0 Introduction

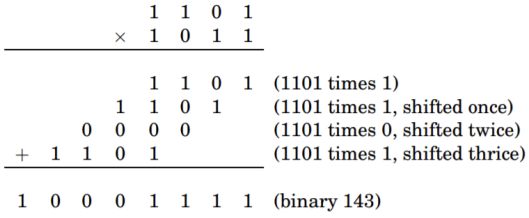
Divide and conquer is the first algorithmic paradigm we will cover in this course; that is, a framework of algorithms that follow the same structure. Divide and conquer is characterized by dividing the problem into some number of subproblems that are each solved recursively, and combining the results intelligently to retrieve the final answer. In this note, we will explore a few algorithms of this nature. To begin, we will examine multiplication, where applying divide and conquer with a neat trick allows us to beat $O(n^2)$.[1]

1 Multiplication

1.1 Grade School Algorithm

To recap, the grade school algorithm for multiplying two numbers x and y is to separately multiply x by each digit of y, left-shift these results by the appropriate amounts, then add them up. For example, to calculate $13 * 11$ in binary, which is $1101 * 1011$, we would perform the following:

Figure 1: Grade School Multiplication. Source: DPV, pg. 24



Bit complexity: we compute n rows, each row having up to $2n$ bits, as each row is left-shifted up to n times. We then do the addition step row by row (not by columns!): we add row 1 (00001101) to row 2 (00011010), then add the result to row 3 (00000000), and so on until we reach the last row.[2] As adding one row to another takes $O(n)$ time and there are $n - 1$ additions to perform, the total time taken to add these rows is $(n - 1) * O(n)$, which is $O(n^2)$.

[1] Kolmogorov, among other mathematicians, conjectured multiplication would be $\Omega(n^2)$. The fastest algorithm now is $O(n \log n)$ discovered in 2019, inspired by FFT, which we'll learn in the next note.

[2] We do the addition row by row since for the purposes of this class, computers can only add two numbers at a time. Therefore, to compute the sum of three numbers, we would have to add the first two, then add the third number to the result.

1.2 Naive Divide and Conquer

A number can be decomposed as the sum of its left and right half: $x = x_L * 2^{n/2} + x_R$ where n is the number of digits of x, in binary (e.g $x = 1011$ is equivalent to $10*100+11$, as 100 in binary is 2^2 in decimal). Let's use this fact to devise a divide and conquer algorithm: we split both x and y into two halves, recursively compute the products of these smaller numbers, and use those to construct our answer. More explicitly, we decompose x and y each into two halves as $x = x_L * 2^{n/2} + x_R$ and $y = y_L * 2^{n/2} + y_R$, then we construct our answer $x * y = x_L * y_L * 2^n + (x_L * y_R + x_R * y_L) * 2^{n/2} + x_R * y_R$, where we solve $x_L * y_L$, $x_L * y_R$, $x_R * y_L$, and $x_R * y_R$ recursively.[3]

Pseudocode:

Algorithm 1 Naive Divide and Conquer Multiplication

```
1:  function MULTIPLY(x, y, n)
2:      if n = 1 then
3:          if x = 1 and y = 1 then
4:              return 1
5:          else
6:              return 0
7:      x_L ← x/2^{n/2}
8:      x_R ← x % 2^{n/2}
9:      y_L ← y/2^{n/2}
10:     y_R ← y % 2^{n/2}
11:     m_LL ← MULTIPLY(x_L, y_L, n/2)
12:     m_LR ← MULTIPLY(x_L, y_R, n/2)
13:     m_RL ← MULTIPLY(x_R, y_L, n/2)
14:     m_RR ← MULTIPLY(x_R, y_R, n/2)
15:     return 2^n * m_LL + 2^{n/2} * (m_LR + m_RL) + m_RR
```

Correctness and bit complexity: The algorithm is correct by induction on n, simply using the distributive property for the inductive step to justify our expression for $x * y$. As for the bit complexity, let's denote $T(n)$ as the overall running time on n-bit inputs. The four $n/2$-bit multiplications are computed recursively, and constructing the answer involves 3 additions that we know take $O(n)$ to calculate, as well as two multiplications by powers of two that are easily calculated by just left-shifting, which is also $O(n)$. So, we get the recurrence relation $T(n) = 4T(n/2) + O(n)$. As we'll see later, this is equivalent to $O(n^2)$, the same as the grade school algorithm.

[3] If n is odd, we can simply left pad with a zero.

1.3 Karatsuba's Algorithm

However, Karatsuba made a clever observation[4] using the distributive property: $x_L * y_R + x_R * y_L = (x_L + x_R) * (y_L + y_R) - x_L * y_L - x_R * y_R$. Now, to construct our expression for $x * y$, we only need to perform three multiplications instead of four - we'll still calculate $x_L * y_L$ and $x_R * y_R$, but instead of also calculating $x_L * y_R$ and $x_R * y_L$, we'll calculate $(x_L + x_R) * (y_L + y_R) - x_L * y_L - x_R * y_R$ from the observation. (Note we've already calculated $x_L * y_L$ and $x_R * y_R$, so there's only one new multiplication here: $(x_L + x_R) * (y_L + y_R)$). There is a slight drawback in that we introduce some additions and subtractions, but those are still done in linear time. The reduction of a multiplication is much more significant: our resulting recurrence relation is $T(n) = 3T(n/2) + O(n)$. Each recursive call now produces one less subproblem, and this effect compounds. This reduces the runtime from $O(n^2)$ to $O(n^{1.59})$!

Pseudocode:

Algorithm 2 Karatsuba's Algorithm

```
1:  function MULTIPLY(x, y, n)
2:      if n = 1 then
3:          if x = 1 and y = 1 then
4:              return 1
5:          else
6:              return 0
7:      x_L ← x/2^{n/2}
8:      x_R ← x % 2^{n/2}
9:      y_L ← y/2^{n/2}
10:     y_R ← y % 2^{n/2}
11:     m_L ← MULTIPLY(x_L, y_L, n/2)
12:     m_C ← MULTIPLY(x_L + x_R, y_L + y_R, n/2)
13:     m_R ← MULTIPLY(x_R, y_R, n/2)
14:     return 2^n * m_LL + 2^{n/2} * (m_C − m_L − m_R) + m_RR
```

2 Master Theorem

2.1 Time Complexity Analysis

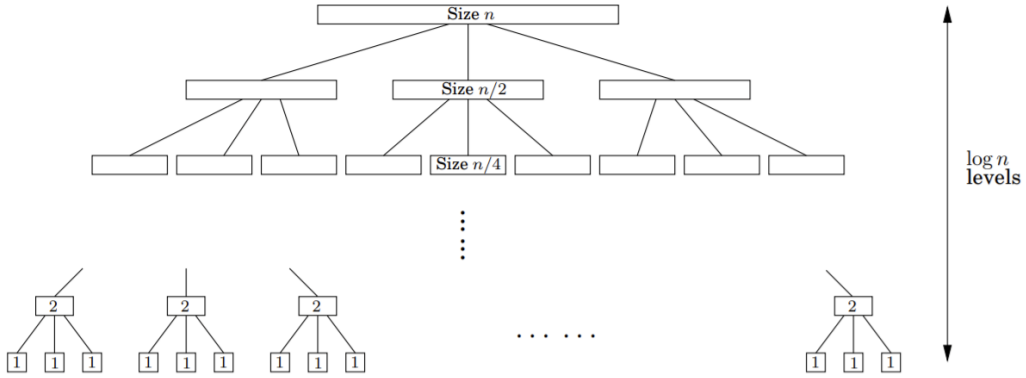
How do we solve these recurrences? In the example above, we can derive the running time of $O(n^{1.59})$ by examining the pattern of the recursive calls:

At each level of recursion, the subproblems are halved in size until they reach 1,[5]

[4] Gauss made the same observation centuries earlier when multiplying complex numbers.

[5] In practice, the base case is not 1, but at 16 or 32 bits depending on the processor, as they can multiply these numbers in a single operation.

Figure 2: Recursive call structure. Source: DPV, pg. 58



therefore the height of the tree is $\log_2 n$. The branching factor is 3, and the result is that at depth k in the tree there are 3^k subproblems, each of size $n/2^k$.

Each subproblem (not subtree) takes a linear amount of time, correlating with the size of the subproblem. Therefore a subproblem of size $n/2^k$ can be solved in $O(n/2^k)$ time, so at depth k of the tree, $3^k * O(n/2^k) = (3/2)^k * O(n)$ time is spent to solve all subproblems.

This means that the total time spent to solve all subproblems is a geometric series, starting at $O(n)$ and ending at $O((\frac{3}{2})^{\log_2 n} * n) = O(3^{\log_2 n} * \frac{n}{n}) = O(3^{\log_2 n}) = O(3^{\log_3 n / \log_3 2}) = O(n^{1/\log_3 2}) = O(n^{\log_2 3})$, which is roughly $O(n^{1.59})$. Since the sum of any increasing geometric series is, within a constant factor, simply the largest term in the series, this is also the time complexity of solving all subproblems combined.

2.2 Master Theorem

The analysis above is standard for divide and conquer algorithms, so let's solve the general form and just reuse the results from here on. Most divide-and-conquer algorithms tackle a problem of size n by recursively solving subproblems of size n/b, then combining these answers in $O(n^d)$ time, for some $a, b, d > 0$ (in the multiplication algorithm, $a = 3$, $b = 2$, and $d = 1$). The running time is therefore $T(n) = aT(\lceil n/b \rceil) + O(n^d)$. We can now derive a closed-form solution for given values of a, b, d.

Master Theorem: If $T(n) = aT(\lceil n/b \rceil) + O(n^d)$ for some constants $a > 0$, $b > 1$, and $d \geq 0$, then

$$T(n) = \begin{cases} O(n^d) & d > \log_b a \\ O(n^d \log n) & d = \log_b a \\ O(n^{\log_b a}) & d < \log_b a \end{cases}$$

Proof: (For the sake of convenience, let's assume that n is a power of b, allowing us to ignore the rounding effect in $\lceil n/b \rceil$.)

As the size of the subproblems decreases by a factor of b with each level of recursion, there are $\log_b n$ levels. The tree has a branching factor of a, so at depth k there are a^k subproblems, each of size n/b^k. The total time complexity of all subproblems at depth k is $a^k * O(n/b^k)^d = O(n^d) * (a/b^d)^k$.

As k goes from 0 (the root) to $\log_b n$ (the leaves), these numbers form a geometric series with ratio a/b^d, and there are three cases:

1. $a/b^d < 1$: The sum is dominated by the first term, $O(n^d)$.
2. $a/b^d > 1$: The sum is dominated by the last term, $O(n^{\log_b a})$, as shown in the multiplication algorithm.
3. $a/b^d = 1$: All $O(\log n)$ terms of the series are equal to $O(n^d)$, and the total complexity is $O(n^d \log n)$.

3 Matrix Multiplication

Matrix multiplication can also be done with a divide-and-conquer algorithm. To recap, the product of two $n \times n$ matrices X and Y is a third $n \times n$ matrix $Z = XY$, where the (i, j)th entry is

$$Z_{ij} = \sum_{k=1}^{n} X_{ik} Y_{kj}$$

This formula implies an $O(n^3)$ algorithm for matrix multiplication, as there are n^2 entries, each taking $O(n)$ time. However, similarly to grade school multiplication, there is a more efficient divide-and-conquer algorithm.

Matrix multiplication is easy to break into subproblems, as it can be performed block-wise. We can carve an $n \times n$ matrix into four $n/2 \times n/2$ matrices:

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \qquad\qquad Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

And their product can be expressed in terms of these blocks and is exactly as if the blocks were single elements.

$$XY = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}$$

However, this particular divide-and-conquer algorithm has eight size-$n/2$ products and $O(n^2)$-time additions, resulting in the recurrence relation $T(n) = 8T(n/2) + O(n^2)$, which comes out to $O(n^3)$ as well.

Strassen discovered with very clever algebra, matrix multiplication can be broken down into only seven $n/2 \times n/2$ subproblems:

$$XY = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix}$$

where

$$\begin{aligned} P_1 &= A(F - H) \\ P_2 &= (A + B)H \\ P_3 &= (C + D)E \\ P_4 &= D(G - E) \\ P_5 &= (A + D)(E + H) \\ P_6 &= (B - D)(G + H) \\ P_7 &= (A - C)(E + F) \end{aligned}$$

The new recurrence relation is $T(n) = 7T(n/2) + O(n^2)$, which works out to $O(n^{\log_2 7})$, roughly $O(n^{2.81})$.

The intuition behind how he discovered this result is not in scope for this class.

4 Sorting

4.1 Mergesort

Let's try applying divide and conquer to some other familiar problems, such as sorting a list. The natural approach would be to divide the list into two halves, recursively sort each half, then somehow merge the two sublists together. This is precisely Mergesort. **Pseudocode:**

Algorithm 3 Mergesort

```
1: function MERGESORT(A[0, n − 1])
2:     if n = 1 then
3:         return A[0]
4:     else
5:         return MERGE( MERGESORT(A[0, ⌊n/2⌋ − 1]), MERGESORT(A[⌊n/2⌋, n − 1])
   )
```

To complete this description, we need to specify the merge step. How do we efficiently merge two sorted lists $A[0..n-1]$ and $B[0..m-1]$ into one sorted list $Z[0..m+n-1]$? The key observation is that the smallest element, $Z[0]$, must be $A[0]$ or $B[0]$, whichever

is smaller. Why? WLOG, let $A[0]$ be the smallest element. By A being sorted, $A[0]$ is smaller than every other element in A, and $A[0]$ is smaller than $B[0]$, which is smaller than every other element in B. Next, $Z[1]$ must be either $A[1]$ or $B[0]$ by the same logic. Notice our subproblem still has two sorted lists - we can solve this recursively.
Pseudocode:

Algorithm 4 Merge

```
function MERGE(A[0, n − 1], B[0, m − 1])
    if n = 0 then
        return B
    else if m = 0 then
        return A
    if A[0] ≤ B[0] then
        return A[0] ∘ MERGE(A[1, n − 1], B[0, m − 1])
    else
        return B[0] ∘ MERGE(A[0, n − 1], B[1, m − 1])
```

Note the circle sign there means concatenation.

Proof Sketch: The merge correctly produces the sorted list since we take the smallest number out of the remaining numbers at each step, as shown in the analysis in the section above. The recursive step follows by induction on n.

Runtime Analysis: The merge step takes $O(n)$ time, as each element is considered once and completed in constant time. The recursive step breaks down the problem into two subparts. Thus, $T(n) = 2T(n/2) + O(n) = O(nlogn)$ from Master Theorem.

As a final note, notice we do all the recursive steps before the merge step. So, during this algorithm's execution, this will create the entire tree of subproblems before ever merging. This suggests we could also do a "bottom-up" formulation where we just start at the singleton arrays and iteratively merge up to the final answer. This may be implemented with a queue.

Pseudocode:

Algorithm 5 Iterative Mergesort

```
function ITERATIVE_MERGESORT(A[0, n − 1])
    Q = []                                   ▷ initialize empty queue
    for i = 0 to n − 1 do
        Q.PUSH([A[i]])
    while |Q| > 1 do
        Q.PUSH(MERGE( Q.POLL(), Q.POLL() ))
    return Q.POLL()
```

5 Median and Selection

Given an unsorted list of numbers, suppose we would like to find some summary statistics to give us a sketch of the data. One choice is the arithmetic mean, i.e the total sum divided by the number of elements, which can be computed easily in $O(n)$. Another choice is the median, i.e the $\lfloor \frac{n+1}{2} \rfloor$-th smallest number, which is often preferable since it is more robust to outliers. However, the median is trickier to calculate - we could resort to an $O(n \log n)$ sort, but that is inefficient, as we likely don't need to construct the entire ordering to search for just a single key element. Let's design an $O(n)$ algorithm. To help us use recursion, we will generalize the median finding problem to selection.

A selection algorithm is an algorithm to find the kth smallest number (aka kth order statistic) in a list for any value of k. For example, $k = 1$ is the minimum element, $k = \lfloor \frac{n+1}{2} \rfloor$-th is the median, and $k = n$ is the maximum (note we are 1-indexing for k).

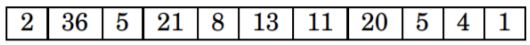
5.1 Quickselect

Let's use divide and conquer. Using our usual format, we could try dividing into two sublists and try to recursively find the kth smallest element in each. But, then how do we use those two to get the kth smallest element of the original list? There's not enough information here, so we'll try a different approach.

How else can we divide the list? One way is through the use of a partition on a pivot v: for a number v, we can split our list into three categories: elements smaller than v, elements greater than v, and elements equal to v.[6] There are ways to do this in $O(n)$, though they are not the focus of this class.[7] Note the sublists are not necessarily ordered.

Example: Splitting the following array on $v = 5$.

Figure 3: Array to be split. Source: DPV, pg. 64



yields the following subarrays

Figure 4: Subarrays after splitting. Source: DPV, pg. 64

This tells us an ordering on S_L, S_V, and S_R: every element in S_R is at least as large as the elements in S_V, and every element in S_V is at least as large as the elements in

[6] We could just use two groups by combining the equals group with one of the other groups, which would yield roughly the same results if there aren't many duplicates.

[7] Two ways include the Hoare and Lomuto partition schemes. This problem is called the Dutch National Flag Problem, proposed by Dijkstra.

S_L. Thus, we can narrow down our search to only one of these sublists. For example above, if $k = 7$, we only need to look at S_R, as the 7th smallest element is in the sublist containing the 6th - 11th smallest numbers. Explicitly,

$$\text{SELECTION}(S, k) = \begin{cases} \text{SELECTION}(S_L, k) & k \leq |S_L| \\ v & |S_L| < k \leq |S_L| + |S_v| \\ \text{SELECTION}(S_R, k - |S_L| - |S_v|) & k > |S_L| + |S_v| \end{cases}$$

So, our algorithm would be to pick a value v, partition our list with respect to v, then recurse on the relevant sublist. We haven't specified how to pick v yet, but are we on the right track to an efficient algorithm? Suppose we were able to split the list by roughly half on each recursive step. Then, $T(n) = T(n/2) + O(n)$ is equal to $O(n)$ by the Master Theorem.

5.2 Pivot Selection

5.2.1 Random (Quickselect)

How do we select v? We must do it quickly, and it should split the list well. One way would be just picking an element at random from the list.

Runtime: In the worst case, if we're really unlucky, our algorithm would pick the smallest/largest element as the pivot on every single iteration, reducing the size of our list by only one; this would make our algorithm run in $(n)+(n-1)+(n-2)+\cdots+1 = O(n^2)$. However, in the best case it picks the middle element every time, which was $O(n)$ from before. What we care about is the average case.

We'll need to calculate the expected running time. Our pivot is random, so to give us some more structure for analysis, let's define some boundaries: call a pivot that falls between the 25-75th percentile a "good" pivot. So, 50% of pivots are good, and it takes on expectation two recursive iterations to get a good pivot by expectation of a geometric distribution. A good pivot reduces our list by at least 25%. So, letting $T(n)$ be the expected runtime, $T(n) \leq T(\frac{3}{4}n) + O(n)$, which is $O(n)$ by Master Theorem. This algorithm is called Quickselect[8].

5.2.2 Deterministic (Deterministic-select) [EXTRA]

We can actually guarantee an $O(n)$ worst case with a clever scheme to get a pivot called median of medians as follows:

1. Group the array into $\lfloor n/5 \rfloor$ groups of 5 elements each

[8]If you are familiar with Quicksort, you might have seen a strong similarity with it and Quickselect - both are divide and conquer algorithms using a random partition with "quick" in their name. This is because they were created by the same person, Tony Hoare.

2. Find the median of each group (as each group has a constant number of elements, finding each individual median is $O(1)$)
3. Create a new array only with the $\lfloor n/5 \rfloor$ medians, and find the true median of this array by recursively calling the selection algorithm on it.
4. Return this median as the pivot.

This pivot is guaranteed to be "good", specifically in the 30-70th percentile. Why? It's greater or equal to than half of the medians, since it's the median of the medians. And each of those medians are greater or equal to the smallest three elements of their respective five element list, since they're the medians of their respective list. So, $p \geq \frac{1}{2} * \frac{n}{5} * 3$, thus $p \geq 3/10 * n$. With a symmetrical argument, $p \leq 7/10 * n$.

Runtime: Let's construct the recurrence. There are two recursive calls: one at step 3 of medians of medians to find the median of the medians, and one from our selection algorithm to recurse on our desired sublist. As for the work per recursive call, we do a linear scan to construct our list of medians, and another linear scan to do our partition in our selection algorithm. Thus $T(n) \leq T(n/5) + T(7n/10) + O(n)$.

We can't solve this with Master Theorem unfortunately. Instead, we can use induction to show $T(n) \leq c * n$ for some constant $c > 0$, thus proving it's $O(n)$ by the definition of Big-O.

Applying the inductive hypothesis, $T(n) \leq c(n/5) + c(7n/10) + d * n \leq (9/10 * c + d) * n$ for some constant $d > 0$.

Remember, we need to just show the existence of some c. To finish showing $T(n) \leq c * n$, we need to just pick c large enough that $(9/10 * c + d) \leq c$, i.e $c \geq 10d$. As for the base case, running deterministic-select on a single element list is trivially constant time.

In practice, efficient implementations of both these algorithms both perform well, with small preference towards Quickselect.[9]

6 Binary Search

Binary Search is the classic algorithm for finding a target number in a sorted list. When searching for a number in a sorted list, it is inefficient to iterate it in its entirety as there is useful structure in a sorted list. A single comparison, $A[i] < x$, tells us not only that x is greater than $A[i]$, but also everything to the left of it, allowing us to discard an entire subarray per comparison. To maximize our number of elements discarded in the worst case, we'll compare our target number to the middle element and recurse on

[9] We can also use selection for sorting, though it's not very efficient - if we iteratively select the ith smallest element for $i = 0..n-1$, we will have constructed the entire sorted list. This is Selection sort.

either the left or right subarray depending on the results of the comparison.[10] Here is an iterative pseudocode of the algorithm, though it can also be done recursively.

Pseudocode:[11]

Algorithm 6 Binary Search

```
1:  function BINARY_SEARCH(A[0, n − 1], k)
2:      l ← 0
3:      h ← n − 1
4:      while l < h do
5:          m ← ⌊(l + h)/2⌋
6:          if A[m] < k then
7:              l ← m + 1
8:          else if A[m] > k then
9:              h ← m − 1
10:         else
11:             return m
12:     return -1
```

[10] Quickselect and Binary Search are different from the other algorithms presented in this note, as their subproblems are not combined; instead, they discard all subproblems except for one, and recurse on that single subproblem. As such, some computer scientists prefer to label this type of algorithm as "*decrease* and conquer".

[11] Implementing binary search may be trickier than you'd expect due to edge cases. There was a bug in Java's implementation of binary search until 2006 due to integer overflow (our pseudocode does not account for it for sake of clarity, though it's an easy fix).

7 Exercises

1. Using Karatsuba's algorithm, find the product of $10 * 11$ (show your work).
2. Use the Master Theorem to solve the following recurrence relations:
   (a) $T(n) = 8T(n/2) + O(n^2)$
   (b) $T(n) = 4T(n/4) + O(n)$
   (c) $T(n) = 4T(n/2) + O(n^5)$
3. Devise a divide and conquer algorithm very similar to Mergesort to find the k smallest numbers in $O(nlogk)$ and analyze its runtime.
4. Run the Quickselect algorithm to find the 3rd smallest number (1-indexed) on $A = [3, 5, 2, 4, 1]$. Experiment what the algorithm's execution would be on different choices of pivots.
5. We define a peak element as an element that is at least as large as all of its neighbor(s). Given a list, devise an algorithm to return an index of a peak element (if there are multiple, just return one of them).

8 Solutions

1. $10 * 11 = 1 * 1 * 100 + (1 * 1 + 0 * 1) * 10 + 0 * 1 = 100 + 10 + 0 = 110$
2. Use the Master Theorem to solve the following recurrence relations:
   (a) $a = 8, b = 2, d = 2, d < \log_b a \rightarrow O(n^3)$
   (b) $a = 4, b = 4, d = 1, d = \log_b a \rightarrow O(n \log n)$
   (c) $a = 4, b = 2, d = 5, d > \log_b a \rightarrow O(n^5)$
3. Algorithm: Split the list into two sublists and find the k smallest numbers in each recursively. Merge the two sublists using the same operation as in Mergesort, but stop after k iterations. At the base case (i.e n/k lists of length k), use a sort to explicitly sort each lists

   Runtime analysis: Master theorem won't work here, since in our base case we do more work than in the rest of our subproblems. At the base case, we sort n/k sublists each of length k, yielding a runtime of $n/k * O(klogk) = O(nlogk)$ For the rest of the subproblems, the merge operation takes k time, as we stop at k iterations. Summing the work per recursion layer everything besides the base case then forms a geometric series $k + 2k + 4k + ... + n/2k * k$, which is $O(n)$. Thus, the runtime is dominated by the base case, $O(nlogk)$.
4. Here's one possibility. Suppose $v = 2$. We partition A into [1], [2], and [3,5,4], so we recurse into the sublist [3, 5, 4], now with $k = 1$. Suppose $v = 4$. Then, our partition yields [3], [4], and [5]. Recursing into [3] reaches our base case, so we return 3.
5. This is Spring 2021 Midterm 1 Question 2. Please refer to the solutions here: https://tbp.berkeley.edu/exams/7225/download/

Feedback Form: https://forms.gle/cM1io6eXyH65ytMP8

Contributors:

- Kevin Zhu
- Jing Yi Lim
- Axel Li