ?` `d = ?`

Closest Pair: Code Trace

```
1  double closest(List<Point> X, List<Point> Y) {
2    int n = X.size(), mid = n / 2;
▶    if (n <= 3) return bruteForce(X);
4    Point m = X.get(mid);  // YL/YR: the halves of Y
5    double dL = closest(X.subList(0, mid), YL);
6    double dR = closest(X.subList(mid, n), YR);
7    double d = Math.min(dL, dR);
8    List<Point> s = new ArrayList<>();  // strip
9    for (Point p : Y)                   // in y order
10     if (Math.abs(p.x - m.x) < d) s.add(p);
11   for (int i = 0; i < s.size(); i++) {
12     Point p = s.get(i);
13     for (int j = i + 1; j < s.size()
14          && s.get(j).y - p.y < d; j++)
15       d = Math.min(d, dist(p, s.get(j)));
16   }
17   return d;
18 }
```

CALL STACK

- **closest(X[5..6])**
- closest(X[5..9])
- closest(X[0..9])
- main()



n = 2 ≤ 3 → brute force: (12,10), (17,17). Closest (12,10)–(17,17), d = **8.60**.

Step 8 / 13

VARIABLES

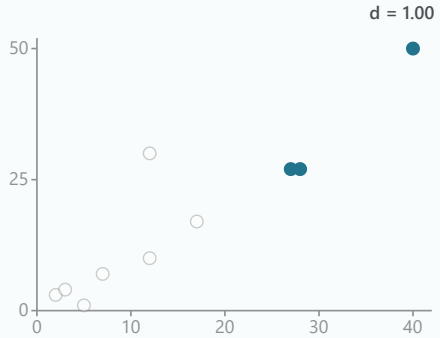
`n = 2` `m = –` `dL = –` `dR = –` `d = 8.60`

Closest Pair: Code Trace

```
1  double closest(List<Point> X, List<Point> Y) {
2    int n = X.size(), mid = n / 2;
▶    if (n <= 3) return bruteForce(X);
4    Point m = X.get(mid);  // YL/YR: the halves of Y
5    double dL = closest(X.subList(0, mid), YL);
6    double dR = closest(X.subList(mid, n), YR);
7    double d = Math.min(dL, dR);
8    List<Point> s = new ArrayList<>();  // strip
9    for (Point p : Y)                   // in y order
10     if (Math.abs(p.x - m.x) < d) s.add(p);
11   for (int i = 0; i < s.size(); i++) {
12     Point p = s.get(i);
13     for (int j = i + 1; j < s.size()
14          && s.get(j).y - p.y < d; j++)
15       d = Math.min(d, dist(p, s.get(j)));
16   }
17   return d;
18 }
```

CALL STACK

- **closest(X[7..9])**
- closest(X[5..9])
- closest(X[0..9])
- main()



n = 3 ≤ 3 → brute force: (27,27), (28,27), (40,50). Closest (27,27)–(28,27), d = **1.00**.

Step 9 / 13

VARIABLES

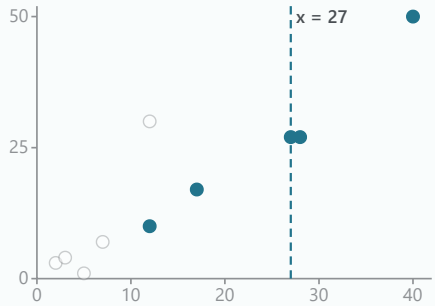
`n = 3` `m = –` `dL = –` `dR = –` `d = 1.00`

Closest Pair: Code Trace

```
1  double closest(List<Point> X, List<Point> Y) {
2    int n = X.size(), mid = n / 2;
3    if (n <= 3) return bruteForce(X);
4    Point m = X.get(mid);  // YL/YR: the halves of Y
5    double dL = closest(X.subList(0, mid), YL);
6    double dR = closest(X.subList(mid, n), YR);
▶    double d = Math.min(dL, dR);
8    List<Point> s = new ArrayList<>();  // strip
9    for (Point p : Y)                   // in y order
10     if (Math.abs(p.x - m.x) < d) s.add(p);
11   for (int i = 0; i < s.size(); i++) {
12     Point p = s.get(i);
13     for (int j = i + 1; j < s.size()
14          && s.get(j).y - p.y < d; j++)
15       d = Math.min(d, dist(p, s.get(j)));
16   }
17   return d;
18 }
```

CALL STACK

- `closest(X[5..9])`
- `closest(X[0..9])`
- `main()`



Both halves returned: dL = 8.60, dR = 1.00 → d = **1.00**.

Step 10 / 13

VARIABLES

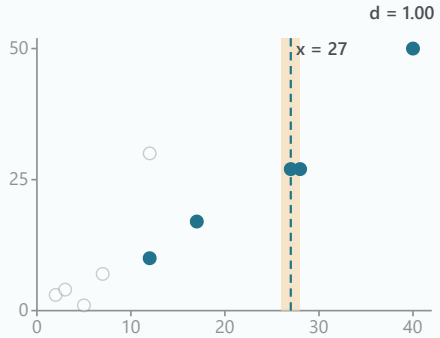
`n = 5` `m = (27,27)` `dL = 8.60` `dR = 1.00` `d = 1.00`

Closest Pair: Code Trace

```
 1 double closest(List<Point> X, List<Point> Y) {
 2   int n = X.size(), mid = n / 2;
 3   if (n <= 3) return bruteForce(X);
 4   Point m = X.get(mid);  // YL/YR: the halves of Y
 5   double dL = closest(X.subList(0, mid), YL);
 6   double dR = closest(X.subList(mid, n), YR);
 7   double d = Math.min(dL, dR);
 8   List<Point> s = new ArrayList<>();  // strip
 9   for (Point p : Y)                   // in y order
▶      if (Math.abs(p.x - m.x) < d) s.add(p);
11   for (int i = 0; i < s.size(); i++) {
12     Point p = s.get(i);
13     for (int j = i + 1; j < s.size()
14          && s.get(j).y - p.y < d; j++)
▶        d = Math.min(d, dist(p, s.get(j)));
16   }
▶    return d;
18 }
```

CALL STACK

- `closest(X[5..9])`
- `closest(X[0..9])`
- `main()`



Strip |x − 27| < 1.00: [(27,27)]; a single point → no comparisons. Returns **1.00**.

Step 11 / 13

VARIABLES

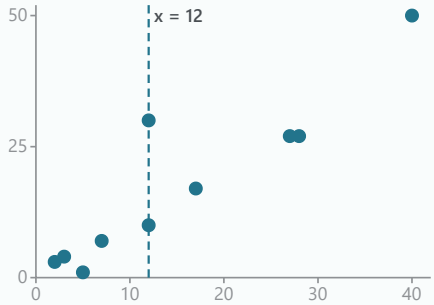
`n = 5` `m = (27,27)` `dL = 8.60` `dR = 1.00` `d = 1.00` `|strip| = 1`

Closest Pair: Code Trace

```
1  double closest(List<Point> X, List<Point> Y) {
2    int n = X.size(), mid = n / 2;
3    if (n <= 3) return bruteForce(X);
4    Point m = X.get(mid);  // YL/YR: the halves of Y
5    double dL = closest(X.subList(0, mid), YL);
6    double dR = closest(X.subList(mid, n), YR);
▶    double d = Math.min(dL, dR);
8    List<Point> s = new ArrayList<>();  // strip
9    for (Point p : Y)                   // in y order
10     if (Math.abs(p.x - m.x) < d) s.add(p);
11   for (int i = 0; i < s.size(); i++) {
12     Point p = s.get(i);
13     for (int j = i + 1; j < s.size()
14          && s.get(j).y - p.y < d; j++)
15       d = Math.min(d, dist(p, s.get(j)));
16   }
17   return d;
18 }
```

CALL STACK

- `closest(X[0..9])`
- `main()`



Both halves returned: dL = 1.41, dR = 1.00 → d = **1.00**.

Step 12 / 13

VARIABLES

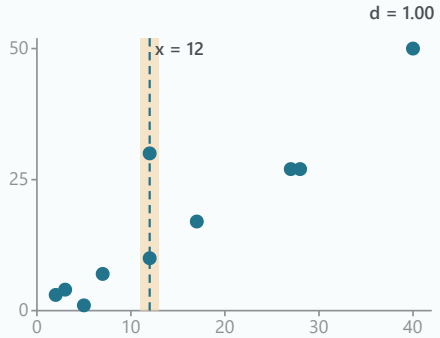
`n = 10` `m = (12,10)` `dL = 1.41` `dR = 1.00` `d = 1.00`

Closest Pair: Code Trace

```
 1 double closest(List<Point> X, List<Point> Y) {
 2   int n = X.size(), mid = n / 2;
 3   if (n <= 3) return bruteForce(X);
 4   Point m = X.get(mid);  // YL/YR: the halves of Y
 5   double dL = closest(X.subList(0, mid), YL);
 6   double dR = closest(X.subList(mid, n), YR);
 7   double d = Math.min(dL, dR);
 8   List<Point> s = new ArrayList<>();  // strip
 9   for (Point p : Y)                   // in y order
▶      if (Math.abs(p.x - m.x) < d) s.add(p);
11   for (int i = 0; i < s.size(); i++) {
12     Point p = s.get(i);
13     for (int j = i + 1; j < s.size()
14          && s.get(j).y - p.y < d; j++)
▶        d = Math.min(d, dist(p, s.get(j)));
16   }
▶    return d;
18 }
```

CALL STACK

- `closest(X[0..9])`
- `main()`



Strip |x − 12| < 1.00: [(12,10), (12,30)]; the y gap exceeds d → **no comparisons at all**. Returns **1.00**.

Step 13 / 13

VARIABLES

`n = 10` `m = (12,10)` `dL = 1.41` `dR = 1.00` `d = 1.00` `|strip| = 2`

Closest Pair: Analysis

Step	Cost	Note
Presort by x and by y	O(n log n)	only once, before the recursion
Divide (X in half, Y into YL/YR keeping the order)	O(n)	no re-sorting
Build the strip	O(n)	Y is already sorted by y
Scan the strip	O(7n) = O(n)	≤ 7 neighbors per point
Total	**O(n log n)**	T(n) = 2T(n/2) + O(n)

Common pitfall

Re-sorting the strip in every call: T(n) = 2T(n/2) + O(n log n) = **$O(n \log^2 n)$**.

In the example

the closest pair of the 10 points is **(27,27)–(28,27)**, d = 1.00. Brute force would compute 45 distances.

Karatsuba: 3 Multiplications Instead of 4

Split the n-digit numbers x, y at m = ⌊n/2⌋: **$x = a \cdot 10^m + b$**, **$y = c \cdot 10^m + d$**.

Naive divide and conquer

$x \cdot y = ac \cdot 10^{2m} + (ad + bc) \cdot 10^m + bd$

4 products: ac, ad, bc, bd

$T(n) = 4T(n/2) + O(n) =$ **$O(n^2)$** → no gain!

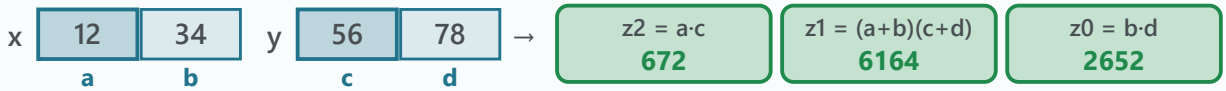
Karatsuba (1962)

$(a + b)(c + d) = ac +$ **$(ad + bc)$** $+ bd$

$\Rightarrow ad + bc = (a + b)(c + d) - ac - bd$

3 products: z2 = ac, z0 = bd, z1 = (a + b)(c + d)

$T(n) = 3T(n/2) + O(n) =$ **$O(n^{1.585})$**



Example: 1234 · 5678 ($m = 2$, $10^m = 100$)	
a = 12, b = 34, c = 56, d = 78	z2 = 12·56 = **672** · z0 = 34·78 = **2652**
z1 = (12 + 34)(56 + 78) = 46·134 = **6164**	z1 − z2 − z0 = 6164 − 672 − 2652 = **2840** (= ad + bc)
$672 \cdot 10^4 + 2840 \cdot 10^2 + 2652 = 6{,}720{,}000 + 284{,}000 + 2652 =$ **7,006,652** ✓	

Karatsuba: Tracing 1234 · 5678

1234·5678

karatsuba(1234, 5678): n = 4, p = 100 → a = 12, b = 34, c = 56, d = 78. Three calls: (34, 78), (46, 134), (12, 56).

Step 1 / 21

```
1  long karatsuba(long x, long y) {
2    if (x < 10 || y < 10) return x * y; // base case
▶    int n = Math.max(digits(x), digits(y));
▶    long p = (long) Math.pow(10, n / 2);
▶    long a = x / p, b = x % p;   // x = a·p + b
▶    long c = y / p, d = y % p;   // y = c·p + d
7    long z0 = karatsuba(b, d);
8    long z1 = karatsuba(a + b, c + d);
9    long z2 = karatsuba(a, c);
10   return z2 * p * p + (z1 - z2 - z0) * p + z0;
11 }
```

VARIABLES

x = 1234 | y = 5678 | n = 4 | p = 100 | a = 12 | b = 34 | c = 56 | d = 78

CALL STACK

karatsuba(1234, 5678)

main()

Karatsuba: Tracing 1234 · 5678



karatsuba(34, 78): n = 2, p = 10 → a = 3, b = 4, c = 7, d = 8. Three calls: (4, 8), (7, 15), (3, 7).

Step 2 / 21

```
 1 long karatsuba(long x, long y) {
 2   if (x < 10 || y < 10) return x * y; // base case
 ▶   int n = Math.max(digits(x), digits(y));
 ▶   long p = (long) Math.pow(10, n / 2);
 ▶   long a = x / p, b = x % p;   // x = a·p + b
 ▶   long c = y / p, d = y % p;   // y = c·p + d
 7   long z0 = karatsuba(b, d);
 8   long z1 = karatsuba(a + b, c + d);
 9   long z2 = karatsuba(a, c);
10   return z2 * p * p + (z1 - z2 - z0) * p + z0;
11 }
```

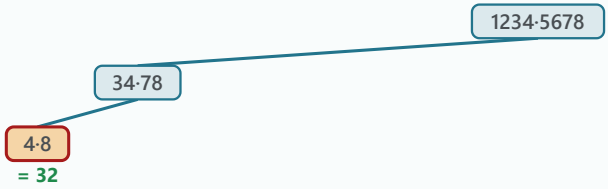
VARIABLES

x = 34 · y = 78 · n = 2 · p = 10 · a = 3 · b = 4 · c = 7 · d = 8

CALL STACK

- karatsuba(34, 78)
- karatsuba(1234, 5678)
- main()

Karatsuba: Tracing 1234 · 5678



4 · 8: a one-digit factor → **base case**, directly 32.

Step 3 / 21

```
1  long karatsuba(long x, long y) {
▶    if (x < 10 || y < 10) return x * y; // base case
3    int n = Math.max(digits(x), digits(y));
4    long p = (long) Math.pow(10, n / 2);
5    long a = x / p, b = x % p;   // x = a·p + b
6    long c = y / p, d = y % p;   // y = c·p + d
7    long z0 = karatsuba(b, d);
8    long z1 = karatsuba(a + b, c + d);
9    long z2 = karatsuba(a, c);
10   return z2 * p * p + (z1 - z2 - z0) * p + z0;
11 }
```

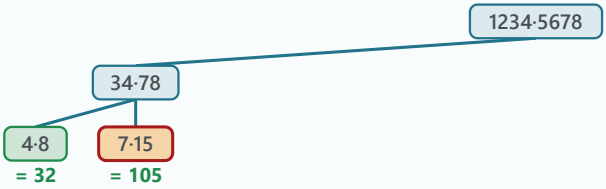
VARIABLES

`x = 4` `y = 8`

CALL STACK

- `karatsuba(4, 8)`
- `karatsuba(34, 78)`
- `karatsuba(1234, 5678)`
- `main()`

Karatsuba: Tracing 1234 · 5678



7 · 15: a one-digit factor → **base case**, directly 105.

Step 4 / 21

```
 1 long karatsuba(long x, long y) {
 2   if (x < 10 || y < 10) return x * y; // base case
 3   int n = Math.max(digits(x), digits(y));
 4   long p = (long) Math.pow(10, n / 2);
 5   long a = x / p, b = x % p;   // x = a·p + b
 6   long c = y / p, d = y % p;   // y = c·p + d
 7   long z0 = karatsuba(b, d);
 8   long z1 = karatsuba(a + b, c + d);
 9   long z2 = karatsuba(a, c);
10   return z2 * p * p + (z1 - z2 - z0) * p + z0;
11 }
```

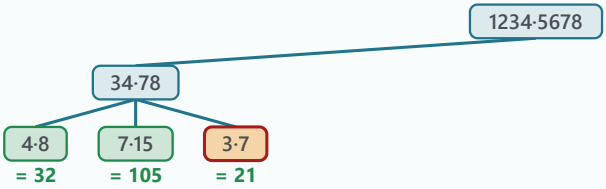
VARIABLES

x = 7 y = 15

CALL STACK

- karatsuba(7, 15)
- karatsuba(34, 78)
- karatsuba(1234, 5678)
- main()

Karatsuba: Tracing 1234 · 5678



3 · 7: a one-digit factor → **base case**, directly 21.

Step 5 / 21

```
1  long karatsuba(long x, long y) {
▶    if (x < 10 || y < 10) return x * y; // base case
3    int n = Math.max(digits(x), digits(y));
4    long p = (long) Math.pow(10, n / 2);
5    long a = x / p, b = x % p;   // x = a·p + b
6    long c = y / p, d = y % p;   // y = c·p + d
7    long z0 = karatsuba(b, d);
8    long z1 = karatsuba(a + b, c + d);
9    long z2 = karatsuba(a, c);
10   return z2 * p * p + (z1 - z2 - z0) * p + z0;
11 }
```

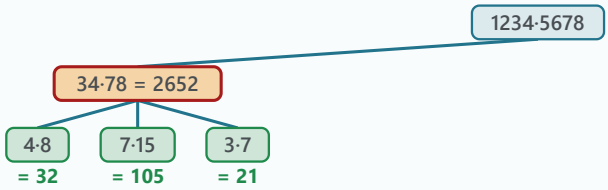
VARIABLES

`x = 3` `y = 7`

CALL STACK

- `karatsuba(3, 7)`
- `karatsuba(34, 78)`
- `karatsuba(1234, 5678)`
- `main()`

Karatsuba: Tracing 1234 · 5678



21·100 + (105 − 21 − 32)·10 + 32 = 2100 + 520 + 32 = **2652**.

Step 6 / 21

```
1  long karatsuba(long x, long y) {
2    if (x < 10 || y < 10) return x * y; // base case
3    int n = Math.max(digits(x), digits(y));
4    long p = (long) Math.pow(10, n / 2);
5    long a = x / p, b = x % p;   // x = a·p + b
6    long c = y / p, d = y % p;   // y = c·p + d
7    long z0 = karatsuba(b, d);
8    long z1 = karatsuba(a + b, c + d);
9    long z2 = karatsuba(a, c);
▶    return z2 * p * p + (z1 - z2 - z0) * p + z0;
11 }
```

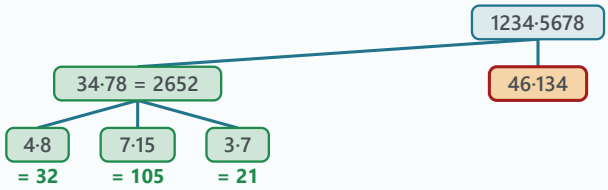
VARIABLES

`x = 34` `y = 78` `p = 10` `z0 = 32` `z1 = 105` `z2 = 21`

CALL STACK

- `karatsuba(34, 78)`
- `karatsuba(1234, 5678)`
- `main()`

Karatsuba: Tracing 1234 · 5678



karatsuba(46, 134): n = 3, p = 10 → a = 4, b = 6, c = 13, d = 4. Three calls: (6, 4), (10, 17), (4, 13).

Step 7 / 21

```
1  long karatsuba(long x, long y) {
2    if (x < 10 || y < 10) return x * y; // base case
▶    int n = Math.max(digits(x), digits(y));
▶    long p = (long) Math.pow(10, n / 2);
▶    long a = x / p, b = x % p;   // x = a·p + b
▶    long c = y / p, d = y % p;   // y = c·p + d
7    long z0 = karatsuba(b, d);
8    long z1 = karatsuba(a + b, c + d);
9    long z2 = karatsuba(a, c);
10   return z2 * p * p + (z1 - z2 - z0) * p + z0;
11 }
```

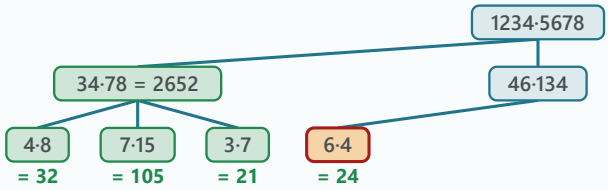
VARIABLES

x = 46 · y = 134 · n = 3 · p = 10 · a = 4 · b = 6 · c = 13 · d = 4

CALL STACK

- karatsuba(46, 134)
- karatsuba(1234, 5678)
- main()

Karatsuba: Tracing 1234 · 5678



6 · 4: a one-digit factor → **base case**, directly 24.

Step 8 / 21

```
1  long karatsuba(long x, long y) {
▶    if (x < 10 || y < 10) return x * y; // base case
3    int n = Math.max(digits(x), digits(y));
4    long p = (long) Math.pow(10, n / 2);
5    long a = x / p, b = x % p;   // x = a·p + b
6    long c = y / p, d = y % p;   // y = c·p + d
7    long z0 = karatsuba(b, d);
8    long z1 = karatsuba(a + b, c + d);
9    long z2 = karatsuba(a, c);
10   return z2 * p * p + (z1 - z2 - z0) * p + z0;
11 }
```

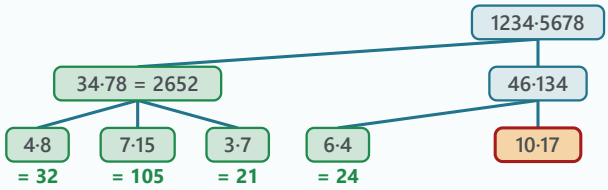
VARIABLES

x = 6 y = 4

CALL STACK

- karatsuba(6, 4)
- karatsuba(46, 134)
- karatsuba(1234, 5678)
- main()

Karatsuba: Tracing 1234 · 5678



karatsuba(10, 17): n = 2, p = 10 → a = 1, b = 0, c = 1, d = 7. Three calls: (0, 7), (1, 8), (1, 1).

Step 9 / 21

```
1  long karatsuba(long x, long y) {
2    if (x < 10 || y < 10) return x * y; // base case
▶    int n = Math.max(digits(x), digits(y));
▶    long p = (long) Math.pow(10, n / 2);
▶    long a = x / p, b = x % p;   // x = a·p + b
▶    long c = y / p, d = y % p;   // y = c·p + d
7    long z0 = karatsuba(b, d);
8    long z1 = karatsuba(a + b, c + d);
9    long z2 = karatsuba(a, c);
10   return z2 * p * p + (z1 - z2 - z0) * p + z0;
11 }
```

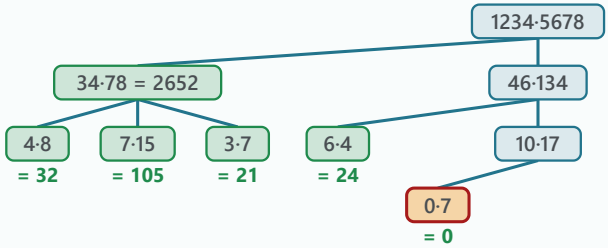
VARIABLES

x = 10 | y = 17 | n = 2 | p = 10 | a = 1 | b = 0 | c = 1 | d = 7

CALL STACK

- karatsuba(10, 17)
- karatsuba(46, 134)
- karatsuba(1234, 5678)
- main()

Karatsuba: Tracing 1234 · 5678



0 · 7: a one-digit factor → **base case**, directly 0.

Step 10 / 21

```
1  long karatsuba(long x, long y) {
▶    if (x < 10 || y < 10) return x * y; // base case
3    int n = Math.max(digits(x), digits(y));
4    long p = (long) Math.pow(10, n / 2);
5    long a = x / p, b = x % p;   // x = a·p + b
6    long c = y / p, d = y % p;   // y = c·p + d
7    long z0 = karatsuba(b, d);
8    long z1 = karatsuba(a + b, c + d);
9    long z2 = karatsuba(a, c);
10   return z2 * p * p + (z1 - z2 - z0) * p + z0;
11 }
```

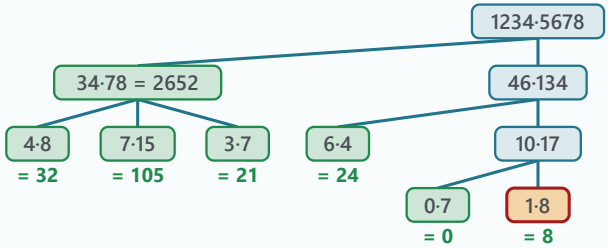
VARIABLES

`x = 0` `y = 7`

CALL STACK

- `karatsuba(0, 7)`
- `karatsuba(10, 17)`
- `karatsuba(46, 134)`
- `karatsuba(1234, 5678)`
- `main()`

Karatsuba: Tracing 1234 · 5678



1 · 8: a one-digit factor → **base case**, directly 8.

Step 11 / 21

```
1  long karatsuba(long x, long y) {
▶    if (x < 10 || y < 10) return x * y; // base case
3    int n = Math.max(digits(x), digits(y));
4    long p = (long) Math.pow(10, n / 2);
5    long a = x / p, b = x % p;   // x = a·p + b
6    long c = y / p, d = y % p;   // y = c·p + d
7    long z0 = karatsuba(b, d);
8    long z1 = karatsuba(a + b, c + d);
9    long z2 = karatsuba(a, c);
10   return z2 * p * p + (z1 - z2 - z0) * p + z0;
11 }
```

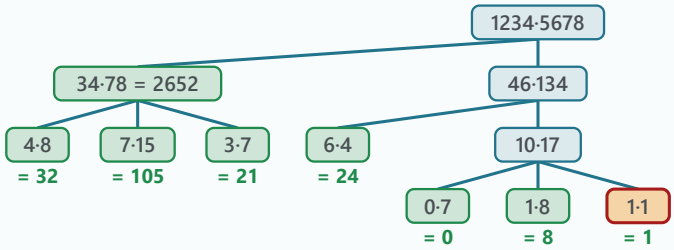
VARIABLES

x = 1 y = 8

CALL STACK

- karatsuba(1, 8)
- karatsuba(10, 17)
- karatsuba(46, 134)
- karatsuba(1234, 5678)
- main()

Karatsuba: Tracing 1234 · 5678



1 · 1: a one-digit factor → **base case**, directly 1.

Step 12 / 21

```
1  long karatsuba(long x, long y) {
▶    if (x < 10 || y < 10) return x * y; // base case
3    int n = Math.max(digits(x), digits(y));
4    long p = (long) Math.pow(10, n / 2);
5    long a = x / p, b = x % p;   // x = a·p + b
6    long c = y / p, d = y % p;   // y = c·p + d
7    long z0 = karatsuba(b, d);
8    long z1 = karatsuba(a + b, c + d);
9    long z2 = karatsuba(a, c);
10   return z2 * p * p + (z1 - z2 - z0) * p + z0;
11 }
```

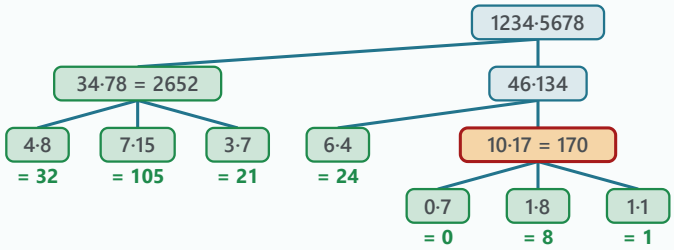
VARIABLES

x = 1 y = 1

CALL STACK

- karatsuba(1, 1)
- karatsuba(10, 17)
- karatsuba(46, 134)
- karatsuba(1234, 5678)
- main()

Karatsuba: Tracing 1234 · 5678



1·100 + (8 − 1 − 0)·10 + 0 = 100 + 70 + 0 = **170**.

Step 13 / 21

```
1  long karatsuba(long x, long y) {
2    if (x < 10 || y < 10) return x * y; // base case
3    int n = Math.max(digits(x), digits(y));
4    long p = (long) Math.pow(10, n / 2);
5    long a = x / p, b = x % p;   // x = a·p + b
6    long c = y / p, d = y % p;   // y = c·p + d
7    long z0 = karatsuba(b, d);
8    long z1 = karatsuba(a + b, c + d);
9    long z2 = karatsuba(a, c);
▶    return z2 * p * p + (z1 - z2 - z0) * p + z0;
11 }
```

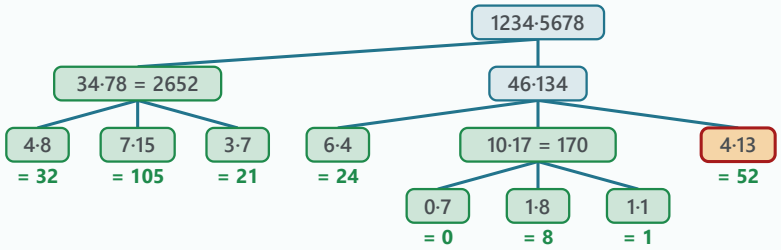
VARIABLES

`x = 10` `y = 17` `p = 10` `z0 = 0` `z1 = 8` `z2 = 1`

CALL STACK

- `karatsuba(10, 17)`
- `karatsuba(46, 134)`
- `karatsuba(1234, 5678)`
- `main()`

Karatsuba: Tracing 1234 · 5678



4 · 13: a one-digit factor → **base case**, directly 52.

Step 14 / 21

```
 1 long karatsuba(long x, long y) {
 2   if (x < 10 || y < 10) return x * y; // base case
 3   int n = Math.max(digits(x), digits(y));
 4   long p = (long) Math.pow(10, n / 2);
 5   long a = x / p, b = x % p;   // x = a·p + b
 6   long c = y / p, d = y % p;   // y = c·p + d
 7   long z0 = karatsuba(b, d);
 8   long z1 = karatsuba(a + b, c + d);
 9   long z2 = karatsuba(a, c);
10   return z2 * p * p + (z1 - z2 - z0) * p + z0;
11 }
```

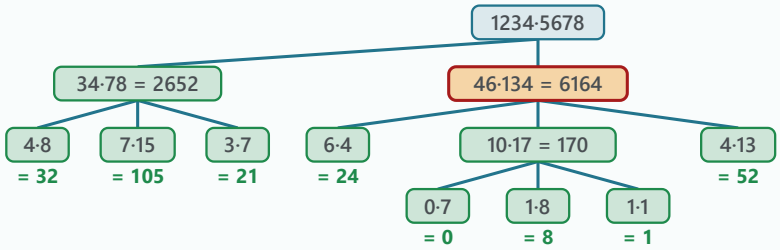
VARIABLES

x = 4 y = 13

CALL STACK

- karatsuba(4, 13)
- karatsuba(46, 134)
- karatsuba(1234, 5678)
- main()

Karatsuba: Tracing 1234 · 5678



52·100 + (170 − 52 − 24)·10 + 24 = 5200 + 940 + 24 = **6164**.

Step 15 / 21

```
1  long karatsuba(long x, long y) {
2    if (x < 10 || y < 10) return x * y; // base case
3    int n = Math.max(digits(x), digits(y));
4    long p = (long) Math.pow(10, n / 2);
5    long a = x / p, b = x % p;   // x = a·p + b
6    long c = y / p, d = y % p;   // y = c·p + d
7    long z0 = karatsuba(b, d);
8    long z1 = karatsuba(a + b, c + d);
9    long z2 = karatsuba(a, c);
▶    return z2 * p * p + (z1 - z2 - z0) * p + z0;
11 }
```

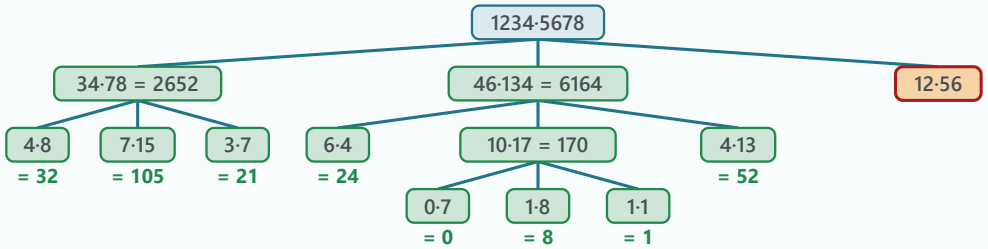
VARIABLES

`x = 46` `y = 134` `p = 10` `z0 = 24` `z1 = 170` `z2 = 52`

CALL STACK

- `karatsuba(46, 134)`
- `karatsuba(1234, 5678)`
- `main()`

Karatsuba: Tracing 1234 · 5678



karatsuba(12, 56): n = 2, p = 10 → a = 1, b = 2, c = 5, d = 6. Three calls: (2, 6), (3, 11), (1, 5).

Step 16 / 21

```
1  long karatsuba(long x, long y) {
2    if (x < 10 || y < 10) return x * y; // base case
▶    int n = Math.max(digits(x), digits(y));
▶    long p = (long) Math.pow(10, n / 2);
▶    long a = x / p, b = x % p;   // x = a·p + b
▶    long c = y / p, d = y % p;   // y = c·p + d
7    long z0 = karatsuba(b, d);
8    long z1 = karatsuba(a + b, c + d);
9    long z2 = karatsuba(a, c);
10   return z2 * p * p + (z1 - z2 - z0) * p + z0;
11 }
```

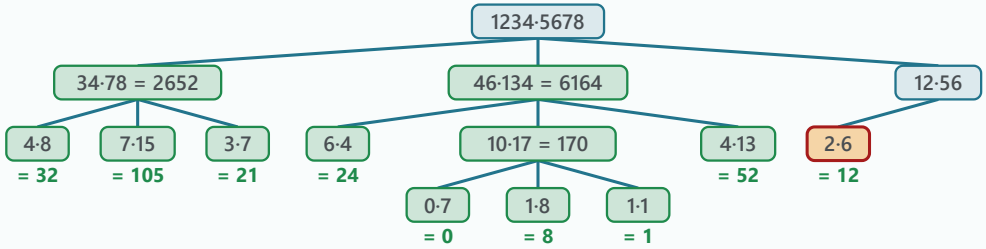
VARIABLES

x = 12 · y = 56 · n = 2 · p = 10 · a = 1 · b = 2 · c = 5 · d = 6

CALL STACK

- karatsuba(12, 56)
- karatsuba(1234, 5678)
- main()

Karatsuba: Tracing 1234 · 5678



2 · 6: a one-digit factor → **base case**, directly 12.

Step 17 / 21

```
1  long karatsuba(long x, long y) {
▶    if (x < 10 || y < 10) return x * y; // base case
3    int n = Math.max(digits(x), digits(y));
4    long p = (long) Math.pow(10, n / 2);
5    long a = x / p, b = x % p;   // x = a·p + b
6    long c = y / p, d = y % p;   // y = c·p + d
7    long z0 = karatsuba(b, d);
8    long z1 = karatsuba(a + b, c + d);
9    long z2 = karatsuba(a, c);
10   return z2 * p * p + (z1 - z2 - z0) * p + z0;
11 }
```

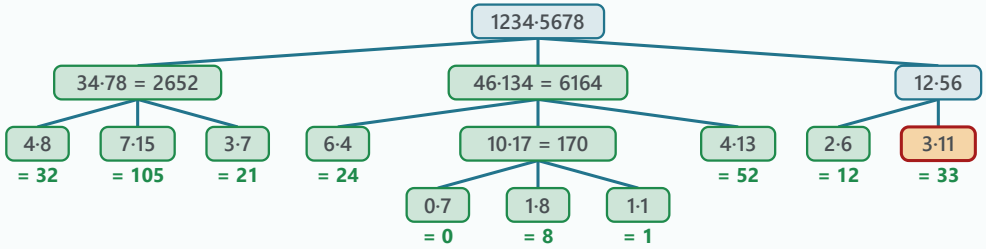
VARIABLES

x = 2 y = 6

CALL STACK

- karatsuba(2, 6)
- karatsuba(12, 56)
- karatsuba(1234, 5678)
- main()

Karatsuba: Tracing 1234 · 5678



3 · 11: a one-digit factor → **base case**, directly 33.

Step 18 / 21

```
1  long karatsuba(long x, long y) {
▶    if (x < 10 || y < 10) return x * y; // base case
3    int n = Math.max(digits(x), digits(y));
4    long p = (long) Math.pow(10, n / 2);
5    long a = x / p, b = x % p;   // x = a·p + b
6    long c = y / p, d = y % p;   // y = c·p + d
7    long z0 = karatsuba(b, d);
8    long z1 = karatsuba(a + b, c + d);
9    long z2 = karatsuba(a, c);
10   return z2 * p * p + (z1 - z2 - z0) * p + z0;
11 }
```

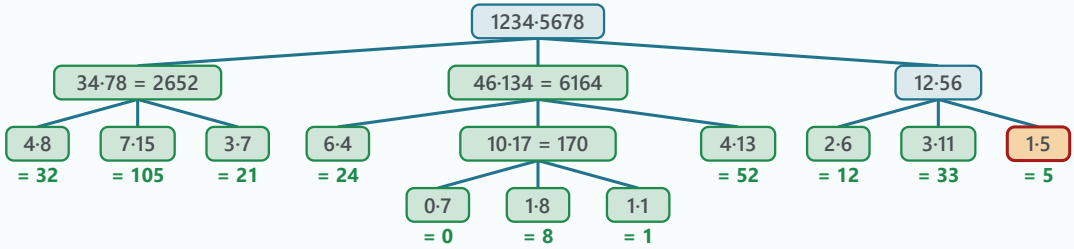
VARIABLES

`x = 3` `y = 11`

CALL STACK

- `karatsuba(3, 11)`
- `karatsuba(12, 56)`
- `karatsuba(1234, 5678)`
- `main()`

Karatsuba: Tracing 1234 · 5678



1 · 5: a one-digit factor → **base case**, directly 5.

Step 19 / 21

```
1  long karatsuba(long x, long y) {
▶    if (x < 10 || y < 10) return x * y; // base case
3    int n = Math.max(digits(x), digits(y));
4    long p = (long) Math.pow(10, n / 2);
5    long a = x / p, b = x % p;   // x = a·p + b
6    long c = y / p, d = y % p;   // y = c·p + d
7    long z0 = karatsuba(b, d);
8    long z1 = karatsuba(a + b, c + d);
9    long z2 = karatsuba(a, c);
10   return z2 * p * p + (z1 - z2 - z0) * p + z0;
11 }
```

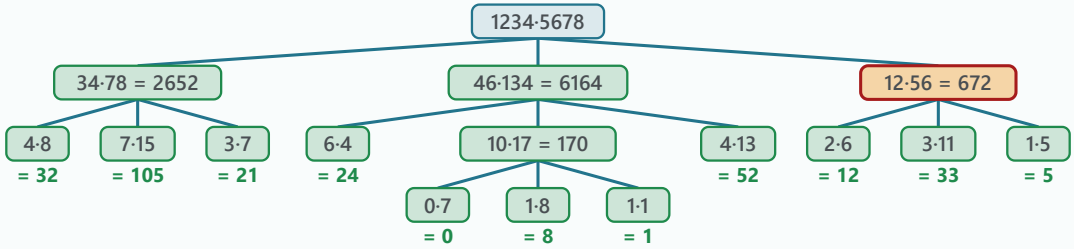
VARIABLES

`x = 1` `y = 5`

CALL STACK

- `karatsuba(1, 5)`
- `karatsuba(12, 56)`
- `karatsuba(1234, 5678)`
- `main()`

Karatsuba: Tracing 1234 · 5678



5·100 + (33 − 5 − 12)·10 + 12 = 500 + 160 + 12 = **672**.

Step 20 / 21

```
1  long karatsuba(long x, long y) {
2    if (x < 10 || y < 10) return x * y; // base case
3    int n = Math.max(digits(x), digits(y));
4    long p = (long) Math.pow(10, n / 2);
5    long a = x / p, b = x % p;   // x = a·p + b
6    long c = y / p, d = y % p;   // y = c·p + d
7    long z0 = karatsuba(b, d);
8    long z1 = karatsuba(a + b, c + d);
9    long z2 = karatsuba(a, c);
▶    return z2 * p * p + (z1 - z2 - z0) * p + z0;
11 }
```

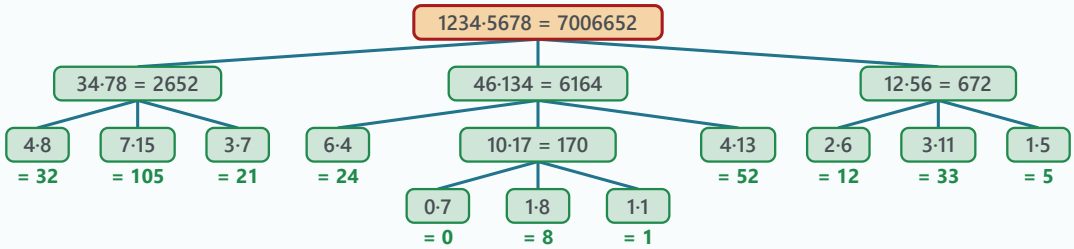
VARIABLES

`x = 12` `y = 56` `p = 10` `z0 = 12` `z1 = 33` `z2 = 5`

CALL STACK

- `karatsuba(12, 56)`
- `karatsuba(1234, 5678)`
- `main()`

Karatsuba: Tracing 1234 · 5678



672·10000 + (6164 − 672 − 2652)·100 + 2652 = 6720000 + 284000 + 2652 = **7006652**.

Step 21 / 21

```
1  long karatsuba(long x, long y) {
2    if (x < 10 || y < 10) return x * y; // base case
3    int n = Math.max(digits(x), digits(y));
4    long p = (long) Math.pow(10, n / 2);
5    long a = x / p, b = x % p;   // x = a·p + b
6    long c = y / p, d = y % p;   // y = c·p + d
7    long z0 = karatsuba(b, d);
8    long z1 = karatsuba(a + b, c + d);
9    long z2 = karatsuba(a, c);
▶    return z2 * p * p + (z1 - z2 - z0) * p + z0;
11 }
```

VARIABLES

`x = 1234` `y = 5678` `p = 100`

`z0 = 2652` `z1 = 6164` `z2 = 672`

CALL STACK

`karatsuba(1234, 5678)`

`main()`

Karatsuba Analysis

- $T(n) = 3T(n/2) + O(n)$ → case 1 → $\Theta(n^{\log_2 3}) \approx \Theta(n^{1.585})$.
- Addition, subtraction, and shifting by 10^m cost $O(n)$; cutting the products from 4 to 3 lowers the exponent from 2 to 1.585.
- The example makes 16 calls and 11 base-case products (the schoolbook method needs 16 one-digit products).
- Slower on small numbers because of the extra additions → use the schoolbook method below a **threshold**.
- Java's `BigInteger.multiply` uses Karatsuba and Toom-Cook for large numbers. Faster still: Schönhage–Strassen $O(n \log n \log \log n)$, Harvey–van der Hoeven $O(n \log n)$ (2019).

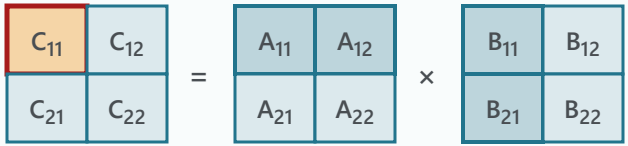
n (digits)	n^2	$n^{1.585}$	ratio
10	100	38	2.6×
100	10,000	1,479	6.8×
1,000	$1.0 \cdot 10^6$	56,871	17.6×
10,000	$1.0 \cdot 10^8$	$2.2 \cdot 10^6$	45.7×
$1.0 \cdot 10^6$	$1.0 \cdot 10^{12}$	$3.2 \cdot 10^9$	309.2×

Matrix Multiplication: Classic and Block

Classic: $\Theta(n^3)$

```
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++)
    for (int k = 0; k < n; k++)
      C[i][j] += A[i][k] * B[k][j];
```

n^2 cells, each needing n products → n^3 products.



C11 = A11·B11 + A12·B21: 2 block products

Block divide and conquer

Split A, B, and C into four n/2 × n/2 blocks:

C11 = A11·B11 + A12·B21
C12 = A11·B12 + A12·B22
C21 = A21·B11 + A22·B21
C22 = A21·B12 + A22·B22

8 block products + 4 block additions: $T(n) = \mathbf{8}T(n/2) + \Theta(n^2) \rightarrow n^{\log_2 8} = \mathbf{\Theta(n^3)}$. Splitting alone is not enough; the number of products must go down.

Strassen (1969): 7 Products Instead of 8

Product	Definition
P1	A11 · (B12 − B22)
P2	(A11 + A12) · B22
P3	(A21 + A22) · B11
P4	A22 · (B21 − B11)
P5	(A11 + A22) · (B11 + B22)
P6	(A12 − A22) · (B21 + B22)
P7	(A11 − A21) · (B11 + B12)

Result blocks
C11 = P5 + P4 − P2 + P6
C12 = P1 + P2
C21 = P3 + P4
C22 = P5 + P1 − P3 − P7

7 products + 18 block additions/subtractions. The blocks are themselves multiplied with Strassen recursively; the base case is 1×1.

In C22, P7 is **subtracted**. Writing "+ P7" makes the program silently produce a wrong matrix; checking the implementation against the classic method on a small example catches the bug.

Strassen Step by Step: A 2 × 2 Example

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

P1	A11 · (B12 − B22)	1 · (8 − 2)	**6**
P2	(A11 + A12) · B22	…	
P3	(A21 + A22) · B11	…	
P4	A22 · (B21 − B11)	…	
P5	(A11 + A22) · (B11 + B22)	…	
P6	(A12 − A22) · (B21 + B22)	…	
P7	(A11 − A21) · (B11 + B12)	…	

P1 = 1 · (8 − 2) = **6**.

Strassen Step by Step: A 2 × 2 Example

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

P1	A11 · (B12 − B22)	1 · (8 − 2)	**6**
P2	(A11 + A12) · B22	(1 + 3) · 2	**8**
P3	(A21 + A22) · B11	…	
P4	A22 · (B21 − B11)	…	
P5	(A11 + A22) · (B11 + B22)	…	
P6	(A12 − A22) · (B21 + B22)	…	
P7	(A11 − A21) · (B11 + B12)	…	

P2 = (1 + 3) · 2 = **8**.

Strassen Step by Step: A 2 × 2 Example

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

P1	A11 · (B12 − B22)	1 · (8 − 2)	**6**
P2	(A11 + A12) · B22	(1 + 3) · 2	**8**
P3	(A21 + A22) · B11	(7 + 5) · 6	**72**
P4	A22 · (B21 − B11)	…	
P5	(A11 + A22) · (B11 + B22)	…	
P6	(A12 − A22) · (B21 + B22)	…	
P7	(A11 − A21) · (B11 + B12)	…	

P3 = (7 + 5) · 6 = **72**.

Strassen Step by Step: A 2 × 2 Example

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

P1	A11 · (B12 − B22)	1 · (8 − 2)	**6**
P2	(A11 + A12) · B22	(1 + 3) · 2	**8**
P3	(A21 + A22) · B11	(7 + 5) · 6	**72**
P4	A22 · (B21 − B11)	5 · (4 − 6)	**−10**
P5	(A11 + A22) · (B11 + B22)	…	
P6	(A12 − A22) · (B21 + B22)	…	
P7	(A11 − A21) · (B11 + B12)	…	

P4 = 5 · (4 − 6) = **−10**.

Strassen Step by Step: A 2 × 2 Example

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

P1	A11 · (B12 − B22)	1 · (8 − 2)	**6**
P2	(A11 + A12) · B22	(1 + 3) · 2	**8**
P3	(A21 + A22) · B11	(7 + 5) · 6	**72**
P4	A22 · (B21 − B11)	5 · (4 − 6)	**−10**
P5	(A11 + A22) · (B11 + B22)	(1 + 5) · (6 + 2)	**48**
P6	(A12 − A22) · (B21 + B22)	…	
P7	(A11 − A21) · (B11 + B12)	…	

P5 = (1 + 5) · (6 + 2) = **48**.

Strassen Step by Step: A 2 × 2 Example

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

P1	A11 · (B12 − B22)	1 · (8 − 2)	**6**
P2	(A11 + A12) · B22	(1 + 3) · 2	**8**
P3	(A21 + A22) · B11	(7 + 5) · 6	**72**
P4	A22 · (B21 − B11)	5 · (4 − 6)	**−10**
P5	(A11 + A22) · (B11 + B22)	(1 + 5) · (6 + 2)	**48**
P6	(A12 − A22) · (B21 + B22)	(3 − 5) · (4 + 2)	**−12**
P7	(A11 − A21) · (B11 + B12)	…	

P6 = (3 − 5) · (4 + 2) = **−12**.

Strassen Step by Step: A 2 × 2 Example

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

P1	A11 · (B12 − B22)	1 · (8 − 2)	**6**
P2	(A11 + A12) · B22	(1 + 3) · 2	**8**
P3	(A21 + A22) · B11	(7 + 5) · 6	**72**
P4	A22 · (B21 − B11)	5 · (4 − 6)	**−10**
P5	(A11 + A22) · (B11 + B22)	(1 + 5) · (6 + 2)	**48**
P6	(A12 − A22) · (B21 + B22)	(3 − 5) · (4 + 2)	**−12**
P7	(A11 − A21) · (B11 + B12)	(1 − 7) · (6 + 8)	**−84**

P7 = (1 − 7) · (6 + 8) = **−84**.

Strassen Step by Step: A 2 × 2 Example

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

C11 = P5 + P4 − P2 + P6	48 + (−10) − 8 + (−12)	**18**
C12 = P1 + P2	6 + 8	**14**
C21 = P3 + P4	72 + (−10)	**62**
C22 = P5 + P1 − P3 − P7	48 + 6 − 72 − (−84)	**66**

C = [[18, 14], [62, 66]]. Check: 1·6 + 3·4 = 18, 7·8 + 5·2 = 66 ✓. 7 products used (classic: 8).

Strassen Analysis and Practice

- $T(n) = 7T(n/2) + \Theta(n^2) \rightarrow$ case 1 $\rightarrow$ $\mathbf{\Theta(n^{\log_2 7}) \approx \Theta(n^{2.807})}$.
- Scalar products for $n = 2^k$: $\mathbf{7^k}$ (classic: 8^k). For $n = 1024$: $7^{10} \approx 2.8 \cdot 10^8$ vs. $8^{10} \approx 1.07 \cdot 10^9$ $\rightarrow$ about **3.8 times** fewer.
- If n is not a power of 2, the matrix is **padded** with zeros.
- **Threshold:** 18 extra additions and temporary matrices make it slow for small n; libraries switch to the classic method for $n \lesssim 64$–128.
- **Numerical stability:** with floating-point numbers, the subtractions can amplify errors.
- **Memory:** temporary blocks at every level $\rightarrow$ extra $O(n^2)$.
- The theoretical record is $\omega < 2.372$ (2024); the constants are huge, so it is not used in practice.

Lesson: trading a product for additions lowers the exponent by reducing **a** by one. Same idea in Karatsuba (4 → 3) and Strassen (8 → 7).

Naive Method vs. Divide and Conquer

Problem	Naive method	Divide and conquer	Recurrence
Sorting	$O(n^2)$ (bubble sort)	$O(n \log n)$	$2T(n/2) + n$
Search in a sorted array	$O(n)$	$O(\log n)$	$T(n/2) + 1$
Exponentiation x^n	$O(n)$ products	$O(\log n)$	$T(n/2) + 1$
Convex hull	$O(n^3)$	avg. $O(n \log n)$	$T(n_1) + T(n_2) + n$
Closest pair of points	$O(n^2)$	$O(n \log n)$	$2T(n/2) + n$
Integer multiplication	$O(n^2)$	$O(n^{1.585})$	$3T(n/2) + n$
Matrix multiplication	$O(n^3)$	$O(n^{2.807})$	$7T(n/2) + n^2$

Where the gain comes from: either the work is split evenly and each level is combined in $O(n)$ (sorting, closest pair), or the number of subproblems is reduced (Karatsuba, Strassen), or half is thrown away (search, exponentiation).

Summary

Topic	Key points
Paradigm	divide → (break down) → conquer → combine; independent subproblems of the same kind; a base case is required
Analysis	$T(n) = aT(n/b) + f(n)$; recursion tree; master theorem: compare $f(n)$ with $n^{\log_b a}$
Basic examples	binary search and fast exponentiation $O(\log n)$; merge sort $O(n \log n)$; quicksort avg. $O(n \log n)$, worst $O(n^2)$
Quickhull	endpoints A, B; farthest C (max cross); points inside the triangle discarded; avg. $O(n \log n)$, worst $O(n^2)$
Closest pair	split by x; $\delta = \min(\delta_L, \delta_R)$; ≤ 7 neighbors in the 2δ strip; $O(n \log n)$
Karatsuba	$ad + bc = (a + b)(c + d) - ac - bd$; 3 products → $O(n^{1.585})$
Strassen	7 block products → $O(n^{2.807})$; $C22 = P5 + P1 - P3 - P7$

SUMMARY

Common Mistakes

Forgetting the base case

Or writing one that is never reached: infinite recursion, `StackOverflowError`. In Karatsuba: `x < 10 || y < 10`.

Unbalanced splits

The last element as pivot on a sorted array, or a single point per step in Quickhull: $T(n) = T(n - 1) + n = O(n^2)$.

Overlapping subproblems

Plain recursive Fibonacci takes exponential time; store results with dynamic programming.

Underestimating the combine cost

Sorting the strip in every call gives $O(n \log^2 n)$; $2T(n/2) + n \log n \neq \Theta(n \log n)$.

Misapplying the master theorem

It does not apply to $T(n) = T(n - 1) + n$, to unequal splits, or to $f(n) = n/\log n$.

Sign errors

In Strassen, $C22 = P5 + P1 - P3 - P7$; a single sign error silently produces a wrong matrix. Comparing with the classic method on a small example catches it.

SUMMARY

Sample Problems

1. Solve $T(n) = 4T(n/2) + n$ with the master theorem.
2. Solve $T(n) = 2T(n/2) + n^2$; check the regularity condition.
3. Compute $21 \cdot 43$ with Karatsuba ($m = 1$).
4. $A = (0,0)$, $B = (10,0)$. Find cross(A, B, ·) for $P = (3,4)$, $Q = (5,-2)$, $R = (7,6)$. Which point does Quickhull choose in the upper set?
5. How many multiplications does `power(3, 10)` perform? Compare with the naive method.
6. When merge sort sorts [5, 1, 4, 2, 3], which two sorted arrays are merged in the final merge?

▶ **Show answers**

SUMMARY

Exercises

1. Find the maximum element of an array with divide and conquer; solve the recurrence.
2. Modify merge sort to count **inversions** ($i < j$, $a[i] > a[j]$). What is the result for [2, 4, 1, 3, 5]?
3. **Maximum subarray**: give a divide-and-conquer solution for [−2, 1, −3, 4, −1, 2, 1, −5, 4] and its running time.
4. Compute A = [[1, 2], [3, 4]] times B = [[5, 6], [7, 8]] with Strassen; try C22 with ± P7.
5. Describe an input on which Quickhull takes $O(n^2)$ time.
6. What would change in the closest pair algorithm if the base case were $n \le 1$ instead of $n \le 3$?

▶ **Answers / hints**

References

- Cormen, Leiserson, Rivest, and Stein, *Introduction to Algorithms*, 4th ed., 2022 (Chapter 4: Divide-and-Conquer; Chapter 33: Computational Geometry).
- Kleinberg and Tardos, *Algorithm Design*, 2005 (Chapter 5: Divide and Conquer).
- Sedgewick and Wayne, *Algorithms*, 4th ed., 2011.
- Barber, Dobkin, and Huhdanpaa, "The Quickhull Algorithm for Convex Hulls", *ACM TOMS* 22(4), 1996.
- Karatsuba and Ofman, "Multiplication of Multidigit Numbers on Automata", 1962; Strassen, "Gaussian Elimination is not Optimal", *Numerische Mathematik* 13, 1969.

The End

Chapter 1: Divide and Conquer

Next chapter: Chapter 2: Greedy Algorithms