

Lecture Note 4

Divide and Conquer

draft of September 29, 2026, by [N. Young](#)

© 2018–2026 [Neal E. Young](#). Text: CC BY-NC-SA 4.0; code: MIT; except where noted. See <https://github.com/nealeyoung/algs101/blob/main/LICENSE>.

Contents

4.1	Bounding sums	1
4.1.1	Naive upper and lower bounds	1
4.1.2	Geometric sums	1
4.1.3	Exercises	2
4.1.4	External sources on bounding sums	2
4.2	Recursion trees for recursive algorithms	3
4.2.1	Exercises	4
4.2.2	External sources on recursion trees and recursive algorithms	6
4.3	An algorithm for the Closest Pair problem	7
4.3.1	Exercises	8
4.3.2	External sources on Closest Pair	9
4.4	An algorithm for Large-Integer Multiplication	10
4.4.1	Exercise	11
4.4.2	External sources on Large-Integer Multiplication	11
4.5	An algorithm for Matrix Multiplication	12
4.5.1	Exercise	13
4.5.2	External sources on fast matrix multiplication	13
4.6	An algorithm for selecting the kth smallest	14
4.6.1	Exercise	16
4.6.2	External sources on deterministic selection	16
4.7	An incorrect algorithm for finding a 2D Local Maximum	17
4.7.1	1D Local Maximum	17
4.7.2	2D Local Maximum	18
4.7.3	Exercise	19
4.7.4	External sources on finding a local maximum	19
4.8	Additional exercises on Divide and Conquer	21

4.1 Bounding sums

Recall that, for example, $\sum_{i=1}^n i^2$ denotes the sum $1^2 + 2^2 + \dots + n^2$. You may have previously learned *closed forms* for some sums, such as $\sum_{i=1}^n i = n(n+1)/2$. Unfortunately, for most sums of interest, closed forms are difficult to find or simply don't exist. Fortunately, when we are analyzing asymptotic running times, we don't care about lower-order terms or constant factors, so we don't usually need exact closed forms for the sums that arise. Instead, for example, we only need to know that $\sum_{i=1}^n i = \Theta(n^2)$. Here are three rules for quickly estimating the big- Θ values of sums.

4.1.1 Naive upper and lower bounds

Here are two rules that work well for sums whose terms grow polynomially (or less) with the index.

Observation 4.1 (Naive upper bound). *Any sum is at most the number of terms times the max. term.*

Three examples:

$$\begin{aligned}\sum_{i=1}^n i &\leq n \cdot \max_{i=1}^n i = n \cdot n = O(n^2) \\ \sum_{i=1}^n i^2 &\leq n \cdot \max_{i=1}^n i^2 = n \cdot n^2 = O(n^3) \\ \sum_{i=1}^n \log i &\leq n \cdot \max_{i=1}^n \log i = n \log n = O(n \log n)\end{aligned}$$

Observation 4.2 (Naive lower bound). *Any sum with non-negative terms is at least half the number of terms times the median term.*

Three examples:

$$\begin{aligned}\sum_{i=1}^n i &\geq (n/2) \cdot (n/2) = \Omega(n^2) \\ \sum_{i=1}^n i^2 &\geq (n/2) \cdot (n/2)^2 = \Omega(n^3) \\ \sum_{i=1}^n \log i &\geq (n/2) \cdot \log(n/2) = \Omega(n \log n)\end{aligned}$$

For the three examples above, the naive upper and lower bounds match (up to constant factors). So together they are enough to determine the value of the sum, up to constant factors (the big- Θ value). For example, $\sum_{i=1}^n \log i = \Theta(n \log n)$.

The bounds will match for any sum whose terms don't increase (or decrease) too fast—specifically, for any sum where the median term is within a constant factor of the maximum term. For example, it will work for sums whose terms grow only polynomially with the index.

4.1.2 Geometric sums

However, it won't work for all sums. For example, try $\sum_{i=1}^n 2^i$. The naive upper bound gives $\sum_{i=1}^n 2^i = O(n2^n)$. The naive lower bound gives $\sum_{i=1}^n 2^i = \Omega(n2^{n/2})$. But these upper and lower bounds are far apart! Indeed their ratio $2^n/2^{n/2} = 2^{n/2}$ is quite large. The answer is actually $\sum_{i=1}^n 2^i = \Theta(2^n)$. Neither the naive upper or lower bound is tight in this case.

We have another rule for such sums. First we formally define the type of sums it applies to:

Definition 4.1. *A sum $\sum_{i=1}^n \alpha_i$ is geometric if, for some constants $c > 1$ and i_0 (indep. of n) either*

- (i) *for each $i \geq i_0$ the i th term is at least c times the previous term (that is, $\alpha_i \geq c \cdot \alpha_{i-1}$), or*
- (ii) *for each $i \geq i_0$ the i th term is at most the previous term divided by c (that is, $\alpha_i \leq \alpha_{i-1}/c$).*

Rule of thumb: if the sum has an exponential term in the summand, it is probably geometric.

Three examples

- $\sum_{i=1}^n 2^i$ is geometric: the ratio of successive terms $2^i/2^{i-1}$ is at least two.
- $\sum_{i=1}^n 2^i/i$ is also geometric: for $i \geq 3$ the ratio of successive terms is $(2^i/i)/(2^{i-1}/(i-1)) = 2(i-1)/i \geq 4/3$.
- $\sum_{i=1}^n i^2 + 1$ is not geometric: the ratio of successive terms $((i+1)^2 + 1)/(i^2 + 1)$ gets arbitrarily close to 1 for large i .

Here's our third rule. It handles any geometric sum:

Observation 4.3 (geometric sums). *Any geometric sum is proportional to its largest term.*

Three examples

$$\begin{aligned}\sum_{i=0}^n 2^i &= \Theta(\max_{i=0}^n 2^i) = \Theta(2^n). \\ \sum_{i=1}^n 2^i/i &= \Theta(\max_{i=1}^n 2^i/i) = \Theta(2^n/n). \\ \sum_{i=0}^{\infty} i^2/2^i &= \Theta(\max_{i=0}^{\infty} i^2/2^i) = \Theta(1).\end{aligned}$$

4.1.3 Exercises

Exercise 4.1. (*Bounding sums*) Use the so-called naive upper and lower bounds, and the rule for geometric sums, to determine the big- Θ value of each of the following sums.

- $\sum_{i=1}^n 1.1^i/i^2$
- $\sum_{i=1}^{\log_2 n} i^2 \log_3 i$
- $\sum_{i=1}^{2^n} (\log_3 i)^2$
- $\sum_{i=0}^n i^3/2^i$

4.1.4 External sources on bounding sums

[CLRS](#) Appendices A-1 and A-2

<code>mergesort($A[1..n]$)</code>	
1. if $n = 1$, return	
2. set $m = \lfloor n/2 \rfloor$	
3. <code>mergesort($A[1..m]$)</code>	(sort the first $n/2$ elements)
4. <code>mergesort($A[m + 1..n]$)</code>	(sort the last $n/2$ elements)
5. <code>merge($A[1..n], m$)</code>	(merge the two sorted halves, in time $\Theta(n)$)

Figure 4.1: Mergesort. The `merge` procedure runs in linear time. We don't give its code.

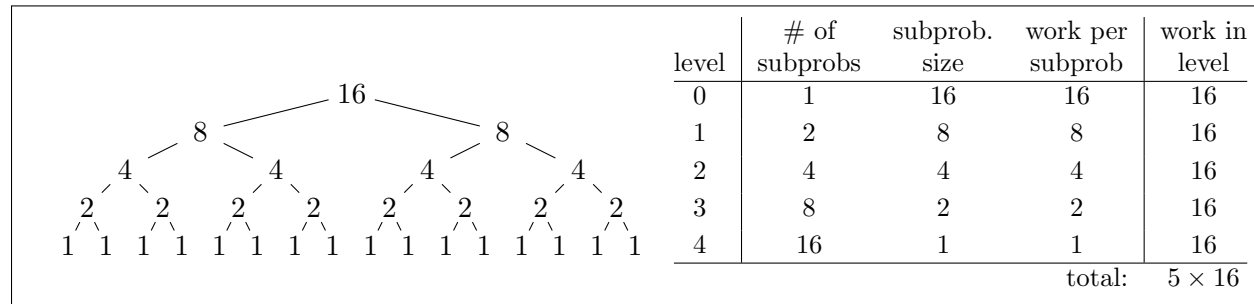


Figure 4.2: On the left, the recursion tree for `mergesort` on an array of size 16. On the right, a table of the work done in each level of the recursion tree.

4.2 Recursion trees for recursive algorithms

Next we discuss how to analyze the run times of recursive algorithms, using `mergesort` (shown in Figure 4.1) as an example. Let $T(n)$ denote the worst-case running time of `mergesort` on any array of size n . Then $T(1) = \Theta(1)$. For $n \geq 2$, by inspection of `mergesort` and using that `merge` takes linear time, we have the recurrence relation

$$T(n) = 2T(n/2) + n.$$

(For now, we assume n is a power of 2, so $n/2$ is an integer.)

Using a *recursion tree* to count the work. A *recursion tree* represents the recursive structure of an execution of a recursive algorithm. The root node represents the top-level call. Each non-root node represents one recursive call. Each node is labeled by the size of the subarray that the corresponding call sorts. The children of each node for a given call represent the two recursive calls that that call makes. Figure 4.2 shows the recursion tree for `mergesort` on an array of size $n = 16$.

As illustrated in the table to the right in the figure, if the input array size n is a power of 2, there are $1 + \log_2 n$ levels. For each $i \in \{0, \dots, \log_2 n\}$, level i has 2^i subproblems, each of size $n/2^i$. The work associated with a subproblem of size $n/2^i$ is $\Theta(n/2^i)$, so the total work at level i is $\Theta(2^i \times n/2^i) = \Theta(n)$. Summing over the $\log n$ levels, the total work is $\Theta(n \log n)$. Hence, $T(n) = \Theta(n \log n)$. Note that the work is balanced evenly across the levels.

If n is not a power of 2, then there are $1 + \lceil \log_2 n \rceil$ levels. In each level i except the last, there are 2^i subproblems, each of size at most $\lceil n/2^i \rceil \leq 2n/2^i$. So, by a similar calculation, the total work $T(n)$ is also $\Theta(n \log n)$ in this case.

Theorem 4.1. *Mergesort runs in $O(n \log n)$ time.*

Variation with heavy leaves. Consider the recurrence relation $T(n) = 3 \cdot T(n/2) + n$. The same approach applies, but now each non-leaf node has **three** children. As before we have

$$T(n) = \sum_{i=0}^{\text{max level}} (\# \text{ subprobs on level } i) \times (\text{work per subprob. on level } i)$$

but now the number of subproblems on level i is 3^i , so the calculation proceeds as follows:

$$\begin{aligned} T(n) &= \sum_{i=0}^{\log_2 n} 3^i (n/2^i) = n \sum_{i=0}^{\log_2 n} (3/2)^i = \Theta(n (3/2)^{\log_2 n}) \quad (\text{because the sum is geometric}) \\ &= \Theta(n \cdot n^{\log_2 3/2}) = \Theta(n^{\log_2 3}) = \Theta(n^{1.58\dots}) \quad (\text{using } b^{\log n} = n^{\log b} \text{ and } \log_2 3/2 = \log_2(3) - 1). \end{aligned}$$

For mergesort, the work was balanced across the levels. But here we get a qualitatively different behavior: *the total work is dominated by the work done at the leaves.*

Variation with heavy root. Consider the recurrence relation $T(n) = 3T(n/2) + n^2$. Then

$$T(n) = \sum_{i=0}^{\log_2 n} 3^i (n/2^i)^2 = n^2 \sum_{i=0}^{\log_2 n} (3/4)^i = \Theta(n^2).$$

In this case *the total work is dominated by the work at the root.*

These are the three common cases: either (a) the work is balanced across the levels, or the sum is geometric, so either (b) the work at the leaves dominates, or (c) the work at the root dominates. A particularly important special case of the latter case (when the work at the root dominates) is when *the algorithm does linear work, then recurses on subproblems whose combined size is a constant factor less than the size of the given problem.*

For example, consider the recurrence relation $T(n) = aT(n/b) + n$, for some $a < b$. That is, the algorithm does linear work, then recurses on a problems, each of size n/b , so having total size $an/b = (a/b)n$ where $a/b < 1$. Then we get

$$T(n) = \sum_{i=0}^{\log_b n} a^i (n/b^i) = n \sum_{i=0}^{\log_b n} (a/b)^i = \Theta(n).$$

The last step follows because $a < b$, so the sum is geometric and dominated by the first term. See also the Selection algorithm in Section 4.6.

Python code: `divide_and_conquer/mergesort.py` (Section 4.9).

4.2.1 Exercises

With solution

Exercise 4.2. (*Recursion-tree method*) For the algorithm below, use the recursion-tree method to find the asymptotic value (using the Θ -notation) for the number of letters printed by the algorithm as a function of n .

<pre>def PrintYs(n): 1. for i = 1 to n log₂ n: print("Y") 2. if n > 1: 3. for j = 1 to 3: PrintYs($\lfloor n/3 \rfloor$)</pre>	<i>Assume n is a power of 3.</i> <i>... print $n \log_2 n$ Y's</i> <i>... make 3 recursive calls, each with input $\lfloor n/3 \rfloor$</i>
---	---

The solution should give

- (i) an appropriate recurrence relation,
- (ii) as a function of the level i in the recursion tree: the number of nodes (that is, calls) in level i , the problem size (the argument to each call) at each node in level i , and the number of letters printed by each call in level i ,
- (iii) the number of levels in the recursion tree,
- (iv) the sum, over all levels, of the number of letters printed by calls in that level, and finally,
- (v) a big- Θ estimate of the value of the sum (as a function of n), obtained by using either the rule for geometric sums or the naive upper and lower bounds (show your work for (v) if you want feedback).

Solution.

- (i) Define $Y(n)$ to be the number of letters printed by PRINTYs given input $n \geq 1$.
Then $Y(1) = 0$ and, for $n > 1$, $Y(n) = n \log_2 n + 3Y(n/3)$.
- (ii) In level i , there are 3^i nodes. Each call has input $n/3^i$ and prints $(n/3^i) \log_2(n/3^i)$ letters.
- (iii) There are $1 + \log_3 n$ levels.
- (iv) The total number of letters printed is $Y(n) = \sum_{i=0}^{\log_3 n} 3^i (n/3^i) \log_2(n/3^i) = \sum_{i=0}^{\log_3 n} n \log_2(n/3^i)$.
- (v) By the naive upper bound, the sum is at most $(1 + \log_3 n)n \log_2 n = O(n \log^2 n)$.
By the naive lower bound, the sum is at least $\frac{1}{2}(1 + \log_3 n)$ times the median term, which is $n \log_2(n/3^i)$ where $i = (1 + \log_3 n)/2$. For this i , 3^i simplifies to $\Theta(\sqrt{n})$, so the median term simplifies to $n \log_2(n/\Theta(\sqrt{n})) = n \log_2 \Theta(\sqrt{n}) = \Theta(n \log n)$. So the sum is $\Omega(n \log^2 n)$.
Thus, the sum, which equals $Y(n)$, is $\Theta(n \log^2 n)$.

Exercise without solutions

Exercise 4.3. (*Mergesort variation*) Consider a variant of Mergesort that, given an array of size N , recurses as usual just until the subproblems are of size $\sqrt{N}$, then solves each subproblem of size $\sqrt{N}$ directly using insertion sort (a quadratic-time algorithm). Use a recursion-tree analysis to determine the asymptotic worst-case running time of this algorithm on arrays of size n . Show your work as described for Exercise 4.2.

Exercise 4.4. (*Recursion-tree method*) Repeat Exercise 4.2 for each of the algorithms below. Give your answers in the same format as the solution given for that exercise. To make the calculations easier, ignore floors in your calculations. (That is, for (a), (c), and (e) assume n is a power of 3. For (b) assume n is a power of 4. For (d) assume n is a power of 2.)

- (a) def PrintAs(n): *Assume n is a power of 3.*
 1. for $i = 1$ to $3n$: print("A") *... print $3n$ A's*
 2. if $n > 1$:
 3. for $j = 1$ to 4: PrintAs($\lfloor n/3 \rfloor$) *... make 4 recursive calls, each with input $\lfloor n/3 \rfloor$*

- (b) `def PrintBs( $n$ ):` *Assume n is a power of 4.*
 1. `for  $i = 1$  to  $n^2$ : print("B")` *... print n^2 B's*
 2. `if  $n > 1$ :`
 3. `for  $j = 1$  to 16: PrintBs( $\lfloor n/4 \rfloor$ )` *... make 16 recursive calls, each with input $\lfloor n/4 \rfloor$*
- (c) `def PrintCs( $n$ ):` *Assume n is a power of 3.*
 1. `for  $i = 1$  to  $5n$ : print("C")` *... print $5n$ C's*
 2. `if  $n > 1$ :`
 3. `for  $j = 1$  to 2: PrintCs( $\lfloor n/3 \rfloor$ )` *... make 2 recursive calls, each with input $\lfloor n/3 \rfloor$*
- (d) `def PrintDs( $n$ ):` *Assume n is a power of 2.*
 1. `for  $i = 1$  to 11: print("D")` *... print 11 D's*
 2. `if  $n > 1$ :`
 3. `for  $j = 1$  to 2: PrintDs( $\lfloor n/2 \rfloor$ )` *... make 2 recursive calls, each with input $\lfloor n/2 \rfloor$*
- (e) `def PrintEs( $n$ ):` *Assume n is a power of 3.*
 1. `for  $i = 1$  to  $2n^3$ : print("E")` *... print $2n^3$ E's*
 2. `if  $n > 1$ :`
 3. `for  $j = 1$  to 9: PrintEs( $\lfloor n/3 \rfloor$ )` *... make 9 recursive calls, each with input $\lfloor n/3 \rfloor$*

4.2.2 External sources on recursion trees and recursive algorithms

- [DPV](#) Chapter 2.2; [KT](#) Chapter 5; [CLRS](#) Chapter 4 (especially 4.4); [LLM](#) Chapter 22;
- [JE](#) Chapter 1, and Appendix II Section 3 (& exercises starting on Page 20), at <https://jeffe.cs.illinois.edu/teaching/algorithms/notes/99-recurrences.pdf>
- An MIT lecture video
 - <https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/lecture-videos/lecture-3-insertion-sort-merge-sort/>

4.3 An algorithm for the Closest Pair problem

Problem definition. The Closest Pair problem is defined as follows:

input: 2D points $P = (p_1, p_2, \dots, p_n)$ where each $p_i = (x_i, y_i) \in \mathbb{R}^2$

output: the minimum Euclidean distance $\delta = \min\{d(p_i, p_j) : 1 \leq i < j \leq n\}$ between any two of the points. Here $d((x_i, y_i), (x_j, y_j)) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$. For $n \leq 1$, the answer is ∞ .

Algorithm. The obvious algorithm (compare all points) takes time $O(n^2)$. We want $O(n \log n)$.

We'll assume the points are sorted by x-coordinate. If not, we can sort them in $O(n \log n)$ time in a pre-processing step. To keep things simple, for now, we also assume that the x-coordinates are distinct. Our goal is to use a divide-and-conquer approach. What about this algorithm?

closest-pair-1($P = (p_1, p_2, \dots, p_n)$)	$(p_i = (x_i, y_i) \in \mathbb{R}^2, \text{ with } x_1 < x_2 < \dots < x_n)$
1. if $n \leq 1$: return ∞	
2. if $n = 2$: return $d(p_1, p_2)$	
3. let $m = \lfloor n/2 \rfloor$	
4. let $L = (p_1, p_2, \dots, p_m)$; let $R = (p_{m+1}, \dots, p_n)$ (partition P into left and right halves by x-coordinate)	
5. let $\delta_L = \text{closest-pair-1}(L)$; let $\delta_R = \text{closest-pair-1}(R)$	
6. let $\delta = \min(\delta_L, \delta_R)$	
7. return δ	(WRONG! See below)

The running time of this algorithm satisfies $T(n) = 2T(n/2) + O(n)$, so is $O(n \log n)$, as desired. But the algorithm is wrong, because it doesn't consider the pairs (p_i, p_j) such that $i \leq m < j$. That is, $(p_i, p_j) \in L \times R$. To fix it, we need to consider those pairs.

Among those pairs, it is enough to consider only those with $d(p_i, p_j) < \delta$ (for δ from Step 6). (Why?)

Let $X = x_m$ be the median x-coordinate (separating L from R) and let $M = \{p_i : |x_i - X| \leq \delta\}$ contain the points lying in a vertical strip of width 2δ centered at X . Any point $p_i \in L \setminus M$ (that is, in L but not M) has distance at least δ from any point in R . Likewise any $p_i \in R \setminus M$ has distance at least δ from any point in L . So we only need to consider pairs $(p_i, p_j) \in M \times M$.

[Draw a picture to help intuition.]

Further, within M , it is enough to compare each point p_i in M only to those points p_j above p_i in M such that $y_i \leq y_j < y_i + \delta$. (Any other point above p_i in M has distance at least δ to p_i .) And there are at most eight such points:

Lemma 4.1. Assume δ_L and δ_R are calculated correctly. For any point $p_i \in M$, there are at most 8 points p_j above p_i in M such that $y_i \leq y_j < y_i + \delta$.

Proof (short form). We use that any $\delta \times \delta$ square contains at most 4 points in L and at most 4 points in R . To prove this observation, imagine partitioning the square into its four $(\delta/2) \times (\delta/2)$ quadrants. Within each quadrant, every pair of points has distance at most $\sqrt{(\delta/2)^2 + (\delta/2)^2} = \delta/\sqrt{2} < \delta$. So R has at most one point in each quadrant (otherwise R would have a pair with distance less than $\delta \leq \delta_R$, contradicting the correctness of δ_R). Likewise for L .

Now, consider any $p_i \in M$. Consider the two $\delta \times \delta$ squares $S_L = [X - \delta, X] \times [y_i, y_i + \delta]$ and $S_R = (X, X + \delta] \times [y_i, y_i + \delta]$. Any point p_j in M with $y_i \leq y_j < y_i + \delta$ must lie in either S_L or S_R . S_L contains no points from R , and (by the observation above) contains at most four points from L . So S_L contains at most four points from P . Likewise S_R contains at most four points from P . The lemma follows. $\square$

Summarizing, it is enough to compare each point p_i in M with the points p_j in M such that $y_i \leq y_j \leq y_i + \delta$, and there are at most eight such points (per p_i). To fix the algorithm, we have it sort the points in M by y-coordinate, then have it compare each point p_i in M to the points p_j in M such that $y_i \leq y_j \leq y_i + \delta$:

closest-pair ($P = (p_1, p_2, \dots, p_n)$)	$(p_i = (x_i, y_i) \in \mathbb{R}^2, \text{ with } x_1 < x_2 < \dots < x_n)$
1. if $n \leq 1$: return ∞	
2. if $n = 2$: return $d(p_1, p_2)$	
3. let $m = \lfloor n/2 \rfloor$	
4. let $L = (p_1, p_2, \dots, p_m)$; let $R = (p_{m+1}, \dots, p_n)$ (partition P into left and right halves by x-coordinate)	
5. let $\delta_L = \text{closest-pair}(L)$; let $\delta_R = \text{closest-pair}(R)$	
6. let $\delta = \min(\delta_L, \delta_R)$	
7. let $M = \{p_i : x_i - x_m \leq \delta\}$	(compute this in $O(n)$ time)
8. let $\delta_M = \min\{d(p_i, p_j) : p_i, p_j \in M, i \neq j, y_i \leq y_j \leq y_i + \delta\}$.	(compute this as described in the text below)
9. return $\min(\delta, \delta_M)$	

In Step 8, Compute δ_M efficiently as follows. Sort the points in M by y-coordinate, then enumerate the points p_i in M in order of increasing y-coordinate. For each such p_i , consider the points p_j in the list after p_i , such that $y_j \leq y_i + \delta$. There will be at most eight of these for each p_i , so the total time (after sorting) is $O(n)$.

As described, this would give time $T(n) = 2T(n/2) + \Theta(n \log n)$, which gives $T(n) = \Theta(n \log^2 n)$.

To save a $\log n$ factor, instead of sorting the points in M by y-coordinate within each recursive call, do the following. Along with P , pass in a second list P' of the points in P , this one sorted by y-coordinate. From P' , the sorted list of points in M can be computed in $O(n)$ time. For the recursive calls on L and R , let L' and R' denote the lists L and R sorted by y-coordinate. Compute each of these (from P' again) in time $O(n)$, then pass L' with L , and pass R' with R , to each recursive call.

The time to preprocess all the points to sort them by x and y coordinate (before any recursion) is $O(n \log n)$. The remaining time satisfies $T(n) = 2T(n/2) + O(n)$, which gives $T(n) = O(n \log n)$.

Theorem 4.2. *There is an $O(n \log n)$ -time algorithm for the Closest Pair problem.*

Above we assumed that all points have distinct x -coordinates. Exercise 4.6 asks you to consider what happens if that's not the case.

Python code: [divide_and_conquer/closest_pair.py](#) (Section 4.9).

4.3.1 Exercises

Exercise 4.5. (*implementation*) Implement the algorithm and test your implementation on Hackerrank.com, at <https://www.hackerrank.com/nealetyong-algs101-closest-pair>.

Solution. Python code: [divide_and_conquer/closest_pair.py](#) (Section 4.9).

Exercise 4.6. (*Closest Pair*) In analyzing the Closest Pair algorithm in Section 4.3, we assumed that each point had a unique x -coordinate. On inputs where that's not the case, can the algorithm give an incorrect output, or fail to run in time $O(n \log n)$? If so, how would you fix the algorithm to deal with such inputs?

4.3.2 External sources on Closest Pair

- [DPV](#) Problem 2.32; [KT](#) Chapter 5.4; [CLRS](#) Chapter 9.2

4.4 An algorithm for Large-Integer Multiplication

Problem definition

input: n -bit integers A and B , with binary representations $A = a_1a_2 \dots a_n$ and $B = b_1b_2 \dots b_n$.

output: the product $C = AB$.

This problem is about multiplying integers that are too large to fit in a machine word. For example, RSA encryption currently requires arithmetic on 2048-bit integers, or larger, while current machine words are typically 64 bits.

Algorithm—first attempt. The standard grade-school algorithm takes time $\Theta(n^2)$. We want something faster. We start by considering the following recursive algorithm:

int-product-1($n, A = a_1a_2 \dots a_n, B = b_1b_2 \dots b_n$) (A and B are n -bit integers; n is a power of two)

1. if $n = 1$: return a_1b_1 (base case)
2. split A into $A_1 = a_1 \dots a_{n/2}$ and $A_2 = a_{n/2+1} \dots a_n$ so that $A = 2^{n/2}A_1 + A_2$.
3. split B into $B_1 = b_1 \dots b_{n/2}$ and $B_2 = b_{n/2+1} \dots b_n$ so that $B = 2^{n/2}B_1 + B_2$.
4. return $2^n(A_1 \times B_1) + 2^{n/2}(A_1 \times B_2 + A_2 \times B_1) + (A_2 \times B_2)$

The last step requires four multiplications of $n/2$ -bit integers. *These are done recursively!*

The two multiplications by powers of 2 are done directly just by shifting the bits.

Correctness. Correctness of the algorithm follows by induction from the equation

$$\begin{aligned} A \times B &= (2^{n/2}A_1 + A_2) \times (2^{n/2}B_1 + B_2) \\ &= 2^n(A_1 \times B_1) + 2^{n/2}(A_1 \times B_2 + A_2 \times B_1) + (A_2 \times B_2). \end{aligned} \quad (4.1)$$

Because of this equation, as long as the four sub-products are calculated correctly (and the additions are done correctly) the result returned will be correct.

Run time. Each addition can be done using the standard algorithm in $O(n)$ time. And each multiplication by a power of two just shifts the bits, so can be done in $O(n)$ time. So the running time satisfies $T(n) = 4T(n/2) + n$. The recursion-tree approach gives

$$\begin{aligned} T(n) &= \sum_{i=0}^{\# \text{levels}} (\# \text{ subproblems in level } i) \times (\text{work per subproblem in level } i) \\ &= \sum_{i=0}^{\log_2 n} (4^i)(n/2^i) = n \sum_{i=0}^{\log_2 n} 2^i = n \Theta(2^{\log_2 n}) = \Theta(n^2). \end{aligned}$$

This isn't any faster than the standard algorithm! Can we do better?

Karatsuba's algorithm. The middle term in Step 4, namely $A_1 \times B_2 + A_2 \times B_1$, satisfies

$$A_1 \times B_2 + A_2 \times B_1 = (A_1 + A_2) \times (B_1 + B_2) - A_1 \times B_1 - A_2 \times B_2. \quad (4.2)$$

The terms $A_1 \times B_1$ and $A_2 \times B_2$ are computed anyway for Step 4, so by computing the right-hand side with this equation, we replace two recursive calls by *one*. This gives us Karatsuba's algorithm:

$\text{karatsuba}(n, A = a_1a_2 \dots a_n, B = b_1b_2 \dots b_n)$	$(A \text{ and } B \text{ are } n\text{-bit integers; } n \text{ is a power of two})$
1. if $n = 1$: return a_1b_1	(base case)
2. split A into $A_1 = a_1 \dots a_{n/2}$ and $A_2 = a_{n/2+1} \dots a_n$ so that $A = 2^{n/2}A_1 + A_2$.	
3. split B into $B_1 = b_1 \dots b_{n/2}$ and $B_2 = b_{n/2+1} \dots b_n$ so that $B = 2^{n/2}B_1 + B_2$.	
4. let $M_1 = \text{karatsuba}(n/2, A_1, B_1)$	$(M_1 = A_1 \times B_1)$
5. let $M_2 = \text{karatsuba}(n/2, A_2, B_2)$	$(M_2 = A_2 \times B_2)$
6. let $M_3 = \text{karatsuba}(n/2, A_1 + A_2, B_1 + B_2)$	$(M_3 = (A_1 + A_2) \times (B_1 + B_2))$
7. return $2^n M_1 + 2^{n/2}(M_3 - M_1 - M_2) + M_2$	

Correctness. Correctness follows by induction from Equations (4.1) and (4.2).

Run time. Recalling that each addition or multiplication by a power of two can be done in $O(n)$ time, the running time now satisfies $T(n) = 3T(n/2) + n$, which implies

$$T(n) = \sum_{i=0}^{\log_2 n} (3^i)(n/2^i) = n \sum_{i=0}^{\log_2 n} (3/2)^i = n \Theta((3/2)^{\log_2 n}) = \Theta(n^{\log_2 3}) = \Theta(n^{1.58\dots}).$$

(Above we use $x^{\log y} = y^{\log x}$ and $\log_2(3/2) = \log_2(3) - 1$.) We have the following result:

Theorem 4.3 (Karatsuba). *There is an $O(n^{\log_2 3})$ -time algorithm for multiplying n -bit integers.*

Note: A recent (2019) algorithm achieves run time $O(n \log n)$.

4.4.1 Exercise

Exercise 4.7. (*Karatsuba's variation*) The sums $A_1 + A_2$ and $B_1 + B_2$ may have $n/2 + 1$ bits due to carry. This complicates the algorithm and the recurrence relation. Professor Bo Zo proposes the following variant to fix this. Does it work?

$\text{int-product-2}(n, A = a_1a_2 \dots a_n, B = b_1b_2 \dots b_n)$	$(A \text{ and } B \text{ are } n\text{-bit integers; } n \text{ is a power of two})$
1. if $n = 1$: return a_1b_1	(base case)
2. split A into $A_1 = a_1 \dots a_{n/2}$ and $A_2 = a_{n/2+1} \dots a_n$ so that $A = 2^{n/2}A_1 + A_2$.	
3. split B into $B_1 = b_1 \dots b_{n/2}$ and $B_2 = b_{n/2+1} \dots b_n$ so that $B = 2^{n/2}B_1 + B_2$.	
4. let $M_1 = \text{int-product-2}(n/2, A_1, B_1)$	$(M_1 = A_1 \times B_1)$
5. let $M_2 = \text{int-product-2}(n/2, A_2, B_2)$	$(M_2 = A_2 \times B_2)$
6. let $M_3 = \text{int-product-2}(n/2, A_2 - A_1 , B_2 - B_1)$	$\leftarrow \text{CHANGED THIS LINE AND BELOW!}$
7. let $s_A = \text{sign}(A_2 - A_1)$ and $s_B = \text{sign}(B_2 - B_1)$	
8. return $2^n M_1 + 2^{n/2}(M_1 + M_2 - s_A s_B M_3) + M_2$	

4.4.2 External sources on Large-Integer Multiplication

- [DPV](#) Chapter 2.1, [KT](#) Chapter 5.5
- [JE](#) Chapter 1, Section 1.9 (has information about the 2019 $O(n \log n)$ -time algorithm)
- An MIT lecture video
 - <https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/lecture-videos/lecture-11-integer-arithmetic-karatsuba-multiplication/>

4.5 An algorithm for Matrix Multiplication

Next we describe a similar algorithm for Matrix Multiplication.

Problem definition

input: $n \times n$ matrices A and B .

output: the matrix product $C = A \times B$. (So $c_{ij} = \sum_{k=1}^n a_{ik}b_{kj}$.)

We assume throughout that we are working with 64-bit integers, so multiplication and addition of two numbers can be done in constant time. We also assume n is a power of 2 (we can reduce the general case to this one).

Algorithm—first attempt. The obvious algorithm takes time $O(n^3)$. We want time $o(n^3)$.

To warm up, consider the following recursive algorithm. Let $A_{11}, A_{12}, A_{21}, A_{22}, B_{11}, B_{12}, B_{21}, B_{22}, C_{11}, C_{12}, C_{21}$ and C_{22} be $\frac{n}{2} \times \frac{n}{2}$ submatrices such that

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

We are given A and B , and want to compute C . The following matrix equation holds for each of the four pairs $i, j \in \{1, 2\}$:

$$C_{ij} = A_{i1} \times B_{1j} + A_{i2} \times B_{2j}. \quad (4.3)$$

This gives the following recursive algorithm:

matrix-product-1 (n, A, B)	(A and B are $n \times n$ matrices; n is a power of two)
1. if $n = 1$: return $[a_{11}b_{11}]$	(base case)
2. for each of the four pairs $i, j \in \{1, 2\}$, compute A_{ij} and B_{ij} as defined above	(time $O(n^2)$)
3. for each of the four pairs $i, j \in \{1, 2\}$:	
4.1. let $C_{ij} = \text{matrix-product-1}(n/2, A_{i1}, B_{1j}) + \text{matrix-product-1}(n/2, A_{i2}, B_{2j})$	
5. return $C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$	

Correctness. Correctness of the algorithm follows from Equation (4.3), by induction.

Running time. To compute the product of two $n \times n$ matrices, the algorithm makes eight recursive calls on $(n/2) \times (n/2)$ matrices. The time spent outside of the recursive calls is $O(n^2)$. So the running time $T(n)$ satisfies $T(n) = 8T(n/2) + O(n^2)$. Using the recursion-tree method, the total time is proportional to

$$T(n) = \sum_{i=0}^{\log_2 n} (8^i) \times (n/2^i)^2 = n^2 \sum_{i=0}^{\log_2 n} 2^i = n^2 \Theta(2^{\log_2 n}) = \Theta(n^3).$$

No better!

Strassen's Algorithm. To improve the running time, we'll reduce the number of recursive calls from eight to seven. To do this, replace Steps 3 and 4.1 with the following steps. First, compute the following seven $\frac{n}{2} \times \frac{n}{2}$ matrices, using seven recursive matrix multiplications:

$$\begin{aligned} M_1 &= (A_{11} + A_{22}) \times (B_{11} + B_{22}) & M_2 &= (A_{21} + A_{22}) \times B_{11} \\ M_3 &= A_{11} \times (B_{12} - B_{22}) & M_4 &= A_{22} \times (B_{21} - B_{11}) \\ M_5 &= (A_{11} + A_{12}) \times B_{22} & M_6 &= (A_{21} - A_{11}) \times (B_{11} + B_{12}) \\ M_7 &= (A_{12} - A_{22}) \times (B_{21} + B_{22}) \end{aligned}$$

Strassen figured out that those seven matrices satisfy the following equations:

Lemma 4.2 (Strassen).

$$\begin{aligned} C_{11} &= M_1 + M_4 - M_5 + M_7 & C_{12} &= M_3 + M_5 \\ C_{21} &= M_2 + M_4 & C_{22} &= M_1 - M_2 + M_3 + M_6 \end{aligned}$$

We don't give the proof, but you can check each equation if you like.

Now we modify the Algorithm in Step 4.1 so it computes each C_{ij} using the four equations above in time $O(n^2)$. This gives Strassen's algorithm.

Correctness. Correctness of the algorithm follows by the lemma (and induction on n).

Run time. The time now $T(n)$ now satisfies $T(n) = 7T(n/2) + O(n^2)$, so the total time is proportional to

$$\sum_{i=0}^{\log_2 n} (7^i) \times (n/2^i)^2 = n^2 \sum_{i=0}^{\log_2 n} (7/4)^i = n^2 \Theta((7/4)^{\log_2 n}) = \Theta(n^{\log_2 7}) = \Theta(n^{2.80...}).$$

(Above we use $x^{\log y} = y^{\log x}$ and $\log_2(7/4) = \log_2(7) - 2$.) We have the following result:

Theorem 4.4 (Strassen). *There is an $O(n^{\log_2 7})$ -time algorithm for multiplying $n \times n$ matrices.*

4.5.1 Exercise

Exercise 4.8. (*Strassen variation*) The standard way of multiply two 3×3 matrices uses 27 multiplications. Professor Bo Zo found a way to do it using 24 multiplications and a constant number of additions (without assuming that multiplication is commutative). He uses this observation as the basis for a variant of Strassen's algorithm that makes 24 recursive calls, each multiplying two matrices of size $n/3 \times n/3$. Does he get a faster algorithm? Show your work!

4.5.2 External sources on fast matrix multiplication

- CLRS Chapter 4.2 (includes exercises), and an MIT lecture video:
- <https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-introduction-to-algorithms-sma-5503-fall-2005/video-lectures/lecture-3-divide-and-conquer-strassen-fibonacci-polynomial-multiplication/>

4.6 An algorithm for selecting the k th smallest

Problem definition. The Selection problem is defined as follows:

input: array $A[1..n]$ of numbers, integer k in $\{1, \dots, n\}$.

output: the k th smallest number in the array

When $k = 1$, the output is $\min_i A[i]$. When $k = n$, the output is $\max_i A[i]$. When $k = n/2$, the output is the median.

Algorithm. One simple way to solve the problem is to sort A , then return $A[k]$. This takes $\Theta(n \log n)$ time. Next we present a faster, linear-time algorithm. We assume for simplicity that all elements in A are distinct.

```

select( $A[1..n], k$ )
1. if  $n \leq 15$ , sort  $A$  in place and return  $A[k]$  (since  $n \leq 15$ , this takes constant time)
2. choose pivot  $p \in A[1..n]$  as described in the text below
3. partition  $A$  around  $p$  into  $A[1..i-1]$ ,  $A[i] = p$ , and  $A[i+1..n]$ 
   (so that elements smaller than  $p$  are in  $A[1..i-1]$  and elements larger than  $p$  are in  $A[i+1..n]$ )
4. if  $k = i$ : return  $A[i]$ 
5. else if  $k < i$ : return select( $A[1..i-1], k$ ) (discard  $A[i..n]$ )
6. else ( $k > i$ ): return select( $A[i+1..n], k-i$ ) (discard  $A[1..i]$ )

```

Correctness. We omit the proof. The idea is that if $k < i$, then the k th smallest in $A[1..n]$ (after Step 3) is the k th smallest in $A[1..i-1]$, and if $k > i$ it is the $k-i$ th smallest in $A[i+1..n]$. Correctness follows from this and induction.

Run time. Step 3 (the partition step) takes linear time. After that, **select** discards some elements of A and calls itself recursively on the rest. If the pivot is chosen badly, this could lead to quadratic running time. For example, if $k = 1$ and the pivot is chosen as the maximum element each time, then each recursive call would discard just one element, so there would be $n-1$ recursive calls, with the i th call taking time proportional to $n-i$, yielding total time $\Theta(\sum_{i=0}^n n-i) = \Theta(n^2)$.

To avoid this, the algorithm uses a complicated scheme to choose the pivot in Step 2. It first partitions the array A into $n/5$ columns, each with 5 elements, as shown below (the diagram assumes n is a multiple of 2 and 5, that is, a multiple of 10):

$A[1]$	$A[6]$		$A[n/2-4]$		$A[n-4]$
$A[2]$	$A[7]$		$A[n/2-3]$		$A[n-3]$
$A[3]$	$A[8]$	$\dots$	$A[n/2-2]$	$\dots$	$A[n-2]$
$A[4]$	$A[9]$		$A[n/2-1]$		$A[n-1]$
$A[5]$	$A[10]$		$A[n/2]$		$A[n]$

$\underbrace{\hspace{15em}}_{n/5}$

It then finds the median within each individual column of 5. This takes constant time for each column, so the total time for this step is linear. There are $n/5$ of these medians. It then *recursively computes the median of these $n/5$ medians, and chooses the pivot p to be the result*. This completes the description of the algorithm. To analyze the run time, we use the following lemma:

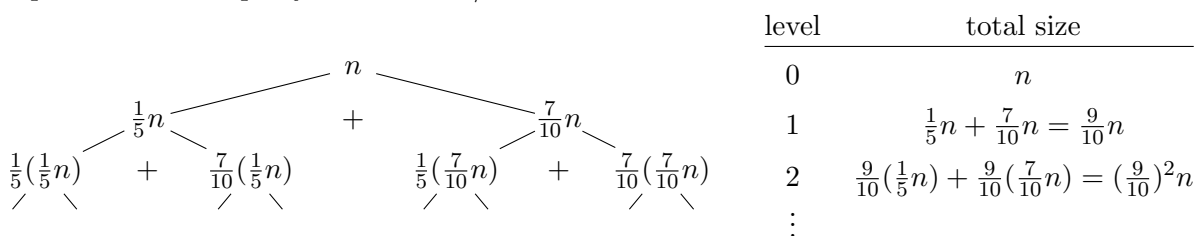
Lemma 4.3. *The rank i of the pivot satisfies $3n/10 \leq i \leq 7n/10$, so the recursive call in Line 5 or 6 has at most $7n/10$ elements.¹*

Proof (short form). The pivot p , being the median of the $n/5$ column medians, is larger than $n/10$ of those medians. And each such median is in turn larger than 2 elements in its column. So p is larger than at least $3n/10$ other elements. By similar reasoning, p is smaller than at least $3n/10$ other elements. So at least $3n/10$ elements are discarded in the recursive call. $\square$

Define $T(n)$ to be the worst-case running time of **select** on n elements. The above reasoning gives (for some constant $c > 0$) the inequality

$$T(n) \leq c \cdot n + T(n/5) + T(7n/10).$$

The next crucial observation is that sum of the subproblem sizes is $n/5 + 7n/10 = 9n/10$, so the total problem size drops by a factor of $9/10$:



This happens at each level of the recursion tree. For each subproblem of any size n' , its two subproblems on the level below have combined size at most $\frac{9}{10}n'$, so the total size at each level is $\frac{9}{10}$ th of the total size at the level above. So, by induction, the sum of the problem sizes in level i is at most $(9/10)^i n$. Since the *work* for each subproblem is linear in the subproblem size, the total work in level i is also $O((9/10)^i n)$. So the total work is $O(\sum_{i=0}^{O(\log n)} (9/10)^i n) = O(n)$ (using here that the sum is geometric).

Correcting for rounding issues. The previous discussion is wrong when n is not a multiple of 10, because it ignores rounding issues. For completeness, we give a more careful, correct analysis, starting with the choice of the pivot. When n is not a multiple of 10, to choose the pivot p , the algorithm partitions into columns of 5 something like this:

1	2	...	$m = \lfloor n/10 \rfloor$	...	$\lceil n/5 \rceil$
$A[1]$	$A[6]$	...	$A[5m - 4]$	...	empty
$A[2]$	$A[7]$		$A[5m - 3]$	...	empty
$A[3]$	$A[8]$	...	$A[5m - 2]$	...	$A[n - 1]$
$A[4]$	$A[9]$	...	$A[5m - 1]$	...	$A[n]$
$A[5]$	$A[10]$	...	$A[5m]$	...	empty

There are $\lceil n/5 \rceil$ columns. The last column has between 1 and 5 elements. After finding the $\lceil n/5 \rceil$ medians of those columns in $\Theta(n)$ time the algorithm recursively finds the median of those medians—specifically, the m th smallest, where $m = \lfloor n/10 \rfloor$, and takes that to be the pivot p .

In the case $k > i$ in Step 6 of the algorithm, it discards p and all elements smaller than p . This will be at least $3m - 2$ elements (not necessarily $3m$, because the last column may have fewer than three elements). Recalling that $m = \lfloor n/10 \rfloor$, in this case at least $3\lfloor n/10 \rfloor - 2$ elements are

¹This observation and some of the following discussion is not necessarily true when n is not a multiple of 10, because we are ignoring rounding issues. See the discussion at the end of the note.

discarded. In the other case ($k < i$ in Step 5), the algorithm discards p and all elements larger than p . The number of such elements is at least

$$\begin{aligned} 3(\lceil n/5 \rceil - m + 1) - 2 &= 1 + 3(\lceil n/5 \rceil - m) \\ &= 1 + 3(\lceil n/5 \rceil - \lfloor n/10 \rfloor) \\ &\geq 1 + 3(n/5 - n/10) \\ &= 1 + 3n/10 \geq 3\lfloor n/10 \rfloor. \end{aligned}$$

Hence the number of remaining elements in the recursive call is at most $n - 3\lfloor n/10 \rfloor + 2$, and

$$T(n) \leq n + T(\lceil n/5 \rceil) + T(n - 3\lfloor n/10 \rfloor + 2).$$

Lemma 4.4. $T(n) = O(n)$.

Proof. We give a proof by induction that $T(n) \leq cn$ for some (large) constant c .

The base cases $n \leq 80$ will hold as long as $c \geq \max\{T(n)/n : n \leq 80\}$.

For $n > 80$, we have

$$\begin{aligned} T(n) &\leq n + T(\lceil n/5 \rceil) + T(n - 3\lfloor n/10 \rfloor + 2) && \text{(see above)} \\ &\leq n + c\lceil n/5 \rceil + c(n - 3\lfloor n/10 \rfloor + 2) && \text{(by induction, using that } n > 80\text{)} \\ &< n + c(n/5 + 1 + n - 3(n/10 - 1) + 2) && \text{(using } \lceil x \rceil < x + 1 \text{ and } \lfloor x \rfloor > x - 1\text{)} \\ &= cn(1/c + 6/n + 9/10) && \text{(algebra)} \\ &\leq cn(1/c + 3/40 + 9/10) && \text{(using } n \geq 80\text{)} \\ &\leq cn && \text{(as long as } c \geq 40\text{)}. \end{aligned}$$

It follows that $T(n) \leq cn$, where $c = \max(40, \max\{T(n)/n : n \leq 80\})$. □

We have proved the following theorem:

Theorem 4.5. *There is a (deterministic) linear-time algorithm for Selection (and Median Finding).*

Above we assumed that all numbers in the array are distinct. Exercise 4.9 asks you to consider what happens if that's not the case.

Python code: [divide_and_conquer/selection.py](#) (Section 4.9).

4.6.1 Exercise

Exercise 4.9. (*Selection algorithm*) In analyzing the Selection algorithm in Section 4.6, we assumed that each number was distinct. On inputs where that's not the case, can the algorithm give an incorrect output, or fail to run in time $O(n)$? If so, how would you fix the algorithm to deal with such inputs?

4.6.2 External sources on deterministic selection

- [JE](#) Chapter 1, Section 1.8 (& exercises); [CLRS](#) Chapter 9.3 (& exercises); MIT lecture video: <https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-introduction-to-algorithms-sma-5503-fall-2005/video-lectures/lecture-6-order-statistics-median/>
- [DPV](#) Ch. 2.4 and [KT](#) Ch. 1.5 discuss a simpler randomized algorithm (random pivot)

4.7 An incorrect algorithm for finding a 2D Local Maximum

4.7.1 1D Local Maximum

For warm-up, we first consider the 1-dimensional problem.

Problem definition. The 1D Local Maximum problem is defined as follows:

input: An array $A[1..n]$ of numbers.

output: A *local maximum*, which is a cell $A[i]$ such that every neighboring cell in the array has a value that is at most $A[i]$. (The neighboring cells are those among $A[i - 1]$ and $A[i + 1]$ that are actually present in A .)

Algorithm. There is a linear-time (that is, $O(n)$ -time) algorithm: just check each cell.

Here is an $O(\log n)$ -time algorithm using divide and conquer:

```

local-max-1D( $A[1..n]$ )
1. if  $n = 1$ , return  $A[1]$ 
2. let  $m = \lceil n/2 \rceil$ 
3. if  $A[m]$  is a local maximum of  $A[1..n]$ , return  $A[m]$ 
4. if  $A[m + 1] > A[m]$ , return local-max-1D( $A[m + 1..n]$ )
5. else ( $A[m - 1] > A[m]$ ) return local-max-1D( $A[1..m - 1]$ )

```

Running time. The running time $T(n)$ satisfies $T(n) = O(1) + T(n/2)$. So $T(n) = O(\log n)$.

Correctness.

Lemma 4.5. *The 1D algorithm is correct.*

Proof (long form).

1. The proof is by induction on n .
2. Let $P(n)$ be the predicate “the algorithm is correct on all inputs of size n ”.
3. By inspection of algorithm Line 1 and the definition of local maximum, $P(1)$ holds.
 - 4.1. Consider any execution of the algorithm on any input A with $n \geq 2$. Assume $P(i)$ for $1 \leq i < n$.
 - 4.2.1. Case 1. Suppose $A[m]$ is a local maximum of $A[1..n]$.
 - 4.2.2. By inspection of algorithm Line 3, the algorithm is correct on this input.
 - 4.3.1. Case 2. Otherwise, suppose $A[m + 1] > A[m]$.
 - 4.3.2. Then any local maximum of $A[m + 1..n]$ is also a local maximum of $A[1..n]$.
(Note: in the case that the local max is $A[m + 1]$, this follows from $A[m + 1] > A[m]$.)
 - 4.3.3. So (by inspection of Line 4), the algorithm is correct on this input.
 - 4.4.1. Case 3. In the remaining case, $A[m - 1] > A[m]$.
 - 4.4.2. Then any local maximum of $A[1..m - 1]$ is also a local maximum of $A[1..n]$.
 - 4.4.3. So (by inspection of Line 5), the algorithm is correct on this input.
 - 4.5. One of the three cases holds, so the algorithm is correct in this input.
5. By Block 4, $P(n)$ holds as long as $P(i)$ holds for $1 \leq i < n$.
6. Since $P(1)$ holds, by induction $P(n)$ holds for all $n \geq 1$. □

4.7.2 2D Local Maximum

Next we consider two dimensions.

Problem definition. The 2D Local Maximum problem is defined as follows:

input: An $n \times n$ array $A[1..n, 1..n]$ of numbers.

output: A *local maximum*, which is a cell $A[i, j]$ such that every neighboring cell in the array has a value that is at most $A[i, j]$. (The neighboring cells are those among $A[i, j + 1]$, $A[i, j - 1]$, $A[i - 1, j]$, $A[i + 1, j]$ that are actually present in A .)

Note that the neighbors of a local max $A[i, j]$ can have the *same* value as $A[i, j]$. For example, in the array to the right, the local maxima are in bold.

0	0	0	0	0	0	0
0	1	1	0	3	4	0
3	2	0	1	3	1	0
2	5	0	3	0	1	0
4	1	0	0	1	3	0
3	3	0	1	2	1	0
0	0	0	0	0	0	0

Algorithm. There is a linear-time (that is, $O(n^2)$ -time) algorithm: just check each cell. We want a faster algorithm. We propose the following algorithm. Assume for simplicity that n is of the form $2^i - 1$ for some integer i . (The “cross” X in Step 2 below is shown in the array above.)

- We assume $n \geq 1$.

local-max($A[1..n, 1..n]$)

 1. Let X be the set consisting of the $2n - 1$ cells lying in column $(n + 1)/2$ or row $(n + 1)/2$ of A .
 2. Check each cell in X to see if it is a local maximum in A . If so, return such a cell. Otherwise:
 3. Let $A[i, j]$ be any cell in X with maximum value (among cells in X).
 4. Let $A[i', j']$ be a neighbor of $A[i, j]$ in A with $A[i', j'] > A[i, j]$. (It exists, as $A[i, j]$ is not a local max.)
 5. Partition $A \setminus X$ (the array A , minus the cells in X) around X into its four $(n - 1)/2 \times (n - 1)/2$ quadrants—the upper left, upper right, lower left, and lower right—in the natural way.
 6. Among those, let A' be the quadrant containing the cell $A[i', j']$.
 7. Recursively compute a local maximum of the subarray A' , and return that cell.

Run time. Recall that the input has size $n \times n = n^2$.

Lemma 4.6. *The algorithm above can be implemented to run in $O(n)$ time.*

Proof sketch. Let $T(n)$ denote the worst-case run time on an $n \times n$ input array. Then $T(1) = O(1)$. Consider running the algorithm on any input with $n \geq 2$. The cross X has $O(n)$ cells, so Lines 2–7 can be implemented to take $O(n)$ time. So $T(n) = T((n - 1)/2) + O(n)$. So $T(n) = O(n)$. $\square$

Correctness. Before we prove correctness, we prove the following utility lemma:

Lemma 4.7. *During any execution of the algorithm that reaches Line 6, the quadrant A' defined there contains a local maximum of A .*

Proof (long form).

1. Consider the following process.

- (i) Put a token on the cell $A[i', j']$.
 - (ii) While the token’s cell is not a local maximum in A do:
 - (iii) Move the token to a neighboring cell with larger value.

2. Each cell in X is no larger than $A[i, j]$ (by Line 3 of the algorithm), and $A[i, j] < A[i', j']$.
3. So each cell in X is smaller than $A[i', j']$.
4. So by inspection of the process it never moves the token into X . The token stays in A' .
5. The process must stop, because each move increases the value of the cell with the token.
6. So the token ends on a cell in A' that is also a local maximum of A . □

Now we prove correctness...

Lemma 4.8 (Wrong!). *The algorithm is correct.*

Proof (long form).

1. The proof is by induction on n .
2. Let $P(n)$ be the predicate “the algorithm is correct on all $n \times n$ inputs”.
3. For any input with $n = 1$, the cross X contains the entire array, so Line 2 returns a local max.
4. So $P(1)$ holds.
- 5.1. Consider any execution of the algorithm on any input A with $n \geq 2$. Assume $P(i)$ for $1 \leq i < n$.
- 5.2.1. Case 1. Suppose the algorithm returns in Line 2.
- 5.2.2. By inspection of Line 2, it returns a local max of A .
- 5.3.1. Case 2. Otherwise the algorithm returns in Line 7.
- 5.3.2. The quadrant A' contains a local max of A (by Lemma 4.7).
- 5.3.3. By Step 5.1, $P((n - 1)/2)$ holds, so the recursive call (on A') returns a local max of A' .
- 5.3.4. By the previous two steps, the algorithm (Line 7) returns a local max of A .
- 5.4. So, in either case, the algorithm returns a local max of A . That is, it is correct on input A .
6. By Block 5, for any $n \geq 2$, if $P(i)$ holds for $1 \leq i < n$, then $P(n)$ holds.
7. By Step 4, $P(1)$ holds.
8. By the previous two steps and induction, $P(n)$ holds for all $n \geq 1$.
9. That is, the algorithm is correct. □

Python code: [divide_and_conquer/local_maximum.py](#) (Section 4.9).

4.7.3 Exercise

Exercise 4.10. (*Local Max proof*) The proof of Lemma 4.8 is wrong.

- (a) In the proof of correctness (Lemma 4.8), where precisely is the error? More specifically, what is the *first* step in the long-form proof that asserts a statement that is not necessarily true? What is the mistake in the reasoning in that line?
- (b) Give an input (with $n \leq 7$) on which the 2D Local Max algorithm in Section 4.7 fails. Describe the execution of the algorithm on that input. We encourage you to verify your answer using the Python implementation in [divide_and_conquer/local_maximum.py](#).

4.7.4 External sources on finding a local maximum

- MIT (“Peak Finding”):
slides: <https://courses.csail.mit.edu/6.006/spring11/lectures/lec02.pdf>
video and lecture notes:

- <https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/lecture-videos/lecture-1-algorithmic-thinking-peak-finding/>

4.8 Additional exercises on Divide and Conquer

Each previous section of this note also contains at least one exercise.

External exercises with solutions

- [KT](#) Page 242, Solved Exercises 1 and 2 (divide-and-conquer algorithms)

Exercise without solution

Exercise 4.11. (*Recursive Bus Routes*) You have been commissioned to design a new bus system that will run along Huntington Avenue. The bus system must provide service to n stops (numbered $1, 2, \dots, n$) on the eastbound route. Commuters may begin their trip at any stop i and end at any stop j where $1 \leq i < j \leq n$. Here are some naive ways to design the system:

- You could have a bus run from stop 1 to stop n , stopping at every stop in between. The system would be cheap because it would only require $n - 1$ route segments, $\{1 \rightarrow 2, 2 \rightarrow 3, \dots, (n - 1) \rightarrow n\}$, for the entire system. However, a person traveling from stop 1 to stop n would have to wait while the bus makes $n - 1$ stops.
- You could have, for every pair of stops i, j with $i < j$, a special express bus from i to j . No commuter would ever have to make more than one stop. However, this system would be expensive as it requires $\binom{n}{2} = \Theta(n^2)$ route segments, $\{i \rightarrow j : 1 \leq i < j \leq n\}$.

Using divide and conquer, you will find a compromise—a set $S(n)$ of $\Theta(n \log n)$ route segments, with the property that a user can get from any stop i to any stop $j > i$ using at most two segments in $S(n)$. That is, *for every i and j with $1 \leq i < j \leq n$, either the (direct) route segment $i \rightarrow j$ is in $S(n)$, or, for some m with $i < m < j$, the set $S(n)$ contains the two segments $i \rightarrow m$ and $m \rightarrow j$.*

- For each base case $n \in \{1, 2\}$, describe a set $S(n)$ with these properties, and having size $n - 1$.
- For each $n > 2$, give a definition for a set $T(n)$ of at most n route segments such that, for every pair of stops (i, j) where i is in the first half ($1 \leq i \leq \lceil n/2 \rceil$) and j is in the second half ($\lceil n/2 \rceil < j \leq n$), a commuter can get from i to j using one or two segments in $T(n)$.
- For each $n > 2$, give a definition for a full set $S(n)$ as in the problem statement. (Extend the partial solution $T(n)$ by recursing twice, once on the first half and again on the second half.)
- Let $N(n) = |S(n)|$ (the number of segments in $S(n)$). Give a recurrence relation for $N(n)$. Use the recursion-tree method to solve it. Show your work as described for Exercise 4.2.

External exercises without solutions²

- [KT](#) Exercises 1–7 (Page 246). [DPV](#) 2.1–2.32 (Page 83). LLM 22.1–22.4 (Page 1017).
- [CLRS](#) Exercises in Section 4.1 (Page 74), and Problems 4–1 through 4–6 (Page 107).
- [JE](#) Chapter 1 Exercises 1–41 (Page 44)

External sources on Divide and Conquer: See the previous sections, especially Section 4.2.2.

²As mentioned in [Lecture Note 1](#), solutions for exercises in [KT](#) and some in [CLRS](#) can be found online.

4.9 Python code

This section lists the Python code for this lecture note. Click on a file's name to open the file on GitHub. Copying code from GitHub is more reliable than copying it from this PDF, since some PDF viewers drop the leading spaces (the indentation) of copied lines.

[divide_and_conquer/mergesort.py](#)

```
#!/usr/bin/env python3

from collections.abc import Iterator
from typing import Any, Protocol

class Ordered(Protocol):
    """a type whose values can be compared with <"""

    def __lt__(self, other: Any, /) -> bool: ...

def mergesort[T: Ordered](A: list[T]) -> list[T]:
    n = len(A)

    if n <= 1:
        return A

    m = int((n + 1) / 2)
    L = mergesort(A[:m])
    R = mergesort(A[m:])
    return list(merged(L, R))

def merged[T: Ordered](L: list[T], R: list[T]) -> Iterator[T]:
    i = j = 0
    while i < len(L) and j < len(R):
        if L[i] < R[j]:
            yield L[i]
            i += 1
        else:
            yield R[j]
            j += 1
    yield from L[i:] + R[j:]

def test(n: int = 1000) -> None:
    import random

    A = [random.randrange(n) for _ in range(n)]
    B = mergesort(A)
    C = sorted(A)
    assert B == C

for _ in range(50):
    test(200)

print("test passed")
```

divide_and_conquer/closest_pair.py

```
#!/usr/bin/env python3

from collections import namedtuple
from collections.abc import Iterable
from math import inf

def minf(*args: Iterable[float]) -> float:
    return min(*args, default=inf)

class Point(namedtuple("Point_", "x y")):
    def distance_squared(self, other: "Point") -> float:
        dx, dy = self.x - other.x, self.y - other.y
        return dx**2 + dy**2 # **0.5

def closest_pair(points: Iterable[tuple[float, float]]) -> float:
    """Return the square of the minimum Euclidean distance between any two of the given points"""

    points = list(points)
    P = set(Point(*p) for p in points)
    if len(P) < len(points): # two equal points
        return 0.0
    Px = sorted(P, key=lambda p: p.x)
    Py = sorted(P, key=lambda p: p.y)

    def cp(Px: list[Point], Py: list[Point]) -> float:
        if len(Px) <= 1:
            return inf

        mid = int(len(Px) / 2)
        median_x = Px[mid].x

        Lx, Rx = Px[:mid], Px[mid:]
        L = set(Lx)
        Ly, Ry = [p for p in Py if p in L], [p for p in Py if p not in L]

        delta_L, delta_R = cp(Lx, Ly), cp(Rx, Ry)
        delta = min(delta_L, delta_R)

        My = [p for p in Py if (median_x - p.x) ** 2 < delta]
        delta_M = minf(p.distance_squared(q) for i, p in enumerate(My) for q in My[i + 1 : i + 9])
        return min(delta, delta_M)

    return cp(Px, Py)

def test_closest_pair(n: int = 30) -> None:
    import random

    P = [(2 * i, 2 * j) for i in range(n) for j in range(n)]
    (a, b) = random.choice(P)
    P += [(a + 1, b + 1)]

    assert abs(closest_pair(P) - 2) <= 1e-100

def test_closest_pair_small(n: int = 8) -> None:
```

```

    """Compare with brute force on a few random points."""
    import random
    from itertools import combinations

    P = [(random.random(), random.random()) for _ in range(random.randint(2, n))]
    brute_force = min(Point(*p).distance_squared(Point(*q)) for p, q in combinations(P, 2))
    assert abs(closest_pair(P) - brute_force) <= 1e-9, P

if __name__ == "__main__":
    for _ in range(10):
        test_closest_pair()
    for _ in range(200):
        test_closest_pair_small()

    print("test passed")

```

[divide_and_conquer/selection.py](#)

```

#!/usr/bin/env python3

from typing import Any, Protocol

class Ordered(Protocol):
    """a type whose values can be compared with < and >"""

    def __lt__(self, other: Any, /) -> bool: ...

    def __gt__(self, other: Any, /) -> bool: ...

def median[T: Ordered](A: list[T]) -> T:
    """return the median of the values in A"""

    return select(A, int((len(A) + 1) / 2))

def select[T: Ordered](A: list[T], k: int, randomized: bool = False) -> T:
    """return the kth smallest of the values in A"""

    n = len(A)
    assert 1 <= k <= n, (k, n)

    if randomized:
        if n == 1:
            return A[0]

        import random

        pivot = random.choice(A)

    else:
        if n < 15:
            return sorted(A)[k - 1]

        medians = [median(A[i : i + 5]) for i in range(0, n, 5)]
        pivot = median(medians)

    A1 = [a for a in A if a < pivot]

```

```

    A2 = [a for a in A if a == pivot]
    A3 = [a for a in A if a > pivot]

    if k <= len(A1):
        return select(A1, k)
    k -= len(A1)

    if k <= len(A2):
        return pivot
    k -= len(A2)

    return select(A3, k)

def test(n: int = 1000) -> None:
    import random

    A = [random.randrange(n) for _ in range(n)]
    k = 1 + random.randrange(n)
    kth = select(A, k)
    kth_r = select(A, k, randomized=True)

    B = sorted(A)
    assert B[k - 1] == kth == kth_r

for _ in range(50):
    test(200)

print("test passed")

```

divide_and_conquer/local_maximum.py

```

#!/usr/bin/env python3

class Subarray:
    def __init__(self, array: list[list[float]], i_range: range, j_range: range) -> None:
        """array is a list of lists; i_range and j_range are ranges"""
        self.array = array
        self.i_range = i_range
        self.j_range = j_range

    def find_nbr_larger_than(self, i: int, j: int) -> tuple[int, int] | None:
        """
        return neighboring cell in subarray with larger value,
        or None if there is no such cell
        """
        for i2, j2 in ((i - 1, j), (i + 1, j), (i, j + 1), (i, j - 1)):
            if i2 in self.i_range and j2 in self.j_range and self.array[i2][j2] > self.array[i][j]:
                return i2, j2
        return None

    def find_local_max(self) -> tuple[int, int]:
        def partition(range_: range) -> tuple[range, int, range]:
            mid = int(len(range_) / 2)
            return range_[[:mid], range_[mid], range_[mid + 1 :]]

        top, mid_i, bottom = partition(self.i_range)
        left, mid_j, right = partition(self.j_range)

```

```

        middle_row = set((mid_i, j) for j in self.j_range)
        middle_col = set((i, mid_j) for i in self.i_range)
        cross = sorted(middle_row | middle_col)

        for i, j in cross:
            if self.find_nbr_larger_than(i, j) is None:
                return i, j

        i, j = max(cross, key=lambda ij: self.array[ij[0]][ij[1]])

        i2, j2 = self.find_nbr_larger_than(i, j)  # type: ignore[misc] # not None, see loop above
        i2_range = top if i2 in top else bottom
        j2_range = left if j2 in left else right

        return Subarray(self.array, i2_range, j2_range).find_local_max()

    @staticmethod
    def from_string(input_: str) -> "Subarray":
        """input_: array as string.  output: array as Subarray"""
        try:
            array = [
                [float(x.strip()) for x in line.strip().split() if x.strip()]
                for line in input_.split("\n")
                if line.strip()
            ]
            if not array:
                raise ValueError
            m, n = len(array), len(array[0])
            if not (m and n and all(len(row) == n for row in array)):
                raise ValueError
            return Subarray(array, range(m), range(n))

        except ValueError as e:
            raise AssertionError("MALFORMED INPUT") from e

if __name__ == "__main__":

    def main(input_: str) -> None:
        A = Subarray.from_string(input_)
        i, j = A.find_local_max()
        if A.find_nbr_larger_than(i, j) is None:
            print("local max:", i, j, "; value:", A.array[i][j])
        else:
            print("NOT LOCAL MAX:", i, j, "; value:", A.array[i][j])

    input_1 = """
    0 1 3
    3 2 3
    2 5 0
    """

    main(input_1)

    input_2 = """
    3 2 2
    2 2 1
    2 1 0
    """

```

```
main(input_2)
```